FVM: Practical Feather-Weight Virtualization on Commodity Microcontrollers

Junchao Li*, Runsheng Hou*, Guangyong Shang,
Huanle Zhang, Xiuzhen Cheng, *Fellow, IEEE*, and Runyu Pan

***Abstract*—Recently, there has been an increasing drive to consolidate multiple microcontrollers into one physical entity, due to advantages in reducing overall costs, enhancing reliability, and simplifying hardware interconnections. To reduce consolidation engineering costs, minimizing system latency and memory footprint is important as well as maintaining compatibility with legacy software. In this paper, we propose a virtualization-based solution called Feather-weight Virtual Machine (*FVM*) that focuses on these goals. *FVM* enables low latency by specializing the virtualization model to Real-Time Operating Systems (RTOSes), achieves small footprint by adapting management policies to microcontroller memories, attains high compatibility by aligning with microcontroller ecosystem idiosyncrasies, finally allowing practical consolidation across a wide range of commodity microcontrollers. We implement and evaluate *FVM* on ARMv6-M, ARMv7-M, and RISC-V architectures with two toolchains and two RTOSes, and it can fit into 20 KiB of RAM with less than 5% latency bloat.**

***Keywords*-Microcontroller consolidation, Virtualization, Microkernel**

I. Introduction

MICROCONTROLLERS are minicomputers that pack all essential hardware – including a CPU, memory, and peripherals – onto a single chip that costs less than $10. They are ubiquitous in automotive, industrial, and consumer products, where they bring intelligence into machines. Typically designed to perform specific tasks, microcontrollers run firmware as a single binary image that boots in bare-metal mode, lacking runtime protection domain isolation. Resource scarcity is often used as a scapegoat for this lack of isolation, as they face stringent restrictions in terms of Size, Weight, Power, and Cost (SWaP-C).

As product complexity increases, more microcontrollers are needed, and they are interconnected through communication interfaces *i.e.* CAN, SPI and IIC. However, the growing complexity of interconnections has led to cost and reliability challenges [1], and the need to consolidate microcontrollers has taken off [2].

Junchao Li, Runsheng Hou, Guangyong Shang, Huanle Zhang, Xiuzhen Cheng and Runyu Pan are with the School of Computer Science and Technology at Shandong University, China. Guangyong Shang is additionally with the Inspur Yunzhou Industrial Internet Co., LTD. (e-mail: {junchaoli, rhou}@mail.sdu.edu.cn, {dtczhang, xzcheng, rypan}@sdu.edu.cn, shangguangyong@inspur.com). Runyu Pan is the corresponding author.

* these authors contributed equally to this work.

This work was partially supported by National Key Research and Development Program of China (Grant No.2022YFB4502001), National Natural Science Foundation of China (Grant No. 62402291, 62302265, U23A20332), and Shandong Province Natural Science Foundation (Grant No. ZR2023QF172, 2024HWYQ-020).

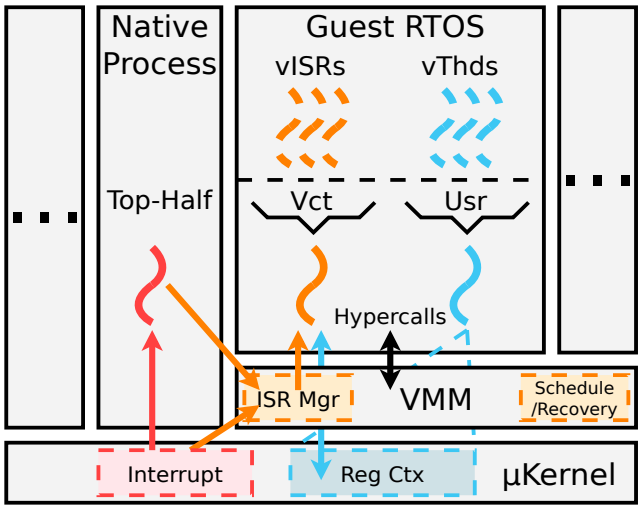

Fig. 1: *FVM* overview. The dashed curves represent guest VM threads (user-level threads) while the solid curves represent microkernel threads (kernel-level threads). The red and orange arrows indicate the fast- and regular- path for interrupt delivery respectively, and the blue double arrows represent that the VCT can modify USR thread kernel-level context directly without microkernel system calls.

The mainstream approach in microcontroller consolidation is to rewrite all legacy firmware and developing an integrated bare-metal image. However, this approach faces three difficulties. First, rewriting legacy code requires substantial effort and forfeits previous certifications. Second, the legacy firmware may rely on incompatible RTOSes or frameworks that are challenging to coexist. Last, the integrated bare-metal image often lacks internal isolation, which can lead to fault or compromise propagation [3], [4].

In the cloud industry, virtualization is broadly employed to perform consolidation without breaking legacy compatibility and isolation [5], [6]. In the microcontroller arena, virtualization is also being explored [7], [8], where each Virtual Machine (VM) corresponds to a virtual microcontroller. Still, several drawbacks prevent their practical use:

- **High execution overheads.** They bloat the interrupt latency and context switch latency by some 5x to 10x, which is unsuitable for many control applications that require tight response times.
- **Large memory footprint.** They require hundreds of kilobytes of memory to use, and cannot be used on mainstream commodity microcontrollers that house some 64KiB of RAM.
- **Specific hardware requirements.** Some require specific hardware that only appears on a certain architecture to

achieve performant isolation. This restricts their applicability to a broader range of hardware [9].

- **Incompatibility with existing ecosystem.** They focus on open-source toolchains which allow flexible pipeline customization, and lack compatibility with industry-standard toolchains that only generate bare-metal images in the end, or the microcontroller ecosystem in a broader sense.

Due to the shortcomings above, many software fault isolation alternatives have been proposed. Yet, when it comes to consolidation, there are some downsides that make them less attractive than virtualization:

- **Requiring access to source code.** They require a single party to hold access to the entire codebase, and cannot cope with situations where source code is held by mutually distrusting third parties.
- **Inability to link against unverified binaries.** They cannot cope with situations where native binaries must be linked against, hence Over-The-Air (OTA) upgrades poses great challenges: it is inherently impossible to check the integrity of the binary updates at development time.
- **High run-time overheads.** Some of them rely on interpretation and run-time bounds-checking which bloats execution time and memory footprint by many times. This could be mitigated but *not prevented* by ahead-of-time compilation.

Other weaknesses are also pronounced, such as fixations on certain customizable toolchains (often LLVM in this case) [10], [11], [12], incompatible toolchain modifications [10], [13], [12], requiring comment markups throughout the entire codebase [14], or potential incompatibilities with software originally written in other languages [15], [16].

In a word, these alternatives have many weaknesses that make them unsuitable for carrying out microcontroller consolidation. This paper introduces *FVM*, a lightweight type II virtualization framework for microcontroller consolidation. As Fig. 1 illustrates, *FVM* integrates VMs, native processes, a hypervisor and a microkernel. Specifically, *FVM* (1) minimizes latency by specializing the virtualization model for RTOSes, (2) reduces footprint by adapting memory management policies to microcontroller memory constraints, (3) maximizes compatibility by aligning with the idiosyncrasies of the microcontroller ecosystem, and (4) enables practical consolidation across a wide range of commodity microcontrollers.

Contributions. The primary contribution of this paper is a demonstration showing that virtualization can be effectively applied for consolidation purposes, even on extremely resource-constrained commodity microcontrollers.

The paper offers the following contributions:

- **we detail the *FVM* design** (§III) that enables state-of-the-art high-performance, low-footprint practical virtualization-based consolidation on commodity microcontrollers;
- **we perform *FVM* implementations** (§IV) on four commodity microcontrollers, three popular architectures, two open-source RTOSes and two industry-standard toolchains, thereby showcasing its wide compatibility and portability;
- **we perform a comprehensive *FVM* evaluation** (§V) across multiple hardware/software combinations, demonstrating its low overheads (as little as 5% execution time) and small footprint (as low as 20KiB).

To the best of our knowledge, *FVM* is the first framework to deliver practical performance while ensuring reliability, security, and compatibility across a wide range of commodity microcontroller consolidation scenarios.

II. Background and Related Work

A. Virtualization

Virtualization allowing multiple pieces of bare-metal software to run in VMs and pool resources on a single physical machine. We call the physical machine "host" and the VMs "guests", and each guest may in turn harness its own processes.

Two major virtualization approaches exist: full virtualization, where the guest OSes run unmodified on the hardware which must be classically virtualizable, and paravirtualization, which modifies the guest OSes to cope with the hypervisor interface but makes no assumptions about the underlying hardware. The virtualization architecture also has two categories: type I, where the hypervisor runs directly on the bare-metal hardware, and type II, where the hypervisor runs alongside a host OS that provides basic services such as drivers, memory management, and scheduling.

Virtualization was initially proposed in the 1960s to support legacy software for mainframes [17], [18]. In modern scenarios, it is often leveraged to perform consolidation while maintaining compatibility and isolation [19], [20], [21], particularly in cloud computing where maximizing hardware utilization is paramount [22].

In recent years, research pivoted towards real-time embedded systems as well [23], [24], [25], [26], [27], [28], [29], [30], [5], [6], [31]. These works mainly focused on embedded microprocessors and generally overlooked microcontrollers. Nevertheless, many of their contributions also apply to microcontrollers and is orthogonal to our work.

B. Memory Protection Facilities

Modern microcontrollers provide memory isolation through Memory Protection Units (MPUs). Unlike Memory Management Units (MMUs), MPUs cannot translate virtual addresses to physical addresses, and only allow dividing the physical address space into regions with varying access permissions. The base, size, and access permissions of the regions are described by a MPU region table within the CPU, which is explicitly updated by the OS kernel, distinct from MMUs that automatically fill their TLBs. hile MPUs have limited regions and cannot perform address translation, they do not generate TLB misses and are more predictable.

Numerous works have leveraged MPUs to provide isolation on microcontrollers. *FreeRTOS*-MPU [32] restricts memory accesses with MPU, but only addresses safety concerns without considering security [33]. Safer Sloth [34] swaps protection domain between thread switches, which guards against transient faults. TockOS [16] relies on Rust for kernel safety and relies on MPU to confine other applications. ACES [13], [35] divides the program into compartments according to

security policies and segregates them with MPUs. uXOM [36] implements execute-only memory with MPUs and unprivileged memory access instructions. Pip-MPU [37] implements a formally verified separation kernel with MPUs. Other works, such as Mbed μVisor [38], uSFI [39] and OPEC [40] also use MPU to confine their protection domains. Compared to virtualization, they require new languages, toolchain modifications or workflow changes that are incompatible with the microcontroller ecosystem.

In addition, some works have explored virtualization [7], [41], [42], [8]. However, they have high execution overheads and large memory footprints that is often many times that of the original RTOS, which makes them impractical for consolidation scenarios.

Other researchers have employed specialized protection hardware to achieve virtualization or isolation [43], [44], [45], [9], [46], [47], [48], [49]. Although these methods are successful, they rely on specific hardware that are not typically present on general-purpose commodity microcontrollers.

Memory protection is also used to increase isolation in lightweight cloud unikernels, which run everything within a single address space [50], [51], [52], [53]. While these works are not directly related to this paper, they do highlight the applicability of memory protection in system design.

