2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)

*SBI*s: Application Access to Safe, Baremetal Interrupt Latencies*

Runyu Pan
School of Computer Science and Technology, Shandong University
rypan@sdu.edu.cn

Gabriel Parmer
The George Washington University
gparmer@gwu.edu

***Abstract*—The continued increase in Cyber-Physical System (CPS) complexity and tightening of Size, Weight and Power (SWaP) constraints are driving the need for consolidation of software tasks onto fewer microcontrollers. Many embedded systems, prominently including those in the Internet-of-Things (IoT), use software packages from multiple untrusted sources, while their network interfaces expose new attack surfaces that are not present in traditional off-line devices. Increased consolidation with untrusted code of various assurance levels complicates system design, and requires increased spatial and temporal isolation between the applications. Current microcontroller protection domain designs are limited by their long interrupt latencies to isolated applications, forcing the system designers to place timing-sensitive application code into the kernel interrupt handlers, trading *spatial* isolation for tightly-bounded *temporal* predictability.**

***SBI* (Secure Baremetal Interrupt) enables *zero-software-cost* delivery of interrupts to protection domains in a secure manner that maintains isolation. We demonstrate an implementation of *SBI* using the new hardware-accelerated interrupt delivery features on *TrustZone-M*-enabled microcontrollers. This implementation reduces interrupt latencies by up to 95%, while maintaining strong *spatial* and *temporal* isolation. We believe *SBI* could significantly enable future real-time systems that require *both* isolation and high responsiveness.**

I. Introduction

The design of traditional cyber-physical systems employs multiple microcontrollers, *e.g.* vehicle electronics, avionics, and programmable logic controllers. Each microcontroller is responsible for a specific type of real-time or best-effort task, and a shared bus interconnect (*e.g.* controller area network) is responsible for mediating communication between them. Currently, this traditional design paradigm is facing challenges from both system complexity and tight Size, Weight and Power constraints. These challenges are further exacerbated with the difficulties in orchestrating development between different microcontroller systems.

The difficulties faced by the traditional multi-microcontroller designs, *e.g.* autonomous vehicles, call for microcontroller consolidation, which replaces multiple microcontrollers with a single consolidated instance. However, traditional standalone microcontroller software commonly runs on the bare-metal, or uses a lightweight Real-Time Operating System (RTOS) without isolation facilities. A rewrite of the software to consolidate multiple of these traditionally stand-alone systems into one may invalidate potential safety certifications and lead to new rounds of engineering effort. Moreover, the timing of different real-time applications may interfere with each other due to the sharing of CPU, potentially threatening schedulability.

With the advent of Internet-of-Things (IoT), software packages from multiple potentially untrusted sources are also required to run on the same piece of hardware. Additionally, the network interfaces expose new attack surfaces that are not present in traditional off-line devices. Contrary to the traditional multi-microcontroller designs, the consolidated design faces various new challenges. An error in one of the software components may corrupt other software components, and one compromised software component may in turn compromise the whole system; the real-time applications may also compete for CPU execution time, which potentially leads to deadline misses. Thus, isolation is required between software components. This includes *spatial isolation* in which memory is segregated between protection domains to ensure integrity, and *temporal isolation* in which all execution is properly prioritized, and the impact of even high-priority malicious or errant execution is bounded.

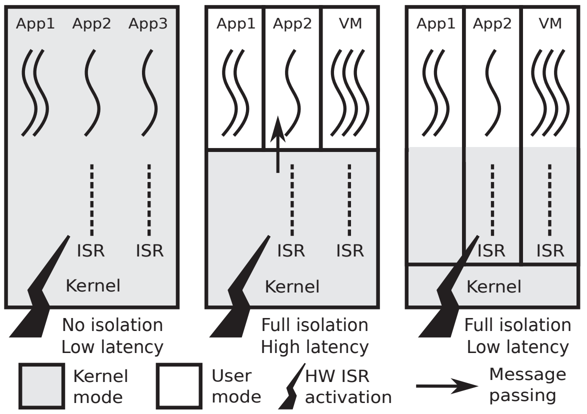

Fig. 1: *SBI-TZ* design overview. Kernel-mode is grey-colored while user-mode is white-colored. Dotted lines indicate Interrupt Service Routines (ISRs), and solid curves indicate application threads. Rectangles indicate protection domains. Lightning bolts indicate ISR activation, and arrows indicate the message passing from the ISRs to applications. The bare-metal OS architecture shown on the left has low interrupt latency but no isolation; the protected OS architecture shown in the middle has full isolation but high latency; the proposed *SBI-TZ* architecture with *SBI* shown on the right has low interrupt latency and full isolation.

Outside of the microcontroller field, virtualization has long

*This material is based upon work supported by the NSF under Grants No. CNS 1815690 and CPS 1837382, and by SRC under grants GRC task 2911.001 and SRC JUMP task 2779.030. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation nor the Semiconductor Research Corporation.

Class	Application	Latency	Explanation
Baremetal	USB I/O emulation	0.5μs	Emulation must be cycle-accurate or communication will fail [1]
Baremetal	1-Wire interface emulation	0.5μs	1-Wire bus requires accurate management of its time slots [2]
Baremetal	Stepping motor control	1μs	Stepping motor with a fine step angle needs rapid pulses [3]
Baremetal	Rotary encoders	1μs	Servo motor reports positions with encoders that toggle rapidly
Baremetal	BLDC motor control	1μs	BLDC servo motor needs accurate control over the transistors
Baremetal	External ADC data logging	1μs	Low-jitter external ADC driving produces high-quality data [4]
Baremetal	Machine vision strobed LEDs	10μs	Use high-speed blinkers for good imaging quality
Baremetal	Communication protocols	10μs	TDMA and FH protocols require accurate data transfer timing
Baremetal	IR remote control decoding	10μs	IR codewords need to be captured with external interrupts
Baremetal	Engine control units	10μs	Engine ignition timing needs to be considerably accurate
Threaded	Quadcopter control loops	100μs	The typical high-criticality quadcopter task period is 1ms [5]
Threaded	SPI interface emulation	1ms	Higher I/O toggling rate means a higher data rate
Threaded	USB packet processing	5ms	The typical timeout for a USB request is 50ms

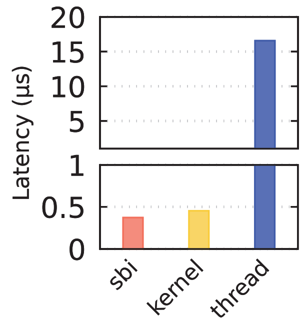

TABLE I: Application latency requirements (left) and comparison between latencies (right). Baremetal-class applications require tight interrupt latencies whereas threaded-class applications do not. Latency means the maximum allowed interrupt latency. `sbi` is the confined *SBI* latency, `kernel` is the privileged kernel interrupt latency, and `thread` is the thread unblock latency upon receiving an interrupt.

been adopted for microprocessor architectures (*e.g.* x86, ARM, MIPS and PowerPC) as a mechanism for providing isolation. A virtualization infrastructure simulates a number of mutually isolated virtual hardware platforms given a single piece of real hardware. This makes it ideal for accommodating applications or systems that used to run on multiple pieces of hardware without significantly modifying the codebase.

Significant research has been performed on microcontroller protection domains [6], [7], [8], [9], and the paravirtualization platform built on them. However, in these designs, the hardware interrupts are serviced by the kernel-level software first before they are passed to any user-level protection domain. These indirections impose high overheads including kernel context switches between protection domains. Such overhead, though acceptable in many cases, is prohibitive for certain applications that directly interact with the timing-sensitive actuators or I/O pins, *e.g.* high-speed servo motor controllers. These applications require tight interrupt latencies only achievable by installing their handlers in the kernel, thus must be isolated from code of lesser timing assurance on the system. Additionally, a faulty or malicious handler should have its negative impact on the rest of the system's software be bounded, despite having low-level access to interrupt-based execution.

This paper introduces *SBI* (Secure Bare-metal Interrupt), a mechanism that (1) enables interrupt service routine activation to be vectored directly to a user-level protection domain, thus bypassing the system kernel and providing bare-metal latencies, (2) interrupts are properly prioritized with respect to the thread scheduler [10], [11], and (3) application ISR execution is constrained to have only a bounded and predictable overhead on other applications and Virtual Machines (VMs). Thus, *SBI* enables bare-metal interrupt handling overheads, while enabling the *spatial* and *temporal* isolation requirements for predictable and secure real-time systems.

