Received: 30 July 2020 | Revised: 1 January 2021 | Accepted: 25 February 2021

DOI: 10.1002/spe.2968

SPECIAL ISSUE PAPER

WILEY

A lightweight virtualization model to enable edge computing in deeply embedded systems

Ramão T. Tiburski[1,2] | Carlos R. Moratelli[3] | Sérgio F. Johann[1] | Everton de Matos[4] | Fabiano Hessel[1]

[1]Polytechnic School, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

[2]Department of Education, Research, and Extension, Federal Institute of Education Science and Technology of Santa Catarina (IFSC), Santa Catarina, Brazil

[3]Department of Control, Automation and Computing, Federal University of Santa Catarina, Santa Catarina, Brazil

[4]Secure Systems Research Centre, Technology Innovation Institute, Abu Dhabi, United Arab Emirates

Correspondence
Ramão T. Tiburski, Polytechnic School, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Rio Grande do Sul 90619-900, Brazil.
Email: ramao.tiburski@ifsc.edu.br

Funding information
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: Finance Code 001.

Abstract

Edge computing paradigm enables moving Internet of Things (IoT) applications from the Cloud to the edge of the network. Modern software engineering approaches are adhering to microservices to enable the deployment of such applications on edge devices. Microservices consist of the disaggregation of an application into smaller pieces that operate independently. Recent works have explored microservices packaged into containers and advocate that containers result in a reduced footprint and avoid the unwanted overhead caused by traditional virtualization. However, containers cannot be used in many deeply embedded systems (DES) due to an underlying operating system's (OSs) requirement. DES are edge devices with minimal resources regarding storage, memory, and processing power. Thus, they cannot afford large and sophisticated OSs. This article presents the Hellfire hypervisor, a lightweight virtualization implementation that enables separation and improves security in IoT applications on DES. Our proposal simplifies the traditional hypervisor approach and reaches devices where the existing techniques fail. The results show that the proposed model has a small footprint of 23 KB while keeping a low average virtualization overhead of 0.62% for multiple virtual machines execution.

KEYWORDS

deeply embedded systems, edge computing, hypervisor, Internet of Things, virtualization

1 | INTRODUCTION

Connected devices around the world have led to the rise of the Internet of Things (IoT). In this paradigm, applications rely on multiple devices, gathering, and sharing data across highly heterogeneous networks.[1,2] Recently, IoT applications started to move toward new strict requirements regarding timeliness and low latency combined with ultra-high availability and reliability.[3] In this regard, the edge computing paradigm has been presented as a promising solution.[1] It refers to the enabling technologies allowing computation to be performed at the edge of the network.[4] In addition, it describes the layer of edge devices used to do some local computing or sensor metering.[5]

The combination of the IoT devices' hardware and software in the edge computing environment is called edge devices.[6] However, edge devices can have different limitations in resources (storage, memory, and CPU). In this work, we use the term "Deeply Embedded System" or simply "DES" to refer to devices starting with a few hundred kilobytes up to some megabytes of storage and memory, typical in IoT applications. These devices often have plenty of connectivity options, graphic accelerators, and other hardware features that enable complex applications. Nevertheless, its memory capacity makes challenging the adoption of rich operation systems. For example, the PIC32mz EF family from Microchip has up to 2 MB for flash and 512 KB RAM with several connectivity options (USB, CAN, Ethernet, and Wifi). Its system-on-a-chip (SoC) has a core MIPS M5150 at 200 Mhz with memory management capacity and a cryptography engine. The PicoCore RT1-V1 has an ARM Cortex-M7 core with 256 MB of flash and 32 MB of RAM and connectivity options such as USB, Ethernet, CAN, and UART. The extensive software layer supported by these boards may require some separation to improve security and modularity, for example, to isolate peripheral control from network connectivity.

Edge computing models aim to optimize IoT applications to move their functionalities to edge devices, rather than outsourcing computations to distant data centers.[7] In new edge computing architectures, modern software engineering approaches are adhering to the microservices paradigm.[3] It allows the development of new distributed computing software systems to achieve high QoS, flexibility, dependability, and other properties due to their autonomic self-behavior, such as self-monitoring, self-adaptation and self-reconfiguration, among others.[8]

Recent works have explored microservices packaged into containers to achieve a high degree of automation, deployment, elasticity, and reconfiguration of IoT applications.[3,8-10] To this end, various container management and orchestration technologies have emerged, including Docker[11] and Kubernetes.[12] Containers are a kind of virtualization at the operating system (OS) level, which package all binaries, libraries, and configuration files in a single box. They are lightweight than server virtualization since it does not contain the complete OS image. Nevertheless, such technologies are limited to hardware capable of executing general-purpose operating systems (GPOS), such as Linux or Windows, that require hundreds of megabytes of storage and memory available on devices, like the Raspberry Pi boards.[3,7-10,13-15] DES devices, as the mentioned PIC32mz and PicoCore, cannot execute GPOS due to memory or processor limitations, but their applications still may require the flexibility provided by containers.

The main contributions of this work are described as follows. First, we present virtualization features considering DES and propose a new lightweight virtualization model that simplifies the traditional approach, enabling edge computing, and the deployment of IoT applications on DES. Thus, delivering the inherent virtualization advantages to many IoT applications cannot afford large and complex OSs. Second, we present the Hellfire hypervisor, an implementation of the proposed model, to validate and demonstrate that it can keep a smaller footprint than existing techniques while keeping a low virtualization overhead. Our experiments show that it can fit in 21 KB of storage and use 2 KB of RAM, representing an overhead of 1.03% and 0.4% on the PIC32mz device.

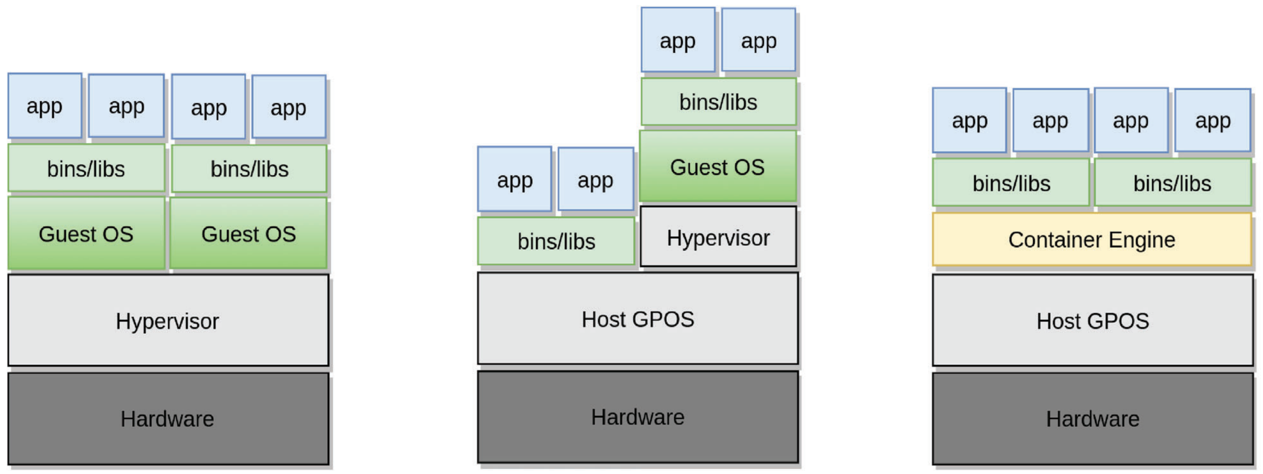
The remainder of this article is organized as follows: Section 2 discusses the use of virtualization on DES. Section 3 presents a virtualization model to fit the DES needs. Section 4 details the implementation of the Hellfire Hypervisor, a lightweight virtualization layer for embedded systems. Section 5 presents experimental results of implementation's footprint, performance, inter-VM communication, real-time, and security. Section 6 presents the related work. Section 7 presents a test case scenario for the deployment of the proposed model. Finally, Section 8 concludes the article.

2 | VIRTUALIZATION ON DES

Edge computing is about processing data streams at least partially on the spot (e.g., directly on DES) in a resource-saving way. It runs specific applications in a fixed logic location and is strongly based on virtualization.[3,7-10,13-19] For a long time, the research community believed that hypervisor-based virtualization was an overkill approach for DES due to its inherent overhead.[7,8,13,15] However, the advances in embedded processors that enabled hardware-assisted virtualization and innovative hypervisor software architectures changed this scene. Recent research has shown the benefit of embedded virtualization to meet DES challenges in edge environments.[3,16,18] However, there is no consensus about which virtualization approach better addresses the needs of DES.

Virtualization means creating a software abstraction layer that gives the applications a different view of the underlying hardware.[14] The following characteristics for virtualized systems are essential in the context of DES:

- **Virtualization approach:** It can be a type-1 hypervisor, a type-2 one, or a containers engine. Type-1 hypervisors (Figure 1(A)) are directly executed on the hardware and are the most adopted approach in server virtualization.[20] Differently, type-2 hypervisors (Figure 1(B)) execute on top of an OS. They are the preferred choice for home and office virtualization since they can run guest OSs side-by-side with user's applications. Containerization (Figure 1(C)) is an OS-level virtualization method to execute multiple isolated systems (containers) using a single kernel. A user process can check different information about the system, such as memory, process trees, files, and directories in a typical OS.



(A) Type-1 Hypervisor. The hypervisor as the first software layers. (B) Type-2 Hypervisor. The hypervisor running over an GPOS. (C) Containerization. Dismiss the use of hypervisors.

FIGURE 1 Comparison between the different virtualization approaches

OS's containers enforce the process isolation limiting and prioritizing the resources (e.g., CPU, memory, I/O, network, among others) without using virtual machines. A container engine executes on top of a host OS, responsible for the separation between applications using the OS's features.

