Using Virtualized Task Isolation to Improve Responsiveness in Mobile and IoT Software

Neil Klingensmith
University of Wisconsin
naklingensmi@wisc.edu

Suman Banerjee
University of Wisconsin
suman@cs.wisc.edu

ABSTRACT

We present Hermes, a hypervisor for MMU-less microcontrollers. Hermes enables high-performance bare metal applications to coexist with real-time operating systems (RTOSes) and other less time-critical software on a single CPU. Hermes creates isolated virtual runtime environments for real-time tasks by adding a layer of abstraction between the hardware I/O devices and the software that services them. Virtualization on low-power mobile and embedded systems also enables some interesting software capabilities like secure execution of third-party apps, online privacy controls, and bare metal performance in a multitasking software environment. These features otherwise require additional hardware (i.e. multiple CPUs, hardware TPM, etc) or may not be available at all. In other projects, we have anecdotally noticed that RTOSes are not always able to respond quickly and deterministically enough to time-sensitive operations, particularly under high I/O load. We validate this observed timing problem by measuring interrupt latency in an RTOS environment and comparing to an experimental implementation of Hermes. In our evaluation we compare runtime performance of several realistic mobile apps on Hermes and FreeRTOS. We find that not only is the interrupt latency lower in the virtualized environment, but it is also much more deterministic—a key figure of merit for real-time software systems.

CCS CONCEPTS

• **Software and its engineering** → *Virtual machines*; • **Computer systems organization** → **Real-time operating systems**; **Embedded systems**.

KEYWORDS

Real-time systems; hypervisor; virtualization

ACM Reference Format:
Neil Klingensmith and Suman Banerjee. 2019. Using Virtualized Task Isolation to Improve Responsiveness in Mobile and IoT Software. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '19), April 15–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302505.3310078

IoTDI '19, April 15–18, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6283-2/19/04…$15.00
https://doi.org/10.1145/3302505.3310078

1 INTRODUCTION

Modern embedded sensing and mobile applications increasingly perform diverse functions, including displaying user interfaces, managing networking, performing real-time data acquisition, and more. Some even allow third-party code to be downloaded and run alongside the factory firmware [24]. Such diversity in runtime requirements poses challenges to software architects, who must manage the often competing needs of different tasks.

To manage the diverse runtime requirements of embedded software, we have developed a lightweight embedded hypervisor we call Hermes [1] , targeted to ARM Cortex-M microcontrollers. Other authors have proposed similar systems for mobile phone environments, but none that we are aware of on MMU-less processors [5, 11, 21].

IoT applications are frequently implemented on CPUs without an MMU in order to save cost and power. While the cost of MMU-equipped Linux-capable processors is going down all the time, energy considerations (especially for mobile applications) are not likely to go away.

The problem we set out to solve is one of I/O latency in such a complex runtime environment. Real-time OS scheduling algorithms cannot guarantee deadlines will be met under high I/O load. People usually solve this problem by running time-critical operations on a separate CPU [20]. For example, high-frequency signal sampling may be implemented in bare metal code running on an independent microcontroller while the user interface, networking, storage, etc. runs on the main device. This approach has a lot of obvious shortcomings: increased hardware and software complexity, power consumption, physical size, verification difficulty, etc.

In this context, "real-time" refers to the schedulability of *user-level code*—the OS has no ability to schedule interrupt service routines triggered by asynchronous I/O events [19], which are managed by the CPU hardware and interrupt controller. The RTOS can reorder the processing of interrupt service routines, but it cannot prioritize user-mode code above ISRs; as we will see, it is this lack of flexibility that causes non-deterministic I/O latency.

Driver-level I/O processing has traditionally been assumed to be a negligible component of overall response time—an assumption that was valid 30 years ago as these real-time scheduling algorithms were being developed. At that time, embedded computers were single-purpose machines that largely performed the same task repetitively.

