iSotEE: A Hypervisor Middleware for IoT-enabled Resource-constrained Reliable Systems

YIXIAO LI¹, AND HIROAKI TAKADA², (Member, IEEE)

¹Graduate School of Informatics, Nagoya University, Nagoya, 464-8601 Japan (e-mail: liyixiao@ertl.jp)
²Institutes of Innovation for Future Society, Nagoya University, Nagoya, 464-8601 Japan (e-mail: hiro@ertl.jp)

Corresponding author: Hiroaki Takada (e-mail: hiro@ertl.jp).

ABSTRACT Embedded systems for critical applications are often based on resource-constrained devices to meet the requirements like performance predictability and energy consumption. To deal with the increased software complexity, many of these systems have adopted reliable RTOSes (Real-Time Operating Systems) with advanced protection functionalities. Meanwhile, the concept of IoT (Internet of Things) is gaining momentum. Many IoT OSes, specialized to provide the large software stack required by IoT applications, have been released. Nevertheless, neither reliable RTOS nor IoT OS can satisfy all the requirements of IoT-enabled reliable systems. Dual-OS configuration (i.e. the coexistence of reliable RTOS and IoT OS) is a promising approach to achieve high reliability and productivity simultaneously. Existing dual-OS solutions, however, depend on additional hardware features (e.g. virtualization extensions, ARM TrustZone), which are unavailable in most resource-constrained devices. This paper presents iSotEE (iSolated Execution Environment), a middleware allowing IoT OS to run inside an isolated environment on top of a reliable RTOS without special hardware. Open-source implementations of iSotEE for Renesas RX (with TOPPERS/HRP3 as reliable RTOS, Amazon FreeRTOS as IoT OS) and ARMv7-M (with two configurations of Zephyr as reliable RTOS and IoT OS) architectures are provided and evaluated. The results show that iSotEE can create reliable systems with a small footprint for resource-constrained devices, high real-time performance for critical applications, and high productivity and throughput for IoT applications.

INDEX TERMS Embedded software, Internet of Things, Real-time systems, Reliability.

I. INTRODUCTION

Embedded systems have been widely used to support critical applications such as industrial control [1], medical devices [2] and national infrastructures [3]. Typically, these systems are required to operate in harsh environments continuously (e.g. for years) and reliably (e.g. ultra-low unrecoverable error rate) [4]. To meet the SWaP-C (size, weight, power and cost) constraints, many embedded systems are based on hardware devices with very limited resources (known as “resource-constrained devices”) [5]. Moreover, hard real-time performance is important when critical applications interact with the physical world [6]. The processors in resource-constrained devices have relatively simple and deterministic design, which can also offer advantages in guaranteeing real-time performance. For example, small on-chip memory can deliver lower latency and higher predictability than external memory [7].

Critical systems must achieve high reliability because a failure could lead to severe consequences (e.g. human injury, property damage). Therefore, the development process typically requires the software to be sufficiently verified...
before releasing the products [8]. The verification cost is expensive and grows fast as the assurance level and software size increase [9]. Contemporary critical systems usually include multiple applications at different assurance levels [10]. Many of them have adopted RTOSes (real-time operating systems) with advanced features for high reliability (hereafter, “reliable RTOSes”) to efficiently deal with the complexity [11]. The partitioning feature in a reliable RTOS, as shown in Fig. 1, allows isolating software components in different domains. Each domain only has access to the granted resources, and is prevented from interfering with another domain. It enables the containment of faults, which is essential to improve the reliability. The isolation also reduces the verification cost, since a domain can be verified separately according to its assurance level, and modifying one domain does not require the whole system to be reverified. The privileged components can access all resources, and are often referred to as the TCB (Trusted Computing Base). The TCB always has the highest assurance level since the reliability of such a system depends on its correctness. Therefore, to keep the verification process affordable, the TCB must be stable (i.e. less frequently updated) and small.

Meanwhile, the concept of IoT (Internet of Things) is gaining momentum in resource-constrained critical systems, because connecting those devices to the Internet has many benefits [12]. Monitoring and managing devices from the cloud can efficiently improve the maintainability and availability [13]. Big data analytics is helpful for creating smart and optimized services [14]. Over-the-air (OTA) updating allows the software to be patched faster, which mitigates the risks of bugs and vulnerabilities [15]. These connected devices, however, are also becoming attractive targets of cyberattacks since they can access valuable assets and sensitive data [16]. Therefore, adding IoT features to a critical system imposes a higher requirement on the reliability.

Building an IoT application typically needs a lot of components (e.g. middleware, device drivers) for common features like connectivity and secure communication. These necessary components are very complex, and thus developing them from scratch could be expensive and time-consuming. Some operating systems specialized for IoT-enabled resource-constrained devices (hereafter, “IoT OSes”) have been released [17]. IoT OSes generally provide a software stack with many useful components to enable efficient application development. Several of them are further officially supported and maintained by cloud service providers for collaborative ecosystems [18]. Reusing the components from open-source IoT OSes can save the effort in building an IoT-enabled critical system greatly, but the conventional ad hoc reuse approach, as shown in Fig. 2, is very inefficient due to the following issues.

- **Low productivity.** IoT OS and reliable RTOS usually use different kernel and subsystem APIs [19]. Besides, the components in IoT OS are often updated very frequently, compared to the traditional components in a critical system. Porting and maintaining them for a reliable RTOS requires a considerable effort, which can lead to low productivity.

