ACRN: A Big Little Hypervisor for IoT Development

Hao Li*
Intel China
hao.l.li@outlook.com

Xuefei Xu
Intel China
anthony.xu@intel.com

Jinkui Ren
Intel China
jack.ren@intel.com

Yaozu Dong
Intel China
eddie.dong@intel.com

Abstract

With the rapid growth of Internet of Things (IoT) and the new emerging IoT computing paradigm such as edge computing, it is prevalent to see that today's real-time and functional safety devices, particularly in industrial IoT and automotive scenarios, are getting multi-functional by combining multiple platforms into single product. The new trend potentially prompts embedded virtualization as a promising solution in terms of workload consolidation, separation, and cost-effective. However, hypervisors, such as KVM and XEN, are designed to run on a server and can not be easily restructured to fulfill the requirements such as real-time constrains from IoT products. Meanwhile, existing embedded virtualization solutions are normally tailored towards specific IoT scenarios, which makes them hard to extend towards various scenarios. In addition, most commercial solutions are mature and appealing but expensive and closed-source.

This paper presents ACRN, a flexible, lightweight, scalable, and open source embedded hypervisor for IoT development. By focusing on CPU and memory partitioning, and meanwhile optionally offloading embedded I/O virtualization to a tiny user space device model, ACRN presents a consolidated system satisfying real-time and general-purpose needs simultaneously. By adopting customer-friendly permissive BSD license, ACRN provides a practical industry-grade solution with immediate readiness. In this paper we will describe the design and implementation of ACRN, and conduct thorough evaluations to demonstrate its feasibility and effectiveness. The source code of ACRN has been released at https://github.com/projectacrn/acrn-hypervisor.

CCS Concepts • **Software and its engineering → Virtual machines**;

*Tencent(leehaoli@tencent.com). Work done while at Intel.

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313816

Keywords embedded virtualization, embedded hypervisor, device model, IoT

ACM Reference Format:
Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. 2019. ACRN: A Big Little Hypervisor for IoT Development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313816

1 Introduction

With the rapid growth of IoT and the vast amount of data generated by numerous IoT devices, edge computing [33] is deemed as a promising paradigm to facilitate data processing, largely or completely, on distributed edge devices. Therefore, it is common to see that today's real-time and functional safety devices, particularly in industrial IoT and automotive scenarios, are combining multiple platforms into single product, in order to fulfill the mixed criticality demands from embedded markets. Although functioning properly, such multi-platform solutions are normally expensive and complicated since they would require more space, weight, hardware, and cabling resources, which inevitably makes such solutions hard to maintain and extend. Fortunately, with the trend of multi-core processors being used in embedded devices, it is feasible to conduct functional partitioning on embedded system nowadays, so that multi-platform solutions can be realized on a single platform with the help of virtualization techniques. Therefore emerging embedded virtualization solutions begin to play important roles in IoT solutions [4][14][15], which could span across multiple domains such as industry, transportation, and so on.

Comparing to multi-platform solutions, virtualization based solutions show superior for some embedded system design and development [24]. By consolidating multiple virtual machines onto a single hardware platform, virtualization techniques can reduce cost and improve manageability. Meanwhile computing resource can be adjusted and optimized flexibly according to application needs. In addition, by allowing consolidation of legacy embedded software, e.g. RTOS, and modern commodity OSes, embedded virtualization can save development cost and reduce time to market.

Although promising, implementing an embedded virtualization solution, a.k.a. embedded hypervisor, is a challenging task. From a hardware perspective, there are enormous kinds of embedded hardware, and they are normally subjected to tight constrains, such as low memory footprint, low power

consumption, real-time, and so on. From a software perspective, there are many types of embedded OSes or RTOSes, such as Zephyr, QNX, Nucleus, VxWorks, and so on, targeting for various purposes. Hypervisors designed to run on servers, such as KVM and Xen, could not be easily restructured to meet the real-time and diversity requirements mentioned above, thus motivating the appearance of various embedded hypervisors [27][37][40][1][20][31][16].

Unfortunately, the intrinsic diversity of embedded systems has been well inherited by existing embedded hypervisors. Although many solutions available already, little could be reused and leveraged directly when designing a new virtualization based embedded solution. The main reason behind this issue is that most of the existing embedded hypervisors target for specific usage scenarios, and inevitably there are often trade-offs in their hardware/software co-design, which makes it very hard to port embedded hypervisors from one platform to another. In addition, different hypervisors might expose different virtual hardware interfaces, which makes software porting, between native and virtual environment, time-consuming and sometimes difficult. Therefore, this limited portability and insufficient adaptability lead to a need to have a generic embedded hypervisor towards IoT development, similar to server virtualization domain.

In this paper, we present ACRN (short for "acorn"), an embedded hypervisor for IoT development. After fully considering the substantial differences between embedded virtualization and datacenter-centric virtualization [14], and investigating existing solutions, we decide to design and develop ACRN from scratch, so that we could explore, innovate, and define the proper architecture, module interface, and so on. For example, we implement ACRN as a lightweight type-1 bare metal hypervisor [28], to pursue the benefits such as easy for industrial certification and minimized attack surface. Meanwhile we implement a small device model (DM) adopting virtio [32], a widely used and tested I/O paravirtualization standard mainly in type-2 hypervisors in cloud and desktop computing worlds, for virtualizing performance critical devices in ACRN. The benefit of such a design is that we can not only aggressively optimize the memory footprint, runtime overhead, and code size of ACRN in the first place, but also provide rich I/O virtualization capabilities. We also leverage hardware virtualization support to realize spatial and temporal isolations, which are fundamental for mixed criticality environment. Finally, we select x86 platform, which is popular in industrial IoT [34] and automotive fields [2][23], as our starting point, to leverage the experiences and practices [3][19] from server world. ACRN will be open to new platform architectures, such as ARM, MIPS, and so on, based on ACRN 's modular architectural design.

In summary, our paper makes the following contributions.

- It introduces ACRN, a flexible, lightweight, scalable, and open source (under permissive license) embedded hypervisor towards IoT development.
- It presents a secure and efficient "VM-Exit-less" solution for real-time (RT) virtualization, and shows how to combine the RT and non-RT virtualization together to form a mixed-criticality system.
- It conducts a thorough evaluation to show that ACRN is an industry-grade solution with immediate readiness.

The rest of the paper is organized as follows. Section 2 outlines the related works on embedded hypervisors. Section 3 describes ACRN 's design goal and system architecture. Section 4 introduces the design and implementation details of ACRN. The experimental results and overall evaluations of ACRN are discussed in Section 5, followed by future work and conclusion in Section 6 and 7.

2 Related Work

As an emerging and hot technology, there are multiple ways to realize embedded virtualization.

First, with the great success of Xen [3] and KVM [19] in cloud and desktop computing worlds, there are interests in porting existing datacenter-centric hypervisors to embedded world, such as RT-Xen [39], Xen-on-ARM [17], and embedded KVM [40], to obtain embedded hypervisors. These solutions allow running Linux, Android, and RTOS on ARM or x86 based embedded systems. This kind of approach is appealing when workload consolidation and software compatibility are preferred over simplicity.

However, existing solutions can not easily fulfill the real-time constrains from IoT usages. Meanwhile, the hypervisor restructuring process often involves with heavy source code modification and tailoring effort. For instance, the event channel in Xen is a powerful mechanism for cloud computing, but for embedded systems, it is heavyweight and hard to optimize due to its complicated API semantics.

Unlike existing solutions, ACRN is developed from scratch by implementing a series of lightweight virtualization primitives, with the help of hardware virtualization support, so that ACRN doesn't need to maintain code compatibility with any existing hypervisor. In addition, ACRN is realized as a type-1 hypervisor, similar to Xen, to have minimum code size and performance implications, which is critical for industrial certifications [9] in embedded world.