But that assumption of single-purposeness is becoming less valid. Modern microcontrollers are equipped with a rich set of peripherals that was unimaginable in the 1980s.Network interfaces, high-speed data acquisition devices, touchscreens, and more all have a diverse

[1] **H**yp**E**rvisor for **R**eal time **M**icrocontroll**E**r**S**
`http://hermes.wings.cs.wisc.edu`

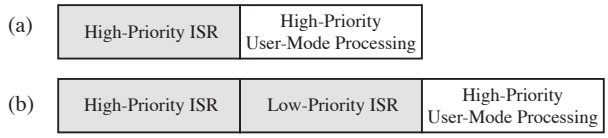

Figure 1: Timeline of (a) high-priority ISR followed by user-mode I/O processing and (b) low-priority ISR co-occurring with a high-priority ISR, delaying high-priority user mode processing.

range of requirements, but they are treated the same by the RTOS and CPU. Exception management for low-priority I/O is always performed before user-mode code can respond to high-priority events, creating a kind of unintended priority inversion (depicted in Figure 1). Consequently, response times to latency-sensitive I/O events are not deterministic, which can result in failure. We explored this problem in [15].

Conventional wisdom among real-time programmers is that ISRs should be as short as possible: clear the interrupt, maybe transfer a few bytes of data, and exit. Userland code should be responsible for responding to the event. In a crowded software environment with multiple drivers and tasks competing for CPU time, this programming method has the effect of delaying the actual response of all I/O events until all ISRs have finished executing. These delays break the assumptions that underlie real-time scheduling algorithms, which require the highest-priority task to always run first. Instead, we are running the driver code associated with low-priority tasks before the user code for high-priority tasks, and RTOSes do not have flexibility to change this behavior.

Hermes is a lightweight virtualization platform that lives between the hardware and the operating system. At its core, Hermes consists of some initialization code and a single interrupt service routine that catches and preprocesses all exceptions before dispatching them to the operating system. In its role as a mediator of exception processing, Hermes can allow multiple operating systems or bare metal applications to run side-by-side on an MMUless microcontroller, dispatching exception processing to the appropriate OS as necessary, much like a hypervisor running on a PC or server. We see several potential advantages to this software architecture:

(1) **Performance.** For time-critical applications, Hermes can provide a thin layer between the software and the hardware. Real time operating systems (RTOSes) on the other hand, often come with a lot of overhead in the form of system call latency for time-critical tasks. This may be unacceptable in applications where time-critical tasks need to coexist with other less critical code like networking or user interface software.
(2) **Privacy.** Because it lives between the hardware and the app, Hermes is in a unique position to intercept and anonymize personal data, coded in I/O transactions. We are exploring this topic in another line of work [16].
(3) **Portability.** Hermes can provide a consistent virtual environment for all higher level software, regardless of the underlying hardware. This could enable, for example, edge computing devices with heterogeneous hardware implementations to run user apps targeted to a common platform.

A diagram of the Hermes software architecture is shown in Figure 3. We are implementing Hermes on an ARM Cortex M7 CPU called the Atmel SAM E70 [6, 7] which has 2 Mbytes of flash and 384 kbytes of RAM. The ARM Cortex M7 core uses an instruction set called Thumb-2, which is a simplified version of the ARM instruction set, allowing most instructions to be encoded in 16 bits. Other Cortex-M cores (M3, M4, etc.) also use the Thumb-2 instruction set and have a similar programmer-visible architecture, which should make it possible to port Hermes to these other models. It also includes many advanced features of the latest ARM microcontrollers such as a floating point unit, a memory protection unit, separate instruction and data caches, and many peripherals. We have tested Hermes by running a FreeRTOS v9.0.0 [2] guest on top of the hypervisor. The contributions made by this work are the following:

- We describe in detail the implementation of a hypervisor built for real time software environments to improve responsiveness of real-time tasks.
- We discuss challenges of implementing a hypervisor on a hardware platform with minimal support for virtualization.
- We evaluate the performance of our hypervisor running multiple real-time tasks in a realistic mobile environment, comparing it to FreeRTOS.

2 BACKGROUND

Real-time operating systems have been around for a long time, and we have had a lot of opportunity to study their schedulers. One may wonder, *can we achieve the same results by modifying an RTOS?*

In principle, we could, but doing so makes the system much more difficult to program. Figure 2 outlines a sequence of events needed to implement our techniques in an RTOS. The key modification is that the ISR would need to disable the scheduler after disabling interrupts (between the first and second bubbles). This would prevent the RTOS scheduler from switching to a new task, which is the cause of the deadlock. The modification would need to be made to the tasks and the drivers, not the internal RTOS code. These modifications would be application specific—we would only make these modifications in the highest priority drivers and associated user code. There may need be some modifications to the way the interrupt controller is configured, but they would be relatively minor. We have not experimented with this.