- **Reduced reliability.** Low-level components are typically designed to run in the kernel space to provide system services. Their large size can significantly bloat the TCB. Their complexity also increases the attack surface and creates many new security threats [20]. Adding these components will make a thorough verification of TCB difficult and costly.

Dual-OS configuration is a promising approach to efficiently reuse an IoT OS [21]. By using a hypervisor, a non-trusted OS (e.g. IoT OS and its applications) can execute in an environment isolated from the trusted OS (e.g. reliable RTOS). The critical services and sensitive data in the trusted side are protected against potential threats in the non-trusted side. The components for IoT are still running on the IoT OS kernel, and thus it is unnecessary to port them for the reliable RTOS. The whole IoT OS software stack is in a less privileged domain rather than the TCB. While Dual-OS configuration can overcome the issues in the ad hoc approach, existing solutions depend on additional hardware features such as ARM TrustZone and virtualization extensions [22]. These features are unavailable in most resource-constrained devices and thus we propose a new dual-OS solution.

The main contributions of this paper are summarized as follows:

1) iSotEE, a middleware allowing IoT OS to run inside an isolated environment on top of a reliable RTOS, is proposed. It is specially designed to meet the requirements of resource-constrained reliable systems.

2) Open-source implementations of iSotEE for Renesas RX and ARMv7-M architectures are provided. The RX target uses TOPPERS/HRP3 as reliable RTOS and Amazon FreeRTOS as IoT OS. The ARMv7-M target uses two different configurations of Zephyr as reliable RTOS and IoT OS. To our knowledge, iSotEE is the first dual-OS solution for RX processor family.

3) These implementations have been evaluated with multiple important metrics for IoT-enabled resource-constrained reliable systems, such as TCB size, productivity, footprint, overhead, and throughput.
The rest of this paper is structured as follows. Section II introduces necessary background. Section III explains the design of iSotEE in detail. The implementations for two targets are described in Section IV and evaluated in Section V. Section VI compares iSotEE with latest related work. Finally, Section VII concludes this paper and discusses some future work.

II. BACKGROUND

In this section, we first overview the hardware characteristics of IoT-enabled resource-constrained devices. The features of different operating systems for resource-constrained devices are then explained.

A. IOT-ENABLED RESOURCE-CONSTRAINED DEVICES

Resource-constrained devices in this paper mainly refer to those devices incapable of running GPOS (general purpose operating system) like Linux. As minimum requirements to build IoT-enabled reliable systems, the hardware should at least have some connectivity for IoT applications, and protection feature for critical applications.

Table 1 shows some representative development boards meeting the requirements above for different hardware architectures. Typically, these boards have limited RAM (a few hundred kilobytes) and ROM (several megabytes). While MPU (Memory Protection Unit) has been widely adopted, advanced features like MMU (Memory Management Unit) and hardware-assisted virtualization are unavailable to most processors.

B. OPERATING SYSTEMS FOR RESOURCE-CONSTRAINED DEVICES

As we have mentioned, both reliable RTOS and IoT OS can support resource-constrained devices. Table 2 compares the features of some open-source reliable RTOSes and IoT OSes.

TOPPERS/HRP (High Reliable system Profile) is a series of open-source µITRON-like reliable RTOS kernels widely used in Japan [23]. HRP3 (version 3 of TOPPERS/HRP series) not only provides space partitioning, but also supports industrial-grade critical applications with advanced features (e.g. time partitioning, µs high-resolution time management, external clock synchronization). However, to develop IoT applications, the user must port and maintain out-of-tree modules which can limit the productivity.

Zephyr [24] is a relatively new RTOS targeting connected resource-constrained devices. It can be flexibly configured to be used as a reliable RTOS or an IoT OS. Its IoT-related modules typically run in privileged mode, which can lead to large TCB size. These modules are not covered by Zephyr’s current scope of safety certification. Therefore, to use Zephyr as a reliable RTOS for critical applications, the IoT features should be considered as out-of-tree due to expensive verification cost required. When using Zephyr as an IoT OS for non-critical applications, the partitioning feature can be disabled to reduce overhead.

Amazon FreeRTOS, Azure RTOS and Mongoose OS are three IoT OSes with official cloud collaboration support. All of them do not provide full-blown partitioning feature for critical applications. Unprivileged tasks in Amazon FreeRTOS can access all kernel objects via RTOS API. Only dynamic loaded modules in Azure RTOS have partitioned memory space. Mongoose OS is a flat RTOS (i.e. RTOS uses a flat memory model without any memory protection functionality).

III. THE DESIGN OF ISOTEE

iSotEE (iSolated Execution Environment) is a hypervisor specially designed to support dual-OS configuration on resource-constrained devices. It has many advantages for building IoT-enabled resource-constrained reliable systems as follows:

- **Portable and lightweight.** iSotEE is a portable middleware that can be implemented in most reliable RTOSes.
It does not require additional hardware features and thus supports various processor architectures. It has a lightweight design with a small overhead and footprint to satisfy the resource constraints.

- **High reliability with reduced TCB size.** The whole software stack of IoT OS, including interrupt handlers, runs in unprivileged mode. Therefore, those complex IoT-related middleware and device drivers can be excluded from the TCB to improve reliability and security.