Second, given by the widespread usage of the industry-proven RTOSes in ARM or MIPS based embedded systems, extending existing RTOSes with virtualization capability is another promising way to realize embedded virtualization. There are many examples of this kind, such as commercial OKL4 Microvisor [16], QNX hypervisor [29], Green Hills INTEGRITY Multivisor [35], PikeOS [38], and Hellfire hypervisor [1]. Among these solutions, except for Hellfire hypervisor

only supporting MIPS architecture, all other ARM-based solutions have already been ported to x86 based embedded systems. Normally the RTOSes of this kind are based on microkernel architecture, and the virtualization capabilities are implemented as RTOS services or modules. Although based on existing RTOSes, it is still feasible for the existing solutions to earn industrial certification like ISO26262 [9], thanks to their small lines of source code, LoC, comparing to commodity OS like Linux.

Unlike existing solutions, ACRN is a pure type-1 hypervisor, instead of relying on any RTOS services. Thus it would be relatively easier to port ACRN to new hardware architectures since no need to port a RTOS first. Besides, not binding to specific RTOS, ACRN can work with various OS/RTOS on demand. In addition, ACRN has smaller LoC, which brings in the benefit of attack surface reduction and easy for certification. More importantly, ACRN is open and free while most commercial solutions are closed-source and expensive.

Third, similar to ACRN, there are hypervisors developed specifically towards embedded systems. The typical examples are open source Xvisor [27], XtratuM [37], Hermes [20], and commercial ETAS hypervisor [31]. One common characteristic of the existing solutions is that they are all type-1 hypervisors supporting para-virtualization. Besides, both Xvisor and ETAS hypervisor also support in-hypervisor device emulation, which further enriches their usage scenarios. However, none of the existing solutions has mature support towards x86 based embedded systems. ACRN takes x86 as a starting point and it supports rich I/O virtualization mechanisms including full emulation, para-virtualization, pass-through, mediated pass-through [36], and so on. Unlike Xvisor and ETAS hypervisor incorporating DM in the hypervisors, ACRN instead moves the major DM and tools into ACRN's service VM, a control VM similar to Xen's Dom0, so that DM crashes won't affect system stability.

Finally, there are some interesting explorations towards embedded virtualization. [6] proposes container-based virtualization as a lightweight alternative to embedded hypervisors, in order to make embedded software deployment and customization easier. However, containers are less isolated from one another than virtual machines since kernel is shared among all containers. [21] proposes a lightweight BIOS-level hypervisor for embedded SoC device, however it only targets at single VM virtualization towards SoC verification scenario. The Jailhouse hypervisor [30] proposes direct hardware assignment and static partitioning techniques to pursue real-time virtualization, and meanwhile leverages "virtio shared-memory transport" to realize inter-cell communication interface. Unlike Jailhouse only supporting pass-through devices and memory sharing across cells, ACRN's runtime "VM-Exit-less" solution towards real-time usage allows device pass-through and sharing simultaneously, which provides more flexibility.

3 System Architecture

Aiming at a generic embedded hypervisor towards various IoT segments, ACRN is designed with the following goals.

- **Big:** ACRN should be rich in functionalities in order to support a variety of IoT usages;
- **Little:** ACRN should be lightweight in code size and memory footprint, which is critical to pass industrial certifications;
- **Real-time:** ACRN should support real-time virtualization, in order to fulfill IoT specific needs;
- **Security:** ACRN should provide isolated environment for workloads in VMs, and ensure system-wide integrity to prevent compromised VM from booting;

To meet the design goals mentioned above, ACRN is designed as a type-1 hypervisor running on top of bare metal hardware and firmware, and the overall system architecture is shown in Figure 1. There are currently three types of VMs supported on ACRN hypervisor, service VM, user VM, and real-time VM (RTVM). Correspondingly we name the RTOS or OS running in them as Service OS (SOS), User OS (UOS), and real-time OS (RT-OS). The service VM, similar to Xen's Dom0, is a control VM where native device driver and ACRN DM are hosted. The user VM is an application VM hosting various OSes such as Linux, Android, Windows, and so on. The RTVM is a special isolated user VM optimized to offer real-time capability.

To offer "big" in functionalities, a lightweight userland program, ACRN DM, is developed to support a wide range of OS, including Linux, Android, Windows, and so on. Besides, to flexibly virtualize the diverse devices in embedded systems, ACRN offers rich I/O virtualization interfaces including virtio, full emulation, mediated pass-through [36], pass-through, and so on.

To achieve "little" in size and hence being easy towards industrial certifications, ACRN hypervisor relies on the service VM and the ACRN DM to manage physical hardware devices and provide I/O mediation to user VM and RTVM. The interaction between ACRN hypervisor and VMs is through clearly-defined portable hypercalls. For performance critical devices like virtual advanced programmable interrupt controller (vAPIC), ACRN implements them within hypervisor for fast responsiveness. To eliminate the implementation complexity, ACRN leverages hardware virtualization support, e.g. Intel VT, to realize virtualization capabilities. To avoid the complexity of restructuring service OS with para-virtualization capabilities, unlike Xen treats dom0 and domU differently, ACRN deprivileges service VM, together with user VM and RTVM, into VMX non-root operation mode in Intel VT, so that all VMs can be virtualized equally, which makes the code size of ACRN very little. With the above strategies, the ACRN hypervisor is simple and lightweight.

To support real-time virtualization, ACRN supports a special isolated user VM, RTVM, to guarantee the real-time

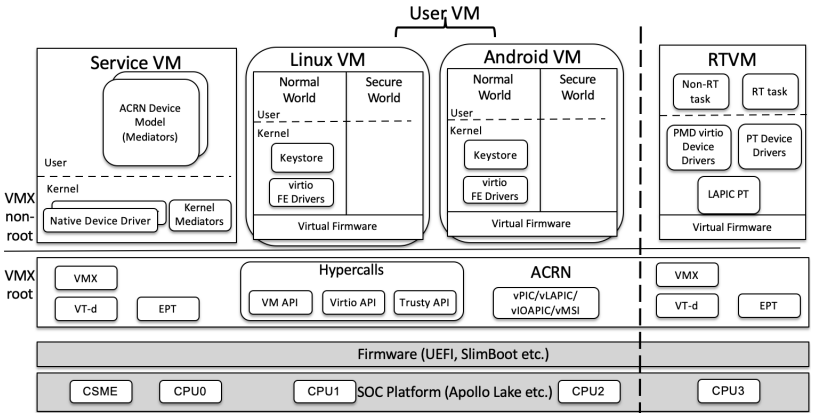

Figure 1. ACRN Architecture.

performance of applications in VMs. The RTVM intends to use minimal emulated device at boot time, and only accesses dedicated pass-through devices for real-time tasks at runtime. The RTVM enables a "VM-Exit-less" solution for real-time virtualization, which is critical for IoT specific usages.

To provide isolated environment for workloads in VMs, ACRN selectively supports secure world and normal world for user VMs like Linux and Android VMs. By offering a few world-related hypercalls, ACRN enables creation of privileged secure world and switch between secure world and normal world. To ensure system-wide security, ACRN supports verified boot process and leverages hardware support, such as converged security and management engine (CSME) [8] on Intel platform, as hardware root of trust. The verified boot process starts from hardware CSME, followed by native firmware, ACRN hypervisor, and service VM. In addition, ACRN extends the verified boot process to each user VM and RTVM, from virtual firmware, UOS/RT-OS, and applications.