- **Underlying GPOS:** It refers to the need for an underlying GPOS (e.g., Linux and Windows), which are complex systems designed to perform well on laptops up to supercomputers and support a wide range of applications, such as accelerated graphics, artificial intelligence, and graphical user interfaces. Such features make GPOS generally large in footprint and also unpredictable for real-time purposes. Although some GPOS (e.g., Linux) are highly customizable, their minimal footprint is still unacceptable for DES.
- **Separation:** It allows the execution of multiple applications or even instances of OSs in separate boxes over the same device. Two other features emerge from the separation: (i) the capacity to break complex applications into smaller software pieces allowing better maintainability and modularity; and (ii) preventing defects in one part of the software to propagate to adjacent regions or the physical platform, improving security as a whole.
- **Real-time support:** Proper real-time support is a common feature required by embedded systems. Typically, GPOS focus on performance and present poor real-time results. Otherwise, real-time operating systems (RTOS) are capable of dealing with timing constraints, but they do not support software separation. Thereby, separation and real-time capabilities are required to coexist in the same system.
- **Memory requirements:** Because of many installed devices, each one's cost keeps an essential role in the IoT field. Thereby, reducing the hardware capacity helps to decrease costs. We consider as memory capacity the device's storage and SRAM size. The occupied portion of this capacity by the software is called the footprint.

The interest in virtualization at the edge has increased.[7] However, as applied to server farms, classic virtualization does not fit most IoT applications at the edge. The intrinsic characteristics (memory constraints, real-time, and security) of such applications have motivated new models for DES virtualization.

3 | VIRTUALIZATION MODEL FOR DES

This section describes a model for DES virtualization that dismisses the use of an underlying GPOS. Although this is true for type-1 hypervisors for cloud computing, their performance-focused development results in some undesired characteristics for embedded systems (e.g., high footprint and power consumption). Despite cloud and embedded applications share some features, others are distinctive. For example, both expect that the virtualization layer provides a strong separation between guests for security purposes. However, cloud computing expects higher performance, while some embedded applications have power consumption and storage constraints.

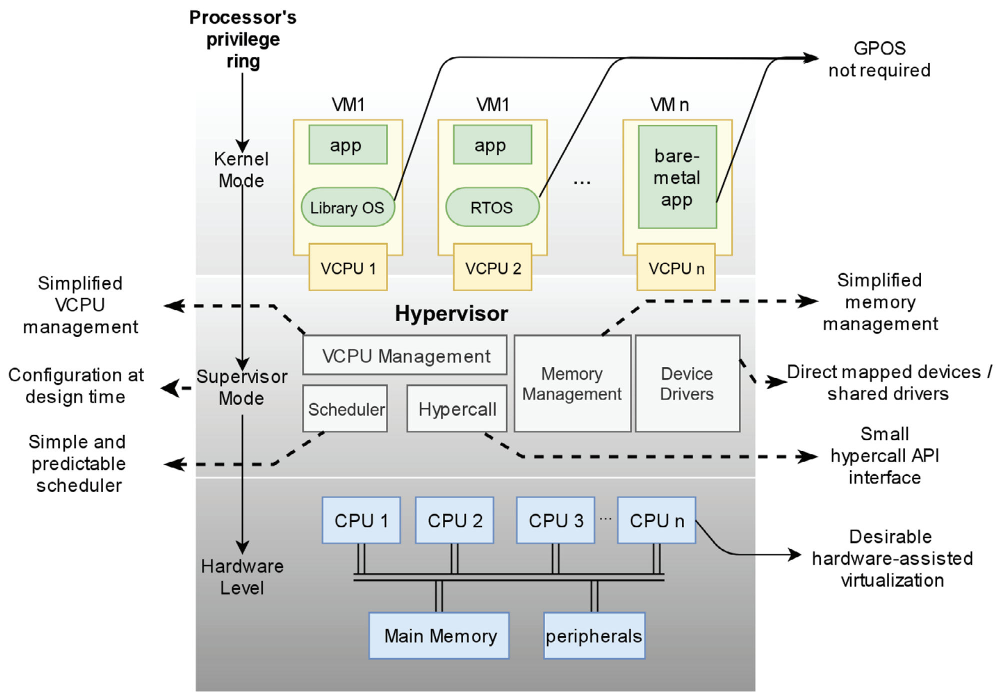

Processor's privilege ring
Kernel Mode
Supervisor Mode
Hardware Level
VM1
app
Library OS
VCPU 1
VM1
app
RTOS
VCPU 2
...
VM n
bare-metal app
VCPU n
GPOS not required
Hypervisor
VCPU Management
Scheduler
Hypercall
Memory Management
Device Drivers
Simplified VCPU management
Configuration at design time
Simple and predictable scheduler
Simplified memory management
Direct mapped devices / shared drivers
Small hypercall API interface
CPU 1
CPU 2
CPU 3
CPU n
Main Memory
peripherals
Desirable hardware-assisted virtualization

FIGURE 2 The proposed virtualization model overview. The doted sets indicate the key model's characteristics that make possible the building of a lightweight hypervisor

Before presenting the virtualization model, it is essential to remember some of the DES's demands. Generally, such systems execute dedicated applications. If a secondary functionality needs to be added, it must be integrated into the *main loop*, affecting the whole system and making hard the management of many different features. The increasing demand for more complex software stacks makes modularity an essential feature for these systems. As said before, the containers approach can be used to bring modularity. However, the implicit overhead cost to keep a GPOS limits the reach for DES. Otherwise, RTOSs may deliver a certain level of modularity but cannot deal with separation.

The proposed model observes some characteristics of DES to build a lightweight hypervisor capable of delivering modularity, security, and computing, where existing techniques fail. Figure 2 depicts the virtualization model. The hardware platform is a typical SoC composed of memories, processors, and peripherals. Here, processors with hardware-assisted virtualization are preferred. Although it is possible to implement a similar approach without such features, dedicated hardware for virtualization can simplify the hypervisor's design, improving performance and security. Besides, common embedded architectures already have defined their support for virtualization, as ARM[21] and MIPS.[22] These processors implement the *privilege ring* that allows for separate actions that can be performed by the hypervisor (supervisor mode) and guests (kernel mode). In addition, to deliver separation, the processor must support a memory management unit (MMU). This hardware module allows the hypervisor to implement memory isolation, a key feature to build modularity and security by separation.[16]

In the proposed model, the hypervisor is the first layer of software (type-1 virtualization), and it performs in the highest processor's privilege level (supervisor mode). Hence, it can control all hardware behavior avoiding that guests change the processor or platform configuration. The hypervisor creates the VM's abstraction implementing two basic functionalities. First, it constructs the memory isolation (using the MMU hardware). Second, it creates the virtual CPU abstraction (VCPU) (i.e., a data structure that keeps the CPU context during context switches). The next software layer is the guest system. They can be RTOSs, Unikernels, or even bare-metal applications that perform with a lower privilege level (kernel mode). Any unexpected behavior (e.g., access to a not allowed memory location) will trap the hypervisor that will take the required actions. Thus, any software in a VM views the system as an entirely independent machine allowing the implementation of wholly separated applications and bringing modularity to the system.

While hypervisors for cloud computing require up to tens of megabytes of footprint, the proposed model needs tens of kilobytes, keeping virtualization advantages such as modularity and security. Although other hardware constraints could be considered, they are less critical. For example, DES communicates just with surrounding devices. Hence, to most of the applications, network throughput is not a significant concern. Nevertheless, some specific applications may require attention over other aspects, as hardware engines for speed up cryptography operations. In this way, we limited our focus to the processor capabilities and memory capacity to build up isolated applications in small devices. To this end, we introduced some simplifications to improve virtualization for DES:

- *Simplified memory management*: The model avoids swapping[23] and simplifies the page tables (PTs)[23] implementation at the hypervisor level. Generally, the number of VMs is known, allowing the system's partitioning in design time. Thereby, the VMs are allocated contiguously in memory, making the management more straightforward because only the base address and size are required for its mapping. This scheme saves memory because no PTs are needed. After all, the memory mapping can be directly written to the MMU control registers (see Section 4.2). Note that this simplification does not affect the memory management implementation at the guest's level because how the hypervisor manages the memory is transparent to the guest. See Section 4.2;
- *Static partitioning*: A hypervisor implementation targeting DES should not support the management interface since it can determine its setup in compilation time. See Section 4.5;
- *Directly mapped devices*: Direct access from applications to peripherals may be needed. If the peripheral does not need to be shared, the hypervisor must allow direct mapping (bypassing) to the applications. This technique avoids the implementation of device drivers at the hypervisor level and improves performance. Otherwise, device drivers for shared peripherals, such as Ethernet, must be implemented and managed by the hypervisor. See Section 4.4;
- *Small hypercall API interface*: Hypercalls are calls invoked from guests to the hypervisor, similar to *syscalls* in a typical OS. The hypercall API allows the implementation of extended services, that is, services provided by the hypervisor to its guests, like inter-VM communication or access to shared devices. It is essential to keep this API simple for two reasons: to reduce the hypervisor's attack surface and to keep the hypervisor as small as possible. See Sections 4.3 and 4.4;
- *Simple and predictable scheduler*: The scheduler must implement proven algorithms to maintain predictability, like the round-robin scheduler. In addition, the interrupt management can be simplified using the *pass-through* technique supported by hardware-assisted virtualization (i.e., interrupts can be redirected to guests without hypervisor intervention). See Sections 4.3 and 4.6;
- *Simplified VCPU management*: The trap-and-emulate technique consists of emulating guest's privileged instructions (i.e., instructions that only can be performed on supervisor mode). However, one single instruction may require tens or hundreds of instructions from the hypervisor side to be emulated, increasing the VCPU management complexity and overhead. Para-virtualization is commonly used to avoid such problems, which require the substitution of privileged instructions on guests by hypercalls. Thus, proper hardware-assisted virtualization can help keep the VCPU management small and straightforward because it allows for eliminating instruction emulation and most of the hypercalls. Nevertheless, hypercalls are still useful for virtualization extended services like inter-VM communication or peripheral sharing. See Section 4.1.