- **Meet hard real-time requirements with reliable RTOS.** iSotEE assumes that reliable RTOS must support hard real-time applications while IoT OS is for soft real-time and best-effort applications. It allows the reliable RTOS to run natively and preempt the IoT OS to minimize the interference. The hypervisor itself is also optimized to reduce performance impact.

- **Increase productivity and throughput with IoT OS.** IoT OS only requires minor modifications to execute on iSotEE. Since there is no need to implement or port IoT software components for reliable RTOS, the development effort can be greatly saved. Further, IoT OS, due to the simplicity of its kernel, can potentially deliver higher throughput than reliable RTOS.

In this section, the architecture of iSotEE is explained at first. Some important concepts are further described later.

### A. ARCHITECTURE OVERVIEW

Fig. 3 shows an overview of a reliable system with iSotEE. The architecture consists of several main components described as follows:

- **Host OS:** It is a reliable RTOS which must at least support space partitioning (e.g. memory protection). The predictability can be further improved if time partitioning is supported.

- **Guest execution environment:** It is an isolated domain created using the partitioning functionality of reliable RTOS. Programs executed in this domain are not trusted and can only access resources explicitly granted by the system designer.

- **Guest software:** It is the software stack for IoT, which runs on a host OS task inside the guest execution environment. It typically includes IoT OS kernel, interrupt handlers, device drivers, middleware and user applications.

- **iSotEE hypervisor middleware:** It provides necessary hypervisor services such as communication and device virtualization. It is implemented as a host OS system component running in privileged mode. More details will be explained later in this section.

- **Paravirtualization API:** It is a library for the guest to request services, such as interrupt management, provided by the hypervisor middleware. An existing IoT OS can be ported to iSotEE as a guest OS by implementing its BSP (board support package) using this API.

This architecture has some noteworthy characteristics:

- The reliable RTOS runs natively on bare metal, just like in a traditional reliable system. For hard real-time applications, the performance penalty can be small and predictable.

- Partitioning is an important yet complex part of resource management in a classic hypervisor design. iSotEE offloads it to the reliable RTOS, in order to make the hypervisor as lightweight as possible.

- Guest OS and its applications run in the same privilege level, which means there is no protection between them. This design can support most use cases since existing IoT OSes are typically based on flat RTOSes without protection. Further, the flat guest OS has the potential to deliver higher throughput than the reliable host RTOS due to smaller overhead.

- All guest software components run on a single host OS task. They are managed by the host as a whole, which can simplify the design and analysis of timing behavior. The host side footprint is also reduced since only a small and fixed amount of resource is required, regardless of the number of tasks in guest OS.

- IoT OS makes use of many open-source software components with various licenses to maximize productivity. Reliable RTOS, on the other side, may include certified/verified proprietary components to improve the reliability of critical applications. iSotEE allows the host and guest software to be separately linked as two independent binaries. This strong isolation can reduce the risk of license compatibility issues.

### B. RESOURCE MANAGEMENT

iSotEE allows the system designer to grant necessary hardware resources (e.g. memory, peripheral devices) to the guest execution environment. Three types of devices are supported to meet the needs in different use cases.

- **Pass-through devices:** The memory areas of these devices are assigned to the guest via the space partitioning functionality in reliable RTOS. The overhead is negligible since guest software can directly access them without the intervention of...
the host. These devices must be assigned with caution since the access requests are not checked by the host. They should be exclusively used by the guest and improper access to them must not harm the host side.

**Protected devices:** Some devices can become attack vectors if they are improperly accessed by the guest. Using DMA (direct memory access) controller is a typical example, which must be checked to ensure that the guest does not access disallowed addresses. Protected devices are always accessed by the guest via the hypercall interface. While the overhead is larger than pass-through devices, improper access can be checked and blocked by the hypervisor.

**Virtual devices:** Devices for the guest must be virtualized if they behave differently than on the bare-metal environment. System tick timer and interrupt controller are two common virtual devices required by most, if not all, guest OSes. Devices shared between the host and guest, such as data storage, may also need virtualization in some cases. Extending an existing host-side device driver is a typical approach to implementing a virtual device. For example, the system designer can use host-aware hypercalls (described in Section III-C) to receive virtual storage service requests from guest OS and delegate them to the file system in host OS. To limit the guest’s accessible files and storage usage, the hypercalls will check the permission for these service requests. In this case, it should be noted that the guest files are also accessible from the host OS, since they are actually stored in the host file system. Therefore, virtual devices implemented in this approach can be used for communicating between guest and host OS via the shared resources.

### C. HYPERCALL INTERFACE

The hypercall interface provides a safe communication mechanism for the guest to request services from the hypervisor. A hypercall is analogous to a system call in the conventional OS, which is usually invoked via a software interrupt (a.k.a. trap instruction). It transfers the control to the privileged hypervisor to execute some trusted function, and then returns to the calling guest program. iSotEE supports two types of hypercalls as follows.

**Bare-metal hypercalls:** These hypercalls are directly executed without doing a context switch to the host OS. They are not allowed to use the host kernel services but the overhead can be reduced. During their execution, all (not only the guest’s) interrupts are disabled to avoid the race condition caused by the host OS. This kind of hypercall must be short and predictable (e.g. a single write to some protected register), since long interrupt disabled time is harmful to hard real-time performance.

**Host-aware hypercalls:** The guest software can also use the host system calls for communication, since it runs on a host OS task. A hypercall via the system call interface is aware of the host system, and thus is able to provide more advanced service using the host OS API. A system call has larger context switch overhead than the bare-metal hypercall, but only disables the interrupts for a few cycles during its execution. For long and complex jobs, this kind of hypercall is more suitable since it has minimal impact on the host real-time performance.