C. Software Fault Isolation (SFI)

Many works bound memory accesses with software constructs, achieving the same as hardware MPUs. These works can be further divided into two categories.

The first category relied on compiler checks or binary verification. TinyOS [54] uses compile-time checks to eliminate possible access races. ARMor [11] leverages binary verification to isolate faults. CRT-C [15] formalizes program compartments and achieves privilege separation with specialized language dialects. Minion [10] and TrustLight [55] determine memory compartments offline and enforce isolation with the help of custom hardware.

The second category leverages managed execution environments originally designed for resource-rich environments, also known as bytecode interpreters. Recently, many of them are ported to microcontrollers, including eBPF [56], [57], [58], [59], Java [60], Python [61], Javascript [62], [63], Lua [64], and even .NET [65] among many others.

However, both are unsuitable for microcontroller consolidation, because they (1) may restrict the choice of programming languages, (2) cannot be linked against third-party binaries without breaking isolation, and (3) induce long latency and large memory overheads when compared to the native binaries.

D. Control Flow Integrity (CFI)

Some works enforce CFI instead of or in addition to isolation, through runtime software or hardware checks, thereby defeating control-flow attacks. RECFISH [14] inserts checks on function calls/returns and maintains a separate MPU-protected return stack. WARduino [66] and aWSM [12] leverage WebAssembly sandbox to provide control- and partial data-flow integrity, and the latter leverages MPU for bounds checking. μRAI [67] stores a list of possible return addresses in Flash memory and verifies returns against them. CaRE [68] leverages *ARMv8-M TrustZone* to accelerate shadow stack operations, whereas Silhouette [69], Kage [70] and SuM [71] achieve the same with MPU and unprivileged memory access instructions. Sherloc [72] exploits hardware tracing buffers that were intended for debugging instead of a shadow stack.

However, their applicability in microcontroller consolidation is limited, because they (1) sometimes do not provide isolation, (2) insert run-time checks that bloat execution latency, (3) cannot be linked against third-party binaries without breaking integrity, and (4) may require customized toolchains or hardware that is not always available in a production setting.

E. Microcontroller Ecosystem Idiosyncrasies

Different from regular application engineering practices, the microcontroller development ecosystems have the following idiosyncrasies: (1) the certified compilers like IAR [73] cannot be customized, (2) the compilers may generate proprietary or esoteric object formats, (3) the linkers may not support sophisticated object symbol manipulations, (4) the debuggers may only support the proprietary executable formats, (5) the toolchain always assume bare-metal, freestanding environments.

These idiosyncrasies render researches involving toolchain or workflow customization less useful in practice, and the *FVM* must therefore account for these factors and stay fully compatible with the current ecosystem.

F. Threat and Task Model.

Threat Model. In *FVM*, we assume a compromised or faulty protection domain (VM) always tries to disrupt others, and such behavior must be confined. *Spatially*, it attempts integrity/confidentiality breaches by unauthorized memory or I/O accesses. *Temporally*, it tries to deny CPU availability to lower-priority domains by executing a dead loop.

Task model. The *FVM* system has two groups of threads: native process and VM. Native process threads always have higher priorities than VM threads, and are scheduled with Fixed-Priority Round-Robin (FPRR). VM threads are scheduled by their respective guest RTOS schedulers when no native process threads are running.

III. *FVM* Design

The main goal of *FVM* is to enable practical virtualization-based consolidation on commodity microcontrollers. The detailed goals include:

- **G1: Low execution latency.** To cope with low latency requirements of the original firmware, the system operations and interrupt responses in the VM must exhibit low overheads and high predictability.
- **G2: Small memory footprint.** To make efficient use of limited memory, the data structures must be compact and the memory allocation must be flexible.
- **G3: Inter-VM isolation.** To ensure security and reliability, the VMs must be confined *spatially* and *temporally* to stop the propagation of faults and compromises.

G4: Ecosystem compatibility. To be practical, the system must be compatible with the microcontroller ecosystem, including architectures, toolchains, RTOSes, workflows, and developer mental models.

A. Virtualization Architecture Overview

As shown in Figure 1, we base the entire virtualization framework on a microkernel that hosts other components as processes. A type II hypervisor manages the guest VMs, within which two cooperating threads emulate a bare-metal vCPU. The lower-priority thread (USR) is responsible for executing all tasks of the guest RTOS, while the higher-priority thread (VCT) handles interrupts and switches task context by modifying USR's context.

Compared with a type I hypervisor design, this choice offers several benefits, including (1) a smaller Trusted Computing Base (TCB), (2) a cleaner separation of policy and mechanisms, and (3) supporting lighter-weight native processes in complement to VMs.

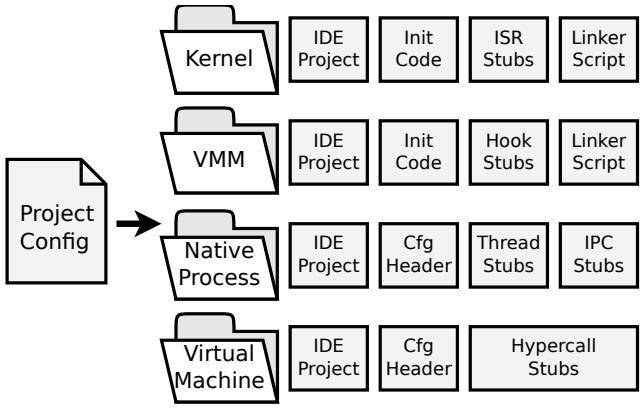

Fig. 2: *FVM* composer workflow. Only the major artifacts generated are shown.

To simplify consolidation workflow, a system composer shown in Figure 2 generates the memory layout, boot-time initialization code, configuration headers and linker scripts. It also produces the Integrated Development Environment (IDE) projects to integrate seamlessly with the toolchain ecosystem.

B. Microcontroller-Specific Microkernel

To support *FVM*, we propose a capability-based microkernel tailored to the microcontroller architectures. Notably, it makes the following design decisions:

Exposing microkernel thread context to user-level. The context switches in RTOSes often employ interrupt vectors to manipulate the registers of the background execution. To eliminate the need for a dedicated system call [74] for such modification, we enable the kernel to save the USR context inside the VM user space instead of the kernel space, so that the context can be modified by VCT directly **(G1)**. However, two security challenges must be overcome to prevent VM escapes: ensuring valid memory access and verifying sensitive register integrity **(G3)**. To this end, we (1) make sure the memory region falls within the physical memory to allow safe kernel access, and (2) insert checks on kernel context restore path to correct illegal sensitive register (*e.g. Cortex-M*'s LR or *RISC-V*'s mstatus) values.

Eliminating kernel object alignment requirements. To make efficient use of limited memory, the kernel objects allocation must be flexible and economical. Notably, MPUs do not perform virtual-to-physical address translation, but do not impose page size and alignment restrictions as Memory Management Units (MMUs) do, either. In these lights, we employ a fine-grained kernel memory allocator rather than a paged allocator, to capitalize on the absence of page size and alignment restrictions. It eliminates the artificial alignment constraints often imposed by other microkernels, minimizing external memory fragmentation **(G2)**. The kernel memories are not tracked in pages but rather in *kernel memory capabilities*, which corresponds to permission to create kernel objects in a memory segment. In this fashion, the kernel memory management policies are also exported to the user-level.

Making in-kernel memory management optional. Given that microcontroller systems tend to statically allocate everything at boot-time, we forgo memory retyping and instead maintain a predetermined boundary between kernel and user memory **(G2)**. Furthermore, we observe that memory access permissions are also rarely modified at run-time. When this is the case, the system composer generates the MPU tables for all processes **(G2)** and store them in *read-only* flash memory **(G3)**, and instructs the kernel to exclude memory management once and for all. This further reduces the memory footprint as well as the attack surface of the kernel **(G3)**. Similar principles can also improve memory management performance in microprocessor-based systems, though this lies beyond the scope of the paper.

Simplifying in-kernel scheduling facilities. In the light of Mixed-Criticality Scheduling (MCS), modern microkernels abandon traditional threads for separate *scheduling* and *execution* contexts. Scheduling contexts hold budgets and policies, while execution contexts store thread register contents. This separation enables flexible policy and subsystem coordination with managed interference. However, the scheduler hot-path now involves both scheduling and execution context objects, increasing overhead for validity checks and data structure operations. Some systems [75] go even further to encode priority as a vector, storing all subsystem scheduler scalar priorities in the scheduling context to avoid priority flattening. In cases where priorities don't flatten, preemption decisions require comparing all subsystem priorities, adding more overheads. Though these overheads don't introduce substantial latency on high-performance processors where cache misses dominate latency, it does become a significant factor on microcontrollers that operate at a rate commensurate to their memory.

In *FVM*, we revert to the traditional thread concept and implement a FPRR policy in the kernel similar to most RTOSes. This policy is *not* a full scheduler: we keep room for bandwidth servers by removing implicit budget replenishes, and each subsystem scheduler now explicitly replenishes its threads' budgets, limiting the interference. If the priorities can be flattened to match the in-kernel FPRR, latencies are minimal, which is often the case for microcontrollers **(G1)**.

Otherwise, we bypass the kernel policy by manually placing inactive servers on a priority lower than the idle thread and currently active servers on higher priorities, allowing user-level policies to take full control. To speed up the operation, our priority change system call accepts up to two thread capabilities and can update them simultaneously **(G1)**. The trade-off here is to optimize for responsive handling of common scenarios, while accepting an increase in latency for the complex systems that require more *controlled interference* than *absolute minimal latency*.

In *FVM*, the in-kernel FPRR policy is directly in charge of the native processes which provide minimal latency, while the user-level hypervisor scheduler is in charge of the VMs that can tolerate more overhead.

Hypercall	Functionality
hyp_int_ena	Enable virtual interrupts.
hyp_int_dis	Disable virtual interrupts.
hyp_vct_phys	Register a virtual vector with a physical vector.
hyp_vct_evt	Register a virtual vector with an event source.
hyp_evt_add	Allow this VM to send to an event source.
hyp_vct_lck	**Lockdown vector mappings and event source permissions.**
hyp_vct_wait	Suspend the VM until future vector activation.
hyp_evt_snd	Send an event to an event channel.
hyp_wdg_clr	Start or clear the virtual watchdog.

TABLE I: *FVM* hypercalls. Sensitive configurations can be locked down with hyp_vct_lck to prevent further modifications. This guarantees security as the boot code for each VM is stored in read-only flash.

C. Type II Paravirtualizing Hypervisor

The type II hypervisor architecture lies on top of the microkernel to support the guest VMs. The hypercalls are shown in Table I. It performs:

CPU virtualization. Instead of pairing each guest thread with a dedicated microkernel thread, as is commonly done in microkernel-based virtualization, we assign all guest tasks within a VM to a single USR thread, and assign all guest interrupt handlers within a VM to a single VCT thread.

Compared with the traditional solution, our design (1) eliminates the need to allocate extra microkernel threads **(G2)**, (2) enables user-level guest thread switching without kernel intervention **(G1)**, and (3) avoids the semantic mismatch between the guest RTOS and microkernel scheduler, which eliminates semantic translation and prevents the resulting execution amplification that happen in [7] **(G1)**.

Compared to Xen [20] which uses one thread per vCPU, our design does not incur End-Of-Interrupt (EOI) atomicity issues. Thus, we avoid making EOI hypercalls on virtual interrupt return paths **(G1)** or writing complex critical region stack fixup code in assembly **(G4)**.