The reason that we consider this a bad solution is that it requires programmers to observe extra rules to prevent deadlocks in their apps. Real time app programming already has specialized rules (eg. no API calls in high-priority ISRs, etc.) that are generally unknown to most non-real time programmers. This technique would increase the difficulty of writing real-time apps, increase the prevalence of bugs, and require programmers to have deeply specialized knowledge of issues around real-time systems programming. Most regular mobile programmers probably do not even know what real time programming is, so it would be unrealistic to expect them to have deep knowledge of the specialized techniques that it requires.

Furthermore, the RTOS solution would make the performance of each individual app mutually dependent on the set of other apps

with which it shares the CPU. As new apps are loaded to a device, the performance of real-time apps might be degraded.

By contrast, the hypervisor solution creates independent virtual runtime environments for each VM which are not affected by other VMs that share the CPU. Each VM in Hermes consists of a group of ISRs and user code. Hermes independently emulates the CPU's interrupt controller, system timer, and other hardware blocks for each VM. By contrast, RTOSes only keep track of the program counter and stack pointer for each running task. Because Hermes has independent control of the state of the interrupt controller for each VM, it can handle enabling and disabling interrupts as the CPU transitions from ISRs to user code, a job that would otherwise be delegated to the individual tasks. This allows us to write and test the code once in Hermes. App programmers do not need to worry about enabling and disabling the scheduler, which makes programming much more intuitive and reduces the possibility of introducing bugs.

We considered several simpler alternative solutions that could be implemented in the RTOS to alleviate I/O latency:

Use the interrupt controller to statically mask low-priority interrupts while executing real-time tasks. Figure 2 outlines the sequence of events needed to mask low-priority interrupts during a high-priority event. Before any exception has been raised, we want all interrupts to be enabled. We would only want to mask low-priority exceptions *after* a high-priority exception has been raised. Low-priority exceptions would then remain masked while the high-priority ISR is executing and until the associated user-mode code has finished.

This requires disabling interrupts in an ISR, processing the I/O event, and re-enabling interrupts in the user-mode task associated with that event. In this scenario, the critical section of code—the code during which interrupts are disabled—spans a return from exception. For this to work reliably, the ISR must return immediately to the associated user-mode task that collects and processes the data from the ISR. This is dangerous because after the return from exception, before executing user mode code, the RTOS will run the scheduler and possibly select a different user-mode task to run.

In fact, there are three points in this sequence of events where the RTOS's scheduler could select a different user-mode task to run[2]. If that happens, the different task may not be aware that the interrupt mask level that was set by the ISR, which may result in a deadlock. Hermes solves this problem by creating a truly isolated execution environment for every guest. Hermes tracks and emulates the CPU state for each guest, including the interrupt mask, interrupt controller, and many other peripherals. When a context switch occurs, Hermes restores the new guest's full emulated CPU state, not just the guest's program counter and stack pointer, as is done in an RTOS. This avoids the problem in Figure 2 where an incorrect CPU state set by an ISR can follow the execution stream into the wrong user-mode task.

Because we are emulating the CPU state independently for every guest on the system, there are fewer rules that programmers need to follow, and there is less possibility of adverse interaction among tasks.

[2]The FreeRTOS documentation instructs users not to call RTOS APIs inside critical sections to avoid deadlocks for this reason.

Disable interrupts in user-level code. This would allow us to process the I/O event in userspace without interruption. It does not solve the problem of ISR-userspace latency, since more than one exception may execute sequentially before user code gets a chance to disable interrupts.

Process I/O events in the ISR.. This would allow us to ensure that our I/O events are processed in a timely fashion. This could be an acceptable solution for a single-purpose bare-metal app with no other tasks running concurrently. The problem with this approach in an RTOS is that it monopolizes the CPU during the entire I/O operation, likely causing other tasks—even higher priority ones—to hang while the I/O event is handled.

Re-prioritize the interrupts. We could use the CPU's interrupt prioritization circuits to execute the time-critical ISR first, before other ISRs. This wouldn't decrease latency in an RTOS environment because lower priority ISRs will always execute before the user space code.