From the perspective of the guest, both kinds of hypercalls can guarantee the atomicity.

A hypercall may also return a shared memory area (e.g. buffers, read-only data structures) to the guest, in order to communicate further with lower overhead.

### D. GUEST OS SCHEDULER SUPPORT

The context switch procedures are essential to support an OS scheduler. Their implementations depend on the target hardware, and thus are usually included in the BSP. To schedule guest OS tasks on the host OS task specified by iSotEE, the native BSP for the guest OS must be modified as follows:

- Context switching will be performed when a task yields the control. It is protected with critical section since it must not be interrupted. Native critical sections are often created by locking CPU with privileged instructions (e.g. disable all interrupts). They must be replaced with implementations provided by the iSotEE paravirtualization API to support the unprivileged guest OS.

- For guest OS with preemptive scheduler, context switching may also be performed inside the interrupt handler. On entering a handler, the context (e.g. stack pointer, registers) of the virtual interrupt controller is different from the hardware controller. The native interrupt handler must be modified accordingly.

It is recommended to assign the lowest priority to the host OS task for guest software, so other host applications can always preempt the guest. The time partitioning feature, if supported by the host OS, can also be used to control the guest CPU usage, for better predictability.

### E. VIRTUAL INTERRUPT CONTROLLER

A virtual interrupt controller is implemented by iSotEE to manage those interrupt sources assigned to the guest (hereafter, “guest interrupts”). It also provides a mechanism to let the guest interrupt handler run on the unprivileged host OS task, just like those guest OS tasks. Some important concepts are explained in this section.

**Interrupt priority:** The guest interrupts should be assigned with lower hardware interrupt priority than other interrupts used by the host OS. By masking the highest priority of guest interrupts during running the host OS applications, the interference with the host can be prevented.

**Interrupt-disabled critical section:** Critical sections can be created by disabling all guest interrupts. This approach typically uses hypercalls to change the interrupt priority mask. Its behavior is very similar to the native critical section since the interrupts are disabled at the hardware level. However, the hypercall overhead is relatively large compared to the native implementation.

**Interrupt-suppressed critical section:** Critical sections can also be created using a shared flag variable. On entering
must be noted that the guest system time cannot be simply

delivers them to the guest like interrupt requests. However, it
by registering a cyclic handler in the host OS. The hypervisor
since the system is started) can also be calculated by counting
periodic interrupt requests (a.k.a. system ticks) to support
preemptive scheduling. The system time (i.e. the elapsed time
for ticking and one free-running timer for system time, to
reduce the overhead. Meanwhile, timers can be exploited for
side-channel attacks on some processor architectures, and
should be assigned to the guest with caution [25].

IV. IMPLEMENTATION
We have implemented iSotEE on two representative IoT-
enabled resource-constrained targets to evaluate our approach,
and made the source code publicly available. Different host OSes, guest OSes and hardware architectures are used to show the high portability of iSotEE.

A. TARGET 1: RENESAS RX ARCHITECTURE
For this target, TOPPERS/HRP3 reliable RTOS is used as
the host OS. It supports both space and time partitioning, and
also provides advanced features for critical applications like
high-resolution (in µs) time management.
Amazon FreeRTOS (version 2019 release) is used as the
guest OS. It is an IoT OS based on the famous FreeRTOS
kernel, including lots of middleware (e.g. network protocols,
security libraries, over-the-air updating) for IoT applications
[18]. It is officially supported by the AWS (Amazon Web
Services) cloud. A collection of demo applications, named
aws_demos, is provided to show the functionalities of
Amazon FreeRTOS. Our implementation has also been tested with these applications.

Renesas Starter Kit+ for RX65N-2MB is used as the target
board. It is one of the Amazon FreeRTOS-qualified hardware
platforms suitable for various IoT applications [26]. It is
based on 120 MHz RX65N SoC with constrained resources
(RAM up to 640 KB, ROM up to 2 MB, no hardware
virtualization support). The Renesas RX processor family has
been adopted in many critical industrial systems, due to its
low power consumption and high performance predictability.
RX65N SoC provides a dedicated DMA controller, called
EtherDMA, for Ethernet communication. Our implementa-
tion has supported EtherDMA as a protected device. To
our knowledge, iSotEE is the first dual-OS solution for RX
processors.

Fig. 5 shows an application scenario example of this target.
iSotEE allows the developer to implement IoT services on
an IoT OS (Amazon FreeRTOS). For IoT applications, the
development cost can be effectively reduced by utilizing the
rich middleware from IoT OS. The critical applications are
developed and natively executed on a reliable RTOS with
guaranteed real-time performance. iSotEE runs the IoT OS
in an unprivileged environment, and thus the critical services,
calculated from virtual ticks, because the guest OS may be
preempted by the host for indeterminate periods of time. This
issue can be handled using a shared variable to store the guest
system time. The hypervisor will update it to the latest value
(e.g. by deriving from the host system time) before delivering
each virtual tick.

On hardware with adequate timer resource, the guest may
also use two dedicated hardware timers, one periodic timer
for ticking and one free-running timer for system time, to
reduce the overhead. Meanwhile, timers can be exploited for
side-channel attacks on some processor architectures, and
should be assigned to the guest with caution [25].