The proposed model follows the microkernel approach. Hence, the hypervisor implements only the necessary hardware control and minimal services, like inter-VM communication. For example, sophisticated network protocols, cryptography, file systems, and other libraries are supported at the VM level. Thereby, a VM with the picoTCP stack can be instantiated as a separate application for network support. The same can be done with cryptographic libraries (e.g., WolfSSL). A VM with an RTOS, like FreeRTOS, can implement real-time services with a predictable execution in parallel. All these features result in a flexible system capable of supporting microservices that all together build IoT applications.

4 | THE HELLFIRE HYPERVISOR IMPLEMENTATION

In this section, we describe the software implementation for the proposed model. We adopted the Microchip PIC32mz as the hardware platform, which comprises an M5150 processor core (MIPS32 architecture) with 512 KB of SRAM and 2 MB of flash. The M5150 core implements the MIPS virtualization extension (MIPSVZ), making it a perfect testbed. Thereby, we created the Hellfire Hypervisor, a hypervisor designed to deliver virtualization for small embedded devices, including IoT sensors and DES. It is open-source software, mostly written in C language with a few lines of Assembly, resulting in around 10k lines of code. It is available online at https://github.com/hellfire-project/hellfire-hypervisor. It was primarily designed to be as small and straightforward as possible; thus, using the hardware support to avoid complex software implementation. The remainder of the section describes the software techniques used to implement the proposed model presented in Section 3.

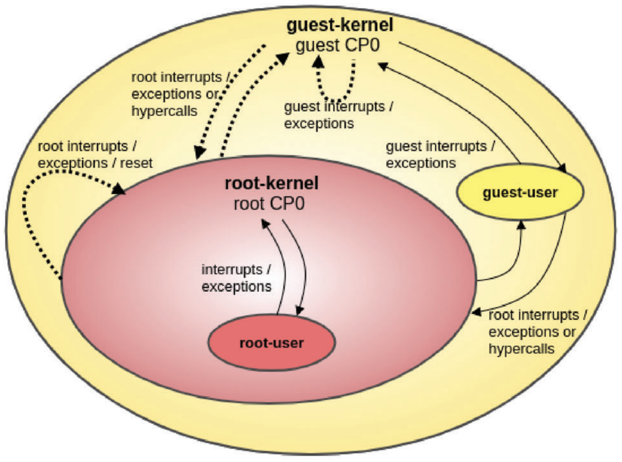
4.1 | Privilege-levels and context-switching

Before understanding the MIPS privilege-ring and the proposed context-switching scheme, it is essential to know that the MIPS has the *coprocessor 0* (CP0), a set of configuration registers accessed by the special instructions *mfc0* and *mtc0* in privileged mode only. In the MIPSVZ specification, a subset of these registers, named guest CP0, are duplicated, and they may be accessed from the VM's privilege level if allowed by the hypervisor.

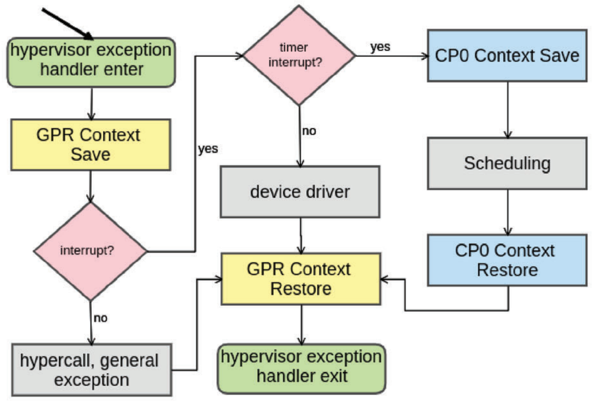
Figure 3(A) shows the complete privilege-ring and all possible ring transitions to the MIPSVZ core. It implements the root-kernel, root-user, guest-kernel, and guest-user ring levels. A GPOS would use only the root-kernel and root-user ring levels that are backward-compatible to the kernel and user-modes from classic architecture. The hypervisor uses the most privileged-ring (root-kernel) and delegates the VMs to the guest-kernel mode. Keeping the VMs in a less privileged ring allows for the hypervisor to create separation. In addition, our implementation uses a subset of the ring transitions, showed in Figure 3(A) as dotted arrows. Note that it is possible to handle interrupts directly in the guest-kernel mode.

The flowchart in Figure 3(B) describes the operations performed during the transitions. On entering the hypervisor exception handler (root-kernel), the GPRs are saved. In the case of a timer interrupt, the CP0 context is saved, the scheduler selects the next VM restoring its CP0 context. Any other interrupt will trigger the corresponding device driver. Other exceptions as hypercalls or guest faults are handled accordingly. Finally, the GPR context is restored, and the control is returned to the guest. Hence, the transition from guest-kernel to root-kernel happens in the following situations: (i) root-kernel interrupts and (ii) guest-kernel exceptions. For example, if an interrupt targeting the root-kernel happens, the processor will jump to the hypervisor's interrupt vector handler. Thus, it will perform the GPR context saving, perform the required operation at the device driver level, restoring the GPR, and jumping back to the guest-kernel mode. During the context restoring, two special operations must be performed: (i) set the Exception Program Counter (a CP0 register that keeps the VM's program counter); and (ii) set the CP0 guest_id register (used to select the translation look-aside buffer (TLB) entries and better-addressed in Section 4.2).

In our implementation, the hypervisor allows for the VMs to access a subset of the guest CP0. Thus, VMs can read the processor status and enable or disable interrupts, avoiding unnecessary traps from guest-kernel to root-kernel. A small penalty is paid; the hypervisor must save/restore the guest CP0 registers on every context-switching. In addition, the M5150 core has a GPR shadow scheme, allowing us to keep up to eight copies of the GPR context in hardware. One shadow page is reserved for the hypervisor, and the other seven are dedicated to guests. Hence, the core can quickly swap register files from shadow copies, that is, the GPR saving/restore routines need only indicate which is the shadow page needed. All this helps to *simplify the VCPU management*, as stated in Section 3.


guest-kernel
guest CP0
root interrupts / exceptions or hypercalls
guest interrupts / exceptions
root interrupts / exceptions / reset
guest interrupts / exceptions
root-kernel
root CP0
guest-user
interrupts / exceptions
root-user
root interrupts / exceptions or hypercalls

(A) The MIPSVZ privilege-ring model and its possible transitions. Dotted arrows show the transitions used by the proposed implementation.



(B) Flowchart to the hypervisor's context-switching and exception handler.

FIGURE 3 MIPS virtualization extension architecture privilege-ring and the flowchart for context-switching

4.2 | Memory virtualization

Memory virtualization plays a vital role in the hypervisor implementation. This subsystem is mainly responsible for the separation, keeping the different domains isolated. Its existence relies on a hardware mechanism present in the processor: the MMU. Memory addresses generated by the processor's core, called virtual addresses (VA), need to be translated to physical addresses (PA) by the MMU controlled by an OS or a hypervisor. The OS needs to keep a PT for each process mapping VA to PA. During context-switching, the OS remaps the processor to the corresponding PT. If a process requires an address translation not present in the PT, a trap is issued (page-fault). Thus, malfunctioning or even malicious memory access is detected and stopped. The MIPSVZ module implements hardware-assistance for memory management with an additional translation level, called two-level MMU. In this scheme, the guest OS configures its virtual memory using the guest CP0 in the same way in a standalone environment. Typically, the hypervisor keeps a PT for each VM and configures the MMU accordingly to the guest's needs. During memory translation, the two-level MMU generates IPA (intermediate physical address) from VA based on the guest's MMU. The PA will result from the combination with the hypervisor's MMU configuration. This scheme avoids modifying the guest OS while reducing the virtual memory configuration and translation traps. Figure 4 describes this mechanism.

Standard hypervisors for cloud computing implement a complete paging mechanism. As stated, the hypervisor keeps a PT to map guest OSs to physical memory. In these systems, the guest OS does not need to be entirely loaded into the main memory to be executed. The hypervisor can implement an on-demand paging mechanism (swapping). Such a scheme reduces memory usage since pages that have not been used recently can be stored in the swapping system. In addition, it avoids the memory external fragmentation problem because the VMs do not need to be allocated contiguously in physical memory. However, this approach has critical drawbacks for DES. First of all, swapping systems and on-demand paging mechanisms impact real-time responsiveness. Nevertheless, some DES do not support swapping due to storage restrictions. Moreover, a complete virtual memory management mechanism implies a more complex hypervisor and, consequently, a larger footprint and more processing requirements.

A *simplified virtual memory management* mechanism brings some advantages to DES. First, it avoids second-stage translation misses keeping the VM whole mapped at the hardware during its execution. Thus, bare-metal applications, RTOSs, or Unikernels that do not implement virtual memory support will not suffer additional delays and jitter due to hypervisor paging management. In addition, the limited number of virtual machines usually required by DES allows for a static configuration. For these systems, memory fragmentation due to contiguous guest OS allocation is not a significant problem. Thereby, our approach simplifies the memory management by combining two distinct techniques: (i) static VM's memory allocation (which is better addressed in Section 4.5) and (ii) avoiding to keep a complete PT scheme in memory. The contiguous memory allocation is represented in Figure 4.

The MIPS M5150 MMU implements a *TLB* that acts as a cache for the PT entries. It has 32 entries, where each entry consists of a guest_id (guest identification number) and a couple of fields to describe the memory map being created.