Interrupt indirection. When a physical interrupt arrives, the microkernel unblocks the hypervisor interrupt indirection thread. This thread then searches for registered VMs and injects virtual interrupts into them by unblocking their VCT threads. To use this mechanism, each VM must first call hyp_vct_phys(phys_id, virt_id) to correspond one of its virtual interrupt sources with the physical interrupt. They must then call virt_vct_set(virt_id, vct_ptr) to configure the handler and hyp_vct_lck() to lockdown virtual-to-physical mappings. Finally, they must enable interrupts by calling hyp_int_ena().

In this design, a privileged Dom0 (as in Xen [20]) is not needed since the VM boot code is burned into *read-only* Flash **(G3)**, and each VM is forced to execute it upon booting. When this is the case, the mappings will remain constant even if the VM compromised later. This model assumes *discretionary access control* where each VM configures its interrupts to align with the traditional microcontroller development mental model, while leaving room for *mandatory access control* when the boot code for all VMs is under the control of a single team.

Device management. *FVM* diverges from traditional virtualization by not performing device virtualization. Instead, it passes peripherals through to the VM directly, as they typically do not share peripherals **(G4)**. It is however possible to allocate dedicated native processes or VMs to manage devices, which can then act as a device manager for other VMs **(G3)**. DMA and Ethernet are typical examples: the mismanagement of DMA may breach spatial isolation, while the sharing of Ethernet connectivity provides network access for all VMs.

Event passing. Event sources are similar to physical interrupts in that they also trigger virtual interrupts. They come from VMs and native processes to achieve inter-VM and VM-process communication. Event source setup follows is similar to physical interrupts, and the primary difference is using hyp_vct_evt(evt_id, virt_id) for registration instead. The event mappings are also locked down by hyp_vct_lck(), which prevents further modifications **(G3)**. The VM can then send events to authorized sources through the hyp_evt_snd(evt_id) hypercall.

VM scheduling. *FVM* employs a Fixed Priority Round-Robin (FPRR) scheduler for scheduling VMs. When a guest VM has no runnable threads, it calls suspends hyp_vct_wait() to suspend its execution until further virtual vector activation, mirroring the behavior of the WFI or WFE instructions in real hardware. When a suspended VM receives a virtual interrupt, it is immediately resumed, and will preempt the current one if it has a higher priority. If a VM does not suspend itself exhausts its allocated budget, the round-robin mechanism will select the next VM for processing. This design deliberately trades interrupt batching efficiency in favor of achieving an absolute minimum latency, thus ensuring real-time responsiveness **(G1)**.

Fault handling. To address faults within VMs, a hypervisor fault handling thread runs at the highest priority. When a fault occurs, this thread unblocks and invokes a customizable handling hook, within which the user can reboot the VM or take other actions **(G3)**. To avoid potential deadlocks or infinite loops from bugs within the VM, *FVM* also provides virtual watchdogs, similar to those found on physical microcontrollers **(G3)**. This is achieved through the hyp_wdg_clr() hypercall, which initializes the watchdog upon the first call and feeds the dog upon subsequent calls.

D. Native Processes

The type-II design enables the coexistence of lighter-weight native processes alongside VMs, similar to KVM-based[1] solutions. The native processes can be used to execute timing-sensitive portions of interrupt "bottom-halves" before passing them to the VM through events. Alternatively, when a total code rewrite is feasible, they can power the entire application, and *FVM* hypervisor can be stripped down to just the fault handling thread, significantly minimizing memory footprint. This enables developers to build even lighter weight systems for extremely resource-constrained devices.

Process initialization. Typical microcontroller toolchains generate images that are directly executed from persistent on-chip Flash memory. During the boot-up, the initialization code copies the data section (*GCC* .data, *ARMCC* .rwdata) from its Flash load address to the RAM run-time address and clears the zero section (*GCC* .bss, *ARMCC* .zidata). This is distinct from Linux-based systems where the kernel is responsible for such initialization before the control transfers to the program. In certified toolchains such as *ARMCC* and *IAR*, the initialization code and segment details are proprietary. To support them without digging into the obscurity, we masquerade each process as a "bare-metal" image and ensure that no thread can run until the process has been initialized **(G4)**. To achieve this, we identify the highest priority thread within each process and set up the main() function to call it. We then configure the highest priority thread to start from the initialization code entry and the stack intended by the linker for the main() function. This results in a "bare-metal" image where the main() function runs the highest priority thread, mimicking a real bare-metal image. The key finding is that the highest priority thread must be the first to run within each process, initializing the process before all other code runs.

This design has an unexpected benefit: when a native process or VM faults, the *FVM* does little recovery work as reinitialization is performed by itself, ensuring bounded interference to other protection domains and avoiding denial-of-service through repeated faulting **(G3)**.

Event sending. To communicate with the VMs, the threads within native processes can also send to event sources to inject virtual interrupts into the VMs, or block on signal endpoints that the VMs can send to.

Latency-sensitive interrupt bottom-halves. In traditional virtualization schemes, all virtual interrupts must be indirected by the hypervisor, which increases VM interrupt latency. As a consequence, these solutions are forced to move latency-sensitive portions of the VM interrupt "bottom-halves" into kernel space, which can compromise kernel integrity.

In *FVM*, the native processes however do not need this indirection and can receive interrupt activations directly from the microkernel. This creates a confined space with less interrupt latency than the VMs, and is ideal for latency-sensitive portions of interrupt "bottom-halves" **(G1)**. The interrupts can be processed in native processes first before triggering events which the hypervisor will translate to virtual interrupt injections. The data associated with the interrupt activations can be passed to VMs through the use of shared memory.

This design also has an unexpected benefit: when the same interrupt source carries two or more service dataflows that have different criticality (i.e. Ethernet or USB packets), the native thread can act as a filter that drop or postpone packets according to customizable user-level policies, achieving flexible *differentiated interrupt handling* [76] **(G1, G3)**.

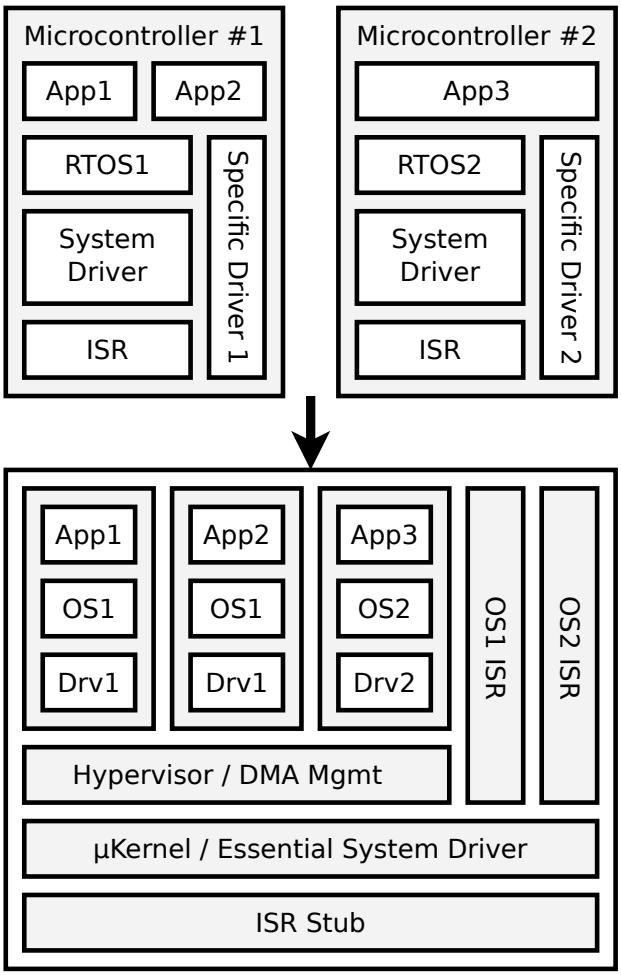

Fig. 3: Example showing two microcontrollers consolidated as one. The first microcontroller has two applications and the second one has one application, and each of them has their respective RTOSes, drivers and ISRs. In the consolidated microcontroller, all applications are confined in their own VMs alongside their drivers (latency-insensitive parts of "bottom-halves"), and the ISR stubs ("top-halves") has been moved to the kernel-level. The ISR routines (latency-sensitive portions of the "bottom-halves") are moved to native processes, which has lower interrupt latencies than the VMs but are still confined.

E. Consolidating Bare-metal Code

To consolidate bare-metal code that runs on different microcontrollers using different RTOSes and frameworks, the key components must be identified, and multiple porting strategies may be applied. As shown in Figure 3, we first separate the entire code base into low-level code that interacts directly with hardware peripherals, and higher-level code that relies on low-level code. The higher-level code including applications and

[1]Some sources classify KVM as type-I, but we classify it as type II because it (1) relies on Linux drivers, (2) is separately loadable as a kernel module.

middleware typically remains unaffected by consolidation and falls outside the scope of discussion. We then separate the low-level code into drivers and RTOSes, each requiring distinct porting strategies to facilitate successful consolidation.

Porting drivers. As a new microcontroller is chosen as the consolidation target, all legacy drivers must be rewritten to fit it while maintaining compatibility with the higher-level code **(G4)**. Most peripherals can be assigned to dedicated VMs as the original firmwares share no peripherals by nature. Peripherals that act as bus masters such as DMA must be managed by the hypervisor or trusted VM proxies **(G3)**. Common routines such as the interrupt controller and system timer initialization must be moved to the microkernel.

The most significant hurdle in porting drivers is the latency-sensitive portion of it. To keep latency low, we place *only* the interrupt flag clearing routines, which cannot be completed at the user-level, into the kernel interrupt handlers **(G1)**, and these are the "top-halves" (ISR stubs). Meanwhile, latency-sensitive portions of the "bottom-halves" (ISR bodies) are moved to native processes, which then pass the interrupts to VMs that host latency-insensitive portions of the "bottom-halves" (within the VMs). Evaluations show that this exhibits an overall interrupt latency comparable to the bare-metal systems.

Porting RTOSes. The porting of RTOSes involves (1) replacing its tick timer with a virtual one, (2) replacing interrupt enable/disable routines with hypercalls, and (3) porting its thread context switch code.

To refit the tick timer, we register its original timer ISR as a virtual timer interrupt, and configure the hypervisor to periodically inject interrupts. Supplanting interrupt disable/enable is not as straightforward, because replacing them with hyp_int_dis() and hyp_int_ena() introduces hypercalls into all RTOS functions that disable interrupts, causing high overheads. Instead, we provide a pair of library functions, virt_int_mask() and virt_int_unmask(), which mask and unmask the VCT virtual interrupt response without stopping its injection. The virt_int_mask() sets the int_mask flag, which is detected by VCT when it unblocks. The VCT then sets a int_pend flag and ignores all incoming interrupts. When virt_int_unmask() is called, int_mask is cleared, and VCT is signalled to process pending interrupts if the int_pend flag is set. This reduces the interrupt enable/disable overheads to that of the bare-metal **(G1)**.

Interrupt context switch code porting is simple because USR's thread context is exposed to VCT, which allows register manipulation to be written entirely in C instead of assembly, eliminating the most architecture-specific and error-prone aspects of it **(G4)**. This is in stark contrast to Xen [77], where such code must be written in assembly. In cases where context switches do not involve interrupts, the original code can be leveraged with minimal effort, within which only interrupt enable/disable operations need to be replaced.