FIGURE 4. Deliver an interrupt request to the guest.

F. VIRTUAL SYSTEM TICK TIMER
In the native environment, an OS kernel typically uses a hard-
ware timer for time management. The timer is set to trigger
periodic interrupt requests (a.k.a. system ticks) to support
preemptive scheduling. The system time (i.e. the elapsed time
since the system is started) can also be calculated by counting
the occurrence of system ticks.

For a guest OS, virtual system ticks can be easily generated
by registering a cyclic handler in the host OS. The hypervisor
delivers them to the guest like interrupt requests. However, it
must be noted that the guest system time cannot be simply
which may interact with the physical world, can be protected against the potential security risk of the IoT service.

B. TARGET 2: ARMv7-M ARCHITECTURE

For this target, both the host OS and the guest OS run Zephyr (v1.14 LTS). However, the host OS uses a reliable RTOS configuration and the guest OS uses an IoT OS configuration.

The reliable RTOS configuration enables space partitioning feature (i.e. CONFIG_USERSPACE), but disables all IoT-related features to minimize the TCB size and reduce the attack surface, as discussed in Section II-B. The IoT OS configuration, on the other hand, is configured as a flat RTOS with IoT features. Since the guest OS is more lightweight than the host and runs in unprivileged mode, this combination of dual configurations can improve the performance and reliability.

ST NUCLEO-F413ZH is used as the target board. It is based on 100 MHz STM32F4 SoC (Cortex-M4F core, 320 KB RAM, 1.5 MB ROM). Its ARMv7-M architecture is very popular in both consumer and industrial resource-constrained devices. STM32F4 SoC provides USB connectivity, and our implementation allows the guest to work as a USB network gadget and a simple HTTP server.

Fig. 6 shows an application scenario example of this target. Zephyr provides system components for both critical and IoT applications. Because these system components run in privileged mode, they must be verified to ensure the reliability of a critical system. However, the IoT-related components are complex and thus the verification cost is very expensive. iSotEE allows the developer to use a non-trusted guest Zephyr OS for IoT services. Since these system components in the guest OS are isolated from the trusted critical components, it is unnecessary to verify them. This approach can effectively improve the reliability and reduce the verification cost of an IoT-enabled critical system.

V. EVALUATION

In this section, we discuss the characteristics of iSotEE for creating IoT-Enabled resource-constrained reliable systems, by evaluating the implementations. For target 1 (RX architecture), the default CCRX compiler optimization settings are used for both host and guest software. For target 2 (ARMv7-M architecture), the default GCC ARM compiler optimization settings provided by Zephyr’s build system are used.

A. SOFTWARE METRICS

We have measured the software metrics with a representative system configuration for each target. The guest OS runs IoT demo applications while the host OS runs system logging service. For target 1, the guest uses middleware for ethernet, TCP/IP, DHCP client and secure sockets to build a simple HTTP server over USB. Table 3 shows the SLOC (source lines of code) of related components, with comments and blank lines excluded. Table 4 shows the memory footprint of host and guest binaries.

Observation 1: iSotEE has a very small TCB size. The lightweight architecture, as described in Section III-A, can exclude the whole IoT software stack from the TCB by only adding less than 230 lines of code to the host software in both targets. It significantly reduces the system verification effort, and thus improves reliability and security.

Observation 2: iSotEE increases the productivity for IoT applications. A huge amount of IoT middleware components can run as guest software without any modification. In both targets, the host OS only requires a tiny patch to support the hypervisor middleware. Besides, Zephyr and FreeRTOS can be ported as guest OS by implementing a small BSP.

Observation 3: iSotEE is generic and lightweight enough for resource-constrained devices. iSotEE does not require more hardware features compared to a reliable RTOS. Its components (i.e. hypervisor middleware, guest execution environment, guest BSP linked with paravirtualization API library) have a small memory footprint in total. Further, in target 2, RAM usage of the whole system is less than 64 KB. It indicates that most resource-constrained devices capable of running a reliable RTOS can be supported.
TABLE 3. SLOC of iSotEE & Guest in Representative Systems

| Category                        | Target 1 Component Lines of Code | Target 2 Component Lines of Code |
|---------------------------------|----------------------------------|----------------------------------|
| Host Software (iSotEE-Related)  | Host OS Patch (TOPPERS/HRP3) 22 | Host OS Patch (Zephyr) 34        |
|                                 | Hypervisor Middleware 204        | Hypervisor Middleware 178        |
| Guest Software (iSotEE-Related) | Paravirtualization API Library 164 | Paravirtualization API Library 137 |
|                                 | Guest BSP (for FreeRTOS) 266     | Guest BSP (for Zephyr as IoT OS) 213 |
| Guest Software (Unmodified)     | FreeRTOS Kernel 7,230            | Zephyr Kernel 6,665              |
|                                 | Middleware (Crypto, Network, ...) 138,769 | Middleware (USB, Network, ...) 39,650 |
|                                 | Demo Applications (aws_demos) 6,167 | Demo Application (http_server) 119 |

TABLE 4. Memory Footprint of Representative Systems

| Binary Component                     | Target 1 RAM | ROM | Target 2 RAM | ROM |
|--------------------------------------|---------------|-----|--------------|-----|
| Host Hypervisor Middleware           | 36            | 671 | 45           | 753 |
| Guest Execution Environment          | 1,508         | 692 | 1,060        | 160 |
| Other (Kernel, Apps, ...)            | 11,610        | 49,981 | 23,471       | 68,451 |
| Guest BSP + Paravirtualization API Library | 32           | 612 | 32           | 612 |
| Other (Kernel, Apps, ...)            | 180,980       | 369,986 | 41,032       | 97,048 |

1 In bytes.
2 User-configured stack size (384 bytes in the example) is included.
3 User-configured stack size (512 bytes in the example) is included.