A practical implementation of *SBI*s is enabled by recent advances in microcontroller designs such as those provided by ARM *TrustZone-M* for interrupt delivery to a "*Normal* World". Upon this foundation, we provide a virtualization infrastructure "*SBI-TZ*" (Figure 1) to enable consolidation of VMs with varying assurances, and fast ISR execution. Due to *TrustZone-M* hardware restrictions, the current implementation only supports one *SBI*-enabled VM per *TrustZone-M*-enabled CPU core. The value of this implementation is an existence proof of the *SBI* concept even on hardware that isn't well suited for it, and motivates a more directed implementation on other hardware in the future. To this end, this paper also discusses some hardware design suggestions that synergize with the proposed *SBI* mechanism, which help to overcome the restrictions that come with the current *TrustZone-M* hardware.

Contributions. The contributions of this paper include:

- an introduction of the *SBI* design (§III) and its adaptation of the hardware-accelerated interrupt delivery features to enable tight interrupt latencies while maintaining strong *temporal* isolation between protection domains;
- a prototype implementation of *SBI* in the Composite microkernel as a foundation for the *SBI-TZ* accelerated microcontroller virtualization prototype (§IV);
- an evaluation of the *SBI-TZ* prototype compared to an existing bare-metal OS without *SBI* to understand the fundamental predictability and efficiency properties of the system's operations (§V); and
- an evaluation of a VM with *SBI-TZ* compared to the same VM without *SBI-TZ* to demonstrate the applicability of *SBI* in real-world virtualization applications (§V).

II. Background and Related Work

Application interrupt latency requirements. The maximum interrupt latencies allowed for typical applications are shown in Table I, alongside different latencies found on a typical microcontroller. It may be observed that the applications are divided into two categories - the threaded-class whose maximum latencies allow execution in OS threads, and the baremetal-class whose maximum latency is tight so that execution in hardware ISRs is required.

However, current microcontroller isolation facilities are limited by their long interrupt latencies to isolated OS threads (`thread`), forcing the system designer to place timing-sensitive application code into the privileged kernel-level ISRs (`kernel`), trading *spatial* isolation for *temporal* predictability. Note that microcontroller manufacturers may have their own names for the `kernel` and `user` modes; for Cortex-M devices leveraged in this paper, we refer to "`privileged handler mode`" as `kernel`, and refer to "`unprivileged thread mode`" as `user` to simplify terminology. Note also that microcontrollers are often clocked in the 10s to 100s of MHz, thus achieving these latencies doesn't allow for much overhead. Though efficient handling of events is often required to minimize the time in a higher-power mode, in this paper we focus on the real-time behavior. In this work, we propose *SBI* which enables bare-metal interrupt latencies (`sbi`) while enforcing both *spatial* and *temporal* isolation.

From / To		Current processor mode			
		K *Secure*	K *Normal*	U *Secure*	U *Normal*
Target	K *Secure*	✓	✗	✗	✗
	K *Normal*	✓	✓	✗	✗
	U *Secure*	✓	✗	✓	✗
	U *Normal*	✓	✓	✓	✓

TABLE II: Access permission matrix for *TrustZone-M* in ARMv8-M processors. K denotes kernel mode, while U denotes user mode.

Direct *TrustZone-M* applications. ARM *TrustZone-M* extensions to microcontroller hardware enable two *domains* that segregate system software: *Normal* and *Secure*. As shown in Table II, *Normal* runs conventional software and has access only to the subset of resources provided to it by *Secure*. *Secure* has access to all system resources, and can map a subset of interrupt vectors to *Normal*. It provides an *asymmetric* model in which *Secure* has access to all resources in *Normal*.

Some researchers have leveraged the *TrustZone-M* to provide protection domain isolation. For example, virtualization of two operating systems has been attempted on *TrustZone-M*-enabled ARM Cortex-M microcontrollers [12], [13]. One of the guest VMs runs in the *Secure* domain while another one runs in the *Normal* domain. These implementations allow hardware-level predictable interrupt latencies. However, (1) only two VMs are allowed, which is inflexible in the case of microcontroller consolidation that requires multiple VMs be orchestrated together, (2) the *Secure* VM have full access permissions to the memory held by the *Normal* VM, breaking the mutual, strong isolation between the two VMs, and (3) the *Secure* VM's execution may indefinitely preempt or interfere with the *Normal* VM's execution, causing the *Normal* VM to miss its deadlines. On contrary, the *SBI-TZ* infrastructure may host multiple VMs with full *spatial* and *temporal* isolation between them.

Language- and software-based isolation. Many microcontroller systems provide isolation via language safety [14], [15]. These have the benefit that software bugs are confined by software checks based on type-safety which leads to low overheads for many common operations. However, these approaches (1) restrict the programming languages used, and make virtualization of separate code bases difficult, (2) prohibit linking against third-party precompiled libraries, which is required in functionality consolidation, and (3) isolates the applications spatially but not temporally.

Other projects use safe languages on microcontrollers, but focus on programmability instead at the cost of performance and predictability [16], [17], [18], [19]. These software VMs allow setting up protection domains regardless of the underlying processor architecture and hardware features given the application bytecode. However, these approaches incur execution efficiency and I/O interaction overheads due to their interpreted execution, and are unpredictable due to potential freeze-the-world garbage collections. The memory footprints of the language VM themselves also put extra pressure on the scarce microcontroller internal memories. The application must be written in the language that the VM supports, which in the automotive domain may require rewriting legacy C/C++ applications. Aside from these downsides, security vulnerabilities also hamper these solutions due to the large code base of language VMs themselves.

Some software systems use verifiers [20] to statically bound the execution of their code. However, these approaches (1) require statically bounded implementations that might not match application execution patterns, (2) require a verifier and a code generator that have been repetitive sources of security issues, (3) complicate dynamic application updates, and (4) still have a performance impact.

Recently, the WebASseMbly (WASM) [21] has been ported to the microcontroller platform [7]. It is an intermediate bytecode assembly language that decouples the language from the VM, and it allows precompilation from bytecode to native code. The shortcomings of software VMs mentioned above are somewhat mitigated with WASM but still not eliminated.

Interrupts as operating system dispatch. Some researchers (*e.g.* [22]) have leveraged ISRs as operating system thread dispatch, and [8] combines this approach with isolation between the applications. These designs reach low interrupt latencies due to their use of hardware ISRs as thread dispatch, and spatial isolation between tasks is enforced by leveraging Memory Protection Units (MPUs). Though [8] has multiple modes, the one that provides *security* properties runs applications at the user-level, thus we focus on its comparison to this work. In this mode, a MPU reprogramming and a kernel- to user-level transition are required for each interrupt to run it at the user-level, which leads to significant latencies. Additionally, [8] requires protection domain annotations throughout the application source code, and all isolation boundaries must be known at compile-time. In contrast, this work combines bare-metal interrupt latencies with *security*, and allows dynamic protection domain boundaries, which is more flexible.

Hardware multitasking support. Some processors feature hardware task switching facilities *e.g.* task gates in 32-bit x86. A specific instruction or an interrupt may activate one of the task gates, and the hardware will automatically context switch to the target task and its protection domain, saving and restoring all the necessary registers. However, though this mechanism in x86 performs hardware task switches, (1) it is only supported on legacy 32-bit x86 that is generally not receiving feature updates, (2) it has been noted to have even more overhead than software approaches, and (3) it provides neither execution budget accounting nor temporal isolation. This makes them unsuitable for *SBI* implementation. On the contrary, the *TrustZone-M* hardware leveraged in this work switches task context and protection domain with far less overhead than software, and we use prevalent debugging facilities to provide budget accounting. In addition, Intel's forthcoming "Sapphire Rapids" processor (not publicly available yet) also has a user-level interrupt feature. However, to the best of our knowledge, it does not switch protection domains, thus cannot be used to preempt lower-priority execution to service event-triggered execution.

Hardware virtualization extensions. Some recent high-performance microcontrollers [23] feature hardware virtualization extensions. When an interrupt to the currently active VM occurs, it will be directly delivered to the VM's kernel mode, eliminating hypervisor-incurred latency [23]. When the target VM is not currently active, the hypervisor still needs to be invoked to switch to it so that the interrupt can be delivered, hampering predictability. By comparison, this work enables bare-metal interrupt latencies no matter the target VM is currently active or not, and may be implemented in low-power or low-cost microcontrollers.

Multi-core systems and resource partitioning. Significant work is performed on memory scheduling [24], task partitioning [25] and cache partitioning [26], [27], [28], [29], [30] for microprocessor-based embedded systems, and it is not uncommon for current microprocessors to have multiple cores that share a last-level cache [31], [32], [33], [34], [35], [36]. Microcontrollers, on the contrary, are simpler and usually don't have multiple cores. In cases where they do, these cores usually don't share a last-level cache. However, the complexity of microcontroller systems is also increasing, and some microprocessor designs may translate to them *e.g.*future Cortex-R-based multi-core automotive microcontrollers. Multi-core capabilities are orthogonal to the *SBI* mechanism. For example, when the memory allocations are cache-colored, the memory accessed by the *SBI*s may be on dedicated cache sets so that they will not be interfered with by other system components; memory accesses of *SBI*s may also be treated specially by the memory access scheduler so that they preempt other requests; *SBI*s on different CPU cores may be assigned different cache sets as well so that they do not interfere with each other. Additionally, *SBI*s of different protection domains may be partitioned among the cores so that they do not compete for CPU.