Disable low-priority interrupts inside high-priority ISRs. When we process a high-priority interrupt, we could disable all lower priority interrupts. This would cause the high-priority ISR to return directly to user-mode code without processing low-priority ISRs. This seems like a hack because it requires the driver code to be tightly coupled to the application: if threads are added, removed, or re-prioritized, the driver code would also need to be modified to reflect the changes in prioritization, likely resulting in bugs. The hypervisor model allows us to think about tasks independently without worrying about complex interactions between user-mode code and drivers.

None of these solutions is a viable alternative because they cannot reduce latency while maintaining a responsive runtime environment for other concurrent tasks.

3 ARCHITECTURE

Generally speaking, there are two classes of problems a hypervisor needs to solve in order to create a virtual runtime environment for its guests. First, it needs to emulate I/O operations in software such that guests think they have access to I/O ports and peripherals even though they do not. Second, it needs to emulate privileged instructions and memory accesses such that guests think they can execute privileged instructions even though they can't.

In both cases, guests should not be able to directly change the physical state of the CPU, for example by masking interrupts or putting the device in a low-power sleep state, because these operations could affect the execution of other guests. Instead, the hypervisor must emulate these privileged operations in software so they affect only one guest, not the state of the entire system.

Interrupt Hooking. Used to implement keystroke loggers in early PCs, interrupt hooking is a technique by which we modify an interrupt vector table entry for a particular I/O event, redirecting it from the operating system's interrupt handler to a different function called an interrupt hook. When the I/O event occurs, the interrupt hook function runs first, inspecting, modifying, or logging data generated by the I/O event. When it finishes, the interrupt hook then calls the operating system's interrupt handler which does

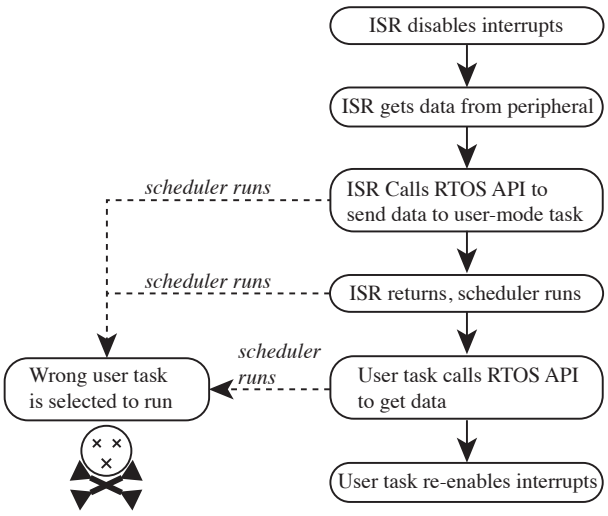

Figure 2: Sequence of events required in an RTOS to disable low-priority interrupts in a high-priority ISR. Interrupts would ostensibly be re-enabled in user mode after the I/O event has finished processing. However, the RTOS could transfer control to a different task, which could cause a deadlock.

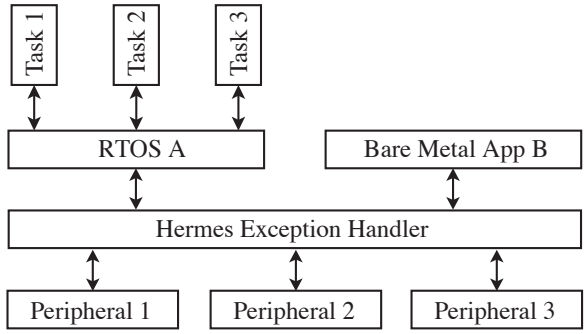

Figure 3: Architecture of the Hermes Hypervisor. The main component of Hermes, its monolithic exception handler, intercepts all exceptions before dispatching them to the guests.

not know that the interrupt hook ran ahead of it. This is the basic concept used by Hermes and other hypervisors to emulate I/O events.

Hermes is a single monolithic interrupt service routine that intercepts all CPU exceptions before they can be processed by the operating system. Figure 3 shows a diagram of the interactions between the Hermes hypervisor and its guests. On boot, the Hermes initialization code sets up the CPU's exception table to point to the Hermes ISR. It then launches the guest operating systems in the ARM CPU's unprivileged execution mode[3].

[3]Normally, operating system code would run in privileged execution mode, but when the RTOS is running as a guest inside Hermes, it executes in unprivileged mode.