4 Design and Implementation

In this section we describe ACRN's detailed design and implementation. We first describe how ACRN achieves spatial and temporal isolation through embedded virtualization techniques. Then we introduce ACRN's rich I/O virtualization interfaces, followed by ACRN's real-time support and optimization. Furthermore, we explain ACRN's security features.

4.1 Spatial Isolation

Spatial isolation is an important feature of virtualization to ensure that VMs only have access to hardware resources, such as registers and memory, which are assigned to them. This feature is fundamental in embedded virtualization since it allows IoT devices to securely collect and process sensitive user data, without worrying about other malicious or compromised VMs could inspect the privacy data.

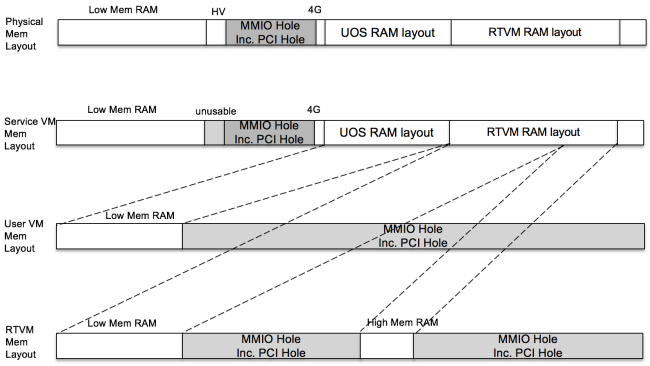

Figure 2. ACRN CPU Memory Partitioning.

ACRN offers spatial isolation mainly through CPU core and memory partitioning to satisfy the isolation demands from IoT usages.

4.1.1 CPU Core Partitioning

To support various CPU scheduling algorithms, particularly the user-configurable CPU core partitioning algorithm, ACRN hypervisor maintains a runqueue list for each physical CPU (PCPU). When a virtual CPU (VCPU) is created, ACRN will select a PCPU for it, according to user configuration, add VCPU into PCPU's runqueue, and then kick off the VCPU execution. When a VMEXIT happens on certain VCPU, ACRN hypervisor will gain control and decide how to schedule the next VCPU execution. For the CPU core partitioning algorithm, ACRN hypervisor simply re-attaches the VCPU onto the same PCPU's runqueue list, so that each VCPU can be binding to specific PCPU and VM.

In addition to CPU scheduling, ACRN leverages hardware virtualization support, such as Intel VT-x, to realize CPU virtualization. In ACRN, most CPU features, such as CPUID, MSR, and so on, are passed through to VMs, to avoid complex and heavyweight CPU emulation.

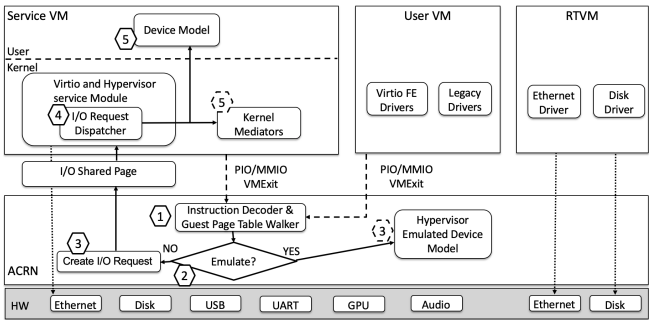

Figure 3. ACRN DM and I/O Handling Flow.

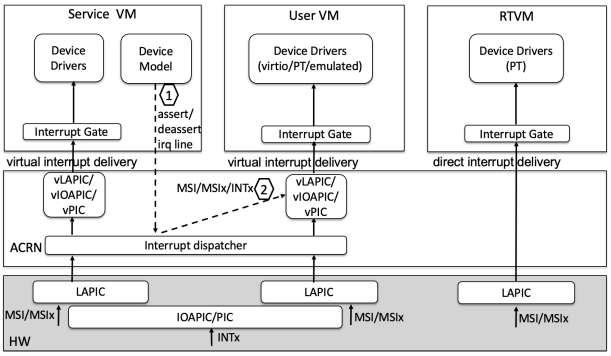

Figure 4. ACRN Interrupt Delivery Flow.

4.1.2 Memory Partitioning

ACRN 's memory partitioning feature aims at providing separation among non-service VMs, rather than the separation between service VM and other VMs, since service VM plays an administrative role in ACRN. To provide memory partitioning mechanism, ACRN leverages hardware-assisted virtualization techniques such as Intel EPT and VT-d. Meanwhile to simplify development, ACRN adopts a straightforward memory model, which is that except for hypervisor region, SOS owns all other system memory, and statically allocates memory for other VMs. Figure 2 shows an example with a user VM and a RTVM, and the memory of both are allocated in different high memory regions of SOS. The hypervisor region, which resides in the low memory region of SOS, is invisible to SOS since the hypervisor mapping is removed from SOS' E820 table, by modifying SOS kernel.

To further reduce the runtime overhead of memory virtualization, ACRN uses pre-built EPT table to map hypervisor region so that the hypervisor-introduced EPT violations could be few, except for the case of in-hypervisor emulated I/O devices, such as virtual RTC and virtual PIC. In addition, to support service VM access other VM's memory for para-virtualization, an SOS kernel module is developed to provide remote memory mapping API. Unlike Xen's powerful but complicated in-hypervsior memory virtualization mechanism, such as grant table, ACRN's remote memory mapping API leverages SOS' memory mapping mechanisms directly, since the memory region of other VM is essentially part of SOS' high memory region. The ACRN hypervisor and SOS kernel module will work together to ensure that the memory regions of other VMs will not be accessed or modified by SOS unintentionally.

4.2 Temporal Isolation

The goal of temporal isolation is to ensure the correct distribution of the CPU time between VMs with different criticality. To achieve this, in addition to assigning dedicated CPU core(s) to each VM, ACRN hypervisor needs to ensure the I/O access and interrupt will be scheduled with the correct VM priority.

4.2.1 I/O Flow

Figure 3 presents ACRN's I/O handling flow, which shows that ACRN handles the I/O flow of service VM, user VM, and RTVM differently.

For service VM, ACRN only traps its access to interrupt devices, shown as step 1~3, and lets other I/O accesses go directly to physical devices, shown as the dotted arrow in the figure. The interrupt devices include programmable interrupt controller (PIC), I/O advanced programmable interrupt controller (IOAPIC), and local advanced programmable interrupt controller (LAPIC). The benefit of virtualizing interrupt devices for service VM and user VM is that each VM can arbitrarily configure the virtual interrupt devices without worrying about interrupt vector conflicts. In addition, all the virtual interrupt devices are emulated within hypervisor for fast responsiveness.

For user VM, except for pass-through devices, ACRN traps all other I/O accesses for I/O virtualization, since all user VMs share the same hardware platform. The I/O access will be forwarded to in-hypervisor virtual interrupt devices, through step 1~3, or to the ACRN DM in SOS, through step 3~5. To trigger the I/O virtualization logic in ACRN DM, an I/O request will be created, by hypervisor, and delivered to I/O dispatcher in "Virtio and Hypervisor service Module(VHM)", an SOS kernel module offering remote memory mapping API, through virtual IRQ. To schedule the I/O access for user VMs, ACRN supports prioritized I/O distribution mechanism within the hypervisor and SOS VHM. By maintaining a ACRN-wide priority list, ACRN hypervisor can query and select the proper I/O request to distribute. In this way ACRN can ensure that high priority I/O request can be scheduled as fast as possible.

For RTVM, by default, ACRN doesn't need to trap any I/O access at runtime since all devices RTVM accesses are pass-through devices.

4.2.2 Interrupt Delivery Flow

ACRN handles interrupt delivery of service VM, user VM, and RTVM differently, as shown in Figure 4.