Embedded virtualization. Embedded virtualization have been proposed on MMU-based microprocessor systems such as MIPS-VZ and ARM Cortex-A9 [37], [38], and commercialized off-the-shelf products have appeared [39], [40], [41]. These virtualization use-cases allow hosting heterogeneous operating systems on the same microprocessor, or port the same software infrastructure over different core counts [42]. A specific use-case is to run feature-rich operating systems that provide commodity high-level API functions alongside a real-time operating system (RTOS). Provided that the hypervisor delivers interrupts to the RTOS with minimal latency, real-time responsiveness of the legacy functionality is delivered. Some research also leverages virtualization to provide isolation between protection domains [42], [37], [43], and enable function integration from multiple firmware binary sources.

However, this research only addresses high-functionality feature-rich embedded systems that are microprocessor-powered, and doesn't discuss the possibility of microcontroller virtualization. In contrast, the virtualization infrastructure proposed in this work provides bare-metal interrupt latencies without sacrificing CPU and memory efficiency, making virtualization possible for MPU-based microcontrollers that are prevalent in resource-constrained embedded systems.

Task model. The system consists of a set of threads, each assigned a fixed priority. Each thread might execute within an application, a VM, and can execute in the user-level or, via system calls, in the kernel. The scheduler implements preemptive fixed-priority scheduling, always choosing the highest-priority thread for execution. The scheduler programs and receives timer notifications for time-triggered activation and preemption. *SBI*s are also assigned fixed priorities. A *SBI* is activated when the hardware interrupt triggers, and the *SBI* has a higher priority than the currently active execution. Additionally, each *SBI* is associated with a deferrable server [44]. A *SBI*'s budget is expended upon its execution, and the *SBI* is suspended should the budget fall to zero. The budget is periodically replenished to a fixed initial value. The deferrable servers enable the system to limit the interference of *SBI*s on lower-priority computations. We'll focus on a single-core model, but this could be extended to multi-core systems with partitioned scheduling. This research assumes a task model that is intentionally simple, and focuses on ensuring that *SBI*s properly implement the model.

Summary. Comparisons to the discussed systems are shown in Table III. It may be observed that the *SBI-TZ* is capable of accommodating all the applications or requirements listed, while other methods have their respective shortcomings.

III. System Design

A. Fundamental Hardware Requirements for SBIs

The critical hardware features to implement *SBI*s are:

- each ISR is assigned to one of the protection domains,
- the interrupt hardware understands how to switch between the protection domains without software intervention,
- such hardware task switches have far less latency than software equivalents, and
- the interrupt hardware provides means to limit the execution for ISRs, e.g. dedicated count-down timers and interrupts on timer expiration.

These features allow the processor hardware to preempt the current protection domain and switch automatically to a protection domain where a higher-priority interrupt is targeting. After executing the interrupt handler, the processor automatically switches back to the former protection domain and resumes execution there. Note that such switches are performed entirely by the hardware, and no extra software overheads are involved. This allows the ISRs to be untrusted and even

Application	Baremetal OS	Language-based OS [14] Compiler-checked OS [8]	Direct *TrustZone-M* OS [12]	Composite	Composite with *SBI*
Multiple applications	✓	✓	✗	✓	✓
Third-party binaries	✓	✗	✓	✓	✓
Temporal isolation	✗	✓*	✓*	✓	✓
Spatial isolation	✗	✓	✓*	✓	✓
Baremetal-class	✓	✓	✓	✗	✓
Threaded-class	✓	✓	✓	✓	✓

TABLE III: Application latency requirements met. Direct *TrustZone-M* OS means directly leveraging the *TrustZone-M* mechanism and running two VMs; Composite means interrupt notification to user-level threads; Composite/*SBI* means leveraging the *SBI* mechanism and directly placing the critical code into the *SBI*. * means partial support depending on the implementation.

potentially malicious while still experiencing hardware-level latencies, hence the name Secure Bare-metal Interrupts (*SBI*).

Given the minimal underlying hardware support, the **guarantees** of *SBI* are as follows.

G1: Bare-metal latency. All *SBI*s are directly routed via hardware, thus avoiding all software overheads, exhibiting bare-metal latency.

G2: Spatial isolation. All *SBI* execution is spatially isolated from the OS kernel and all protection domains that they do not have access to, so that the *SBI*s cannot read from or write to memory beyond their applications or VM's protection domain.

G3: Temporal isolation. The *SBI* execution is bounded, which means that no *SBI* may execute for longer than system-defined limits and thus hamper system responsiveness. Each protection domain has a *SBI* time budget, and when this budget is expended, the execution of all *SBI*s in that protection domain has to stop until that budget is replenished again.

B. Spatial Isolation

The *SBI*s are untrusted and hence must be spatially isolated from the rest of the system, except for the protection domain they target. As each protection domain corresponds to a dedicated hardware access control register set for restricting its memory and I/O access at the kernel-level, we program these registers much like how we program the *MPU* to restrict the accesses of these *SBI*s. When a *SBI*-enabled protection domain has its memory map changed, we program both the *MPU* and the corresponding kernel-level access control registers. When the protection domain executes at the user level, the *MPU* is responsible for restricting the accesses of its threads; when the protection domain executes at the kernel level, the dedicated access control registers are responsible for restricting the access of its *SBI*s. In this way, we spatially isolate both the user- and kernel-level execution of the protection domain from the rest of the system.

C. Temporal Isolation

Even if the *SBI*s are spatially isolated, they may enter infinite loops either due to software faults or malicious intents and hamper the responsiveness of the system. To prevent this, each *SBI*-enabled protection domain has a budget assigned to all its *SBI*s, and when the *SBI*s execute, the budget expends. As the hardware provides dedicated per-domain timers only programmable by the kernel, we preprogram these timers with the execution budgets of each respective protection domain. As the *SBI*s in the protection domain execute, the corresponding timer counts down and an interrupt targeting the kernel will fire when the budget expends. This allows the kernel to preempt the execution of the *SBI*s and disable further interrupts to the protection domain when the budget is overrun. When more budget is allocated to this protection domain by the scheduler, the kernel will reprogram the timer with this budget and reenable the *SBI*s targeting this domain.

D. SBI-enabled Virtualization Infrastructure

Microcontroller virtualization has been used to run multiple legacy code-base on the same microcontroller, facilitating functionality consolidation. Each VM is provided with the ability to call hypervisor services *e.g.* memory management, I/O management and scheduling. In public designs, the interrupt signals need to pass through the hypervisor before they are sent to the VM, which introduces extra latencies that render many real-time applications difficult to implement.

In this paper, we leverage the *SBI* mechanism to implement *SBI-TZ*, a hardware-accelerated virtualization infrastructure whose VM interrupts are delivered with bare-metal latencies (Figure 2). To achieve this, all timing-critical VM interrupt handlers are implemented with *SBI*s. When the interrupt is triggered, the hardware automatically switches to the accelerated VM and executes its interrupt handlers. Thus, the use of *SBI* makes bare-metal latencies possible for the legacy applications without sacrificing mutual isolation between them.

IV. Implementation

A. SBI-enabled Composite μ-Kernel

The *SBI-TZ* is implemented with the Composite μ-kernel [45] that has a strong security model based on capability-based access control. In Composite, all kernel objects such as threads, communication endpoints and protection domains are only accessed through unforgeable tokens called *capabilities*. These capabilities are shared between protection domains via *delegation*. Each protection domain has a *capability table* that tracks its access to kernel objects. A *component* is synonymous with a protection domain, and includes both a page-table to constrain memory access, and a capability table to constrain kernel object access.

At boot time, a single *constructor* component with access to all system resources (aside from the kernel) is tasked with creating the rest of the components including VMs and real-time applications, and separates resources between them. In *SBI-TZ* that support both native composite applications and *FreeRTOS* VMs, the management components include the scheduler, the memory manager and the I/O manager that manages device and interrupt access. Applications such as VMs use highly optimized synchronous invocations and asynchronous signals to request and handle I/O communication. In Composite, privileged components such as the I/O manager use a *hardware capability* to connect an interrupt vector to a user-level thread that will handle that interrupt. *SBI-TZ* expands the kernel logic for this to enable connecting *SBI*s directly to the corresponding thread, thus enabling direct hardware dispatch to the protection domain. In *SBI-TZ*, the I/O manager is extended to be aware of the *SBI*s, and is responsible for redirecting them to the correct VM. *SBI-TZ* may also be implemented in other systems that provide protection domains on microcontrollers.