Broadly, there are three kinds of exceptions that Hermes needs to handle:

Faults are exceptions caused by software running on the CPU. In general, faults are caused by privileged instructions or memory operations being run in unprivileged mode. When a guest causes a fault, it is a cue to Hermes that the guest's virtual execution state needs to be modified. For example, when a guest attempts to return from an exception, it causes a fault because the instruction it uses is privileged, and the guest is running in unprivileged mode. The Hermes exception handler will trap the fault and modify the guest's virtual state to indicate that the guest is running in virtual unprivileged mode. The different kinds of faults that Hermes handles are explained in detail below.

Direct I/O Interrupts are interrupts triggered by peripherals that are exclusively owned by one virtual machine. An example of this would be a hardware serial port. Hermes handles all direct I/O interrupts in the same way: by passing control to a hardware-specific ISR in the appropriate guest. There is no specialized code for individual direct I/O interrupts. Hermes handles these exceptions by setting up a virtualized exception stack frame for the guest which emulates the stack frame that would have been created by the hardware if the guest VM were running directly on the bare metal. It then returns the CPU state to unprivileged thread mode and passes control to the guest's interrupt service routine. The guest's ISR will then process the exception. As it does so, it will perform privileged operations such as accesses to privileged memory regions, execution of privileged instructions, and exception return. These privileged operations will be trapped by faults and emulated by Hermes.

Indirect I/O Interrupts are interrupts triggered by peripherals that may be shared among multiple guests. An example of this would be a bridged Ethernet interface where the NIC hardware can be shared by multiple guests with different virtual MAC addresses. These are different from direct I/O interrupts because the driver code for different peripherals must be inside the Hermes hypervisor. Indirect I/O interrupts are associated with peripheral devices that are emulated by the hypervisor, so the hypervisor must also store some virtual state information for each guest VM that uses the emulated peripheral. In general, Hermes will expose some virtualized hardware interface to the guest, which is either a full or simplified version of the underlying hardware peripheral's interface. When the guest modifies the virtual peripheral's state, Hermes will record those changes in its internal data structures and use the guest's settings in future interactions with the hardware peripheral.

A key difference between direct and indirect I/O interrupts is where the peripheral-specific driver code lives. For direct I/O exceptions, the driver code is part of the guest, and indirect I/O exceptions, the driver code is in the hypervisor. A consequence of this is that for shared peripherals using indirect I/O exceptions, the Hermes driver may have to significantly modify the hardware peripheral's settings to make the virtualized peripherals behave as expected from the guest's perspective.

Sleep: If a guest has nothing to do and needs to wait for an I/O event before it continues processing, it can put its virtual CPU into a sleep state using the standard ARM sleep instruction `wfe`. This is the same method an app would use to put the CPU into a low-power sleep state if it were running on the bare metal. When a

guest goes to sleep, the scheduler will be notified, and it will not be scheduled to run until it gets an I/O event from a peripheral that it owns. Other guests will then be allowed to use its share of the CPU. There are several types of faults that Hermes handles:

Privileged register access generates a (possibly imprecise) bus fault. These are normally associated with attempts to access peripherals. The Hermes bus fault handler either emulates the access in software (if supported) or executes the instruction directly. At the moment, we have only implemented emulation of a small subset of the privileged registers, most of which are related to critical functions like setting the vector table offset register or exception prioritization.

Privileged instruction generates a usage fault. These are associated with attempts to change the execution state of the processor (modifying stack pointers, exception masking, etc.). Hermes never directly executes privileged instructions. Instead, it decodes the instruction and modifies the guest's emulated state by updating fields in a data structure.

SVCall handler is a stub that just jumps to the guest's SVCall handler, which is executed in unprivileged mode.

I/O Exceptions handler is a stub that jumps to the guest's implementation of the handler for that exception, which is executed in unprivileged mode. Hermes looks up the address of the guest's handler in the guest's exception table. If the guest attempts to execute a privileged instruction or access a privileged memory region from within its exception handler, that action will trap to the Hermes exception handler so it can be emulated.

Guest return from exception creates a memory management fault. Hermes catches the fault and returns execution to the guest's user mode code.

3.1 Establishing Guest Priorities

The ARM Cortex-M interrupt controller includes an interrupt priority mask register called `BASEPRI` that can disable interrupt sources lower than a given priority. The mask register is normally set by an RTOS to enter/exit critical code sections. Hermes does not allow guests to directly set the architectural `BASEPRI` register—that is a privileged operation that Hermes emulates in software. Instead, it maintains a virtual `BASEPRI` register for each guest which is used as the guest's priority.