To schedule interrupts among VMs, ACRN hypervisor manages all physical interrupts by default. If RTVM is enabled, ACRN will pass through per CPU core LAPIC to RTVM, and meanwhile reserve a portion of IOAPIC vectors for RTVM, so that RTVM can access physical interrupt devices directly, without triggering any VMExit. In addition, as shown in the figure, RTVM only handles interrupts through MSI/MSIx.

For service VM and user VM, ACRN hypervisor handles two types of interrupts, physical interrupt from pass-through device, and virtual interrupt from ACRN DM. For the physical interrupt from pass-through device, hypervisor will map it from physical interrupt to virtual interrupt and then inject it into either service VM or user VM. For the virtual interrupt from ACRN DM, through MSI/MSIx or INTx, ACRN hypervisor will inject them into user VM through virtual LAPIC/IOAPIC/PIC, shown as the step 1~2 in the figure. To correctly and properly schedule the interrupts among user VMs, ACRN hypervisor maintains a ACRN-wide priority list for hypervisor to decide the right interrupt to deliver.

4.3 Rich Embedded I/O Virtualization

Embedded I/O virtualization differs from traditional server I/O virtualization in several ways. First, some embedded I/O devices to be virtualized are critical for embedded world but not commonly seen in server world. The typical examples include audio, USB, touch, CSME, and so on. Second, unlike standard and advanced hardware support in server world, the embedded platforms are normally versatile, and sometimes limited, in hardware support, such as APIC virtualization and posted interrupt support. Third, the typical usage case in embedded world is to run particular workload for particular functionality on particular hardware resource, thus the resource pass-though and partition are more preferred than resource migration and overcommit, such as VM migration and memory overcommit.

ACRN offers several solutions to address the unique challenges of embedded I/O virtualization.

First, to enable easy embedded I/O virtualization, similar to server world, ACRN DM supports four typical I/O virtualization mechanisms, including full emulation, para-virtualization, pass-through, and mediated pass-through [36], as shown in Figure 3. User can simply leverage her server I/O virtualization experience to develop embedded I/O virtualization on ACRN. From our experience, para-virtualization(virtio) is mostly used since most embedded I/O devices are performance-critical and meanwhile many existing virtio frontend drivers, available in Linux and FreeBSD, can be directly reused. To develop virtio backend driver, user can use ACRN's virtio backend service (VBS) framework, which includes two sets of API, VBS in userland (VBS-U) and VBS in kernel (VBS-K). Backend driver based on VBS-U API is easy to implement and debug since it is userland application, while the driver leveraging VBS-K API can achieve higher performance, but pay with a little kernel space development complexity. ACRN's VBS is compliant to virtio 0.95 and 1.0 specifications.

Table 1. ACRN Rich Embedded I/O Virtualization.

Devices	Virtualization Mechanism
LAPIC/IOAPIC	Full emulation/Pass-through
GPU	Mediated pass-through
Audio/Ethernet/Block	Virtio (in-kernel)-perf. critical
Touch/CSME	Virtio (in-user)-perf. critical
USB XHCI	Full emulation-unchanged driver
USB XDCI	Pass-through-unshared device

Second, regarding the embedded I/O devices not having standard hardware virtualization support, for example LAPIC and IOAPIC on platform without hardware APIC virtualization support (APICv), user can either emulate them in hypervisor for fast responsiveness, or pass-through them in RTVM for real-time performance, as mentioned in section 4.2.2 and 4.4.

Table 1 lists the major I/O devices supported on ACRN, and their virtualization strategies.

4.4 Real-Time

The RTVM in ACRN is a special isolated user VM optimized to pursue real-time performance. Comparing to regular user VM running commodity OS such as Linux, the RTVM has several distinct optimizations.

To optimize runtime real-time performance, RTVM is designed to access dedicated hardware resource, passed through by hypervisor, in order to avoid the additional latency and nondeterministic timing introduced by SOS, which plays a key role in serving user VMs with the hardware resource sharing. The dedicated hardware devices include LAPIC, disk, network, and so on. One key benefit of this manner is that VMExit can be eliminated completely. Therefore to adapt to the RTVM requirement mentioned above, RT-OS could be modified to only access dedicated hardware resource at runtime. For example, rather than using legacy interrupt and IOAPIC, RT-OS is encouraged to use MSI/MSIx only.

However, the drawbacks of only using dedicated devices in RTVM are obvious. The number of RTVM supported on ACRN is limited by the availability of hardware resource. What is worse, under certain industrial scenario, pass-through device, such as network card, is not only binding to specific RTVM, but also dedicated to specific RT task, for instance the congestion-less PLC (Programmable Logical Controller) control task. Consequently, RTVM may lose the flexibilities, normally offered by non-RT tasks, including in-field logging/debugging support, communication with outside world, and so on. Therefore how to guarantee the "VM-Exit-less" performance for RT-tasks and device sharing capability for non-RT tasks simultaneously is a challenging task. ACRN

addresses this challenge by leveraging virtio poll mode driver (PMD), originating from the Data Plane Development Kit (DPDK), to share device like network card with other VMs. The PMD consists of a frontend driver in RT-OS and a backend driver in ACRN DM. The frontend PMD driver only posts I/O request into virtio shared ring, without triggering any VMExit to notify the backend PMD driver, and immediately relaxes the CPU for other non-RT tasks. Similarly, the backend driver only processes the request in the shared ring without interrupting the frontend driver. Both frontend and backend PMD drivers will be waken up periodically to process the I/O request in the shared ring.

Another pass-through device needs special handling is LAPIC, since direct LAPIC pass-through might lead to a potential performance attack caused by RTVM continuously sending inter-processor interrupt (IPI) to CPUs belonging to other VMs. To address this issue, ACRN configures LAPIC into X2APIC mode, in which mode guest IPI can only happen with MSR instructions, which can be selectively intercepted and prohibited by ACRN hypervisor.

For devices that are unique but necessary for RT-OS boot up, RTVM relies on device emulation either in hypervisor or in ACRN DM. Fortunately, this kind of devices won't be used at runtime, and therefore it won't impact the latency and jitter of RT tasks. These devices include PCIe host bridge and RTC device from our experience supporting PREEMPT_RT Linux. If verified boot process is required for RTVM, it also needs ACRN DM's support to access virtual root of trust device, as mentioned in the next subsection.

4.5 Security

Currently ACRN targets two main security features, verified boot [12] and trusted execution environment (TEE) [11].

By supporting verified boot, ACRN ensures all executed code come from a trusted source. For service VM, the verified boot logic starts from hardware-protected root of trust, such as CSME, which verifies and hands over execution to bootloader. Then the bootloader initializes hardware platform, verifies ACRN hypervisor and SOS kernel image together, and passes control to ACRN hypervisor to launch service VM if the verification passes. For user VM, ACRN reuses service VM's verified boot process by ensuring that 1) a unique virtual SEED, used for integrity verification, can be retrieved by each user VM, and 2) a virtual root of trust is available for each user VM. The virtual SEED is generated by combining native SEED and VM UUID, and the virtual root of trust is emulated within ACRN DM.

ACRN selectively supports virtualization-based TEE, including normal world and secure world, for user VMs, as shown in Figure 1. The UOS, Linux or Android, can run in normal world, while the secure OS, such as Trusty [13], can run in secure world. The ACRN hypervisor acts as a secure monitor to isolate and switch two worlds to realize the "one VM, two worlds" architecture. By design, two worlds share the same CPU resources to avoid affecting other VMs. By maintaining separate VMCS and EPT data structures for each world, and meanwhile reserving a certain segment of guest memory as trusty-only-memory (TOM) for secure world, both worlds can be created and switched with ACRN's world-switch hypercall. In addition, secure world is able to access the entire memory of VM, whereas the normal world cannot access the TOM. Any access to TOM from normal world will lead to an EPT violation and ACRN will inject a page fault to normal world.