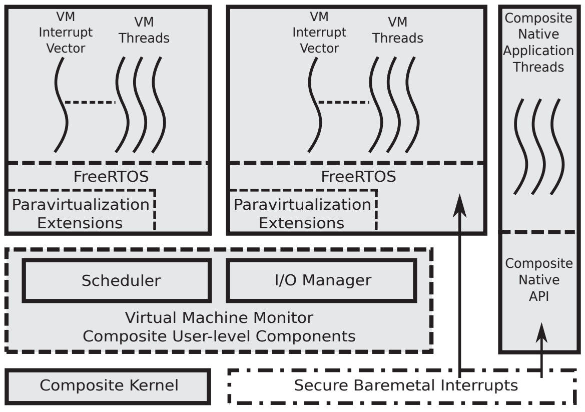

Fig. 2: Virtualization infrastructure. Memory isolation is denoted by the dark solid lines, logical separation is denoted by the dashed lines, and VM monitor is denoted by the dotted lines. Specifically, memory separation by the security extension is denoted by the dark dash-dotted lines, and is in white background.

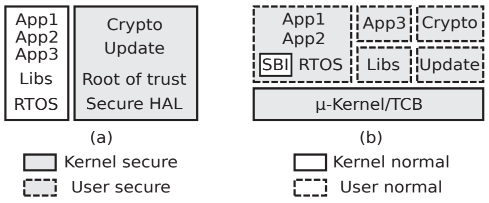

Fig. 3: Comparison between conventional *TrustZone-M* systems (a) and *SBI-TZ* (b). Contrary to the conventional design, the *SBI*s are in *Kernel Normal*, while other software components are in *Secure*. Protection domains are denoted by rectangles. In (b), the Composite kernel is also the Trusted Computing Base (TCB).

B. TrustZone-M Hardware.

TrustZone-M was marketed as a virtualization extension; however, it is in fact a security extension that separates the system resources, including CPU registers, memory and I/O into *Secure* and *Normal* worlds. It is important to note that the *Secure* and *Normal* are orthogonal to the `kernel` ("privileged handler") and `user` ("unprivileged thread") modes, for a combination of four CPU states: *Kernel Secure*, *Kernel Normal*, *User Secure* and *User Normal*. As shown in Table II, CPU running in *Secure* mode may access both *Secure* and

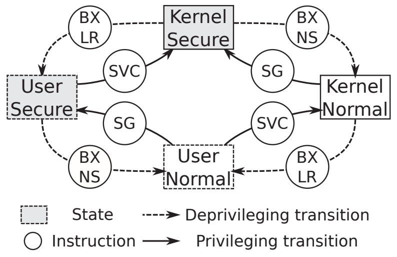

Fig. 4: *TrustZone-M* design by ARM. Its original intention is to isolate the key secure software components from the other software components.

Normal resources, while in *Normal* mode only accesses to *Normal* resources are allowed. The resources include CPU, memory and I/O. Switching from *Secure* to *Normal* needs only a specialized branch instruction (*BXNS*)[1], however switching from *Normal* to *Secure* requires executing a specific secure gate (*SG*)[2] instruction residing in the *Secure* but *Normal*-callable code memory. As shown in Figure 4, the *TrustZone-M* design optimizes for these cases, thus supporting fast switches between *Secure* and *Normal* worlds, and enabling the vectoring of interrupts directly from one state to another.

That said, *TrustZone-M* has notably *asymmetric* access to resources: the *Secure* is trusted to access its own resources *and* that of *Normal*. This makes it challenging to provide strong isolation of code executing in *Normal* from *Secure*. The *TrustZone-M* enables direct transitions from *User Secure* directly into *User Normal* using *BXNS*, thus enabling whatever is running in *Secure* to cause arbitrary control flow in *Normal*. *SBI* prevents this by disabling computation in *User Normal*, and instead enabling isolated VM execution in *User Secure*, and coordination with *SBI*s in *Kernel Normal*. To sum up, we carefully design *SBI-TZ* to take the naturally *asymmetric* trust of *TrustZone-M*, and create a *symmetric* protection domain model that isolates the applications.

C. TrustZone-M as an Interrupt Accelerator

In this particular work, we implement *SBI-TZ* in a *TrustZone-M*-enabled microcontroller. In conventional *TrustZone-M*-based designs, we run the critical small code-bases in *Secure* while the bulk of the operating system and all of its applications execute in *Normal*, which is shown in Figure 3 (a). However, interrupts to be handled by ISRs in the *Normal* RTOS cannot be handled by a *Secure* handler. Doing so would enable the ISR to access not only the *Normal* RTOS, but *also* the *Secure* code, which violates isolation guarantees. Previous work has multiplexed *TrustZone* [46] to share it between multiple VMs; in contrast, to enable selected isolated VMs to reach bare-metal interrupt latencies while isolating all applications from each other, we choose to leverage *TrustZone-M* in an *unconventional* way as shown in Figure 3 (b). The *SBI*s are placed in the *Kernel Normal* while the Composite

[1]Once `BXNS` is executed, the processor will switch to *Normal* and begin execution at the location designated by the `BXNS`'s register operand.

[2]Once `SG` is executed, the processor will switch to *Secure*, and then the instructions immediately following it will be executed.

kernel is placed in *Kernel Secure*; all applications run in the *User Secure* while the *User Normal* is empty. As shown in Figure 3 (b), the *Normal* is *only* used for *SBI* execution. The protection domain for the VM/application that uses *SBI*s spans the *Normal* and *Secure*, but is identical in both. The *TrustZone-M MPU* is banked between *Secure* and *Normal* and may only restrict memory accesses at the user level. Due to the fact that the *Secure MPU* only isolates application execution in the *User Secure* but not *SBI* execution in the *Kernel Normal*, the *TrustZone-M* Security Attribution Unit (SAU) must be programmed to *confine* the *SBI* execution. Also, should any *User Secure* application attempt to maliciously switch to the *SBI*s, *SBI-TZ* guarantees that an exception is delivered to the Composite kernel. This is ensured as the *Normal MPU* is locked into enabling access to no memory at the user-level. As the *Normal MPU* is locked, all *MPU* references follow refer to the *Secure MPU* unless otherwise noted. To sum up, we'll be able to support *SBI*s and multiple mutually mistrusting protection domains *simultaneously*, enabling *both* tight latencies and true multi-tenancy.

Timing-sensitive *SBI*s run at high priorities in the system, and this is necessary because they need to meet tight deadlines. To ensure that *SBI*s cannot monopolize the CPU or prevent other tasks from meeting their deadlines, we rely on hardware facilities to track the cycles executed in *Normal*. Should the *SBI*s execute enough to expend their budget, the *Normal* execution cycle tracking hardware will generate an interrupt. The kernel uses this interrupt to suspend the *SBI*s until a budget replenishment. Unfortunately, *TrustZone-M* alone does not provide such facilities. However, broadly deployed debugging facilities [47] do, thus *SBI-TZ* uses these to ensure temporal isolation.

To summarize how these mechanisms approach the **guarantees** laid out in §III, we have the following findings:

G1: *SBI-TZ SBI*s are executed in *Normal* using *TrustZone-M*'s support for nested interrupt handers. This support enables the preemption of even non-preemptible kernel code in *Secure*, and performs hardware context switches to guarantee bare-metal overheads for interrupt dispatch.

G2: *TrustZone-M* implements a set of SAU access control registers for *Normal* that allow the kernel to preprogram and restrict the access of the *SBI*s, enforcing spatial isolation. The access control registers for *TrustZone-M* provide ample flexibility when it comes to memory range programming, so we may always program it to reflect the same protection domain that the *MPU* is enforcing, placing no restrictions on memory allocations. However, they do not allow setting independent read, write and execute permissions for the memory segments. Additionally, the *TrustZone-M* only supports two worlds on each CPU core. Thus, a single *SBI*-enabled protection domain can execute on each core.

G3: Composite provides a *TCaps* abstraction[3][48] that allows programming execution budgets, and this mechanism is adapted for *SBI*s to place a bound on their execution time, enforcing temporal isolation. Note that the *TrustZone-M* itself does not provide cycle-accurate execution accounting of the *SBI*s. However, we adapt a common debugging facility to enable this tracking. A privileged, programmable counter decreases as the *Normal* world is executing, and when the value reaches zero, an interrupt is activated so that the kernel may stop the execution of the *SBI*s. We use this facility to interrupt *SBI* execution only if it overruns its budget.

To this end, all three guarantees are met in *SBI-TZ* by carefully leveraging the *asymmetric TrustZone-M* to provide hardware-level interrupt latencies, yet strong *symmetrical* spatial and temporal isolation.