TABLE 5. Execution Time of the Paravirtualization API

| Operation                                      | Target 1 Hypercall CPU Cycles | Target 2 Hypercall CPU Cycles |
|------------------------------------------------|-------------------------------|-------------------------------|
| Enter interrupt-disabled critical section      | Bare-metal 27                | Bare-metal 62                 |
| Exit interrupt-disabled critical section       | Bare-metal 22                | Bare-metal 58                 |
| Enter interrupt-suppressed critical section    | Unused 5                     | Unused 10                     |
| Exit interrupt-suppressed critical section     | Unused 21                    | Unused 10                     |
| Mask a guest interrupt                         | Bare-metal 34                | Bare-metal 84                 |
| Unmask a guest interrupt                       | Bare-metal 32                | Bare-metal 83                 |
| Pop a guest interrupt from pending queue       | Bare-metal 58                | Unused 15                     |
| Read the virtual system timer value            | Unused 5                     | Unused 10                     |
| Use EtherDMA to send a frame<sup>a</sup>       | Host-aware 250               | Host-aware 487                |
| Use EtherDMA to receive a frame<sup>a</sup>    | Host-aware 487               | Host-aware 257                |
| Configure clock for USB controller<sup>b</sup> | Bare-metal 17                | Bare-metal 52                 |
| Skeleton bare-metal hypercall<sup>c</sup>      | Host-aware 145               | Host-aware 145                |

<sup>a</sup> Only target 1 implements EtherDMA device.
<sup>b</sup> Only target 2 implements USB controller.
<sup>c</sup> Baselines measured for reference purpose.

TABLE 6. Host OS Overhead Introduced by iSotEE

| Reason / Purpose                  | Target 1 Operation | Overhead<sup>a</sup> | Target 2 Operation | Overhead<sup>b</sup> |
|-----------------------------------|--------------------|-----------------------|--------------------|-----------------------|
| Track guest trap frame for interrupt delivery | An interrupt preempts a task | +8                   | An interrupt preempts a task | +8                   |
|                                   | Context switch between tasks | +6                   | A task resumes from an interrupt | +8                   |
|                                   | A task yields the control | +7                   |                                 |                       |
| Mask guest interrupts when running host apps | A task starts execution | +1                   | Context switch between tasks | +13                  |
|                                   | A task resumes execution | +1                   | A task yields the control | +14                  |

<sup>a</sup> All tasks here refer only to host OS tasks.
<sup>b</sup> Measured in cycles.

B. PERFORMANCE OVERHEAD

The paravirtualization API is used by the unprivileged guest to request hypervisor services. Its execution time, as shown in Table 5, is important to the performance of guest OS.

Observation 4: The paravirtualization API can support soft real-time IoT applications with reasonable performance. Most operations only cost less than 0.5 $\mu$s for target 1 (120 MHz CPU), and less than 0.9 $\mu$s for target 2 (100 MHz CPU). Accessing the EtherDMA protected device in target 1 is relatively complex and costs about 4 $\mu$s at most. In many constrained industrial IoT scenarios, the network latency requirements are in the order of 10 milliseconds [27]. Therefore, we believe the performance is quite acceptable for soft real-time applications.

VOLUME 4, 2016
Observation 5: It is important to use different types of hypcalls properly. The invocation overhead of a host-aware hypcall is multiple times (145 vs 17 in target 1, 145 vs 52 in target 2) of a bare-metal hypcall. Implementing hypervisor services with bare-metal hypcalls as possible can improve performance. Meanwhile, for long and complex operations (e.g. EtherDMA), host-aware hypcalls should be used to reduce the impact on the host real-time performance.

Table 6 shows the performance impact of iSotEE on the host OS, which is essential to those critical applications.

Observation 6: iSotEE can satisfy hard real-time requirements of the host software. The guest interrupts are masked during the execution of host applications. Several operations in the host OS are patched to support the hypervisor but only small and predictable overheads are introduced. Masking interrupts is faster in target 1, because RX architecture has dedicated instruction to set priority level while ARMv7-M requires register access.

C. THROUGHPUT BENCHMARK

Thread-Metric is an open-source benchmark suite for measuring RTOS [28]. Each benchmark application repeatedly executes some test program for a period of time, and then outputs a counter value (i.e. the number of iterations). We use Thread-Metric to evaluate the throughput of a system with iSotEE. The results are shown in Fig. 7.

For target 1, the host OS is measured with three configurations: (1) HRP3 without iSotEE as baseline (2) HRP3 with iSotEE and a dummy (busy loop) guest (3) HRP3 with iSotEE and FreeRTOS as guest OS. The guest OS is measured with three configurations: (1) FreeRTOS in the native environment (2) FreeRTOS as a guest using interrupt-suppressed critical sections (3) FreeRTOS as a guest using interrupt-suppressed critical sections.

The target 2 is measured with similar configurations. Zephyr in the host side is configured as reliable RTOS kernel with space partitioning. Zephyr in the guest side is configured as flat RTOS kernel.