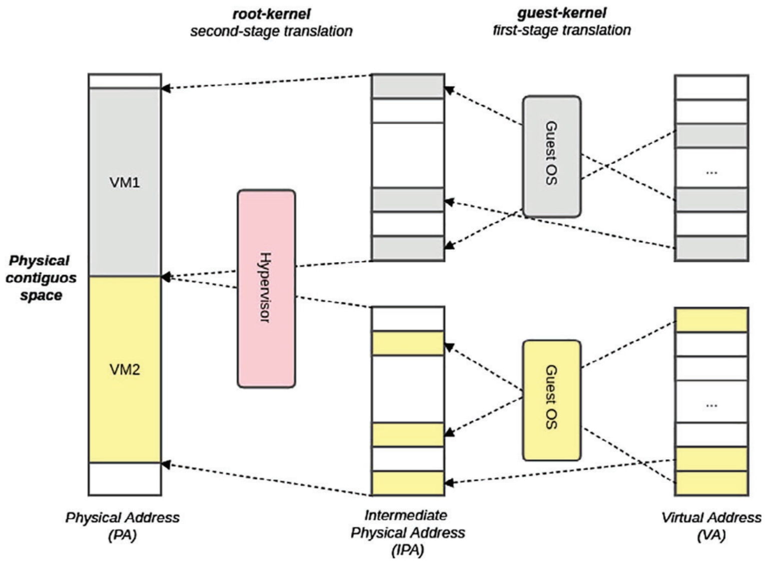


FIGURE 4 Hellfire Hypervisor memory management strategy using the MIPS virtualization extension two-level memory management unit hardware support. VMs are contiguously mapped in the physical memory

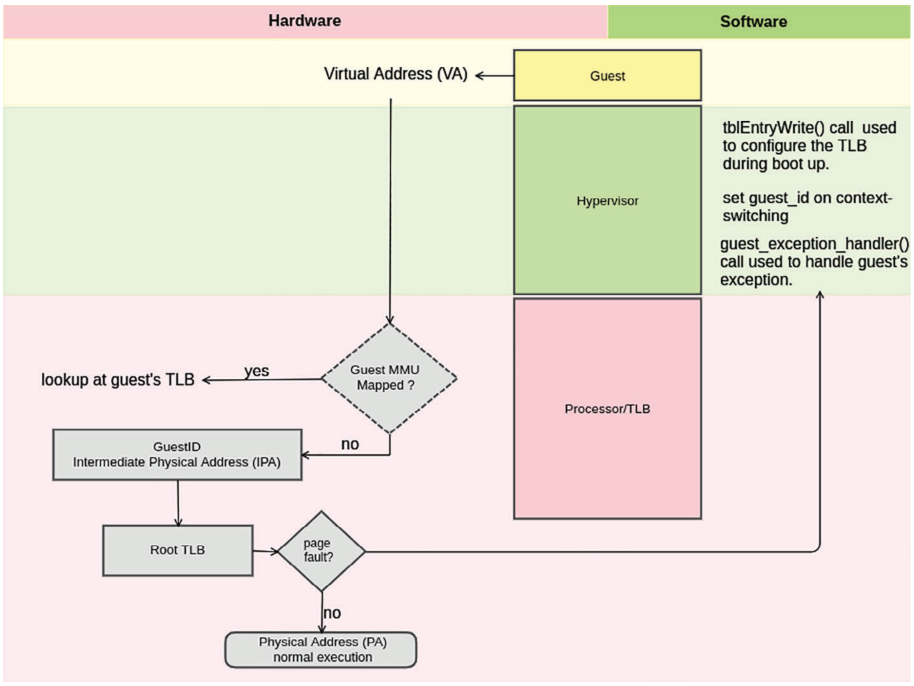

Hardware
Software
Virtual Address (VA)
Guest
Hypervisor
tblEntryWrite() call used to configure the TLB during boot up.
set guest_id on context-switching
guest_exception_handler() call used to handle guest's exception.
Guest MMU Mapped ?
yes
lookup at guest's TLB
no
GuestID Intermediate Physical Address (IPA)
Processor/TLB
Root TLB
page fault?
no
Physical Address (PA) normal execution

FIGURE 5 Hardware translation look-aside buffer (TLB) and hypervisor actions during address translation

During compilation, we generated a fixed-size array of elements that describe the memory mapping. On boot time, the hypervisor calls the *tlbEntryWrite()* routine for each element of the array. This routine translates the elements to the hardware TLB, allowing us to map IPA to PA. Among the hardware entries, guest_id has an important role; it indicates which VM an entry belongs to. With all needed entries loaded on hardware, the hypervisor only needs to set the guest_id during context-switching to remap the memory to another VM. This scheme is depicted in Figure 5, which clarifies the actions performed by the software (hypervisor) and the hardware. Once written the memory mapping to the TLB during boot, the software only needs to write the correct guest_id during context-switching and stop a VM execution in a page-fault case. From the hardware view, for each guest's memory access, a translation must generate the PA from the VA. Hence, the hardware checks the guest level TLB for memory mapping and generates the IPA. In our case, as we avoid the use of GPOSs or complex memory schemes on guests, the IPA is the same as the VA. The IPA is combined with the guest_id to find a second-level TLB (root TLB) match. If the translation is successful, the execution proceeds uninterrupted.

This scheme makes the context-switching lightweight and avoids to keep huge PTs on memory. Finally, the contiguous mapping does not affect the guest's memory management, allowing it to implement a complete management scheme. However, the focus of our hypervisor is more straightforward guests.

4.3 | I/O virtualization

Some hardware peripherals need to be shared among the VMs, such as Ethernet and timers. For example, when a guest tries to read or schedule a timer interrupt, the hypervisor will need to intercept these actions by traps or using para-virtualization to share the device properly. Similarly, the hypervisor may implement a network switch layer to allow guests to access the external world. As a consequence, all I/O may need to be controlled by the hypervisor. Both examples are complex in terms of implementation and lines of code. In addition, they may impose performance penalties on the hypervisor. VirtIO[24] surged as an effort to standardize the I/O interfaces for Linux hypervisors consisting of a set of Linux modules. Nonetheless, simplified subsystems are essential for embedded hypervisors. For example, the ability to map a peripheral directly to a VM redirecting its interrupts can save many efforts and diminish the hypervisor attack surface.

The proposed implementation supports *directly mapped devices*, which requires to map noncontinuous memory regions to a VM. Usually, I/O devices are mapped to specific PA. For example, a VM may have mapped 32 Kbytes of RAM allocated in the physical memory from 0x1000_0000 to 0x1000_8000. If the same guest requires access to a peripheral at the PA 0x1F00_0800 the hypervisor must configure a TLB-entry to match it. The static partitioning approach allows for defining all direct-mapped devices in a configuration file, as stated in Section 4.5.

4.4 | Inter-VM communication

The proposed hypervisor defines a hypercall interface for communication among VMs. This implementation adopts a message passing mechanism based on para-virtualization. The hypervisor routes messages among VMs using the address, size, and ID destination, which are hypercall parameters configured by the guest. Thereby, the hypervisor does not make any assumptions about the message formatting; this is entirely responsible for communicating guest OSs. For example, suppose a multitask guest OS needs to demultiplex[25] incoming messages among different tasks. In that case, it may add a header to the message indicating the origin and destination task id. In this case, the communicant guests must agree about the header format.

Each VCPU implements its incoming message queue as a limited circular buffer, statically allocated for performance purposes. A message targeting a determined VCPU will be copied to its queue, and the hypervisor will insert a virtual interrupt to the guest. The next time that the guest is executed, it will handle the virtual interrupt and call a hypercall to retrieve the message. Figure 6 describes hypervisor behavior while redirecting messages between guests. The VM 2 invokes the HCALL_IPC_SEND_MSG hypercall (1), causing a message copy from its buffer to the ring buffer of the destination VCPU. After, the hypervisor injects a virtual interrupt (2) in the VCPU 1. In the next execution, VM 1 will handle the interrupt executing the HCALL_IPC_RECV_MSG hypercall. Thus, the hypervisor will copy the ring buffer's message to the target buffer (3), completing the message's sending.

4.5 | Static partitioning

Cloud hypervisors implement management interfaces to allow users to configure all system elements. Cloud computing requires to instantiate or stop VMs without overall system interruption. Migration or reconfiguration should not influence the execution of the other guests. As discussed prior in this section, typical DES applications restrict the number of VMs, and usually, they must be executed during all device's runtime. Beyond simplifying the memory subsystem, as seen in Section 4.2, static partitioning benefits from these characteristics. Static partitioning consists of determining the system resources allocated to each guest at design time. For example, memory space, scheduling priorities, directly mapped devices, among other resources, are estimated by the developers and defined programmatically before compilation. Despite this method being less flexible than using an underlying GPOS or a hypervisor with a management interface, it brings two advantages: a small attack surface (making the system more robust against attacks) and simplicity.