D. SBI Integration Overview

We integrate *SBI*s into the Composite system with the **guarantees** in mind.

Spatial isolation. To enforce spatial isolation (**G2**), both the *MPU* and *SAU* hardware must be administered. The *MPU* is responsible for isolating the applications in *User Secure*. The *SBI*s execute in *Kernel Normal* thus the *MPU* is incapable of restricting its memory accesses alone, and we leverage the *SAU* to restrict their memory accesses.

Each address range programmed in the *SAU* are labeled with one of the two security attributes: *Normal* or *Secure* but *Normal*-callable which may contain Security Gate `(SG)` instructions that are callable from *Normal* code. The `(SG)` instruction, when executed, switches the CPU from *Normal* to *Secure*. As such, the *SAU* registers are used to control *Normal*-accessible memory.

In this implementation, we program the *SAU* so that all RAM of the *SBI*-enabled component is marked as *Normal*. The *MPU* and the *SAU* are programmed to cover identical address ranges so that the component is executable from both *Normal* (*SBI*s) and *Secure* (application code). The *SAU* may only be programmed in *Secure*, so that malicious *SBI*s never expand their set of accessible memory, thus enforcing strong spatial isolation between the *SBI* code and the rest of the system.

For *SBI*s, *TrustZone-M*'s Nested Vectored Interrupt Controller (`NVIC`) is enabled to directly deliver interrupts to handlers in *Normal*. In particular, the `NVIC`'s Interrupt Target Non-Secure (`ITNS`) registers are configured so that all the *SBI*s now target *Normal* by hardware. When the *Normal* interrupts fire, all *Secure* CPU registers are saved onto the *Secure* stack then set to zero by hardware, and the CPU switches to the *Kernel Normal* state to execute the *SBI*s. This prevents accidental leakage of *Secure* register information to untrusted *SBI*s.

[3]*TCaps* are time budgets that fit into the capability-based access model. They integrate CPU management into a capability-based access-control system and distribute authority for scheduling. They enable controlled delegation of time between different schedulers, and track budgets associated with execution. *SBI* budgets and accounting are integrated into the *TCaps* abstraction, thus integrating *SBI*s into the timing and access control mechanisms of the system. Please see [48] for details.

Temporal isolation. To enforce temporal isolation (**G3**), we need to guarantee that the *SBI*s cannot execute for longer than system-defined limits, though they may preempt the Composite kernel. To this end, the ARM standard debugging facility registers and its special features are leveraged so that we may (1) track time spent in different worlds – most notably, in *Normal*, thus *SBI* execution, and (2) preempt the *SBI*'s execution with a high-priority interrupt when that time reaches a threshold, even Composite is a non-preemptible kernel.

The *SBI*s are executed at a higher priority than the Composite kernel without any intervention so the kernel may not know when they started execution and how long they executed for. To complicate things further, the *SBI*s may `preempt` even the Composite kernel, and this makes the original *TCaps* support of Composite unfit for enforcing execution budgets of *SBI*s. Thus, a hardware mechanism is needed so that the *Kernel Normal* execution time is accounted for whenever the handlers execute, and the kernel needs to be notified of when the budget depletes so that it can preempt the *SBI*.

In particular, we use the Data Watchpoint and Trace (`DWT`) facility which is broadly deployed in Cortex-M microcontrollers. The `DWT` built into the processor has a 32-bit `DWT_CYCCNT` counter that (1) may be programmed to only account for the number of CPU cycles spent in the *Kernel Normal* state, where the *SBI*s execute.This is done by programming the `CYCDISS` (bit [23]) and `CYCCNTENA` (bit [0]) of the `DWT_CTRL` register. If the `DEMCR` and `DWT_FUNCTIONn` registers are also configured, (2) the Debug Monitor Exception (`DebugMon`) may be fired when the `DWT_CYCCNT` value equals a predefined value in the corresponding `DWT_COMPn`. This gives us an opportunity to fire a *Kernel Secure* interrupt when the *SBI*s overrun their budget, effectively implementing a finite budget server for them.

Composite activates threads in response to interrupts, and executes them using budgets defined by *TCaps*. *TCaps* enable access control for time [48] and are the mechanisms by which time is accounted to, and limit interrupt thread execution. User-level management components (*e.g.* the scheduler) define policies for replenishing these budgets, and we use a deferrable server policy [44] in this work. To interface the software abstractions with *SBI*s, we introduce a new *TCaps* type called *hardware TCaps*. Different from other *TCaps*, it is associated not with any thread but with the *Normal* world. When transferred time to, the *SBI*s will be enabled and the `DWT_COMPn` will be programmed with the budget allotted for their execution. When the budget is expended, the processor will be taken to the `DebugMon_Handler` executing in *Kernel Secure*, where we (1) disable the *SBI*s and (2) modify the preempted *SBI*'s stack so that an immediate return from it will be performed. In this way, the currently executing *SBI* will be suspended and the interrupt stacks will be unwinded correctly, and we're back in the execution context on the *Secure* side. The *SBI*s will be re-enabled by the hypervisor only when more budget is allocated to the *Kernel Normal* vectors, and the `DWT_COMPn` register will be programmed accordingly so that the next budget overrun interrupt will fire when this budget is exhausted. The system scheduler is responsible for replenishing the hardware *TCaps* on a regular basis so that the time allocated manages to service the *SBI*s.

DMA engines. It is necessary to administer the DMA engines securely so that they are not exploited to bypass memory access control. The *MPU* and *SAU* settings only apply to the CPU but not to the DMA engines, and manufacturer-specific extensions are required to confine DMA accesses so that the *SBI*s may directly initiate them. When such manufacturer-specific extensions are not present, the system requires a trusted intermediary that programs the DMA engines securely. Our hardware platform does provide such manufacturer-specific extensions, however we do not enable them for simplicity of the implementation. Instead, we leverage the I/O manager as the trusted intermediary to perform DMA transfers.

E. SBI-TZ Virtualization Overview

SBI-TZ builds upon a public work [49] on microcontroller paravirtualization in *MPU*-enabled Composite. Here we provide an overview of this public work, and detail the changes for *SBI-TZ*.

CPU virtualization. Composite system scheduling facilities that supports user-level scheduling policies are leveraged so that each *FreeRTOS* VM uses its own scheduler. Between different *FreeRTOS* VMs, hierarchical scheduling is applied, and strict temporal isolation is enforced by Composite's *TCaps* support so that the VMs don't interfere with each other [49]. In this particular implementation, the root scheduler uses a fixed-priority round-robin scheduling strategy, as does *FreeRTOS*.

Memory virtualization. In *MPU*-enabled Composite, each VM is associated with a Path-Compressed radix Trie (*PCTrie*) at creation time. The *PCTrie* is responsible for holding all the descriptors of memory regions that it has access to [49]. The memory manager is responsible for translating VM memory allocation and deallocation requests into *PCTrie* operations, which finally updates the *MPU* register[4] content and enforces spatial isolation.

I/O virtualization. To enforce strong isolation, all I/O operations, as well as interrupts, are virtualized so each VM is provided with a restricted set of hardware that it may interface. In Composite microkernel, I/O virtualization on *TrustZone-M* devices is much like memory management due to the fact that all devices are memory-mapped. The devices monopolized by one VM only are directly mapped to their address space, enabling zero-overhead access to device data. Nevertheless, some devices may be shared between different VMs. In that case, each VM makes requests via protected component invocation to the I/O manager to operate the device.

Accelerated interrupt handling and passing. We've discussed how the *TrustZone-M* is leveraged to enable *SBI*

[4]Region Number Register (RNR), Region Base Address Register (RBAR) and Region Attribute and Size Register (RASR).

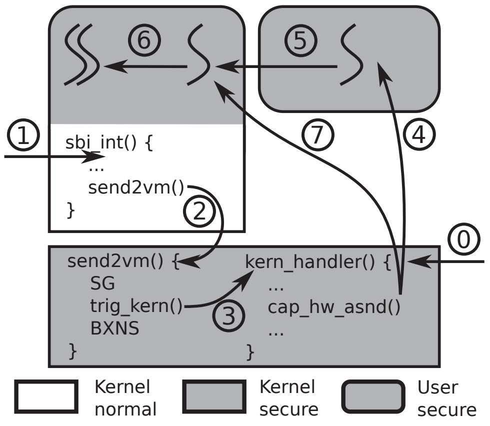

Fig. 5: Accelerated interrupt delivery. Memory isolation is denoted by the dark solid lines, execution resulting from interrupts is denoted with arrows. Numbers denote the step sequence. Rounded rectangles denote *User* and rectangles denote *Kernel*. Accelerated *SBI*s start from step 1, while non-accelerated interrupts start from step 0.