F. System Composer

We introduce a system composer for *FVM* to simplify the setup process. The composer takes an XML description file that outlines the toolchain, compiler options, and memory requirements for each native process or VM. For each VM, you can also specify the RTOS that it intends to run. The composer then produces all necessary configuration files, IDE projects/-makefiles, and automatically ports the RTOSes for VMs. This composer is designed to be out-of-the-box compatible with both proprietary and open-source bare-metal toolchains that allow custom linker scripts and generate binary images, which applies to virtually all toolchains.

Configuration generation. The generation process begins by reading memory requirements from the description file to allocate memory for native processes and VMs, as well as reserve space for the microkernel and hypervisor. Other settings such as priorities and timer tick intervals are also extracted from the description file. With such information, configuration headers are generated for the microkernel and hypervisor, detailing kernel object allocation steps, memory trunk addresses, and other settings. The allocation of memory trunks needs to take MPU region restrictions into account, which we discuss in §IV.

Project organization. To seamlessly integrate with the bare-metal ecosystem, we masquerade each system component (the kernel, hypervisor, native processes and VMs) as individual "bare-metal" projects. For each component, the IDE projects including configuration headers, user-modifiable hooks, and custom linker scripts are created. Compiling each project produces native executables and raw binary trunks (.bin or .hex) that are transcribed into data arrays later. This strips away all potentially conflicting library symbols without requiring executable interlinkability between projects. The arrays are then mapped to fixed addresses as raw data in the microkernel project to create a combined system image. These steps make no assumptions beyond the core functionality of the toolchains, ensuring compatibility across all of them **(G4)**.

In addition, this also allows to choose different toolchains for different projects: certified toolchains can be used for critical components like the kernel and hypervisor, while open-source alternatives can be leveraged for less critical VMs. This eliminates the need to switch compilers when the original projects employ different ones **(G4)**.

Native debugging support. For a consolidated system, effective interactive debugging measures are a must. However, proprietary embedded debuggers only recognize their own symbol format, as well as assuming a "bare-metal" microcontroller. Debugging the microkernel is straightforward since its project produces the final system image. However, debugging other protection domains can be challenging due to missing symbols. To overcome this, we capitalize on the "debug without download" feature which all embedded debuggers support, as they are designed to peek running microcontrollers in the field (**G4**).

We download the system image from the microkernel project first, and then launch a debugging session from the target project without performing the download. The debugger loads symbols directly from the toolchain-specific executable, while the actual running binary remains the system image. In the physical address space of a microcontroller, the symbol

locations of project will always coincide with the system image, allowing the debugger to run flawlessly. The debugger will suspend execution upon the main() function of the project, mimicking a bare-metal debugging session. From there, actual debugging can commence.

IV. *FVM* Implementation

Chip	CPU	Flash/RAM	Toolchain	RTOS
STM32L071CBT6	CM0+	128K/20K	ARMCC/GCC	FreeRTOS/RMP
STM32F405RGT6	CM4F	512K/192K	ARMCC/GCC	FreeRTOS/RMP
STM32F767IGT6	CM7F	1M/512K	ARMCC/GCC	FreeRTOS/RMP
CH32V307VCT6	RV32	192K/128K	GCC	FreeRTOS/RMP

TABLE II: *FVM* implementation combinations.

To demonstrate the general applicability of *FVM*, we carry out implementations across four popular chips of three common architectures, two industry-standard toolchains, and two open-source RTOSes, for a total of seven combinations shown in Table II.

We believe that the diverse combinations effectively demonstrate the flexibility and applicability of *FVM*.

A. FVM for ARMv6-M: the Minimal

At the lower end, we choose the *STM32L071CBT6* featuring 128KiB Flash and 20KiB RAM, paired with a MPU-enabled *ARMv6-M Cortex-M0+* core @ 32MHz. This is a extremism proof that virtualization can be technically possible on even the most constrained hardware. In practice, a sensible *Cortex-M0+* choice would be similar to *STM32G0B1RET6*, which houses 512K/144K while remaining sub-$1.

Memory management. The *Cortex-M0+* MPU has 8 regions with power-of-2 sizes, where each region's base address must align with its size. Additionally, each region has 8 subregions that can be individually enabled or disabled. For example, we consider a memory segment with base address 0x20000000 and size 0x600. Its size is not a power-of-2 and would have required two regions; when subregions are leveraged, a single region with base address 0x20000000 size 0x800 will suffice with its two tailing subregions disabled. he system composer's static region allocation algorithm is similar to that of [7], which tries to minimize the number of regions required for a given memory map by considering both regions and subregions. It works by finding the smallest power-of-2 region size, deciding on the base address, leveraging subregions when possible, and recursively mapping residual segments until all are covered. When it runs out of regions, an error is reported.

The *Cortex-M0+* faults upon memory access permission violations, but does not provide address information. This makes the dynamic region swapping algorithm in [78] inapplicable, and there's little additional point to keep the in-kernel memory management facility. Thus, we disable it and store generated MPU protection tables in read-only Flash.

RTOS porting. We port two open-source RTOSes, *FreeRTOS* and *RMP*, to the *FVM*. We registered their original timer interrupts to the virtual timer source using virt_vct_set(), and replaced their interrupt disable/enable calls with virt_int_mask() and virt_int_unmask(), respectively. We also replaced system initialization code with a simple hyp_int_ena() call. No further change beyond the context switch code is required.

Listing 1: Assembly

```
PendSV_Handler PROC
MRS     R0,  PSP
LDR     R2,  =pxCurrentTCB
STR     R0,  [R2]
STMIA   R0!, {R4-R7}
MOV     R4,  R8
...
STMIA   R0!, {R4-R7}

PUSH    {R3, LR}
BL      vTaskSwitchContext
POP     {R2, R3}

LDR     R0,  [R2]
MOV     R1,  R0
ADDS    R0,  R0, #16
LDMIA   R0!, {R4-R7}
MOV     R8,  R4
...
LDMIA   R1!, {R4-R7}
MSR     PSP, R0
BX      R3
ENDP
```

Listing 2: Transcribed C

```
void VCT_PendSV_Handler(void) {
    unsigned long* SP;
    SP = USR_REG->Reg.SP;

    *(--SP) = USR_REG->Reg.R4;
    ....
    *(--SP) = USR_REG->Reg.R11;
    *(--SP) = USR_REG->Reg.LR;

    *pxCurrentTCB = SP;
    vTaskSwitchContext();
    SP = *pxCurrentTCB;

    USR_REG->Reg.LR = *(SP++);
    USR_REG->Reg.R11 = *(SP++);
    USR_REG->Reg.R10 = *(SP++);
    USR_REG->Reg.R9 = *(SP++);
    USR_REG->Reg.R8 = *(SP++);
    ....
    USR_REG->Reg.R4 = *(SP++);
    USR_REG->Reg.SP = SP;
    return;
}
```

Porting *FreeRTOS* context switch assembly is straightforward as it relies solely on the PendSV_Handler() software interrupt triggered by the portYield(). We translate the assembly vector (Listing 1) to C (Listing 2), and register it with the VCT. The resulting C code is simpler than the assembly due to *Cortex-M0+*'s not supporting LDMIA on high registers R7-R11.

The *RMP* context switch code additionally included an assembly snippet that switches context without triggering the PendSV interrupt. To integrate it, we replaced its CPSID and CPSIE with virt_int_mask() and virt_int_unmask() calls, respectively. We fitted this snippet to the *FreeRTOS* as well to improve its performance.

Sensitive register fixes. We notice that the LR stores information about where an interrupt would return, with 0xFFFFFFF1 indicating HANDLER mode (kernel-level) and 0xFFFFFFFD indicating THREAD mode (user-level). Hence, malicious LR modifications by VCT can lead to VM execution with kernel privileges, and we defeat this by forcing LR to 0xFFFFFFFD upon interrupt exits.

System composer. *Cortex-M0+* is supported by various toolchains including open-source *GCC* and *clang* as well as proprietary certified *Keil µVision* and *IAR*. We choose *GCC* and *ARMCC* given their high popularity. We employ *Makefile* in combination with *GCC* and *Keil µVision* in combination with *ARMCC*to reflect typical usage patterns found in real-world projects.

For *Makefile-GCC*, we use the *GCC* syntax .s assembly files, generate .ld format linker scripts, and create sub-Makefiles for the microkernel, hypervisor, native processes, and VMs. A top-level makefile will invoke the sub-makefiles to produce the final system image. This process is quite straightforward due to the openness of the *Makefile* format.

Things are more complicated for *Keil µVision-ARMCC*, as the .uvproj project format is proprietary and and no formal description of it is publicly available. We experimented on different project structures and build options to reverse engineer the format, which turned out to be a XML file that

contains the build options and references the source files. In this light, we use the *ARMCC* syntax .asm assembly files, generate .sct format scatter file (linker script equivalent), and create the .uvproj project the microkernel, hypervisor, native processes, and VMs. The *Keil µVision* also supports a .uvmpw workspace format that could hold multiple .uvprojs so that they can be compiled together with a single click in the IDE. We again reverse-engineered this format to reflect the correct build sequence for our system.

B. FVM for ARMv7-M: the Mainstream

The *ARMv7-M* adds FPU and DSP instructions to the base *ARMv6-M* instruction set, and also generates precise fault addresses. We selected the 168MHz *STM32F405RGT6* that has 512KiB Flash and 192KiB RAM to represent the *Cortex-M4F*, and the 216MHz *STM32F767IGT6* 1MiB Flash and 512KiB RAM to represent the *Cortex-M7F*.

The *Cortex-M4F* is slower but more common, while the dual-issue *Cortex-M7F* is more performant but rarer. Additionally, the *Cortex-M7F* features double precision FPU and a cache, which are absent in the *Cortex-M4F*. Though *ARMv7-M* chips are significantly faster than *ARMv6-M* ones, we don't detail the basic porting as it remains similar. We focus on fault handling and FPU management here, which are unique for *ARMv7-M*.

Fault handling. *ARMv7-M* generates precise fault addresses for memory access permission violations, which allows us to implement dynamic region swapping similar to [78]. This treats MPU table as a software-filled TLB, which kicks out temporarily unused memory segments when regions are used up. However, the processor automatically pushes and pops registers (R0-R3, R12, LR, PC, xPSR) to the user-level stack upon interrupt entry and exit. When access permission faults are caused by such stacking, the BFSR or MMFSR containing the faulting address is not updated, and the regions cannot be swapped in due to this reason. To avoid intangible stack-related faults, we ensure the stack is always allocated on a memory segment retained in MPU regions, regardless of swapping. Other faults are handled similarly to the *ARMv6-M*: we restart the process as always.

FPU management. The *Cortex-M4F* and *Cortex-M7F* has identical FPU registers despite that the latter support double-precision. Their FPU context switching is hence the same from a system standpoint.

The FPU has two unique features: automatic stacking (FPCCR.ASPEN) which pushes FPU registers to the user-level stack upon interrupts if FPU is DIRTY, and lazy stacking (FPCCR.LSPEN) which postpones such pushing until FPU is accessed during a context switch. The former minimizes interrupt latency, while the latter avoids excessive pushes when context switches are absent. All threads start with an INITIAL FPU indicating that they have never tainted it. For optimal performance, we enable both features and skip FPU context switches when the current and next threads both report an INITIAL FPU.