Table 2. HW/SW Configuration for ACRN Evaluation.

Hardware Info	
Board	Intel Apollo Lake NUC6CAYH
CPU	Intel Celeron Processor J3455
Cores	4
Memory	8GB
Storage	SSD 128GB
Software Info	
Hypervisor	ACRN/RT-Xen/KVM/Jailhouse
Device Model	ACRN DM/QEMU
Service OS	Linux 4.14
User OS	Linux 4.14
RT-OS	PREEMPT_RT Linux/LITMUS^{RT}
BIOS	UEFI/SBL
Benchmarks	UnixBench/IOzone/Cyclictest

5 Experimental Results

In this section we conduct evaluations of ACRN. We first describe the hardware and software configuration for our evaluation. Then we will mainly focus on the following 5 questions:

1) Is ACRN lightweight, comparing to prior systems?
2) How is ACRN's virtualization overhead?
3) Is ACRN scalable and how does its performance vary as more VMs being used?
4) How is ACRN's real-time performance?
5) How is ACRN's security overhead?

5.1 Experimental Setup

Table 2 shows the hardware and software setup for ACRN evaluation. For hardware platform, the ACRN evaluation is mainly based on Intel Apollo Lake platform, an x86 IoT platform. For software systems, other than ACRN hypervisor and DM, we use Linux 4.14 as both SOS and UOS, and choose Linux with PREEMPT_RT [10] patches as RT-OS.

For hypervisor performance comparison, we mainly target at KVM, Xen, and Jailhouse, since we successfully enabled them and tuned their performance on the same hardware.

Table 3. Single VM setup for ACRN Evaluation.

	User VM	RTVM	Service VM
VCPU #	1	1	1
Memory	2GB	2GB	2GB/Whole
Storage	4GB by virtio	Pass-through	Pass-through

Table 4. Hypervisor LoC Comparison.

	KVM	Xen	Xvisor	OKL4	Jailhouse	ACRN
LoC	17M	309K	356K	393K	38K	30K

Table 5. Device Model LoC Comparison.

	QEMU	Xvisor DM(in-hypervisor)	ACRN DM
LoC	1M	25K	43K

The OKL4 and Xvisor are well-known embedded hypervisors as well, but we only compare their lines of code with ACRN, since we didn't succeed in enabling them on the same hardware during this paper writing.

For performance evaluation, we mainly use UnixBench [22] for CPU and memory benchmarking. For I/O, we leverage IOzone [25] benchmark for storage device . For real-time evaluation, we use Cyclictest [7] as the benchmark.

Table 3 shows the single VM setup for ACRN evaluation. To unify the evaluations within this section, single user VM or RTVM is assigned with one VCPU and 2GB memory. For service VM setup, there are two cases. When evaluating performance of single service VM, it will be assigned with one VCPU and 2GB memory, through SOS kernel command-line configuration. When supporting other VMs, the SOS kernel will own all memory before allocating them to user VMs or RTVMs.

5.2 Lightweight

To verify if ACRN is lightweight, we compare it with the existing open source hypervisors and DMs. The comparison metric is lines of code (LoC), which can be easily calculated with "cloc" utility. Table 4 and Table 5 list the comparison results of ACRN hypervisor and DM respectively.

From the results it can be seen that ACRN is very lightweight. To most hypervisors in comparison, the LoC of ACRN hypervisor is around 10% of them, mainly because ACRN offloads the majority of I/O virtualization to the user space ACRN DM, and meanwhile offers spatial and temporal separations through CPU core and memory partitioning mechanisms. The LoC of Jailhouse hypervisor is comparable to ACRN, since Jailhouse is also a partitioning hypervisor but tightly binding to Linux. Unlike Jailhouse, ACRN doesn't rely on Linux to bootstrap itself, and supports various usage scenarios such as entertainment and real time.

The LoC of ACRN DM is comparable to the in-hypervisor DM in Xvisor, and smaller than the "giant" QEMU used in KVM and Xen, because ACRN DM only needs to make sure that all I/O virtualization mechanisms (emulation, para-virtualization, pass-through, and mediated pass-through) are supported, rather than emulating as many hardware platforms as possible. Another benefit of moving DM into user space is that the attack surface of hypervisor can be very small, which improves system security [26][18].

5.3 Baseline Virtualization Overhead

To evaluate the baseline virtualization overhead of ACRN, we measure the performance of single service VM, user VM, and RTVM respectively, and compare them with the native performance. To make a fair comparison, no user VM or RTVM is launched when evaluating the performance of single service VM, and no workload is running in service VM when evaluating performance of single user VM or RTVM.

For baseline performance evaluation, we thoroughly measure 1) interrupt overhead, 2) PIO/MMIO processing cost, and 3) CPU, memory, and I/O virtualization overhead.

5.3.1 Interrupt Overhead

To measure ACRN's interrupt overhead, we develop several micro-benchmarks within hypervisor to measure the extra interrupt handling time introduced by ACRN. The micro-benchmarks mainly record VMExit information, such as reason and frequency, so that they can be useful for I/O virtualization analysis as well. For interrupt latency analysis, we boot up each type of VM and run these micro-benchmarks for a period of time T, for example 80 seconds in our experiment, and collect interrupt related data shown in Table 6. For each factor causing extra interrupt handling time, we mainly focus on the following 3 metrics, 1) $M1$: how many times it happens per second; 2) $M2$: how many CPU cycles spent in ACRN to handle it; and 3) $M3$: time percentage of each factor in the time interval T.

In ACRN, virtualization introduced extra interrupt handling time could be mainly derived from two events, "External Interrupt" and "Interrupt Window". The "External Interrupt" metric indicates VMExit caused by either physical interrupt from any pass-through device, or IPI interrupt from other CPU core for virtual interrupt injection. However, the virtual interrupt injection to VMs may not be successful and thus become pending within current VMExit-VMEntry interval, since guest might be under interrupt disable state or within its interrupt service routine. The pending interrupt needs to be re-injected again by ACRN hypervisor with the help of hardware "Interrupt Window" support, and this is why "Interrupt Window" metric is taken into account as well when calculating ACRN's interrupt handling time.

Table 6. ACRN Interrupt Overhead Measurement with Micro-benchmarks.

VMExits due to	Service VM			User VM			RTVM		
	M1	*M2*	*M3*	*M1*	*M2*	*M3*	*M1*	*M2*	*M3*
External Interrupt	4480	2331	0.47%	258	2584	0.03%	0	0	0.00%
Interrupt Window	6240	614	0.17%	541	724	0.02%	0	0	0.00%

From the data we can observe that more "External Interrupt" VMExits happen within service VM, than user VM and RTVM, and thus more time is spent in ACRN hypervisor. As explained in section 4.2.2, both service VM and user VM share the physical interrupt devices through ACRN's device emulation, and RTVM owns dedicated physical interrupt device exclusively to pursue real-time performance. Service VM owns most physical devices so that more virtual interrupts, mapped from physical interrupts, are injected into service VM by hypervisor. On the other side, RTVM can use dedicated interrupt devices without triggering any VMExit.

One interesting result is that the "Interrupt Window" event happens more frequently than the "External Interrupt" event in both service and user VM. The reason is that our evaluation platform doesn't support posted-interrupt feature, without which ACRN has to leverage "Interrupt Window" feature to trigger VMExit from VM, and then re-injects the pending interrupt later. On hardware platforms with posted-interrupt support, this overhead can be efficiently eliminated.