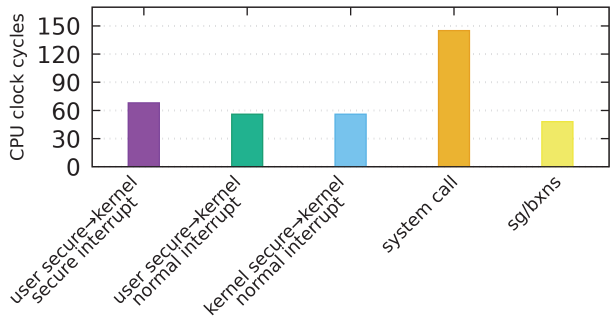

Fig. 6: Hardware overheads. `user secure→kernel secure` is the interrupt latency from *User Secure* to *Kernel Secure*, `user secure→kernel normal` is the interrupt the latency from *User Secure* to *Kernel Normal*, `kernel secure→kernel normal` is the interrupt latency from *Kernel Secure* to *Kernel Normal*, `system call` is system call overhead including the system call stub register pushes and pops, and `sg/bxns` is the total time to perform security gate transition from *Kernel Normal* to *Kernel Secure* and returning from *Kernel Secure* to *Kernel Normal*.

execution. However, it is common in RTOSes for ISRs to activate threads that further process the device's data. In this case, *SBI-TZ* must provide means to activate the application's threads (in *Secure*) from the *SBI*. First, we use the *TrustZone-M* facilities for efficiently communicating from *Normal* to *Secure* to invoke the Composite kernel, which then routes the notification using Composite activates to the application threads in *Secure*.

As shown in Figure 5, when the *SBI* gets activated ①, it will begin execution via direct activation from hardware. After handling the activation, the *SBI* may need to notify a thread in a VM of this event. A small veneer function beginning with `SG` is linked into a *Normal*-callable *Kernel Secure* address, and is responsible for triggering a dedicated interrupt routine in the Composite kernel.

When the *SBI* wants to notify the underlying accelerated VM, it executes the veneer ② where an interrupt in the *Kernel Secure*-side Composite kernel is triggered after ③. The *SBI* then finishes its execution, and the *Secure* side resumes its execution. If the *Secure* side was not executing in the kernel, the *Kernel Secure* interrupt now activates and notify (`asnd`) ④ the I/O manager which will notify the VM as well ⑤. After the VM internal handler is notified, the thread running in the VM will be notified subsequently by a `xQueueSendFromISR()` ⑥. The kernel may also notify the VM internal handler directly where only one VM needs to be notified ⑦, bypassing the I/O manager. If the *Secure* side was executing in kernel vector, upon exiting of the kernel, the *Kernel Secure* interrupt vector will be activated, performing the notification.

V. Evaluation

Hardware configurations. For the evaluation, we use an ARM Cortex-M33 microcontroller running at 150MHz (LPC55S69JBD100). Since this processor has a low clock frequency, no cache is implemented in the CPU.

Software configurations. We run the Composite system with the supporting components including the I/O manager and system scheduler. Due to memory footprint constraints, we only evaluate the overheads of a single VM as the rest of the memory is required to store measurement results. We use *FreeRTOS* version 9.0.0, and the `gcc` compiler version 9.2.1, with the -O3 optimization flag in all cases.

Measurement criteria. For interrupt latency measurements, the time between the triggering of the interrupt and the activation of the corresponding ISR or handler thread is measured. For *round-trip* context switching measurements, the time between switching to another thread and switching back to the original thread is measured. For communication measurements, the time between the sending of the sender and receiving of the receiver is measured.

All measurements are repeated 10000 times; the average and maximum values are calculated. All bar graphs in this section depict the average (the bottom darker bar) and maximum (the lighter top bar) measurements.

A. Microbenchmarks

Hardware overheads. Many system operations require interaction with the hardware features such as privilege modes, security gates, and interrupts. Three types of interrupts are leveraged in *SBI-TZ*: *User Secure*-to-*Kernel Secure* (for regular interrupts), *User Secure*-to-*Kernel Normal* (for *SBI*s when the system was executing the components), and *Kernel Secure*-to-*Kernel Normal* (for *SBI*s when the system was executing the kernel). To understand the operating system, virtualization and *SBI* overheads, we first investigate the hardware overheads for the relevant operations. These provide *upper bounds* on the performance of the software abstractions that use them. Figure 6 includes the hardware overheads for privilege mode transitions, security state transitions, and interrupts.

We measure the bare-metal latency for an interrupt. Such latency is defined as the time that the interrupt gets triggered to the time that the first instruction in the ISR gets executed. In contrast, system call overhead is measured using the handler in Composite which is modified to return immediately, and accounts for both system call vector entering & returning, system call routing logic, and the saving & restoring of additional registers.

The costs of security transitions involve hardware operations on the register sets and redirection of the control flow.

Discussion. These results show that the performance of the hardware is not prohibitive for real-time applications, especially the bare-metal interrupt overheads. When comparing the different `int` latencies, we observe that they are all within 70 cycles. When we compare the `SG` and `BXNS` overheads with interrupt latencies, we see that these overheads are significantly smaller because they don't trigger exceptions. Also, notice that all system overheads are fairly deterministic, and the maximum value deviates little, if any, from the average value.

Operating system operation overheads. To investigate the cost of various primitive system abstractions in Composite and *FreeRTOS*, we compare: (1) Composite component execution with the user-level scheduling and kernel bypassing library [9], and (2) *FreeRTOS* which has *no protection facilities* thus represents the overhead of a state-of-the-art lightweight RTOS. These are depicted in Figure 7. For both systems, the core operations include interrupt handling, thread context switching, and inter-thread communication or IPC. For Composite, both `intra`- and `inter`- component values are shown. As *FreeRTOS* does not provide protection domains, only `intra` values are shown.

Both systems divide interrupt handling into "top half" ISR execution, and "bottom half" handler thread context execution. We define bare-metal interrupt latency as the time measured between the interrupt firing and the start of the ISR execution, and define thread interrupt latency as the time measured between the end of the ISR execution and the start of the handler thread context execution. For both systems, we activate an asynchronous send mechanism at the end of the ISR and make the handler thread block in the receiving end of that mechanism. For Composite, this is done by using the `asnd` and `rcv` pair, while for *FreeRTOS* this is accomplished with the `xQueueSendFromISR()` and `xQueueReceive()`.

Discussion. In the context switch and message passing case, *FreeRTOS* is the winning system that gives definitive lower bounds on the overheads of these operations. However, this comes at a price of no isolation. For Composite, many kernel operations are bypassed when possible thanks to the user-level scheduling mechanism, especially in cases where no protection domain switches are necessary (`intra`). When protection domain switches are included in the overhead (`inter`), the user-level scheduling mechanism no longer bypasses the kernel and extra overheads of kernel component switches are imposed. Despite this, the Composite `intra` overheads are generally on par with the *FreeRTOS* despite providing protection, and this refutes the performance argument to go without protection domains.

When comparing the bare-metal interrupt latencies, we observe that the Composite's is almost identical to the *FreeRTOS*'s, meaning that they are all fit for applications that require tight interrupt latencies. Despite this, the Composite provides protection domains for *SBI*s, whereas the *FreeRTOS* does not provide any protection domains; this refutes the interrupt latency argument to go without protection domains.

Interestingly, Composite's synchronous communication mechanism (synchronous invocation, `sinv`) is much faster than *FreeRTOS*'s queue mechanism despite extra protection domain switching overheads. This is because Composite is heavily optimized for inter-component synchronous communications, leveraging the *thread migration* mechanism.

To better understand the impact of these metrics for real systems, we discuss a few applications and their interrupt latency requirements listed in Table I. These applications are representatives from multiple domains *e.g.* consumer electronics, automotive electronics, precision instruments and general industry control. The maximum latencies are typical requirements for these applications and are intended to provide a general impression of what interrupt latencies they expect.

Discussion. From the Table I we can see that the applications are divided into two categories. There are (1) threaded-class applications *e.g.* SPI emulation that only require responsiveness at or above a millisecond level, and there are (2) baremetal-class applications *e.g.* motor control that require tight interrupt latencies that are typically in the 0.5-10 microsecond range. The bare-metal OSes (*e.g. FreeRTOS*) fulfills the (1) threaded-class by invoking handler threads from the ISRs using the queue communication (`xQueueSendFromISR()` and `xQueueReceive()`) mechanism. As they are designed without security considerations altogether, the (2) baremetal-class is fulfilled by placing the application code directly into the ISRs. However, vanilla Composite may only invoke handler threads from the ISRs, because placing the application code into the kernel-level ISRs gives them system privileges so that they may break the isolation between protection domains. This means that the vanilla Composite will be unable to run applications with tight latency requirements, or will be forced to run them without protection domains, which cancels the most important benefit of Composite. With the addition of *SBI* to Composite, protection domains are now in place around these ISRs as well, which isolates them from the rest of the system while still providing bare-metal responsiveness. This enables the Composite to support the (2) baremetal-class applications as well.