Observation 7: iSotEE degrades the host throughput slightly. The overheads in Table 6 and virtual ticks in Section III-F will consume some CPU time. The maximum degradation is about 4% for target 1 (observed in IPP benchmark), and about 6% for target 2 (observed in BP benchmark). Different guests (dummy vs IoT OS) can present similar results, which indicates the degradation is mainly determined by iSotEE hypervisor rather than the executed guest software.

Observation 8: Host OS may support critical applications with higher throughput. Critical applications, which tend to process interrupts frequently, run on the host OS by the iSotEE design. HRP3, like most reliable RTOSes, is optimized for better interrupt handling performance. IP and IPP benchmarks also confirm that HRP3 delivers high throughput compared to FreeRTOS. This cannot be observed in target 2 since the host and guest use same Zephyr OS with different settings.

Observation 9: Guest OS supports IoT applications with higher throughput. IoT OSes typically include many tasks to support various complex components such as network stacks. The performance of scheduling and messaging between tasks is important for IoT applications. Since FreeRTOS is a flat RTOS without protection between the kernel and applications, it has lower overhead than HRP3 reliable RTOS in many task operations. The idea is similar to unikernel [29] in cloud computing, which improves performance by using hypervisor to replace the OS protection functionality. CS, PS, and MP benchmarks confirm that FreeRTOS guest delivers high throughput compared to HRP3. The differences are even more significant in target 2. It indicates that, although features of both reliable RTOS and IoT OS are provided by Zephyr, using them separately with iSotEE has remarkable benefits for reliability and performance.

Observation 10: Using interrupt-suppressed critical sections can effectively improve throughput. The overhead of interrupt-disabled critical sections, compared to interrupt-suppressed ones, is about twice in target 1 and six times in target 2. As a result, IntSup configuration shows significantly better throughput than IntDis in many benchmarks, for both FreeRTOS and Zephyr.

Observation 11: Native FreeRTOS is outperformed by the guest implementation in some cases. This phenomenon can be observed in CS, PS, MP, SP, and MA benchmarks. We have found that it is caused by inefficient context switch implementation in the native BSP. In FreeRTOS (Native), the cooperative scheduler (i.e. task yield) shares the same context switch procedure with the preemptive scheduler. It can simplify the implementation but also includes unnecessary operations. As described in Section III-D, the guest

VOLUME 4, 2016
BSP should provide different context switch procedure for each case, which leads to better performance. The native BSP of Zephyr for ARMv7-M also shares the context switch procedure. However, the ARMv7-M architecture provides a hardware optimization for this purpose called PendSV. Therefore, the guest only shows slightly better performance in a few benchmarks like PS and SP.

VI. RELATED WORK

Dual-OS solutions can offer many benefits, since neither reliable RTOS nor IoT OS can satisfy all the requirements. Although hypervisor is a well-established technology for high-end embedded systems (e.g., those capable of running Linux), to our knowledge, iSotEE is the first open-source solution supporting low-end resource-constrained devices [30]. Due to this situation, we cannot quantitatively compare iSotEE with existing solutions under similar experimental conditions. Instead, to show the novelty of our solution, this section qualitatively compares iSotEE with the state-of-the-art related works on IoT-enabled embedded real-time systems (not limited to the resource-constrained environment).

ACRN [31] and Bao [32] are two latest open-source hypervisors for high-end embedded systems. ACRN is a Linux Foundation project under active development, and provides many advanced features for safety-critical embedded systems [33]. It relies heavily on Intel Virtualization Technology (Intel VT), and thus cannot be used on non-Intel platforms. Meanwhile, Bao has a portable design and supports two architectures (ARMv8-A and RISC-V). Both of them implement resource isolation via hardware-based approach, and thus their targets are limited to high-end processors with large memory size and virtualization support. On the other hand, iSotEE uses a software-based paravirtualization approach only depending on minimal hardware features (e.g., several kilobytes memory and MPU), which can support most processors for resource-constrained devices. Generally, the performance overhead and operating system modifications are two potential drawbacks of software-based paravirtualization. Our evaluation results have shown that iSotEE has small and predictable real-time performance overhead, and the modifications to host and guest OS are less than 300 lines of code.

MultiZone [34] and ILTZVisor [21] are two latest hypervisors for resource-constrained devices. MultiZone uses a software-based approach similar to iSotEE [20] and supports low-end RISC-V and ARMv7-M processors with MPU. Although MultiZone is able to run more than two OSes, all OSes are virtualized in unprivileged mode. Meanwhile, iSotEE allows the reliable RTOS to execute natively in privileged mode with hard real-time guarantee. ILTZVisor is a dual OS solution specially designed for ARMv8-M processors with TrustZone feature. It can also be extended to support multicore platforms [35]. As a hardware-based approach, it runs unmodified reliable RTOS and IoT OS with native performance, but the lack of portability limits its usage for resource-constrained devices compared to our solution. Since a trusted component like hypervisor must be verified in a critical system, the source code availability is important [36]. However, unlike iSotEE, both MultiZone and ILTZVisor are not open-source.

Apart from hypervisor, TEE (trusted execution environment) is another approach to enhance the reliability of IoT-enabled systems [37]. While a hypervisor mainly focuses on protecting the critical real-time tasks from the interferences of non-trusted applications, TEE ensures the confidentiality and integrity of data against possible attacks [38]. Typically, a TEE is not able to run a full-featured OS like a hypervisor, but can provide stronger isolation with its minimal attack surface. For resource-constrained devices using ARM processors, most TEEs are based on the hardware TrustZone feature [39]. Since hardware-based hypervisors on those devices also use TrustZone [35], they cannot coexist with TEE due to resource contention. iSotEE, as a software-based approach, is able to cooperate with TEE to further improve the reliability.