To make the definition of the system's setup easier, we created a building scheme involving a configuration file and a tool to process it. Thereby, the partitioning is written in a structured file parsed by the *libconfig*, a C/C++ library for processing configuration files. This library has a compact and readable content format that is more appropriate than XML, similar to JSON schemes. In the configuration file, the user gives details about the system to be built. An example is given in Figure 7. The figure's left side shows a configuration file example with a section called *system* where it is defined

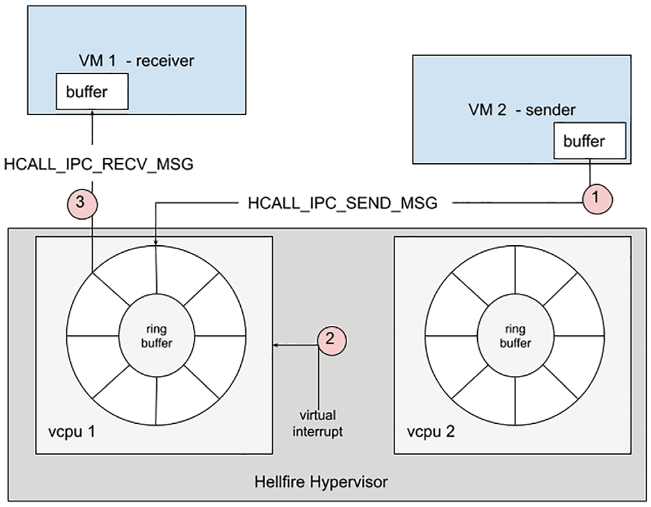
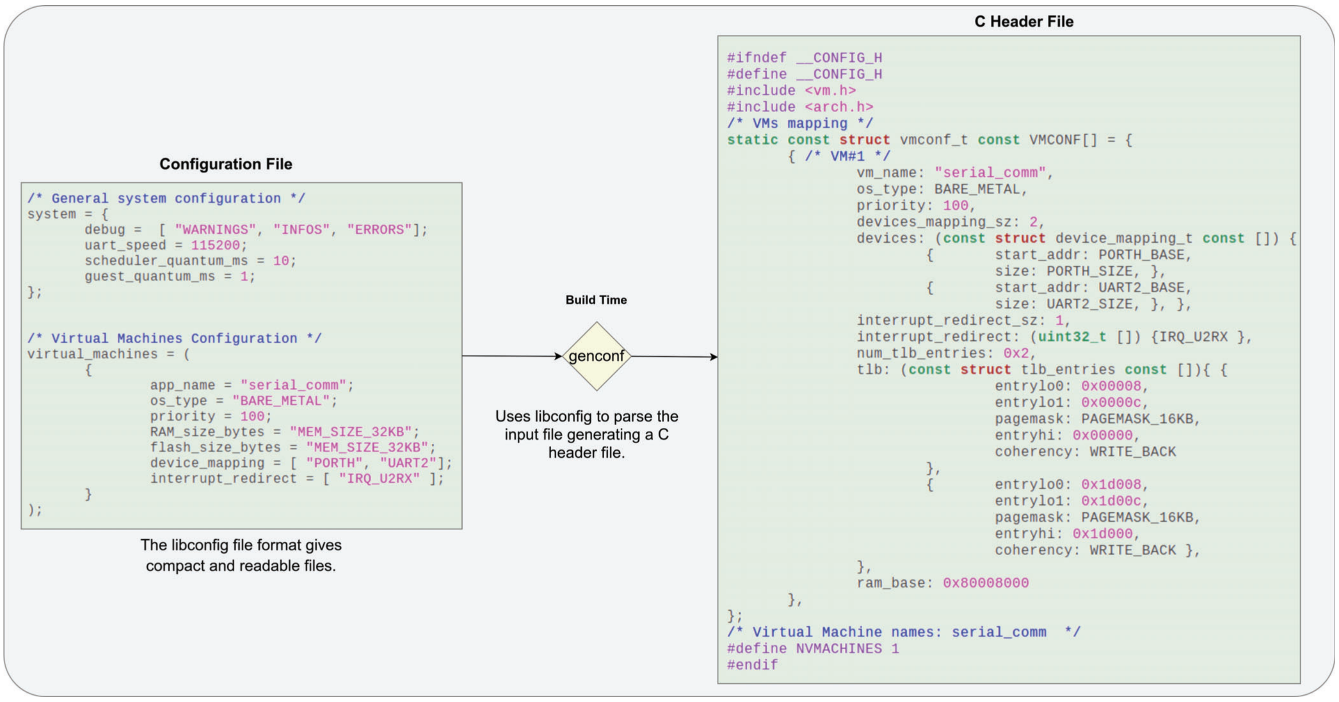


FIGURE 6 Example of inter-VM communication involving two guests


Configuration File
Build Time
genconf
Uses libconfig to parse the input file generating a C header file.
The libconfig file format gives compact and readable files.
C Header File

```
/* General system configuration */
system = {
        debug =  [ "WARNINGS", "INFOS", "ERRORS"];
        uart_speed = 115200;
        scheduler_quantum_ms = 10;
        guest_quantum_ms = 1;
};

/* Virtual Machines Configuration */
virtual_machines = (
        {
                app_name = "serial_comm";
                os_type = "BARE_METAL";
                priority = 100;
                RAM_size_bytes = "MEM_SIZE_32KB";
                flash_size_bytes = "MEM_SIZE_32KB";
                device_mapping = [ "PORTH", "UART2"];
                interrupt_redirect = [ "IRQ_U2RX" ];
        }
);
```

```
#ifndef __CONFIG_H
#define __CONFIG_H
#include <vm.h>
#include <arch.h>
/* VMs mapping */
static const struct vmconf_t const VMCONF[] = {
        { /* VM#1 */
                vm_name: "serial_comm",
                os_type: BARE_METAL,
                priority: 100,
                devices_mapping_sz: 2,
                devices: (const struct device_mapping_t const []) {
                        {       start_addr: PORTH_BASE,
                                size: PORTH_SIZE, },
                        {       start_addr: UART2_BASE,
                                size: UART2_SIZE, }, },
                interrupt_redirect_sz: 1,
                interrupt_redirect: (uint32_t []) {IRQ_U2RX },
                num_tlb_entries: 0x2,
                tlb: (const struct tlb_entries const []){ {
                                entrylo0: 0x00008,
                                entrylo1: 0x0000c,
                                pagemask: PAGEMASK_16KB,
                                entryhi: 0x00000,
                                coherency: WRITE_BACK
                        },
                        {       entrylo0: 0x1d008,
                                entrylo1: 0x1d00c,
                                pagemask: PAGEMASK_16KB,
                                entryhi: 0x1d000,
                                coherency: WRITE_BACK },
                },
                ram_base: 0x80008000
        },
};
/* Virtual Machine names: serial_comm  */
#define NVMACHINES 1
#endif
```

FIGURE 7 Configuration file versus the C header generated by the genconf tool

debug flags, serial speed, and the hypervisor scheduler's quantum. Following, a section called *virtual_machines* allows for creating a list of VMs specifying their scheduling priority, memory size, storage, mapped devices, and interrupts.

During the building time, the configuration file is read by a tool called gentool. We implemented this tool to help the developers to configure the system from a higher abstraction level view. Gentool knows the platform architecture details and creates the partitioning based on the number of VMs and the required size for each one. It outputs a C header used in the rest of the compilation process. The VM's configuration is grouped by a data structure, called *struct vmconf_t*, a fixed-size array that keeps the meta-data processed from the configuration file, see the right side of Figure 7. Note that the gentool's output is substantially more complex than the input file, especially memory partitioning. The tool considers the memory and storage sizes to optimize the allocation scheme for a given VM's set. The resulting allocation is stored in a fixed size array of *struct tlb_entry* elements. This information is read by the hypervisor during booting time and used to configure the processor's TLB.

Furthermore, the gentool keeps details about hardware devices and interrupts. The *device_mapping*propriety in the configuration file gives an array of device names to be mapped to the guest. The tool creates a structure called *struct device_mapping_t* that keeps the memory addresses and size of the memory-mapped devices allowed to the guest. Similarly, the *interrupt_redirect* is an array that keeps interruptions that must be redirected to the guest. Finally, the gentool gives a convenient way to configure the system and promote the hypervisor and the VM build.

4.6 | Real-time support

Several aspects may impact real-time responsiveness, as paging and swapping schemes or scheduling policies. Techniques as on-demand paging[23] or swapping bring execution unpredictability because when a required page is not present in memory, the process or VM is blocked until the data is loaded. The loading time may vary depending on the system load and the kind of storage involved. RTOSs overcome these problems by simplifying their implementation. For example, no memory management and a more straightforward software stack with predictable scheduling algorithms.

Our hypervisor implementation finds a thread-off between memory management's advantages, as separation, and the drawbacks in responsiveness. Thereby, it implements only the memory management features needed to provide separation. No additional schemes like on-demand paging or swapping are provided. As seen in Section 3, the DES characteristics dispenses complex memory management techniques. In addition, we implement a predictable round-robin scheduling algorithm with priority. Another feature of our hypervisor is the ability to support directly mapped devices, bypassing the hypervisor to access certain devices, avoiding additional overhead, and improving responsiveness. This feature is associated with the interrupt pass-through, where interrupts can be redirected to a VM without hypervisor intervention.

4.7 | Porting to other platforms

We designed the Hellfire Hypervisor software structure to facilitate the port to other architectures. In fact, beyond MIPS32, it already supports RISC-V for both rv32 and rv64 architectures. It implements a Hardware Abstraction Layer (HAL), providing a set of function calls that must be implemented to move the hypervisor across architectures. The HAL gives adequate support to build a higher software layer that is architectural independent. During the port, subsystems such as memory management, general exceptions, and interrupt control must be rewritten to reflect the new processor hardware that comprehends a small part of the software. For example, of 9800 hypervisor code lines, only 1530 corresponds to the M5150 and 1294 to the rv32 HALs. In addition, the gentool helps keep the same configuration files across architectures since it abstracts the configuration details, generating the adequate header file.

5 | EVALUATION AND RESULTS

This section brings a quantitative and qualitative analysis of our implementation. We analyzed the resulting hypervisor footprint and determined the virtualization performance impact using a well-known benchmark. In addition, we tested the inter-VM communication delay for a set of two applications and provided some real-time results. Finally, we present a brief qualitative analysis of security.