This is important as the system designer may now run multiple protection domains (*e.g.* applications and VMs) while still enjoying bare-metal interrupt latencies without additional kernel indirection.

Other comparable technologies that aim to provide multi-tenancy either lacks the ability to link against third-party binaries(language-based OS [14] and compiler-checked OS [8]), or are unable to support more than two protection domains altogether (direct *TrustZone-M* OS [12]), and some of their isolation properties depend on the detailed system im-

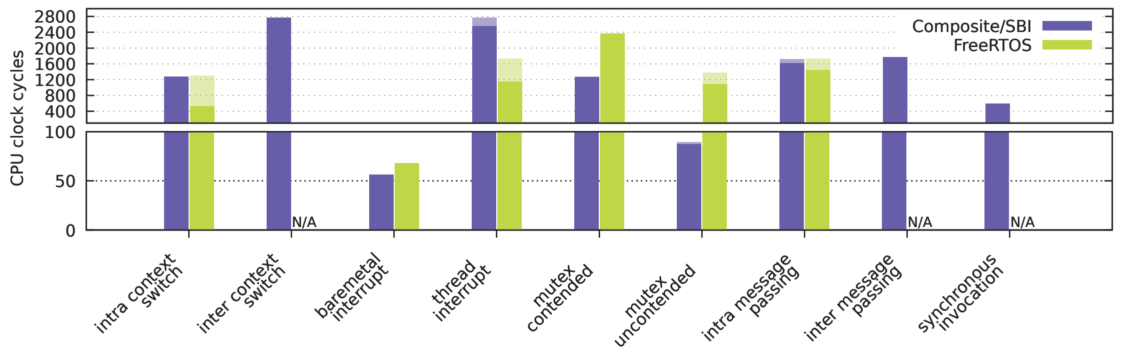

Fig. 7: Overheads of primary system operations, in Composite and *FreeRTOS*. `context switch` is the *round-trip* context switch time, `baremetal interrupt` is the bare-metal interrupt latency for *FreeRTOS* and *SBI* latency for Composite, `thread interrupt` is the interrupt latency to notify threads, `mutex contended` is the contended mutex latency, `mutex uncontended` is the uncontended mutex acquire/release overhead, `message passing` is the message-passing time. For *FreeRTOS*, mutex operation is through semaphore APIs (`xSemaphoreTake()` and `xSemaphoreGive()`), message-passing is through queue APIs (`xQueueSend()` and `xQueueReceive()`); for Composite, mutex operation is through `crt_lock_take()` and `crt_lock_release()`, message-passing is through `crt_chan_send()` and `crt_chan_rcv()`. For Composite, we list both inter- and intra- component values, and the overhead of the synchronous communication mechanisms (`synchronous invocation`).

plementation. For example, [8] is unable to provide temporal isolation, and [12] is unable to prevent the *Secure* VM from accessing the *Normal* VM.

B. Virtualization Acceleration Evaluation

SBI **performance.** To demonstrate the effectiveness of *SBI* in functionality consolidation, we evaluate the VM operation overheads and interrupt latencies with and without *SBI*. We measure in particular the latency of each step in delivering the interrupt, for both the *SBI*-enabled VM and the normal VM, to better understand the detailed implications of the *SBI* mechanism.

As shown in Figure 9, in terms of top-half interrupt latency, the *SBI*-enabled VM far outperforms the normal VM by orders of magnitude and is approaching bare-metal measurements, and this validates our design goal to accelerate the interrupts to hardware speeds. When considering the bottom-half performance, the *SBI*-enabled VM is slightly more sluggish than the normal VM due to its use of an extra trampoline to notify the kernel.

Discussion. Firstly, as shown in Figure 8, the context switch (`context switch`), mutex (`mutex contended`) and queue (`message passing`) operations generally exhibit overheads of more than 1500 cycles, which will have to be avoided when the applications require tight latencies under one microsecond. The `mutex uncontended` is an exception, because it does not involve substantial scheduler operations. Compared to *FreeRTOS* measurements in Figure 7, these operations are more expensive, as is expected from the paravirtualization approach which provides the additional benefit of increased isolation for legacy code-bases.

Secondly, the results on interrupt latencies confirm that the critical interrupt vectors should be hardware-delivered to minimize the latency (**G1**). As shown in Figure 9, compared to the normal VM, the *SBI* mechanism dramatically decreases the top-half latency for those *SBI*-enabled VMs. This is due to the normal VM's need to propagate the interrupt signal between too many agents: from the interrupt source hardware

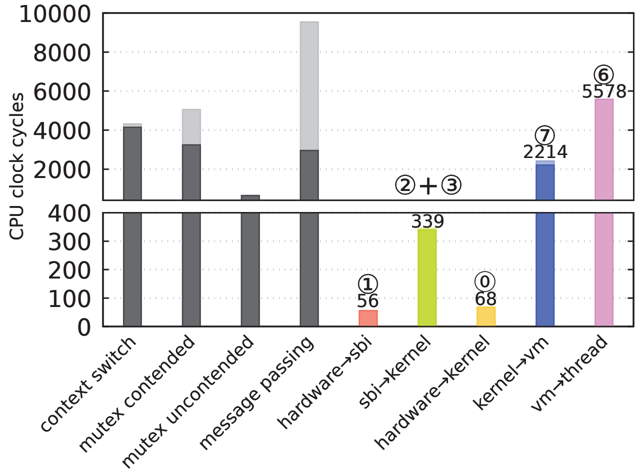

Fig. 8: Overheads of primary system operations in paravirtualized *FreeRTOS* with and without *SBI*. `context switch` is the *round-trip* context switch time for both VMs, `mutex contended` is the contended mutex latency for both VMs, `mutex uncontended` is the uncontended mutex acquire/release overhead for both VMs, and `message passing` is the *FreeRTOS* queue send/receive time for both VMs. `hardware→sbi` is the time from interrupt firing to *SBI* for the *SBI*-enabled VM, `sbi→kernel` is the time from *SBI* to the kernel trampoline for the *SBI*-enabled VM, `hardware→kernel` is the time from interrupt firing to the kernel ISR for the normal VM. `kernel→vm` is the time from Composite ISR to *FreeRTOS* interrupt vector inside the VM for both VMs, `vm→thread` is the time from *FreeRTOS* interrupt vector to the *FreeRTOS* receiving thread for both VMs. See Figure 5 for circled number meanings.

to the kernel (`hardware→kernel`), and then to the VM handler thread (`kernel→vm`). This involves the kernel code that does asynchronous communication, scheduling primitive maintenance, and context switches, and here we're assuming that the I/O manager is bypassed thanks to the Composite's *TCaps* mechanism. If the I/O manager is not bypassed, more indirections will occur and hence more overhead will be imposed. As shown in Table III, the *SBI*-enabled VM is capable of accommodating all the applications listed, as long as the hardware still catches up.

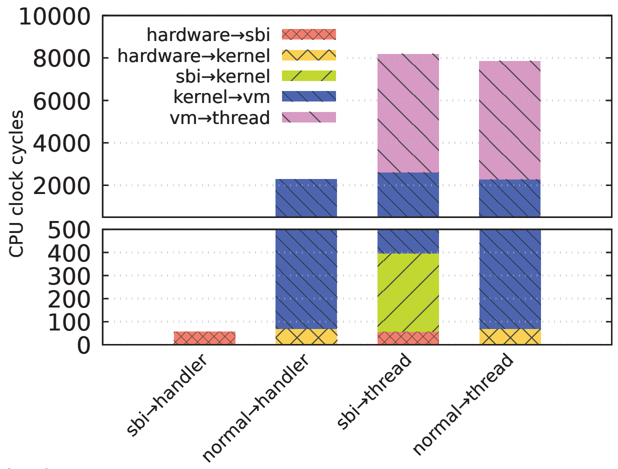

Fig. 9: Interrupt latency comparisons between paravirtualized *FreeRTOS* with and without *SBI*. `sbi→handler` and `normal→handler` represent the respective total latency from interrupt firing to the top-half isolated vector handler, while `sbi→thread` and `normal→thread` represent the total latency from interrupt firing to the bottom-half application thread.

However, for bottom-half latency, that is, the time to deliver the interrupt to a *FreeRTOS*-aware thread, the *SBI*-enabled VM (`sbi→thread`) takes slightly longer than the normal VM (`normal→thread`). This is due to its use of trampolines (`sbi→kernel`) to notify the kernel. While this may sound like a trade-off between top-half and bottom-half latencies, we argue that the top-half computation is only performed when we have very tight jitter/latency requirements. Bottom half computation, on the other hand, is much more common for processing pipelines that have less strict latency bounds, where the normal thread activation overhead is acceptable and *SBI* mechanism will not be used. It is only in cases where both the top-half and bottom-half are run and the latency of both is critical will this matter. Even so, the additional trampoline overhead is around 300 cycles, which is not prohibitive in many application cases.