When automatic stacking is enabled, touching the FPU clears LR.[4] upon interrupt entry and interrupts will stack the FPU registers. However, since the USR's LR can be modified by user-level in parallel to become INITIAL when multiple CPUs exist, and the LR observed by the microkernel is untrustworthy. If LR is used as a DIRTY indicator, the DIRTY FPU won't be cleaned up when switching to an INITIAL thread; this can leak previous FPU context. To this end, we treat the USR's FPU context as always DIRTY and reinitialize it when we switch to an INITIAL thread.

When lazy stacking is enabled, it occurs when the microkernel saves the FPU context into the thread kernel object. A malicious thread may set its stack pointer to invalid addresses to trigger faults in kernel-space. To this end, we ignore lazy stacking errors (LSPERR and MLSPERR); any malicious FPU stacking attempts only harm the attackers.

C. FVM for RISC-V: the Emerging

The *RISC-V* is an emerging architecture that targets scenarios spanning from cloud computing to embedded. Its openness makes it suitable for microcontroller designs that require flexibility. We choose the 144MHz *CH32V307VCT6* (*RV32IMAFC* core) that has 192KiB Flash and 128KiB RAM to represent *RISC-V*. This port proved the most challenging and we elaborate on the unique difficulties below. In terms of RTOS porting and system composer, it is however similar to the *Cortex-M*.

Nonstandard hardware caveats. The *RISC-V* is an open architecture and manufacturers are *free* to implement their own flavor despite the ratified standards. Unfortunately, this flexibility is not without downsides. Take the *CH32V307VCT6* as an example, it (1) is missing the mandatory machine mode timer (mtime), (2) does not generate precise fault addresses, (3) has a modified interrupt vector table, (4) does not disable machine-mode interrupts upon entry (mstatus.mie remains set), (5) does not block user-level accesses when all Physical Memory Protection (PMP; *RISC-V* MPU equivalent) entries are "off", and (6) adds user-level accessible global interrupt control CSRs (gintenr and intsyscr) that compromise temporal isolation.

We began the port consulting ratified manuals, only to find out that we're *very* wrong. We fix these issues by (1) providing custom timer routines and interrupt vector tables, (2) carefully placing all vectors at the same priority so that they cannot preempt each other, (3) sacrificing one PMP entry to disable all background user-level accesses, and (4) using a binary scanner to make sure no code access gintenr and intsyscr.

Memory region allocation. The *RISC-V* PMP is unique in that it supports both Naturally-Aligned-Power-Of-Two (NAPOT) and Top-Of-Range (TOR) modes. In NAPOT mode, a single entry defines a power-of-2 aligned memory segment similar to the *Cortex-M* MPU. The TOR mode however allows declaring unaligned segments using two entries, each specifying the base and limit address. To use the regions efficiently, our algorithm merges contiguous segments, and checks if the merged segment can fit into a NAPOT entry; if not, we use two entries working in TOR mode instead. Due to the absence of precise PMP faults, the dynamic region swapping algorithm [78] cannot be used.

FPU management. The *RISC-V* FPU management is even more complex than *ARMv7-M*, featuring an additional CLEAN status indicating whether the FPU has been accessed since the last context saving. This allows skipping FPU context saving if the FPU remains untouched since last save. To take advantage of this feature, we follow the designer's intention and only save thread FPU contexts when they're DIRTY, and then mark the FPU as CLEAN again to prevent excessive saves. All threads start with an INITIAL FPU status until their first FPU access makes them DIRTY.

However, the USR FPU status reflected in mstatus is untrustworthy because VCT can arbitrarily modify it. Instead of treating USR's INITIAL FPU statuses as DIRTY like on *ARMv7-M*, we treat them as CLEAN, allowing the microkernel to reinitialize the FPU upon switching to an initial thread while skipping context saving for USR. This way, malicious VMs can only hurt themselves by losing their own FPU context when launching an attack.

V. EVALUATION

In our evaluation, we examine the following key questions that directly correspond to our goals listed in §III:

Q1: How much processing overhead does the *FVM* introduce and is this acceptable?

Q2: How much memory bloat does the *FVM* impose and is this acceptable?

Q3: What changes to a RTOS is needed to port it to the *FVM* and is this acceptable?

Q4: Can the *FVM* recover from VM faults or attacks and reboot the affected VMs individually?

To address these questions, we perform a comprehensive benchmark on four chips introduced in §IV. In all benchmarks, we run two native processes alongside one VM. We use *GCC* 13.2.1 and *ARMCC* 6.3.1 for compilation with the -O3 compiler flag, as well as *FreeRTOS* 9.0.0 and *RMP* 0.2.7 for virtualization evaluation. The *Cortex-M0+* uses an -Os size optimization flag instead, which better reflects the real-world scenarios.

Each benchmark is repeated 10000 times and we report both the mean (average; darker bottom bar) and worst-case (maximum; lighter top bar) values. We do not provide standard deviation values as they are virtually invisible in graphs.

A. Bare-metal Performance (Q1)

We start by benchmarking the baseline performance of vanilla *FreeRTOS* and *RMP* using both *GCC* and *ARMCC* toolchains, which is shown in Figure 4 and 5. *ARMCC* is only compatible with the ARM architecture and thus does not apply to the *RISC-V* processor.

For *FreeRTOS*, we measure the (1) context switch latency (`ctx`), (2) notification latency (`ntf`), (3) semaphore latency (`sem`), (4) message-queue latency `msg`, and (5) their respective interrupt versions with (`.../i`).

For *RMP*, we measure the (1) context switch latency (`ctx`), (2) mailbox latency (`mbx`), (3) semaphore latency (`sem`), (4) message-queue latency `msg`, (5) bounded message-queue latency `bmsg` and (6) their respective interrupt versions

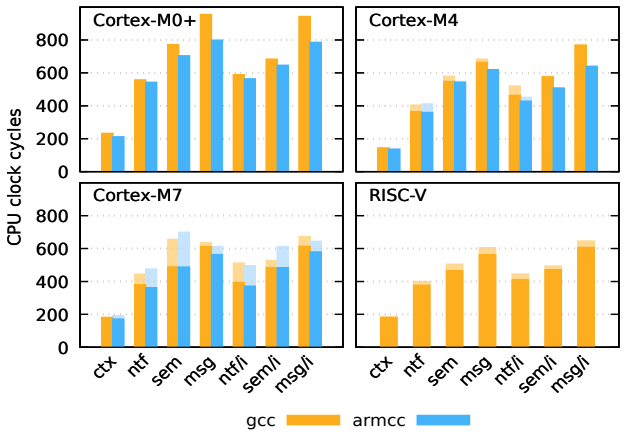

Fig. 4: Bare-metal *FreeRTOS* performance. "`ctx`" is one-way context switch overhead (taskYIELD), "`ntf`" is notification latency (xTaskNotify), "`sem`" is semaphore latency (xSemaphoreGive), "`msg`" is message-queue latency (xQueueSend), and "`.../i`" are respective interrupt latencies (...FromISR). The bare-metal performance typically falls within several microseconds, except for the *Cortex-M0+* which has a low clock rate.

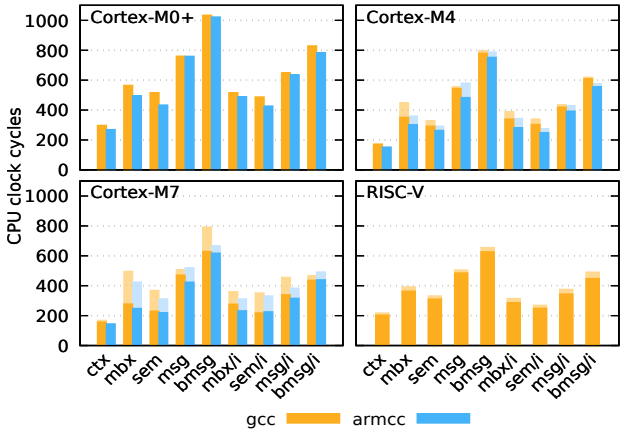

Fig. 5: Bare-metal *RMP* performance. "`ctx`" is one-way context switch overhead (Thd_Yield), "`mbx`" is thread mailbox latency (Thd_Snd), "`sem`" is semaphore latency (Sem_Post), "`msg`" is message-queue latency (Msgq_Snd), "`bmsg`" is bounded-size message-queue latency (Bmq_Snd) and "`.../i`" are respective interrupt latencies (..._ISR). Notably, the *RMP*'s semaphore ("`sem`") is faster than *FreeRTOS*'s, but bounded message queue ("`bmsg`") is slower than *FreeRTOS*'s; this is due to *RMP*'s semaphore-centric construction as opposed to the *FreeRTOS*'s queue-centric construction.

(`.../i`). *RMP*'s mailbox is roughly equivalent to the *FreeRTOS*'s notification as it is tied to a thread and allows to pass light-weight messages between them. *RMP*'s message queues are separated into regular unbounded queues and bounded queues, and the latter is equivalent to *FreeRTOS* message queues.

Discussion. On all architectures and toolchains, both RTOSes perform well and almost all system operation latencies stayed below 1000 CPU cycles. This translates to a few microseconds as all processors but the *Cortex-M0+* runs at more than 100 MHz. As these results are what is achievable by vanilla RTOSes, they represent overheads that are acceptable

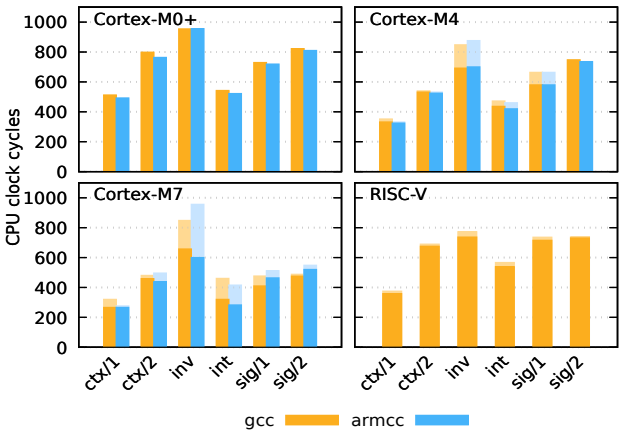

Fig. 6: Microkernel basic operation performance. It is somewhat comparable to that of both RTOSes, although it exhibits slightly higher latencies due to the need to reprogram the MPU during protection domain switches.

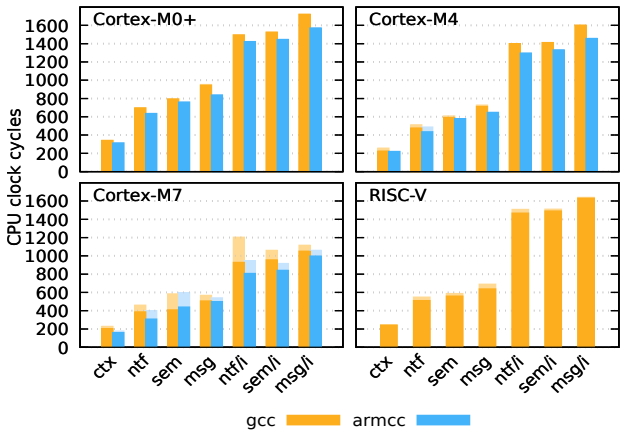

Fig. 7: Virtualized *FreeRTOS* performance. The abbreviations in here and Figure 9 have the same meaning as in Figure 4.

in applications that already leverage these RTOSes.