5.1 | Footprint analysis

The footprint aspects are important for small devices since the hypervisor and the application's size must be acceptable. We measured the hypervisor size for configurations with one, two, three, and four VMs, with the inter-VM communication, Ethernet, USB, and interrupt redirection drivers enabled. The one VM system consists of a simple blink led application. The two VMs implement the ping/pong application and use the inter-VM communication mechanism to exchange messages. The three VMs system consists of a combination of the blink and the ping/pong applications. To the four VMs system, we included a VM that performs the Coremark benchmark (see Section 5.2 for benchmark results). We have compiled the source-code using GCC 4.9.2 (Codescape GNU Tools 2016.05-03 for MIPS MTI Bare Metal) with Binutils 2.24.90 and the optimization levels O0 (no optimization), O2 (most of the supported optimizations that do not involve a space/speed trade-off), O3 (all O2 optimizations more optimizations for speed that may increase the footprint), and Os (optimize for space usage). In addition, we used the compiler flag *-micromips*. MicroMIPS is a code compression instruction set architecture that offers 32-bit performance with 16-bit code size for most instructions and is supported by the M5150, which allows for significant code reduction. Table 1 shows the results. All numbers are given in bytes, and only the hypervisor footprint is considered (VM's size is not included). The column *text+ro* means the size of the instructions

TABLE 1 Footprint results for the Hypervisor (bytes)

	GCC optimization level				
	data+bss	text+ro			
#VMs	all opt.	O0	O2	O3	Os
1	2016	32,632	21,328	25,496	20,156
2	2028	34,952	21,548	25,716	20,344
3	2048	37,620	21,684	25,852	20,468
4	2068	40,104	21,788	25,984	20,584

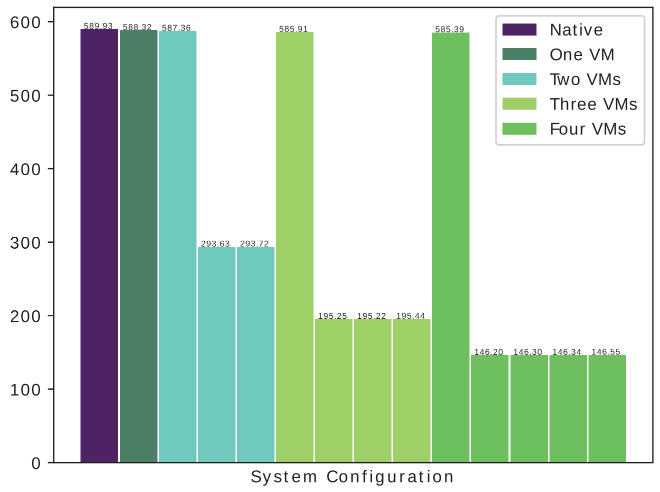


FIGURE 8 Coremark's score for an increasing number of VMs

and the read-only segments (kept in flash storage), while *data+bss* is the sum of the global initialized data and noninitialized data (loaded to RAM during boot). As the optimization levels do not affect the *data+bss* size, Table 1 shows only a column for all results.

It is seen that the compiler optimization level plays an important role. For example, the one VM system has a total footprint of 34,648 bytes (text+ro plus data+bss) for O0 optimization and 22,172 bytes when optimized for Os, a reduction of 36%. In all optimization levels, we can see a small increase in text+ro and data+bss sections with additional VMs. This happens because it is allocated a *struct vmconf_t*(see Section 4.5) in the read-only section for each new VM. In addition, a *struct vcpu_t* (a data structure that keeps the execution status of a VCPU) is allocated in the data section for each VM. Finally, based on footprint results, we can see that using optimization levels O2 or Os, it is possible to keep the total footprint around 23 KB, which is very optimistic for a hypervisor.

5.2 | Performance analysis

Coremark[*] is a benchmark used to measure embedded processors' performance. It was designed to run on microprocessors from 8 to 64-bit. It implements algorithms such as list processing, matrix manipulation, state machine, and cyclic redundancy check, all everyday operations in embedded applications. The Coremark result is a score number that can be used to compare performance among different processor families. Our goal was to determine the virtualization impact by comparing the native Coremark score with the hypervisor under different system configurations. We prepared five different setups: native, one VM, two VMs, three VMs, and four VMs systems. For the native setup, we performed the Coremark as a standalone application, that is, without the hypervisor. The remaining setups consist of a different number of parallel VMs running the Coremark application. Thereby, comparing the native resulting score to the score of different system setups, we expect to find the hypervisor overhead. In all setups, the optimization level used was 02, and the hypervisor scheduler quantum was configured for 5 ms, that is, it performs context-switching every 5 ms.

Figure 8 show the results. The native execution resulted in a Coremark score of 589.93, while the one VM system was 588.32 giving a performance penalty of 0.27%. For the remaining setups, the CPU time will be equally shared among all VMs, that is, in the four VM system, each VM will have only 25% of the CPU time. Thus, the resulting score will be divided among the VMs. Observe that, for the two systems VM, Figure 8 shows the resulting score for each VM and a column bar with the sum. In this case, VMs' score was 293.63 and 293.72, resulting in a total of 587.36, which gives an overhead of 0.43%. Using the same technique, we found overheads of 0.68% and 0.76% for the three and four VMs systems. These are very optimistic numbers that result from two main reasons: (i) the low hypervisor code complexity and (ii) the MIPSVZ hardware features, especially the TLB and the GPR shadows that keep the context-switching lightweight.

The MIPS 5150 implements performance counter registers[22] that can be programmed to count different kinds of hardware events, for example, number of executed instructions or invoked hypercalls. Thereby, we used the register counters to determine the impact of the different setups over the cache. For this, we programmed the counters to issue the number

[*]https://www.eembc.org/coremark/

of data and instruction cache misses. As expected, the cache misses for data and instructions increases exponentially with the addition of VMs. For the native execution, the data and instruction cache misses were 192 and 651, respectively. The one, two, three, and four VMs systems resulted in 10,167, 492,228, 897,950, and 3,903,672 data cache misses, and 712, 155,134, 617,441, and 3,788,338 instruction cache misses. The M5150 processor core has only 16 KB for instruction cache and 4 KB for data cache. The context-switching between VMs changes the spatial memory location abruptly, forcing new cache lines to be loaded. Additional VMs mean different spatial locations being accessed, and the amount of cache has not been enough. This problem may be minimized, increasing the scheduler quantum to, for example, 10 ms, causing two times less context-switching.

5.3 | Inter-VM communication delay

We tested the inter-VM communication focusing on evaluating the latency of message exchanges between VMs. Thus, we implemented two applications. The first one works as an echo server that replies to all received messages. The second one sends messages of 256 bytes, repeatedly calculating its round-trip time. We call them the ping-pong application. As a result, we obtained an average round-trip time of 199.97 μs after 1000 messages. Thereby, the inter-VM mechanism presented in Section 4.4 is an efficient and secure way to implement communication on the virtualized platform.

5.4 | Real-time analysis

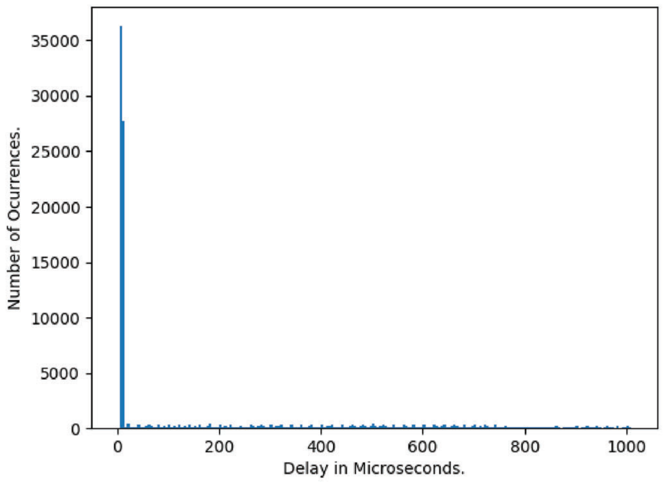
For real-time applications, it is essential to understand the behavior of the system regarding the response delay. To measure a VM's responsiveness in our hypervisor, we tested the response time for interrupt handling. For this, we implemented a VM capable of receiving interrupts from an I/O pin (external source) and generate outputs to another I/O pin. Hence, we used a function generator to issue interrupts every 10 ms. We measured the instants of the generated interrupt and the response in the output pin for each interrupt. The time difference is the total delay to the system to react to an external event. We tested the responsiveness in three situations:

1. the system idle, composed of one VM: the VM for the latency test;
2. the system under moderate load, composed of two VMs: the VM for latency test and the blink LED application;
3. the system under heavy load, composed of four VMs: the VM for latency test, the blink LED application, and the ping-pong application.

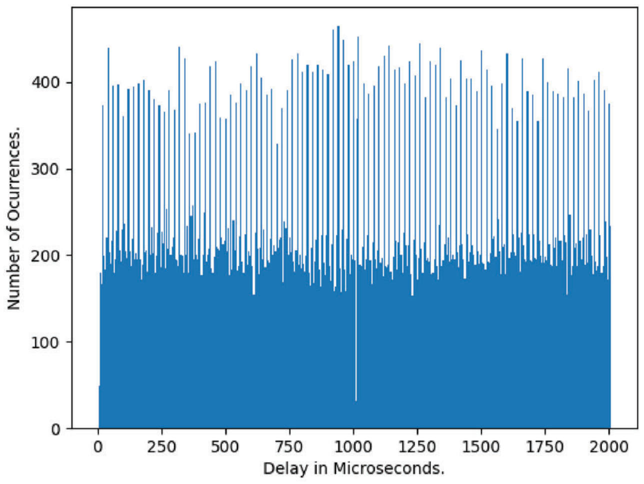
The ping-pong application generates a heavy load on the system since it changes messages exhaustively. For each situation, we generated 100,000 interrupts obtaining the response delay for each one. For the system idle test, the average response time was 8 μs. Since there is not scheduling (only one VM), the interrupts are directly sent to the VM (interrupt pass-through), resulting in a fast response. In the moderate and heavy load tests, interrupts may happen when the target VM is not in execution, requiring rescheduling. The rescheduling may happen when the interrupt arrives, or it is postponed if the hypervisor needs to attend to other VMs. Figure 9 presents the response delay histograms showing the time distribution. For the system under moderate load (two VMs), see Figure 9(A), the average response time was 173.23 μs with a minimal of 8 and maximum of 1008 μs. For the system under heavy load (four VMs), see Figure 9(B), the average response time was 1010.48 μs with a minimal of 8 and maximum of 2008 μs. We can see a variation depending on the system load, but it is possible to determine the worst-case response time making the resulting system behavior predictable.