VII. CONCLUSION AND FUTURE WORK

We have introduced the iSotEE hypervisor middleware to provide a novel dual-OS solution for embedded systems. An IoT-enabled system based on iSotEE achieves high reliability and productivity, by using a reliable RTOS as the trusted host OS for critical applications and an IoT OS as the non-trusted guest OS for IoT applications. To support most resource-constrained devices, the generic design of iSotEE does not depend on special hardware features. Open-source implementations for Renesas RX and ARMv7-M architectures are provided and evaluated. The results show that a reliable system with iSotEE can meet resource constraints with a small footprint, guarantee real-time performance for critical applications, and deliver high throughput for IoT applications.

As future work, we plan to port iSotEE for more targets (e.g., nbed OS, RISC-V architecture) and extend the approach to multi-core devices. This paper mainly focused on reliability, but we believe iSotEE can be further improved to provide TEE (Trusted Execution Environment) for security.

REFERENCES

[1] M. Gaiceanu, “Cyber-physical systems for industrial applications,” in 2019 6th International Symposium on Electrical and Electronics Engineering (ISEEE). IEEE, 2019, pp. 1–3.
[2] N. Dey, A. S. Ashour, F. Shi, S. J. Fong, and J. M. R. Tavares, “Medical cyber-physical systems: A survey,” Journal of medical systems, vol. 42, no. 4, p. 74, 2018.
[3] C. Alcaraz and S. Zeadally, “Critical infrastructure protection: Requirements and challenges for the 21st century,” International journal of critical infrastructure protection, vol. 8, pp. 53–66, 2015.
[4] V. Narayanan and Y. Xie, “Reliability concerns in embedded system designs,” Computer, vol. 39, no. 1, pp. 118–120, 2006.
[5] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, “Operating systems for Internet of Things low-end devices: Analysis and benchmarking,” IEEE Internet of Things Journal, vol. 6, no. 10, pp. 10 375–10 383, 2019.
[6] G. C. Buttazzo, Hard real-time computing systems: predictable scheduling algorithms and applications. Springer Science & Business Media, 2011, vol. 24.
[7] L. Wehmeyer and P. Marwedel, “Influence of memory hierarchies on predictability for time constrained embedded software,” in Design, Automation and Test in Europe. IEEE, 2005, pp. 600–605.
[8] S. S. Prabhu, H. Kapil, and S. H. Lakshmaiah, “Safety critical embedded software: Significance and approach to reliability,” in 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), IEEE, sep 2018.

[9] L. S. Azevedo, D. Parker, Y. Papadopoulos, M. Walker, I. Sorokos, and R. E. Araújo, “Exploring the impact of different cost heuristics in the allocation of safety integrity levels,” in Model-Based Safety and Assurance. Springer International Publishing, 2014, pp. 70–81.

[10] A. Burns and R. I. Davis, “A survey of research into mixed criticality systems,” ACM Computing Surveys (CSUR), vol. 50, no. 6, pp. 1–37, 2017.

[11] T. Li, T. Ishikawa, Y. Matsubara, and H. Takada, “A platform for LEGO MINDSTORMS EV3 based on an RTOS with MMU support,” in Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7–11, 2015. ACM, 2015, pp. 54:1–54:6.

[12] K. Parveen, A. Ali, and G. Asadullah, “Survey on operating systems for the applications of the Internet of Things,” Journal of Information Communication Technologies and Robotic Applications, pp. 9–16, 2018.

[13] P. P. Ray, D. Dash, and D. De, “Edge computing for Internet of Things: A survey, e-healthcare case study and future direction,” J. Network and Computer Applications, vol. 140, pp. 1–22, 2019.

[14] O. Hahn, E. Baccelli, H. Petersen, and N. Tsiftes, “Operating systems for low-end devices in the Internet of Things: A survey,” IEEE Internet of Things Journal, vol. 3, no. 5, pp. 720–734, 2016.

[15] S. Pinto and C. Garlati, “User mode interrupts: A must for securing embedded systems,” in Proceedings of the Embedded World Conference, 2019.

[16] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, “Virtualization on TrustZone-enabled microcontrollers? voilà!” in 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, apr 2019.

[17] S. Pinto and N. Santos, “Demystifying arm TrustZone,” ACM Computing Surveys, vol. 51, no. 6, pp. 1:1–36, jan 2019.

[18] Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada, “A platform for LEGO MINDSTORMS EV3 based on an RTOS with MMU support,” in OSPERT 2014, 2014, pp. 51–59.

[19] F. Jaskani, S. Manzoor, M. Amin, M. Asif, and M. Irfan, “An investigation on several operating systems for Internet of Things,” EAI Endorsed Transactions on Creative Technologies, vol. 6, no. 18, p. 160386, jan 2019.

[20] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” ACM SIGARCH Computer Architecture News, vol. 41, no. 1, pp. 461–472, 2013.

[21] M. Cinque, D. Cotonore, L. D. Simone, and S. Rosiello, “Virtualizing mixed-criticality systems: A survey on industrial trends and issues,” Future Generation Computer Systems, dec 2021.