When comparing architectures, *Cortex-M7F* is approximately 20% more efficient cycle-wise than *Cortex-M4F*, which in turn is 20% more efficient than *Cortex-M0+*; the *RISC-V* processor of our choice is comparable to *Cortex-M7F*. The performance difference between *Cortex-M* processors less pronounced than their CoreMark. This might be attributed to certain system operations that limit the dual-issue capability of the more advanced *Cortex-M7F* by causing hard delays or pipeline flushes. For most architectures, the maximum and average does not differ much, with the *Cortex-M7F* being the sole exception due to its additional cache. On the other hand, thanks to its simple pipeline and low clock rate, the *Cortex-M0+* has almost identical maximum and average.

For both RTOSes, context switching is the fastest because only the scheduler is involved. Message queue operations are significantly heavier than notifications. IPC interrupt latencies generally match their normal versions. When comparing both RTOSes, *RMP*'s notification mechanism and unbounded queue exhibit slightly less latency compared to *FreeRTOS*'s mailbox and (bounded) queue. On the contrary, *RMP*'s bounded queue is slower than *FreeRTOS*'s queue because it is implemented with a semaphore and an unbounded queue. Other performance figures are otherwise similar as both systems are heavily optimized.

When comparing the two toolchains, *GCC* is slightly lower than *ARMCC* on *Cortex-M0+* and *Cortex-M4F*, but achieves parity on *Cortex-M7F*. This means that being compatible with proprietary toolchains in addition to open-source ones might have performance benefits for certain systems.

B. Microkernel Performance (Q1)

As the *FVM* relies on the underlying microkernel, the microkernel performance must be benchmarked so that we can understand other evaluations that follow. As shown in Figure 6, we measure (1) intra-process context switch latency (`ctx/1`), (2) inter-process context switch latency (`ctx/2`), (3) thread migration/return round-trip latency (`inv`), (4) interrupt signaling latency (`int`), (5) intra-process signaling latency (`sig/1`), and (6) inter-process signaling latency (`sig/2`).

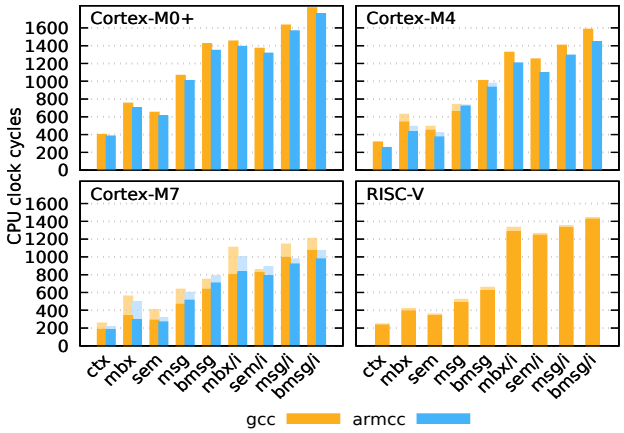

Fig. 8: Virtualized *RMP* performance. The abbreviations in here and Figure 10 have the same meaning as in Figure 5.

Discussion. The *FVM*'s underlying microkernel is comparable to the bare-metal RTOSes in terms of intra-process operations despite its addition of capability-based security. However, in inter-process measurements, an additional 100 cycle overhead is imposed by MPU reprogramming operations, and this translates to less than one microsecond on all processors but the *Cortex-M0+*. The microkernel has less interrupt latency than both RTOSes, because its signal endpoint is lighter-weight than RTOS message-passing IPCs. A signal delivers just the activations without the message, while the messages can be passed in shared memory when needed. Thread migration is also provided for synchronous communication, which is twice as fast as an asynchronous signal round-trip.

C. Guest Virtualization Performance (Q1)

To evaluate how the RTOSes perform after they have been virtualized, we measure the same system operation latencies as we've measured in bare-metal environment, which is shown in Figure 7 and 8.

Discussion. We focus on two groups of latencies. First, the context switch and IPC latencies between the threads. Second,

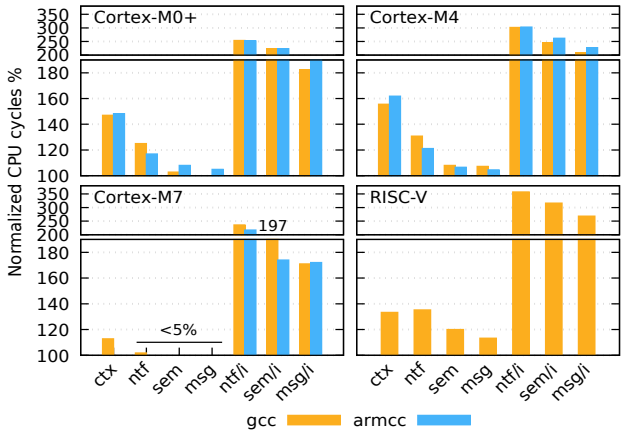

Fig. 9: Virtualized *FreeRTOS* performance normalized by the original bare-metal performance that serves as the 100% baseline.

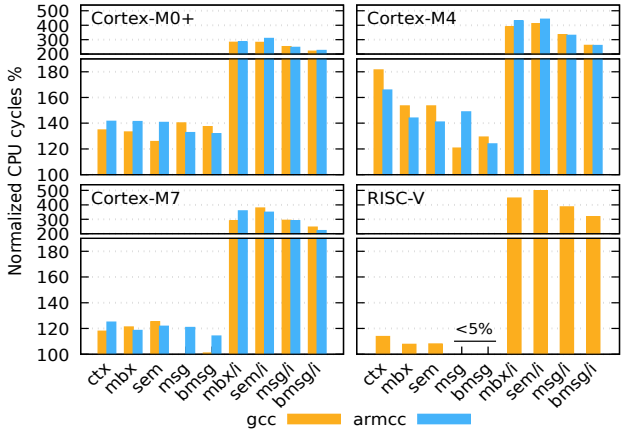

Fig. 10: Virtualized *RMP* performance normalized by the original bare-metal performance that serves as the 100% baseline.

the interrupt latencies from the virtual interrupt handler to the receiving thread. Our results show that for both RTOSes, the first group of latencies remains similar to bare-metal performance. This is because a single USR thread corresponds to all guest threads, and guest thread switching can occur at the user-level without invoking the microkernel. However, when it comes to the second group where IPCs originating from virtual interrupts, the results are different. Even when we expose the USR context to the VCT so that it can be modified efficiently without additional system calls, the resulting latencies remain much higher than those of bare-metal. This is due to the fact that the virtual interrupt injection causes two kernel-level context switches: from USR to VCT upon virtual interrupt activation, and from VCT back to USR upon virtual interrupt exiting. Note that these context switches are unavoidable even if we map one guest thread to one microkernel thread.

When we normalize virtualized measurements by bare-metal measurements as shown in Figure 9 and 10, these observations become much more pronounced. The performance bloat of the first group remained below 60% for all combinations of toolchains and architectures. In a few measurements involving more substantial guest system operations, the difference is hardly noticeable, *i.e. FreeRTOS*'s IPC virtualization overhead is below 5% in *Cortex-M7F*. The second group shows a significant performance penalty: the virtual interrupt latencies overheads are at least 180% of their bare-metal counterpart and can reach up to 500% in some cases, *i.e. RMP*'s semaphore interrupt latency in *RISC-V*.

From these results we can safely conclude that keeping timing-sensitive portions of interrupt "bottom-halves" within guest threads is a disaster regardless of how optimized the virtualization infrastructure might be. Do note that this analysis only considers the latency from the virtual interrupt handler to the receiving guest thread; the additional latency introduced by the hypervisor's interrupt relaying mechanism, which occurs when an interrupt travels from the hardware interrupt handler to the virtual interrupt handler, has not been accounted for yet and will be explored in our next subsection.

D. Interrupt Latency (**Q1**)

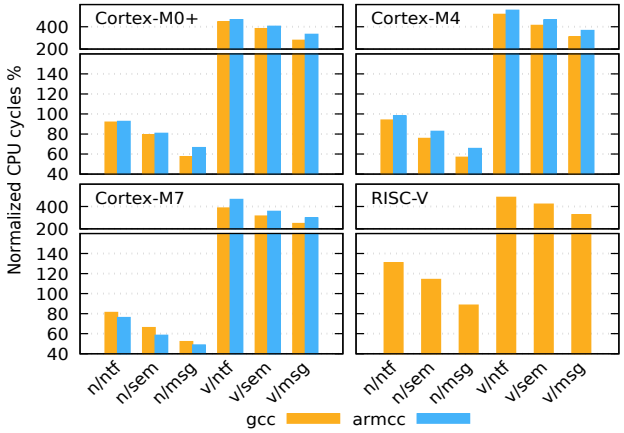

Fig. 11: Native process and VM interrupt relay latency normalized by various bare-metal *FreeRTOS* interrupt latencies that serve as the 100% baseline. The "`n/...`" normalize native process interrupt latency to various bare-metal *FreeRTOS* interrupt latencies, while the "`v/...`" normalize VM interrupt relay latency to various *FreeRTOS* bare-metal interrupt latencies. The abbreviations here have the same meaning as in Figure 4.

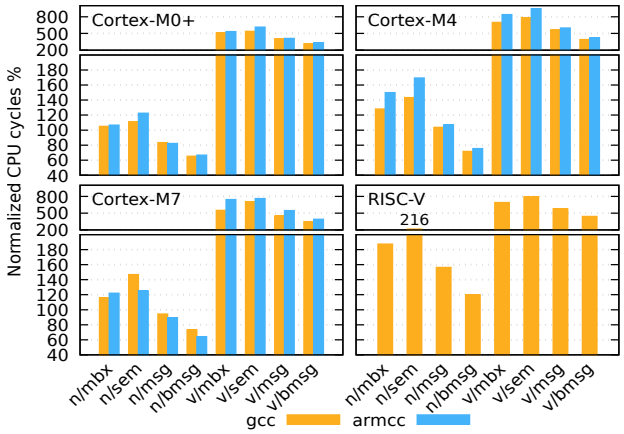

Fig. 12: Native process and VM interrupt relay latency normalized by various bare-metal *RMP* interrupt latencies that serve as the 100% baseline. The notations and abbreviations here have the same meaning as in Figure 11 and 5.

To evaluate the real-time responsiveness of the virtualized RTOSes, we investigate the latency overhead imposed by *FVM*. Instead of showing the raw numbers, we normalize them by various bare-metal RTOS interrupt latencies to make a cleaner comparison, which is shown in Figure 11 and 12. We also show the native process interrupt latency side-by-side, which is normalized as well. For *FreeRTOS*, the `v/ntf`, `v/sem` and `v/msg` values normalize VM interrupt relay latency by bare-metal notification, semaphore and message queue interrupt latencies respectively. For *RMP*, the `v/mbx`, `v/sem`, `v/msg` and `v/bmsg` values normalize VM interrupt relay latency bare-metal mailbox, semaphore, unbounded message queue and bounded message queue interrupt latencies respectively. The `n/...` values have similar meanings except that the value being normalized are native process interrupt latencies.
Discussion. The interrupt relaying latency is generally 400% of any bare-metal latency, and can reach as high as 800% in some cases (`v/...`). This underpins the fact that even placing latency-sensitive portions of interrupt "bottom-halves" into virtual interrupt handlers, where the bare-metal "top-halves" originally reside, is not a viable solution. Moreover, keeping "bottom-halves" within guest threads becomes even less attractive due to the extreme latency bloat: the total latency from hardware interrupt handler to guest thread can reach over ten times of that of the bare-metal, as seen in the case of *RMP* semaphore latency on *Cortex-M4F*, where the combined latency reaches 2376+1248=3624 cycles compared to just 340 cycles for the bare-metal. However, our type-II hypervisor design shines here: we can relocate latency-sensitive portions of the "bottom-halves" into native processes, where they enjoy native process interrupt latency which is generally on par with that of the bare-metal.