Generally the interrupt handling time of ACRN hypervisor is lightweight, occupying time less than 0.8% in service VM, and less than 0.1% in user VM. For RTVM, ACRN offers a VM-Exit-less solution for real-time performance.

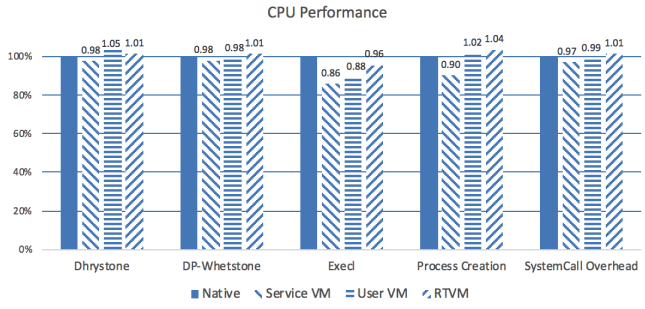

Figure 5. Normalized CPU Performance.

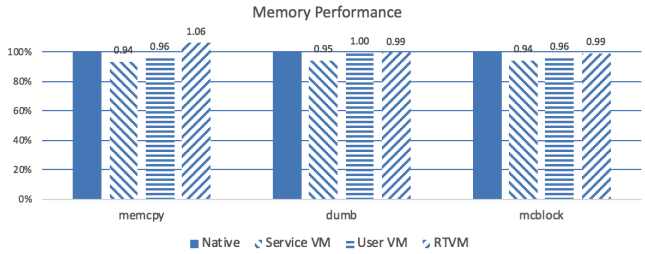

Figure 6. Normalized Memory Performance.

5.3.2 PIO/MMIO Processing Cost

To evaluate ACRN 's PIO/MMIO processing cost, we reuse the micro-benchmarks and metrics for interrupt analysis and measure ACRN's I/O processing time, shown in Table 7. The PIO or MMIO requests are dependent on OS behavior, thus there might be zero request captured during our experiment. Besides, since GPU virtualization is only used for user VM, rather than service VM and RTVM, "N/A" is used to denote this case.

From the data we can see that less PIO or MMIO caused VMExits happen in service VM, comparing with user VM, since most physical devices, except for interrupt devices, are pass-through devices to service VM. For RTVM no VMExit is captured, since RTVM by default accesses dedicated devices at runtime without triggering any VMExit.

Unlike service VM and RTVM, except for pass-through device, all other devices seen by user VM are virtualized by ACRN DM. Hence most PIO and MMIO access from UOS will cause a VMExit, which enables ACRN to kick-off the I/O flow mentioned in section 4. This is why we find relatively large CPU cycles waiting for I/O request to be processed. Among the virtual I/O devices, virtual GPU is one of the heaviest I/O devices for user VM, and it occupies around 23% of the user VM time.

5.3.3 CPU/Memory Performance

Figure 5 and Figure 6 show the CPU and memory performance comparisons between native and VMs. For each VM data, we run the same OS and applications between native and VM, and we normalize VM data to native data. From the data we can see that both CPU and memory performance of VMs are comparable, and they achieve similar results comparing to the native performance.

5.3.4 I/O Performance

To measure ACRN I/O performance, we take storage device as an example since it is a fundamental I/O device in ACRN. Figure 7 shows the normalized disk I/O performance comparison between native and VMs. From the figure we can see that both service VM and RTVM achieve comparable disk I/O performance to native platform, since the storage device is pass-through device to them. The user VM's disk I/O is

Table 7. ACRN I/O Processing Measurement with Micro-benchmarks.

VMExits due to	Service VM			User VM			RTVM		
	*M*1	*M*2	*M*3	*M*1	*M*2	*M*3	*M*1	*M*2	*M*3
PIO	27	374	~0.00%	262	514004	5.98%	0	0	0.00%
MMIO (by vGPU)	N/A	N/A	N/A	1874	283161	23.56%	N/A	N/A	N/A
MMIO (by other I/O)	0	0	0.00%	5	717270	0.10%	0	0	0.00%

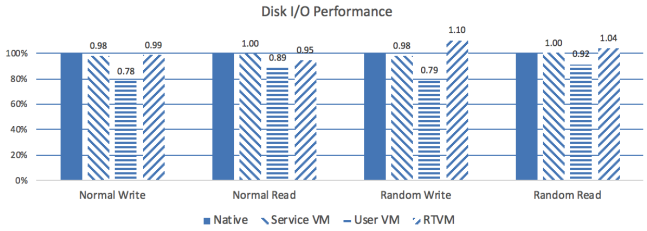

Figure 7. Normalized Disk I/O Performance.

based on ACRN virtio API, and its average read performance is around 90% of the native performance, whereas the average write performance is around 80% of the native. The read performance of user VM is better than write performance in that we enforce a write-through policy for each write operation from user VM, in order to prevent possible data loss in case a sudden power-off event happens.

5.4 Scalability

To evaluate the scalability of ACRN, we launch multiple VMs on our evaluation platform, and examine if there is any performance interference among them. Due to the various possible combinations of service VM, user VM, and RTVM, we mainly focus on the combinations shown in Figure 8. Our platform has four CPU cores, so we can launch one service VM and three user VMs at most. For RTVM which occupies dedicated hardware exclusively, we can only launch one service VM and two RTVMs at most, limited by the number of storage devices, which is three on our platform. To make a fair comparison, each VM is assigned with one CPU core and 2GB memory. In addition, same benchmarks are executed simultaneously within each VM.

Figure 8 (a) shows the scalability results of service VM. From service VM's results we can see that CPU performance is not affected no matter how many user VMs are used. However, both memory and I/O performance are gradually degraded as more user VMs are launched.

For the memory performance degradation, we observe that when three UOSes executing the same memory benchmark as SOS, only one UOS among the three will cause SOS memory performance to drop around 50%. The reason is that the CPU on our evaluation platform shares L2 cache per two cores, which means two CPU cores will compete for memory bandwidth. In our experiment, two cores are assigned to service VM and one user VM respectively, and each VM could get roughly 50% bandwidth since they run same memory benchmark. Except for this hardware limitation, other two user VMs won't affect the memory performance of service VM too much, less than 10%, indicating that ACRN adds very limited overhead to memory performance.

For the I/O performance degradation, there are mainly two reasons. First, service VM and user VMs share the same hardware platform through ACRN DM, thus they will compete for the shared resources. Second, the ACRN DM lacks of quality-of-service (QoS) support, which is a work-in-progress feature during our evaluation. Without QoS, the I/O request from user VMs will be eventually queued in SOS. That is why we observe the gradual descent in I/O performance. However, the I/O degradation can be avoided by device pass-through, as shown in Figure 8 (c).

Figure 8 (b) shows the results of user VM. Similar to service VM results, the CPU performance of user VM is not affected as more user VMs being launched, and the I/O performance is degraded due to lacking of QoS support within ACRN DM. Interestingly, the memory performance of the first user VM is not affected by another two user VMs. As explained above in the service VM case, this is because the CPU core of the first user VM share the same L2 cache with the CPU core of service VM. We select another user VM as our target and rerun the memory performance test, and we observe around 50% performance drop as certain user VM being activated, as expected.

Figure 8 (c) shows the RTVM related results. Since RTVM by default accesses dedicated hardware device exclusively at runtime, no performance interference is measured between the two RTVMs.