C. Discussion of Additional SBI-TZ Properties

***SBI* and system security.** Despite the fact that *Secure* is trusted to access its own resources *and* that of *Normal*, we have been able to maintain both *spatial* and *temporal* isolation between different protection domains. The spatial isolation of *SBI*s is achieved by placing them in *Kernel Normal*, where they can access neither the kernel nor other protection domains in *Secure* (enforced by the SAU). The spatial isolation between protection domains and the kernel in *Secure* is achieved by programming the *Secure*-side MPU. Strong memory isolation properties between VMs, services and applications, are provided and orchestrated by the capability-based access control mechanisms of the Composite microkernel [49]. The temporal isolation of *SBI*s is achieved by using broadly deployed debugging facilities that have a budget consumption-triggered interrupt that is integrated into the temporal capability abstraction of the Composite microkernel [48]. This guarantees that the *SBI* execution cannot cause unbounded interference on lower priority threads and *SBI*s.

***SBI* implementation complexity.** Paravirtualization relies on modifications to the lowest levels of the virtualized OS which we wish to modify as little as possible to maximize compatibility with legacy codebases. The paravirtualization infrastructure itself requires 363 Source Lines of Code (SLOC) modifications to *FreeRTOS* to virtualize it. Compared to this already small effort, the *SBI* complexity is negligible as it only requires the programmer to configure the hardware at boot-time to set the correct vectors to target *Kernel Normal*. No modifications of the interrupt vectors themselves are necessary except that they must be linked to the *Normal* code memory.

***SBI* memory footprint.** In the experiment, we focus on a single VM as we *devote a large amount of memory to measurement logging for evaluation*. Nevertheless, the multi-VM support from the system in [49] is maintained. Figure 9 in [49] demonstrates the overheads for switching between VMs, and performing IPC between them. The memory footprint overhead for *SBI*s is shown in the Table IV, and they are not prohibitive for many cases.

Class	Kernel		VM		Native	
	ROM	RAM	ROM	RAM	ROM	RAM
SBI-enabled	94397	33044	14260	13508	12494	12424
SBI-disabled	93137	33044	13852	12484	12094	11400

TABLE IV: Memory footprints for each software module in the system. The numbers are in bytes.

The memory overhead of *SBI*s is small compared to the rest of the system. A native Composite application that isn't memory-optimized increases by only 400 bytes SRAM, and 1024 bytes ROM.

***SBI* and multi-core.** As discussed in §I, §III and §IV, the current *SBI-TZ* implementation only supports one *SBI*-enabled VM per *TrustZone-M*-enabled CPU core. If future microcontrollers feature more than one *TrustZone-M*-enabled CPU core, the *TrustZone-M* facility of each core may be independently programmed. This will allow multiple *SBI*-enabled VMs (up to the number of CPU cores), where the *SBI*s of each VM are handled by a different core. Note that even on single-core microcontrollers that only allow a single *SBI*-enabled VM, the *SBI*-enabled VM may have more than one *SBI* vector. Also, there's no limitation beyond memory for running *SBI*-disabled VMs for consolidation [49]. When the VMs can tolerate 20us interrupt latency, running them as SBI-disabled VMs is sufficient, as in [49].

VI. Conclusions

This paper introduces *SBI*, an infrastructure to provide bare-metal interrupt latencies for protection domains while still maintaining strong isolation along all of the CPU, memory, and I/O dimensions. Furthermore, *SBI-TZ*, a microcontroller virtualization infrastructure with tight interrupt latencies is built upon it.

It is shown that, by leveraging hardware mechanisms to deliver interrupts, a low-performance microcontroller will be able to afford the protection domain overheads while providing interrupt latencies only rivaled by bare-metal systems.

References

[1] "V-USB. A software-only implementation of a low-speed USB device for Atmel AVR controllers, https://obdev.at/products/vusb/index.html, retrieved 12/14/20," 2020.

[2] M. I. Products, "Application note 126. 1-wire communication through software." 2002.

[3] Z. Cheng, R. West, and Y. Ye, "Building real-time embedded applications on qduinomc: A web-connected 3d printer case study," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[4] A. Devices, "8-/6-/4-channel das with 16-bit, bipolar input, simultaneous sampling adc." 2010.

[5] A. Farrukh and R. West, "smartflight: An environmentally-aware adaptive real-time flight management system," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[6] F. Paci, D. Brunelli, and L. Benini, "Lightweight IO virtualization on MPU enabled microcontrollers," in *Proceedings of the Embedded Operating Systems Workshop co-located with the Embedded Systems Week (ESWEEK)*, 2016.

[7] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical software fault isolation for reliable embedded devices," in *EMSOFT*, 2020.

[8] D. Danner, R. Muller, W. Schröder-Preikschat, W. Hofer, and D. Lohmann, "SAFER SLOTH: efficient, hardware-tailored memory protection," in *RTAS*, 2014.

[9] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: OS support for near zero-cost, configurable scheduling," in *RTAS*. IEEE, 2020, pp. 160–173.

[10] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt scheduling with low overhead for real-time kernels," in *RTCSA*, 2006.

[11] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional PC hardware," in *RTAS*, 2006.

[12] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-Enabled Microcontrollers? Voilà!" in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[13] H. M. E. Araújo, "Ltzvisor: a lightweight trustzone-assisted hypervisor for low-end arm devices," Ph.D. dissertation, Universidade do Minho, 2018.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *PLDI*, 2003.

[15] L. Amit, C. Bradford, G. Branden, G. Daniel, P. Pat, D. Prabal, and L. Philip, "Multiprogramming a 64 kB computer safely and efficiently," in *SOSP*, 2017.

[16] "MicroPython for microcontrollers: http://www.micropython.org, retrieved 10/6/17."

[17] "eLua project: http://www.eluaproject.net, retrieved 10/6/17."

[18] "MicroEJ for microcontrollers: http://www.microej.com, retrieved 10/6/17."

[19] "mJS embedded javascript engine for C/C++: http://github.com/cesanta/mjs, retrieved 10/6/17."

[20] J. Corbet, "Bpf: the universal in-kernel virtual machine," *Linux Weekly News, Eklektix Inc*, 2014.

[21] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web Up to Speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '17, 2017.

[22] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as interrupts," in *2009 30th IEEE Real-Time Systems Symposium*. IEEE, 2009, pp. 204–213.

[23] S. Otani, N. Otsuki, Y. Suzuki, N. Okumura, S. Maeda, T. Yanagita, T. Koike, Y. Shimazaki, M. Ito, M. Uemura *et al.*, "A 28nm 600mhz automotive flash microcontroller with virtualization-assisted processor for next-generation automotive architecture complying with iso26262 asil-d," in *2019 IEEE International Solid-State Circuits Conference (ISSCC)*, 2019.

[24] D. Hoornaert, S. Roozkhosh, and R. Mancuso, "A memory scheduling infrastructure for multi-core systems with re-programmable logic," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, 2021.

[25] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, 2020.

[26] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous mpsoc platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[27] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Des. Autom. Electron. Syst.*, 2016.

[28] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, 2004.

[29] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006.

[30] R. Pan and G. Parmer, "MxU: Towards predictable, flexible, and efficient memory access control for the secure iot," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[31] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *ECRTS*, 2013.

[32] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS*, 2013.

[33] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *EuroSys*, 2009.

[34] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson, "Cache sharing and isolation tradeoffs in multicore mixed-criticality systems," in *RTSS*, 2015.

[35] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS*, 2013.

[36] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *RTAS*, 2019.

[37] C. Moratelli, S. Johann, M. Neves, and F. Hessel, "Embedded virtualization for the design of secure iot applications," in *2016 International Symposium on Rapid System Prototyping (RSP)*, 2016.

[38] A. Iqbal, N. Sadeque, and R. I. Mutia, "An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems," *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 2009.

[39] K. Sandström, A. Vulgarakis, M. Lindgren, and T. Nolte, "Virtualization technologies in embedded real-time systems," in *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013.

[40] D. Rossier, "Embeddedxen: A revisited architecture of the xen hypervisor to support arm-based embedded virtualization," *White paper, Switzerland*, 2012.

[41] R. Mijat and A. Nightingale, "Virtualization is coming to a platform near you," *ARM white paper*, 2011.

[42] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008.

[43] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014.

[44] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[45] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[46] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[47] A. Limited, "ARM v8-M architecture reference manual." 2015.

[48] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *RTSS*, 2017.

[49] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *RTAS*, 2018.