This effectively splits our interrupts into three sections rather than two: the extremely sensitive "top-sections" remains in kernel hardware interrupt handlers, simply clearing interrupt flags and reenabling interrupts. Meanwhile, the "middle-sections" can now reside within native processes, performing relatively sensitive data processing. Once the "middle-sections" are complete, the interrupt can be reported to the hypervisor through event channels, and the least sensitive "bottom-sections" are then executed within the VMs. This way, the total interrupt latency to the first application code execution — which are the original "bottom-halves" — remains mostly unaffected (`n/...`). Due to the fact that the microkernel signals only contain activations but no messages, the pure interrupt latency is actually lower than that of the bare-metal RTOS which carries messages, and could reach as low as 50% *i.e.* in the case of *FreeRTOS* on *Cortex-M7F*. In a word, when the "bottom-halves" are moved to native processes, the *FVM* has comparable interrupt latency to bare-metal systems.

Combining subsection V-A, V-C, V-D and specific application requirements in [79], we can safely conclude that the virtualization latencies are acceptable in most applications that already adopt bare-metal RTOSes.

*E. Power Draw (***Q1***)*

To perceive the processing overheads from a power draw perspective, we measure the microcontroller (with minimum external circuitry) 3.3V supply currents when running the benchmark and report the average, as shown in Table III. This exercises all systems to the fullest extent and amplifies the difference that can be attributed to *FVM*.

Toolchain	Test	*Cortex-M0+*	*Cortex-M4F*	*Cortex-M7F*	*RISC-V*
GCC	*FreeRTOS*	7.85	53.10	90.13	31.98
	v*FreeRTOS*	9.41	60.66	107.44	32.60
	RMP	8.31	50.90	94.94	32.36
	v*RMP*	9.51	55.62	107.12	3.04
ARMCC	*FreeRTOS*	7.93	53.69	91.03	N/A
	v*FreeRTOS*	9.48	60.07	105.45	N/A
	RMP	8.26	50.03	96.64	N/A
	v*RMP*	9.43	56.20	106.35	N/A

TABLE III: Average current (mA) when running the benchmark nonstop. The "`v...`" entries represent the virtualized RTOSes.

Discussion. It could be observed that the power overheads are mild in all cases – some 10% increase – which can be attributed to extra CPU mode switches and enabling of MPUs. However, this difference is only exposed when the system is exercised, and real-world applications rarely exercise system calls as strong as benchmarks do. Even in cases where they do, a 10% increase is quite acceptable when compared to SFI approaches that can cause 100% overheads on all execution, including that of the application [12].

*F. Comparison Study (***Q1***)*

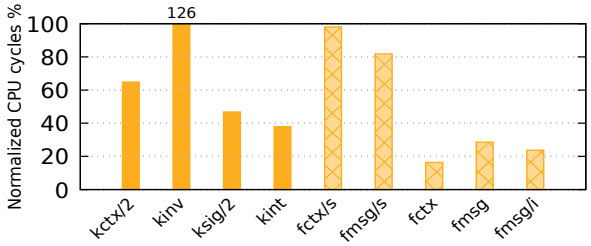

Fig. 13: Microkernel and virtualized *FreeRTOS* performance normalized by that of [7], which serves as the 100% baseline. The solid bars ("`k...`") represent microkernel measurements, while the striped bars ("`f...`") correspond to virtualized *FreeRTOS* measurements. The abbreviations here have the same meaning as in Figure 4 and 6; the additional "`.../s`" do not use the fast context switching path.

To better understand the contributions of this work, we compare it to the existing *Composite* virtualization framework [7] which only requires standard MPUs. We do not consider others [9], [41] because they either require custom hardware that is not ubiquitous across all architectures or use specific instruction set extensions. o make an apples-to-apples comparison, we use the GCC version 5.4.1, and reduce optimization to -O2 to match [7].

The normalized *FVM* performance results are presented in Figure 13, where the *Composite* values serve as the baseline.

The microkernel inter-process context switch latency (`kctx/2`), thread migration latency (`kinv`), inter-process asynchronous signal latency (`ksig/2`) and interrupt latency (`kint`)

are normalized by their corresponding *Composite* kernel values. All these values are inter-process measurements and are shown by the solid bars. Additionally, the *FVM* virtualized *FreeRTOS* context switch latency (`fctx`), message queue latency (`fmsg`) and message queue interrupt latency (`fmsg/i`) are normalized by corresponding *Composite* virtualized *FreeRTOS* values. These are shown by the striped bars. To demonstrate the effectiveness of user-level context switches, we disable the fast-path in our virtualized *FreeRTOS* and perform all context switches through VCT activations. The context switch latency (`fctx/s`) and message queue latency (`fmsg/s`) without user-level fast-path are normalized by *Composite* virtualized *FreeRTOS* values as well.

Discussion. *FVM* microkernel outperforms *Composite* microkernel in most operations (except for `kinv`, which *Composite* heavily optimizes) thanks to its simplified scheduling facilities. Among all measurements, the interrupt latency is significantly reduced by approximately 60% (`kint`), which is more than twice as fast. Despite *Composite*'s *TCaps* mechanism allowing for direct VM interrupt delivery without hypervisor indirection, our design's native process interrupt latency remains superior.

Virtualized *FreeRTOS* measurements demonstrate that our virtualized *FreeRTOS* context switch is over five times faster than that of *Composite* (`fctx`). While message queue latencies are also vastly improved (`fmsg`), it is somewhat less pronounced due to the overheads associated with message passing. This performance boost is attributed to user-level guest context switches that don't require microkernel intervention, as well as the decoupling of guest context switches and microkernel context switches. This could be cross-validated by looking at the measurements with user-level fast-path disabled (`fctx/s` and `fmsg/s`), where *FVM* is roughly on par with *Composite*. The four fold improvement in message queue interrupt latency (`fmsg/i`) can also be attributed to simplified scheduling facilities. This reason is somewhat intricate: *Composite* corresponds each *FreeRTOS* thread to a *Composite* thread and thus needs to translate *FreeRTOS* scheduling semantics to *Composite TCaps* scheduling semantics. In this translation, a dedicated scheduler thread is triggered to handle scheduler notifications generated by the virtual interrupt vector activation, which delays the scheduling of the guest thread. n contrast, our system doesn't require this translation as we model the vCPU instead of making a direct correspondence.

Even with the *Composite*'s *Slite* user-level context switching extension enabled [80], *Composite* is still slower than *FVM*. Take context switch as an example, *Composite-Slite* virtualized *FreeRTOS* has a latency of 2071 cycles on *Cortex-M33F* [80], whereas *FVM* virtualized *FreeRTOS* only have a latency of 224 cycles on *Cortex-M4F*, despite *Cortex-M33F* being about 15% more performant IPC-wise. This is due to the *Composite*'s many-to-many threading model, as well as its additional cost of synchronizing user-level scheduler states with the kernel. Note that *Slite* is a user-level mechanism, so it does not improve inter-process context switch or interrupt latency. In a word, a straightforward FPRR scheduler is better suited for microcontrollers with simple microarchitectures than MCS-capable scheduling facilities; the latter would cause unacceptable overhead. For a microkernel that needs to support both microcontrollers and microprocessors, a more practical approach would be to make MCS facilities optional and move them entirely to the user-level, and when complex MCS facilities are necessary, more powerful microprocessors can be leveraged to handle the extra overhead.

G. Memory Footprint (**Q2**)

To understand how scalable the system is on limited microcontroller on-chip memory, we compile each system component and delineate their memory usage, as shown in Table IV. We also provide bare-metal numbers including and excluding drivers for comparison. Note that the numbers already include all run-time created kernel objects, and both RTOSes are built with a comprehensive benchmark that thoroughly exercises their system calls, allowing for a practical memory footprint to be reflected.

Toolchain	Component	*Cortex-M0+*	*Cortex-M4F*	*Cortex-M7F*	*RISC-V*
GCC	Kernel	55.8/5.89	72.0/6.72	72.6/8.83	65.5/6.33
	Hypervisor	15.9/1.86	21.5/2.28	22.0/2.50	22.2/2.45
	d*FreeRTOS*	14.7/2.61	17.8/5.22	17.0/4.91	23.5/4.91
	b*FreeRTOS*	9.8/2.60	14.4/4.85	11.8/4.90	17.9/4.85
	v*FreeRTOS*	10.8/2.47	14.1/4.70	14.3/4.70	14.7/4.71
	d*RMP*	16.1/1.67	21.0/1.70	24.9/1.98	27.9/1.70
	b*RMP*	11.4/1.66	19.5/1.68	19.6/1.96	25.2/1.68
	v*RMP*	13.2/1.55	20.4/1.56	20.6/1.56	25.8/1.53
ARMCC	Kernel	55.5/5.89	95.0/6.72	94.7/8.83	N/A
	Hypervisor	15.0/1.86	41.3/2.28	31.5/2.50	N/A
	d*FreeRTOS*	12.1/2.58	14.2/4.98	14.6/4.88	N/A
	b*FreeRTOS*	7.8/2.57	11.3/5.11	10.5/4.87	N/A
	v*FreeRTOS*	9.1/2.46	12.8/4.69	13.0/4.69	N/A
	d*RMP*	13.4/1.64	21.8/1.65	22.0/1.95	N/A
	b*RMP*	9.3/1.63	19.7/1.64	17.8/1.93	N/A
	v*RMP*	11.1/1.53	20.0/1.56	20.0/1.56	N/A

TABLE IV: Memory footprints of all software components, in kilobytes. The numbers are listed in a Flash/RAM format, with the first representing Flash and the second representing RAM. The "`d...`" entries denote the bare-metal versions of the two RTOSes with minimal necessary drivers, the "`b...`" entries denote their bare-metal versions with all drivers stripped, and the "`v...`" entries represent their virtualized counterparts.

Discussion. The microkernel always uses the most Flash due to its complexity. The hypervisor is a close second as it contains the bulk of VM scheduling, event passing, interrupt relaying, hypercall handling, and fault handling. However, we argue that Flash is inexpensive and abundant on microcontrollers, as a sub-$1 *STM32G0B1RET6* has 512K Flash.

The microkernel and hypervisor uses approximately 10K RAM, and can be easily fit into microcontrollers that have more than 64K RAM. When the microkernel is used without the hypervisor, the RAM footprint shrinks to about 3K, which could fit into any microcontroller with 16K or more RAM.