5.5 Real Time

To evaluate the real-time performance, we compare ACRN with native and several popular real-time hypervisors including Jailhouse, RT-Xen, and KVM. To collect real-time data, we use PREEMPT_RT [10] Linux and LITMUS^{RT} [5] as RT-OSes and leverage the popular cyclictest, which is considered the gold standard of real-time Linux performance [7], to report scheduling latency. Table 8 shows the detailed configuation. To obtain the best performance of competitors, we use LITMUS^{RT} as Xen Domain OS, and add "$isolcpus = 1$

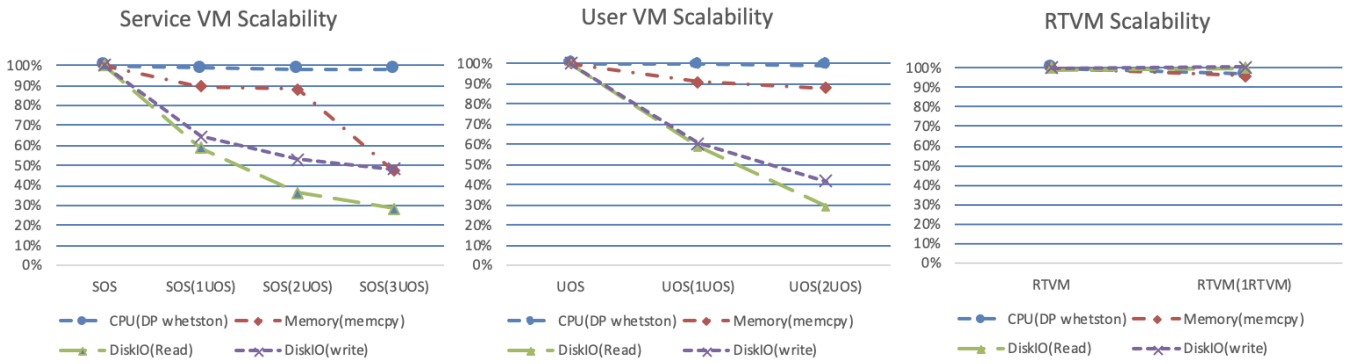

Figure 8. VM Performance Scalability: (a) Service VM (b) User VM (c) RTVM

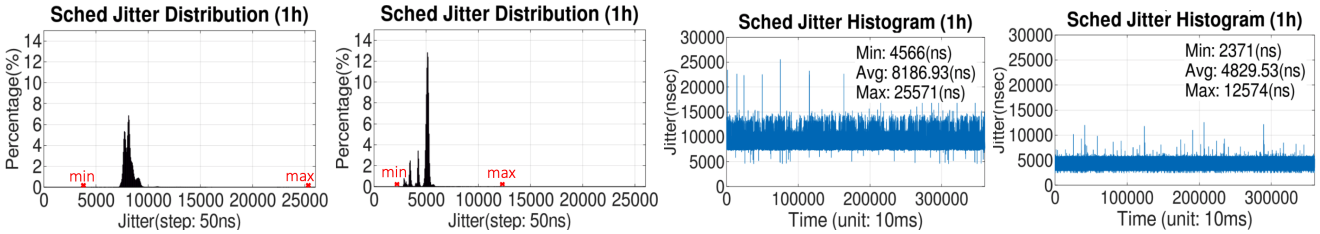

Figure 9. Jailhouse vs. ACRN: (a) Jailhouse (b) ACRN (c) Jailhouse (d) ACRN

Table 8. Real-time Configuration.

Target	Dom0/host/ root cell/SOS	DomU/guest/ non-root cell/RT-OS
Native	PREEMPT_RT	N/A
RT-Xen v2.2	LITMUS^{RT}	LITMUS^{RT}
KVM	Linux 4.14	PREEMPT_RT
Jailhouse	PREEMPT_RT	PREEMPT_RT
ACRN	Linux 4.14	PREEMPT_RT

$irqaffinity = 0\ idle = poll$" to KVM's host kernel command-line to pin real-time tasks to specific CPU core and meanwhile to prevent OS from sleeping. We run each measurement for one hour and deem native performance as the baseline jitter, before collecting jitter data introduced by hypervisors.

Table 9 shows the scheduling latency from cyclictest in microseconds. The "Max" column indicating the worst-case latency is the most important data since it represents jitter introduced by hypervisors. Therefore, we calculate the rightmost column by comparing the worst-case latency between native and VMs. From the results we can see that ACRN RTVM achieves the best performance, followed by Jailhouse. Both are comparable to native data, which shows that the jitter introduced by hypervisors are negligible, thanks to their "VM-Exit-less" solutions. To further evaluate the contribution of ACRN's LAPIC pass-through, we implement a special RTVM using virtual LAPIC, similar to normal user VM. The data shows that RTVM's LAPIC pass-through significantly improves RTVM's real-time performance, twice at least, comparing to RTVM using virtual LAPIC. The performance of KVM real-time guest is better than RT-Xen, after we pin real-time tasks to specific CPU core. The performance of RT-Xen DomU is the worst in our evaluation, and we suspect it is because RT-Xen is still a work-in-progress project.

To further understand the performance difference between Jailhouse and ACRN, we plot their scheduling jitter distribution and histogram figures, shown in Figure 9. From the results we can see that ACRN's jitter is more stable and lower than Jailhouse's jitter. To further investigate the reason, we find that Jailhouse generates quite a few VMExits at runtime, whereas ACRN doesn't. Jailhouse's VMExit is caused by LAPIC working in XAPIC mode, so that Jailhouse cell has to access LAPIC through MMIO, which triggers VMExit. In contrast, for security reason mentioned in section 4.4, ACRN configures LAPIC to work in X2APIC mode, so that RT-OS accesses LAPIC through MSR, which is passed through to RTVM and thus no VMExit happens. The reason that both hypervisors configure LAPIC differently is because Jailhouse relies on Linux to configure LAPIC, whereas ACRN configures LAPIC itself. By default, Linux will configure LAPIC to work in X2APIC mode when the number of CPU cores is

Table 9. The cyclictest results (scheduling latency) in microseconds.

Target	Min	Ave	Max	Normalized Sched Latency Jitter
native	3.14	5.41	11.82	1
RT-Xen DomU	12.00	435212.00	855145.00	72347.29
KVM real-time guest	8.00	21.00	417.00	35.28
Jailhouse non-root cell	4.57	8.20	25.57	2.16
ACRN User VM	5.81	57.75	6548.20	553.99
ACRN RTVM + vLAPIC	6.11	9.50	37.95	3.21
ACRN RTVM	2.37	4.83	12.57	1.06

Table 10. ACRN Boot Time and Verified Boot Overhead in milliseconds.

Boot Phase	No VB	δVB Extra Time	δVB/No VB
Bootloader	750	50 (phase 2)	7%
Hypervisor	55	N/A	N/A
Service VM	2150	N/A	N/A
User VM	2510	1089 (phase 3)	43%
RTVM	2265	950 (phase 3)	42%

above certain amount, such as 256, which is not embedded-friendly. After we force Jailhouse to work in X2APIC mode, we observe Jailhouse is as "VM-Exit-less" as ACRN on x86 architecture.

5.6 Security Overhead

ACRN's TEE is under development to ensure all secure applications, e.g. Trusty applications, are fully functionally during this paper writing, thus we leave its evaluation as our future work. This section evaluates ACRN's verified boot overhead.

ACRN relies on its verified boot (VB) process to ensure all executed code come from a trusted source. There are mainly three verification phases involved during ACRN system boot: 1) hardware root of trust verifying native bootloader, 2) native bootloader verifying ACRN hypervisor and SOS, and 3) virtual bootloader verifying UOS and RT-OS. The phase 1) verification is carried out fully by hardware and thus its execution time is too short to be measured, therefore we only show the overhead of phase 2) and phase 3) in Table 10.