5.5 | Security analysis

In Reference 17, we demonstrated how security could be delivered in DES using the Hellfire Hypervisor. A security architecture involving trust mechanisms and virtualization was proposed, where a secure boot allows for a root of trust environment and, ultimately, ensuring a chain of trust. Afterward, a trustworthy virtualization layer is booted up. The intrinsic virtualization characteristics, such as separation, ensure protection during the VM's boot and runtime states. Cryptography algorithms can be used to provide integrity/authenticity to the system. Results for footprint and overall performance were presented, showing that the technique is feasible for DES.



(A) Response delay for the system with a moderate load (2 VMs).



(B) Response delay for the system with a heavy load (4 VMs).

FIGURE 9 Histograms for interrupt responsiveness for system under moderate and heavy loads

6 | RELATED WORK

Virtualization is a well-established technology, with various hypervisor solutions, mainly due to many use cases ranging from servers[20,26] to embedded systems.[19,27-29] Xen[20] is an open-source and type-1 hypervisor that relies on a privileged VM, called Dom0, to manage nonprivileged VMs and interface with peripherals. KVM[26] is also an open-source but was designed as a hosted hypervisor and integrated into Linux's kernel. Although initially developed for desktop and server-oriented applications, both hypervisors have found their place in the embedded space.[27,28]

In literature, there are relevant studies that propose lightweight implementations of hypervisors.[19,29] The work in Reference 29 presents Muen Separation Kernel, an open-source microkernel, formally proven to contain no runtime errors at the source code level. It uses Intel's hardware-assisted virtualization technology (VT-x) to provide strong separation and make its implementation simpler. The Muen project makes use of emulation by employing the Bochs IA-32 emulator. The kernel has approximately 2700 lines of code. The work does not present performance results.

Authors in Reference 19 present Bao, a lightweight hypervisor implementation that uses a static partitioning architecture, supporting Armv8 and RISC-V platforms. Bao strongly focuses on isolation for fault-containment and real-time behavior. Its kernel has approximately 5600 lines of code. Tests were executed in Xilinx ZCU104, featuring a Zynq-US+ SoC with a quad-core Cortex-A53 running at 1.2 GHz, per-core 32K L1 data and instruction caches, and a shared unified 1 MB L2/LLC cache. The hypervisor code and benchmark applications were compiled using the Arm GNU Toolchain version 8.2.1 with -O2 optimizations. Results regarding memory show that it needs 23 KB of storage and 17 KB during runtime. To assess virtualization performance overhead, authors employed the MiBench Embedded Benchmark Suite.[30] Preliminary evaluation shows Bao generates an average virtualization overhead of 1.25% (one VM) and 32.50% for multiple VMs executions.

The work in Reference 31 presents Xvisor, an open-source type-1 hypervisor, focused on providing a monolithic, lightweight, portable, and flexible virtualization solution. It supports ARM virtualization extensions to provide full-virtualization and para-virtualization through optional VirtIO compatible device drivers. It can map interrupts directly to guests, allowing guest interrupts to be handled without the hypervisor's intervention. In addition, it provides memory isolation between hypervisor, guests, and guest applications using the third privileged level from ARM's virtualization support. The kernel has approximately 440K lines of code. Experimental results show that Xvisor ARM guest has lower CPU overhead and higher memory bandwidth than KVM ARM guest and Xen ARM DomU.

Authors in References 32-34 present seL4, a high-assurance, high-performance OS microkernel. seL4 is the most advanced member of the L4 microkernel family. It supports virtual machines that can run a fully fledged guest OS. Subject to seL4's enforcement of communication channels, guests and their applications can communicate with each other and native apps. In terms of source-code size, the kernel is about 9400 SLOC (ARM and RISC-V). In terms of executable code

		Memory		
Work	Lines of code	Storage	RAM	Footprint
Muen[29]	2700	75 KB	16 KB	91 KB
Bao[19]	5600	41 KB	18 KB	59 KB
Xvisor[31]	440, 000	1–2 MB	4–18 MB	5–20 MB
seL4[32-34]	9400	138 KB	24 KB	162 KB
Hellfire (this work)	9800	21 KB	2 KB	23 KB

TABLE 2 Comparison with related works

size, the kernel has about 138 KB. Its RAM size is about 24 KB. Regarding virtualization overhead, the authors do not present numerical results but argue that seL4 adds minimal overhead.

There has been an increasing interest in containers in recent years, which are a vital element of modern cloud computing and play an important role in emerging concepts.[15] However, it is a solution that still requires a GPOS. On the other hand, specially designed type-1 hypervisors enable the implementation of smaller virtualization layers compatible with embedded system constraints.

Table 2 shows that embedded virtualization can build hypervisors with small memory footprints and low virtualization overheads. Type-1 embedded hypervisors can be used for more restricted devices requiring additional modularity and security, adding the advantage of supporting RTOSs, unikernels, or bare-metal applications. Another advantage of type-1 architecture is its strong separation. A deep concern about edge computing is security. Therefore, to improve the security of sensitive domains through a separation is essential.

Our solution, the Hellfire Hypervisor, was designed especially for DES. It is custom-made during compilation time, that is, its data structure length is defined during the building process. The expressive results presented in Section 5 and Table 2 show that the Hellfire hypervisor can enable edge computing in DES due to its small footprint, low virtualization overhead and inter-VM communication delay, and real-time support while enforcing security by separation.

7 | TEST CASE SCENARIO AND BENEFITS TO SMART CITY APPLICATIONS

We also evaluated the proposed approach in a scenario that highlights its capabilities for IoT deployment. The scenario, presented in Figure 10, depicts a typical smart city application that monitors the air quality in urban areas. Air quality is usually monitored by networks of fixed stations strategically placed in the city, where each station can measure a wide range of pollutants (Figure 10(A)). In this scenario, each station has a DES, which was implemented in a MIPS32 processor core running at 200 Mhz, with 2 MB of flash memory and a 512 KB SRAM. The DES was connected to some sensors simulated by software to monitor the environment: light, CO_2 (carbon dioxide), CH_2O (methanal), and temperature. The DES's software components were implemented and are shown in Figure 10(B). The air quality monitoring application was divided into three applications: *data acquisition*, *data reasoning*, and *network communication*. They execute into each DES in order to monitor the air quality in different parts of the city. The *data acquisition* application receives raw data from sensors and sends them to the *data reasoning* application through inter-VM communication. The *data reasoning* performs data filtering and aggregation to make decisions. For example, it can generate alarms for pollution peaks or aggregate and compact relevant data to reduce communication. Finally, the *network communication* application implements the required network stacks to send data to the fog/cloud monitoring device (Figure 10(C)). Regarding memory and storage requirements, we observed a requirement of about 32 KB of flash and 16 KB of SRAM to communicate the *data acquisition* application with sensors using any of the following interface options: I2C, UART, SPI, or USB. The *data reasoning* application had the same requirements since it does not implement complex software stacks. *Network communication* was achieved using the picoTCP stack[†]. It required 128 KB for storage and 64 KB of SRAM to support TCP/IP and HTTP protocols. Thus, resulting in a total footprint of 288 KB (SRAM and storage) for the three applications execution.

The proposed hypervisor can be useful not only in this experiment but in all IoT smart scenarios composed of DES devices, such as smart home care, smart building, smart lighting, and smart parking, as highlighted in Figure 10(D). The use of DES devices in the described scenarios means the reduction of cost and power consumption.