The overheads for RTOSes are quite interesting. When the footprints without any drivers (`b...`) are compared to their virtualized counterparts (`v...`), the virtualized versions use more Flash/RAM due to their inclusion of the paravirtualizing library. However, when looking at the footprints with common manufacturer-provided drivers *e.g. STM HAL* (`d...`), the virtualized versions are less bloated because they do not contain hardware drivers anymore. This means that a consolidated system actually use *less* memory than the original system combined, which reflects reduced costs.

Thanks to our simpler kernel design, when the footprints are compared to [7], we decrease microkernel Flash and RAM usage by 46% and 76%, and VM Flash and RAM usage by 72% and 84%, respectively. The removal of unnecessary kernel object alignment requirements is also a contributing factor. Moreover, our two-thread vCPU model does not need scheduler semantic translations, and avoids creating additional kernel objects at runtime, which further reduces memory usage. n conclusion, the *FVM* memory footprints are acceptable on commodity microcontrollers.

*H. RTOS Porting Effort (***Q3***)*

The *FVM* is a paravirtualization facility that requires modifications to the guest OS or firmware. To assess the adaptation effort on existing microcontroller code, we measure the Source Lines of Code (SLoC) that must be modified to port both RTOSes, as shown in Table V.

RTOS	File	*ARMv6-M*	*ARMv7-M*	*RISC-V*
FreeRTOS	C source	+51/-23	+48/-33	+116/-38
	ARMCC asm	+22/-49	+16/-46	N/A
	+fastyield	+111	+91	N/A
	GCC asm	+22/-49	+16/-46	+8/-114
	+fastyield	+108	+87	+122
RMP	C source	+49/-3	+50/-4	+101/-5
	ARMCC asm	+1/-45	+1/-33	N/A
	+fastyield	+109	+91	N/A
	GCC asm	+1/-45	+1/-33	-115
	+fastyield	+106	+91	+122

TABLE V: Modifications of RTOS source, in Source Lines of Code (SLoC). The numbers are listed in a +/- "diff" format, indicating additions (+) and deletions (-). The fastyield entries denote the additional user-level assembly code that accelerates guest thread context switches.

Discussion. The modifications to the C source code are relatively minor, with around 50 lines changed for both *ARMv6-M* (*Cortex-M0+*) and *ARMv7-M* (*Cortex-M4F* / *Cortex-M7F*), and 120 lines changed for *RISC-V*. The *RISC-V* port requires more changes due to its additional 32 registers compared to the *ARM*'s 16. ompared to the SLoC of the bare-metal RTOS hardware abstraction layers (in the case of *ARMv7-M FreeRTOS*, 483 lines), these modifications only accounted for only about 1/5 of the original code. Moreover, most of the assembler modifications are deletions, and the error-prone context switch code can be rewritten in C, greatly simplifying the porting process. Notably, all these changes are in the hardware abstraction layers, leaving the original RTOSes untouched. When the user-level fastyield optimization is implemented, an additional assembler code is required, which is typically around 100 lines. We also compare the total SLoC in the virtualized *FreeRTOS* HAL to [7], where our *FreeRTOS* HAL includes only 283 SLoC instead of [7]'s 363 SLoC. This is because the *FVM* don't need to translate *FreeRTOS* scheduling semantics to the microkernel's.

*I. VM Anomaly Detection and Recovery (***Q4***)*

To evaluate the system's resilience to errors or attacks, we intentionally inject references to (1) null pointers, (2) memories of other protection domains, and (3) nonexistent capabilities into the otherwise correct VM logic.

Discussion. For all combinations of toolchains, architectures, and RTOSes, *FVM* can detect the anomaly immediately and reboot the offending VM without affecting the entire microcontroller, which accomplishes its design goals. However, *FVM* does not defend against ROPs which does not cause permission violations, and this can be complemented by CFI techniques such as sandboxes [59], [12]. In is also noteworthy that sandboxes can also be used in lieu of *FVM* to achieve SFI on even legacy microcontrollers that does not feature a MPU, at the cost of much higher execution overheads [12].

VI. Conclusions

The need to reduce costs, improve reliability and enhance security has been driving microcontroller consolidation in automotive, industrial and commercial scenarios. To this end, the paper proposes *FVM*, a virtualization framework for performing practical consolidation on commodity microcontrollers. We optimize the *FVM* design towards the microcontroller ecosystem and implement it on four chip architectures, three industry-standard toolchains, two open-source RTOSes, for a total of seven combinations. As the evaluation demonstrates, *FVM* reduces the latency overheads down to *near zero*, reduces the virtualization footprint bloat by up to 72%, and imposes minimal effort to port existing systems to it. We believe that *FVM* enables practical virtualization-based microcontroller consolidation in automotive, industrial, and commercial applications.

References

[1] K. Vipin, "CANNoC: An open-source NoC architecture for ECU consolidation," in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2018.

[2] "Adaptive AutoSAR. The AUTOSAR Runtime for Adaptive Applications (ARA), https://www.autosar.org/standards/adaptive-platform, retrieved 12/14/20," 2020.

[3] 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.

[4] C. Morales-Gonzalez, M. Harper, M. Cash, L. Luo, Z. Ling, Q. Z. Sun, and X. Fu, "On building automation system security," *High-Confidence Computing*, 2024.

[5] B. Sá, J. Martins, and S. Pinto, "A first look at RISC-V virtualization from an embedded systems perspective," *IEEE Transactions on Computers*, 2021.

[6] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. Audsley, and Z. Dong, "Towards hard real-time and energy-efficient virtualization for many-core embedded systems," *IEEE Transactions on Computers*, 2022.

[7] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[8] N. Klingensmith and S. Banerjee, "Hermes: A real time hypervisor for mobile and IoT systems," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018.

[9] 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.

[10] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching." in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[11] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "ARMor: Fully verified software fault isolation," in *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, 2011.

[12] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical software fault isolation for reliable embedded devices," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2020.

[13] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *IEEE Symposium on Security and Privacy (SP)*, 2017.

[14] R. J. Walls, N. F. Brown, T. L. Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[15] A. Khan, D. Xu, and D. J. Tian, "Low-cost privilege separation with compile time compartmentalization for embedded systems," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[16] L. Amit, C. Bradford, G. Branden, G. Daniel, P. Pat, D. Prabal, and L. Philip, "Multiprogramming a 64 kB computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[17] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the ibm system/360," *IBM Journal of Research and Development*, 1964.

[18] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, 1974.

[19] "Linux KVM: http://www.linux-kvm.org, retrieved 9/16/12."

[20] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 1–14.

[22] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds," *ACM SIGCOMM Computer Communication Review*, 2009.

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

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] P. Parra, A. Da Silva, B. Losa, J. I. García, Ó. R. Polo, A. Martínez, and S. Sánchez, "Tailor-made virtualization monitor design for cpu virtualization on leon processors," *ACM Transactions on Embedded Computing Systems*, 2023.

[32] "FreeRTOS: http://www.freertos.org, retrieved 5/1/13."

[33] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Good motive but bad design: Why ARM MPU has become an outcast in embedded systems," *arXiv preprint arXiv:1908.03638*, 2019.

[34] D. Danner, R. Muller, W. Schröder-Preikschat, W. Hofer, and D. Lohmann, "SAFER SLOTH: efficient, hardware-tailored memory protection," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[35] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *Proceedings of the 27th USENIX Conference on Security Symposium (CSS)*, 2018.

[36] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Effient execute-only memory on ARM Cortex-M," in *USENIX Security Symposium (USENIX Security)*, 2019.

[37] N. Dejon, C. Gaber, and G. Grimaud, "Pip-mpu: Formal verification of an mpu-based separation kernel for constrained devices," *International Journal of Embedded Systems and Applications*, 2023.

[38] "Mbed µVisor: https://github.com/armmbed/uvisor, retrieved 6/2/24."

[39] Z. B. Aweke and T. Austin, "uSFI: ultra-lightweight software fault isolation for iot-class devices," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[40] X. Zhou, J. Li, W. Zhang, Y. Zhou, W. Shen, and K. Ren, "OPEC: operation-based security isolation for bare-metal embedded systems," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[41] A. K. Sundar Rajan, A. Feucht, L. Gamer, I. Smaili, and N. D. M., "Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls," *Journal of Systems Architecture*, vol. 82, p. 37–48, 2018.

[42] "F9 microkernel: https://github.com/f9micro/f9-kernel, retrieved 6/2/24."

[43] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *Security and Privacy in Communication Networks: 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings 6*, 2010.

[44] H. Xia, J. Woodruff, H. Barral, L. Esswood, A. Joannou, R. Kovacsics, D. Chisnall, M. Roe, B. Davis, E. Napierala *et al.*, "CheriRTOS: A capability model for embedded devices," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

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

[46] 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.

[47] H. Almatary, M. Dodson, J. Clarke, P. Rugg, I. Gomes, M. Podhradsky, P. G. Neumann, S. W. Moore, and R. N. Watson, "Compartos: Cheri compartmentalization for embedded systems," *arXiv preprint arXiv:2206.02852*, 2022.

[48] M. Schönstedt, F. Brasser, P. Jauernig, E. Stapf, and A.-R. Sadeghi, "Safetee: combining safety and security on arm-based microcontrollers," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.

[49] D. Oliveira, T. Gomes, and S. Pinto, "utango: an open-source tee for iot devices," *IEEE Access*, 2022.

[50] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[51] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[52] G. Li, D. Du, and Y. Xia, "Iso-unik: lightweight multi-process unikernel through memory protection keys," *Cybersecurity*, 2020.

[53] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: A library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[54] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for network sensors," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[55] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: a security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[56] "extended Berkeley Packet Filter (eBPF). https://ebpf.io/."

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

[58] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers," in *ACM/IFIP International Middleware Conference (Middleware)*, 2022.

[59] S. Kubica and M. Kogias, "μbpf: Using ebpf for microcontroller compartmentalization," in *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, 2024.

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

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

[62] "Duktape. An embeddable Javascript engine with a focus on portability and compact footprint, https://duktape.org, retrieved 12/14/20," 2020.

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

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

[65] W. R. Davis, P. A. Laplante, and B. I. Sandén, "A real-time virtual machine implementation for small microcontrollers," *Innovations in Systems and Software Engineering*, 2012.

[66] R. Gurdeep Singh and C. Scholliers, "WARDuino: A dynamic webassembly virtual machine for programming microcontrollers," in *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2019.

[67] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[68] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[69] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[70] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow protection on Real-Time embedded systems with kage," in *31st USENIX Security Symposium (USENIX Security)*, 2022.

[71] W. Choi, M. Seo, S. Lee, and B. B. Kang, "SuM: Efficient shadow stack protection on ARM Cortex-M," *Computers & Security*, 2024.

[72] X. Tan and Z. Zhao, "Sherloc: Secure and holistic control-flow violation detection on embedded systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[73] "IAR Embedded Workbench: https://www.iar.com, retrieved 6/2/24."

[74] A. Lackorzynski *et al.*, "L4linux porting optimizations," *Master's thesis, TU Dresden*, 2004.

[75] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *38th IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[76] Z. Ruan, A. Njavro, and R. West, "Usb interrupt differentiated service for bandwidth and delay-constrained input/output," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.

[77] D. Chisnall, *The definitive guide to the xen hypervisor*. Prentice Hall, 2008.

[78] 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.

[79] R. Pan and G. Parmer, "SBIs: Application access to safe, baremetal interrupt latencies," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[80] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: OS support for near zero-cost, configurable scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.