From the data we can see that the overhead of phase 3) is much larger than phase 2) overhead. The reason is that the native bootloader leverages hardware engine to speed up the encryption/decryption algorithms, while the virtual bootloader only uses software implementations. The hardware acceleration to virtual bootloader will be added in the future. Generally the overhead of each verified boot phase, 7%, is acceptable if hardware acceleration is enabled.

6 Future Work

While we believe ACRN offers a practical industry-grade solution, more future work remains.

First, ACRN aims at achieving functional safety in order to support safety-critical embedded system development. One big prerequisite of this goal is to make ACRN comply with MISRA-C standard, which is a subset of C99 standard aiming to facilitate code safety, security, portability, and reliability in the context of embedded systems. To realize this, we are in the process of addressing and fixing the coding violations, discovered by MISRA-C test suite.

Second, although both secure world and normal world can be created and switched within one user VM on ACRN, we are still in the process of ensuring all secure applications are fully functional. Thus we leave the thorough evaluation of ACRN TEE as our future work.

Third, although start from x86 architecture, ACRN is not tightly binding to x86 platform. It is very open to other CPU architectures such as ARM, MIPS, and so on. In addition, ACRN is not binding to Linux. Based on the clearly-defined interface between ACRN and SOS, users have the flexibility to leverage different OS or RTOS as SOS.

7 Conclusion

This paper presents ACRN, an embedded hypervisor towards IoT development. By offering a flexible, lightweight, real-time, and scalable design, ACRN provides a practical industry-grade solution with immediate readiness. In this paper we have elaborated the design and implementation of ACRN's key components, and the evaluation results have showed the effectiveness of ACRN as a generic embedded virtualization solution.

Acknowledgments

We thank the anonymous reviewers and our shepherd Tim Merrifield for valuable feedback. Like most production scale systems, there are many more contributors than the authors of this paper: Christopher Cormack, Matthew Curfman, Jeff Jackson, Jason Chen, Fengwei Yin, Yu Wang, Lei Rao, and the virtualization team of Intel open source technology center.

References

[1] A. Aguiar, S. J. Filho, F. G. Magalhães, T. D. Casagrande, and F. Hessel. 2010. Hellfire: A design framework for critical embedded systems' applications. In *2010 11th International Symposium on Quality Electronic Design (ISQED)*. 730–737. https://doi.org/10.1109/ISQED.2010.5450495

[2] GENIVI Alliance. 2018. GENIVI Compliant™ Products. https://www.genivi.org/compliant-products.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. 164–177.

[4] Manfred Broy. 2006. Challenges in Automotive Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1134285.1134292

[5] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*. IEEE Computer Society, Washington, DC, USA, 111–126.

[6] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito. 2016. Exploring Container Virtualization in IoT Clouds. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. 1–6. https://doi.org/10.1109/SMARTCOMP.2016.7501691

[7] Felipe Cerqueira and Björn B. Brandenburg. 2013. A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUS RT. In *Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT'13)*. 19–29.

[8] Intel Corporation. 2017. Getting Started with Intel® Active Management Technology (AMT). https://software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt.

[9] International Organization for Standardization. 2011. 26262: Road vehicles-Functional safety. https://www.iso.org/standard/43464.html.

[10] The Linux Foundation. 2017. PREEMPT RT. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup/.

[11] GlobalPlatform. 2018. Trusted Execution Environment. https://globalplatform.org/specifications/technical-overview/.

[12] Google. 2017. Verified Boot. https://source.android.com/security/verifiedboot/.

[13] Google. 2018. Trusty TEE. https://source.android.com/security/trusty/.

[14] Gernot Heiser. 2008. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES '08)*. ACM, New York, NY, USA, 11–16. https://doi.org/10.1145/1435458.1435461

[15] G. Heiser. 2011. Virtualizing embedded systems - why bother?. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 901–905.

[16] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems (APSys '10)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/1851276.1851282

[17] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park, and C. R. Kim. 2008. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *2008 5th IEEE Consumer Communications and Networking Conference*. 257–261. https://doi.org/10.1109/ccnc08.2007.64

[18] Kenta Ishiguro and Kenji Kono. 2018. Hardening Hypervisors Against Vulnerabilities in Instruction Emulators. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec'18)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3193111.3193118

[19] Avi Kivity Qumranet, Yaniv Kamay Qumranet, Dor Laor Qumranet, Uri Lublin Qumranet, and Anthony Liguori. 2007. KVM: The Linux virtual machine monitor. 15 (01 2007).

[20] Neil Klingensmith and Suman Banerjee. 2018. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications (HotMobile '18)*. ACM, New York, NY, USA, 101–106. https://doi.org/10.1145/3177102.3177103

[21] Hao Li, Dong Tong, Kan Huang, and Xu Cheng. 2010. FEMU: A Firmware-based Emulation Framework for SoC Verification. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS '10)*. ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/1878961.1879007

[22] Byte Magazine. 2018. UnixBench. https://github.com/kdlucas/byte-unixbench/.

[23] IHS Markit. 2019. Teardown - Tesla 2018 Model 3 Autopilot and Media Controller. https://technology.ihs.com/607719/teardown-tesla-2018-model-3-autopilot-and-media-controller.

[24] Carlos Moratelli, Sergio Johann, Marcelo Neves, and Fabiano Hessel. 2016. Embedded Virtualization for the Design of Secure IoT Applications. In *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype (RSP '16)*. ACM, New York, NY, USA, 2–6. https://doi.org/10.1145/2990299.2990301

[25] William Norcott and Don Capps. 2016. IOzone. http://www.iozone.org/.

[26] Junya Ogasawara and Kenji Kono. 2017. Nioh: Hardening The Hypervisor by Filtering Illegal I/O Requests to Virtual Devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, New York, NY, USA, 542–552. https://doi.org/10.1145/3134600.3134648

[27] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. 2015. Embedded Hypervisor Xvisor: A Comparative Analysis. In *Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '15)*. IEEE Computer Society, Washington, DC, USA, 682–691. https://doi.org/10.1109/PDP.2015.108

[28] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM* 17, 7 (July 1974), 412–421. https://doi.org/10.1145/361011.361073

[29] QNX. 2017. QNX Hypervisor. http://blackberry.qnx.com/en/products/hypervisor/index.

[30] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look Mum, no VM Exits! (Almost). *CoRR* abs/1705.06932 (2017). arXiv:1705.06932 http://arxiv.org/abs/1705.06932

[31] D. Reinhardt and G. Morgan. 2014. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. 189–198. https://doi.org/10.1109/SIES.2014.6871203

[32] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[33] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3 (2016), 637–646.

[34] SIEMENS. 2017. Industrial PCs for the Digital Factory. https://w3.siemens.com/mcms/automation/en/pc-based-automation/Documents/simatic-ipc-en.pdf.

[35] Green Hills Software. 2018. INTEGRITY Multivisor. https://www.ghs.com/products/rtos/integrity_virtualization.html.

[36] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 121–132. http://dl.acm.org/citation.cfm?id=2643634.2643647

[37] S. Trujillo, A. Crespo, and A. Alonso. 2013. MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems. In *2013 Euromicro Conference on Digital System Design*. 260–265. https://doi.org/10.1109/DSD.

2013.37

[38] Freek Verbeek, Oto Havle, Julien Schmaltz, Sergey Tverdyshev, Holger Blasum, Bruno Langenstein, Werner Stephan, Burkhart Wolff, and Yakoub Nemouchi. 2015. Formal API Specification of the PikeOS Separation Kernel. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 375–389.

[39] S. Xi, J. Wilson, C. Lu, and C. Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. 39–48.

[40] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan. 2010. Performance analysis towards a KVM-Based embedded real-time virtualization architecture. In *5th International Conference on Computer Sciences and Convergence Information Technology*. 421–426. https://doi.org/10.1109/ICCIT.2010.5711095