[†] https://github.com/tass-belgium/picotcp

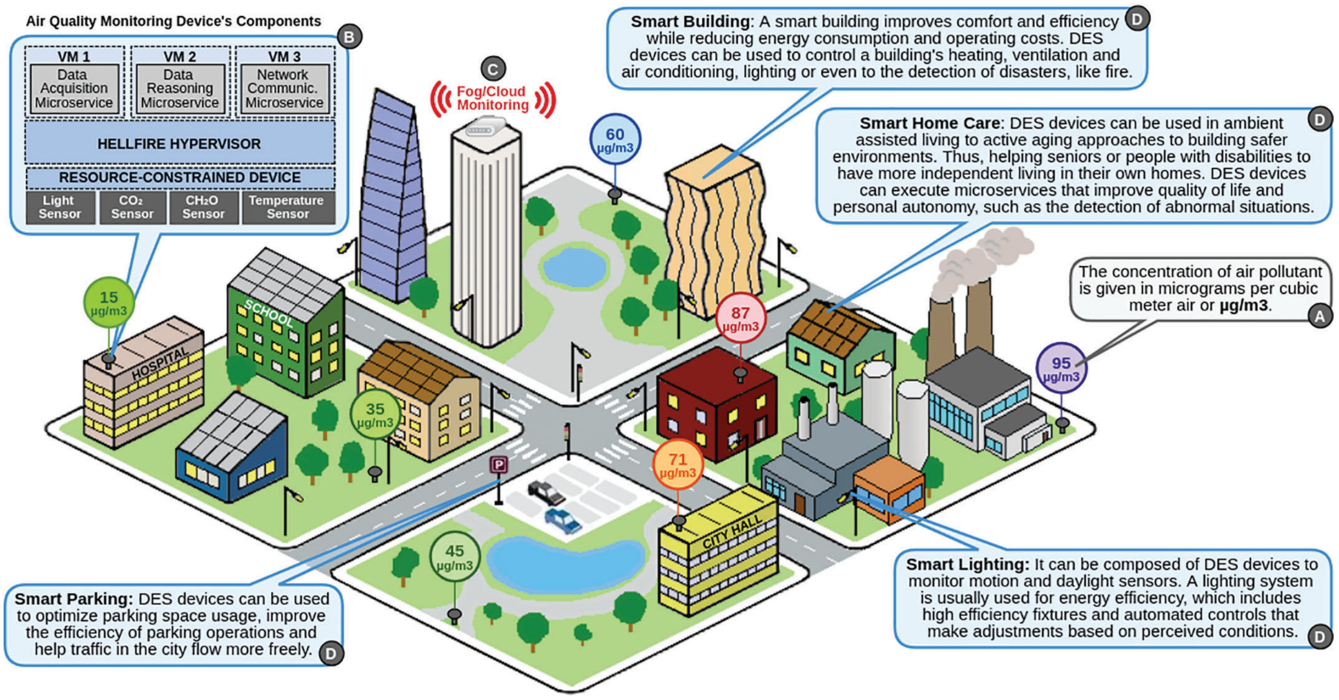

Air Quality Monitoring Device's Components
B
VM 1 Data Acquisition Microservice
VM 2 Data Reasoning Microservice
VM 3 Network Communic. Microservice
HELLFIRE HYPERVISOR
RESOURCE-CONSTRAINED DEVICE
Light Sensor
CO₂ Sensor
CH₂O Sensor
Temperature Sensor
C
Fog/Cloud Monitoring
Smart Building: A smart building improves comfort and efficiency while reducing energy consumption and operating costs. DES devices can be used to control a building's heating, ventilation and air conditioning, lighting or even to the detection of disasters, like fire. D
Smart Home Care: DES devices can be used in ambient assisted living to active aging approaches to building safer environments. Thus, helping seniors or people with disabilities to have more independent living in their own homes. DES devices can execute microservices that improve quality of life and personal autonomy, such as the detection of abnormal situations. D
The concentration of air pollutant is given in micrograms per cubic meter air or µg/m3. A
Smart Lighting: It can be composed of DES devices to monitor motion and daylight sensors. A lighting system is usually used for energy efficiency, which includes high efficiency fixtures and automated controls that make adjustments based on perceived conditions. D
Smart Parking: DES devices can be used to optimize parking space usage, improve the efficiency of parking operations and help traffic in the city flow more freely. D
15 µg/m3
35 µg/m3
45 µg/m3
60 µg/m3
71 µg/m3
87 µg/m3
95 µg/m3
HOSPITAL
SCHOOL
CITY HALL
P

FIGURE 10 Air quality monitoring scenario in urban areas. In addition, smart cities scenarios that can benefit from our approach

8 | CONCLUSION

In this article, we proposed a virtualization model to enable IoT applications in DES. We presented the virtualization model and described our implementation, called Hellfire hypervisor. Following this, we evaluated the implementation and presented a quantitative and qualitative analysis to demonstrate its effectiveness. To complement the results, we presented application scenarios that can benefit from the proposed model.

The Hellfire hypervisor does not compete with current methodologies, like containerization. However, it is complementary since it can reach devices that container engines cannot afford, enabling edge computing on devices with minimal CPU, storage, and memory resources.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

ORCID

Ramão T. Tiburski https://orcid.org/0000-0002-7350-5195
Everton de Matos https://orcid.org/0000-0001-5888-0969

REFERENCES

1. Ahmed E, Ahmed A, Yaqoob I, et al. Bringing computation closer toward the user network: is edge computing the solution? *IEEE Commun Mag*. 2017;55(11):138-144. https://doi.org/10.1109/MCOM.2017.1700120.
2. Matos DE, Tiburski RT, Moratelli CR, et al. Context information sharing for the Internet of Things: a survey. *Comput Netw*. 2020;166:1-19. https://doi.org/10.1016/j.comnet.2019.106988.
3. Ahmed E, Ahmed A, Yaqoob I, et al. Orchestration of microservices for IoT using docker and edge computing. *IEEE Commun Mag*. 2018;56(9):118-123. https://doi.org/10.1109/MCOM.2018.1701233.
4. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: vision and challenges. *IEEE IoT J*. 2016;3(5):637-646.
5. Iorga M, Feldman L, Barton R, Martin MJ, Goren NS, Mahmoudi C. Fog computing conceptual model. Technical report, NIST SP 500-325; Gaithersburg, MD; 2018.

6. Sunyaev A. *Internet Computing Principles of Distributed Systems and Emerging Internet-based Technologies*. Switzerland, AG: Springer Nature; 2020.
7. Naha RK, Garg S, Georgakopoulos D, et al. Fog computing: survey of trends, architectures, requirements, and research directions. *IEEE Access*. 2018;6:47980-48009. https://doi.org/10.1109/ACCESS.2018.2866491.
8. Taherizadeh S, Stankovski V, Grobelnik M. A capillary computing architecture for dynamic internet of things: orchestration of microservices from edge devices to fog and cloud providers. *Sensors*. 2018;18(9, 2938):1-23.
9. Souza A, Cacho N, Noor A, Jayaraman PP, Romanovsky A, Ranjan R. Osmotic monitoring of microservices between the edge and cloud. Paper presented at: Proceedings of the IEEE 20th International Conference on High Performance Computing and Communications IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). Exeter, UK; 2018:758-765.
10. Morabito R, Petrolo R, Loscri V, Mitton N. LEGIoT: a lightweight edge gateway for the Internet of Things. *Future Generat Comput Syst*. 2018;81:1-15.
11. Singh S, Singh N. Containers & Docker: emerging roles & future of cloud technology. Paper presented at: Proceedings of the IEEE. 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT). Bangalore, India; 2016:804-807.
12. Abdollahi Vayghan L, Saied MA, Toeroe M, Khendek F. Deploying microservice based applications with kubernetes: experiments and lessons learned. Paper presented at: Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD). San Francisco, CA; 2018:970-973.
13. Morabito R. Virtualization on Internet of Things edge devices with container technologies: a performance evaluation. *IEEE Access*. 2017;5:8835-8850. https://doi.org/10.1109/ACCESS.2017.2704444.
14. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper Syst Rev*. 2007;41(3):275-287. https://doi.org/10.1145/1272998.1273025.
15. Morabito R. Consolidate IoT edge computing with lightweight virtualization. *IEEE Netw*. 2018;32(1):102-111. https://doi.org/10.1109/MNET.2018.1700175.
16. Moratelli C, Johann S, Neves M, Hessel F. Embedded virtualization for the design of secure IoT applications. Paper presented at: Proceedings of the International Symposium on Rapid System Prototyping (RSP). Pittsburgh, Pennsylvania, 2016:2-6.
17. Tiburski RT, Moratelli CR, Johann SF, et al. Lightweight security architecture based on embedded virtualization and trust mechanisms for IoT edge devices. *IEEE Commun Mag*. 2019;57(2):67–73.
18. Pinto S, Gomes T, Pereira J, Cabral J, Tavares A. IIoTEED: an enhanced, trusted execution environment for industrial IoT edge devices. *IEEE Internet Comput*. 2017;21(1):40-47. https://doi.org/10.1109/MIC.2017.17.
19. Martins J, Tavares A, Solieri M, Bertogna M, Pinto S. Bao: a lightweight static partitioning hypervisor for modern multi-core embedded systems. In: Bertogna M, Terraneo F, eds. *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020). 77 of OpenAccess Series in Informatics (OASIcs)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2020:3:1-3:14.
20. Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization. *SIGOPS Oper Syst Rev*. 2003;37(5):164-177. https://doi.org/10.1145/1165389.945462.
21. ARM ARM architecture reference manual. ARMv7-A and ARMv7-R edition. ARM DDI C; 2012:406.
22. Imagination Technologies Ltd Virtualization module of the MIPS32 architecture. Technical report, Imagination Technologies Ltd; 2013.
23. Silberschatz A, Galvin PB, Gagne G. *Operating System Concepts*. 9th ed. New York, NY: Wiley Publishing; 2012.
24. Russell R. Virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper Syst Rev*. 2008;42(5):95-103. https://doi.org/10.1145/1400097.1400108.
25. Tanenbaum AS, Wetherall DJ. *Computer Networks*. 5th ed. Upper Saddle River, NJ: Pearson Prentice Hall; 2011.
26. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the linux virtual machine monitor. Paper presented at: Proceedings of the Linux Symposium; 2007:225-230; Ottawa, Ontario, Canada.
27. Dall C, Nieh J. KVM/ARM: the design and implementation of the linux ARM hypervisor. Paper presented at: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems; 2014:333-348; New York, NY.
28. Xi S, Wilson J, Lu C, Gill C. RT-Xen: towards real-time hypervisor scheduling in Xen. Paper presented at: Proceedings of the 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT). Taipei, Taiwan; 2011:39-48.
29. Buerki R, Rueegsegger AK. Muen-an x86/64 separation kernel for high assurance. University of Applied Sciences Rapperswil (HSR), Technical report; 2013.
30. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. MiBench: a free, commercially representative embedded benchmark suite. Paper presented at: Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). Austin, TX; 2001:3-14.
31. Patel A, Daftedar M, Shalan M, El-Kharashi MW. Embedded Hypervisor Xvisor: a comparative analysis. Paper presented at: Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Turku, Finland; 2015: 682-691.
32. Klein G, Elphinstone K, Heiser G, et al. SeL4: formal verification of an OS kernel. *SOSP '09*. New York, NY: Association for Computing Machinery; 2009:207-220.

33. Klein G, Andronick J, Elphinstone K, et al. Comprehensive formal verification of an OS microkernel. *ACM Trans Comput Syst*. 2014;32(1):1–70. https://doi.org/10.1145/2560537.
34. The seL4; 2020. https://cdn.hackaday.io/files/1713937332878112/seL4-whitepaper.pdf. Accessed November 05, 2020.

How to cite this article: Tiburski RT, Moratelli CR, Johann SF, de Matos E, Hessel F. A lightweight virtualization model to enable edge computing in deeply embedded systems. *Softw Pract Exper*. 2021;51:1964–1981. https://doi.org/10.1002/spe.2968