If a guest elevates its `BASEPRI` register to disable certain interrupt sources, Hermes elevates that guest's priority among the pool of currently active guests. At any given time, the guest with the highest `BASEPRI` value will be given access to the CPU. Under some circumstances, Hermes may also set the architectural `BASEPRI` register to the value configured by the guest, disabling interrupt sources for low-priority guests. This is how we establish virtual isolation among guests—by disabling low-priority interrupt sources and maintaining separate states for the virtual interrupt controller in each guest. In contrast, operating systems—real time or otherwise—do not maintain separate virtual hardware states for different tasks or processes. Instead, OSes prefer to provide programmers with APIs and drivers to interact with the physical hardware. Nothing in principle prevents them from virtualizing memory or I/O in the way that hypervisors do. It's just that our custom is to c all runtime environments that manage separate virtual state for hardware peripherals hypervisors and runtime environments that do not manage virtual state for peripherals operating systems. So the question about whether we can modify the operating system to get the real-time behavior that we want is really about the architectural and philosophical differences in implementation between these two runtime environments. Probably if we redesigned operating systems today from the ground up using modern hardware and software, we would end up with a design somewhere between an operating system and a hypervisor: more virtual state than an OS, but more handholding for I/O (like drivers) than a hypervisor.

3.2 Opportunities

Running a hypervisor on embedded IoT equipment enables some interesting possibilities for IoT software.

3.2.1 Distributed Processing on a Single Chip. Many embedded hardware designs use a distributed computation model to separate a complex task into several independent execution environments. For example, a board might have one network processor, one sampling processor, and a main CPU, each performing its own specific task independently of the others. This type of design complicates the hardware and software and likely drives up the cost, size, and energy requirements of the equipment. With a hypervisor, we can run all software on a single CPU while maintaining isolation by running each independent application in its own VM. CPU and resource allocation can be strictly controlled by the hypervisor to ensure that deadlines are met.

3.2.2 Security and Privacy. Mobile device privacy is an interesting area of research in which we try to limit the amount of personally identifying information that users divulge about themselves in the course of using mobile and IoT apps. In a parallel line of investigation, we have used Hermes as a platform for capturing information about users in mobile apps that is collected from sensors like video cameras and microphones on board mobile and IoT devices [16]. The sensor data streams captured by Hermes can be processed by anonymization software in real time before being handed off to the appropriate app. The advantage of operating system and hypervisor-level techniques over the more common network middlebox approach [8, 12] is that the sensor data is unencrypted at the hardware level. A hypervisor-based privacy agent does not need to worry about decrypting and SSL stream to inspect or modify its content.

Authenticating the software on an unattended embedded device is still an open problem. A few proposed solutions [4, 22] rely on measuring the timing of some arbitrary computational operation. The hypervisor may be able to serve as a root of trust for virtualized applications by implementing a virtualized trusted platform module (vTPM) [1] to be used by underlying software components. It may be possible to implement a virtual TPM in software using either ARM TrustZone [23] or an on-chip cryptographic accelerator [13].

3.2.3 Loadable Apps. Apps written by users or third parties could be easily selected from an app store and run natively on the IoT device, allowing flexibility to the end user without overloading the cloud services. A major challenge for this model of app distribution is that systems without an MMU must have all their code compiled together before runtime. Third party apps, when loaded

dynamically, could overwrite each other's memory regions during execution. Furthermore, allowing third party apps to run on an IoT platform creates software versioning headaches because we must make sure that the OS and user app share the same library and system call implementation: when we upgrade one, we must upgrade all.

A hypervisor could solve both problems by running each user app inside an independent virtual machine. When we need to context switch away from one user app, we can snapshot it into a peripheral memory device (external SRAM or flash) and load a different app into the same memory space. This would allow us to run multiple apps with a single virtual memory space without the use of an MMU. From the app's perspective, it would be running on an unshared virtual machine, and it would not need to share common drivers with the RTOS, making software compatibility much simpler.

3.3 Challenges

By running a hypervisor on a Cortex-M CPU, we are using the device in a way that was not intended by its designers. Emulation of privileged instructions and memory regions, I/O, etc. exposes some interesting features, optimizations, and design decisions in the CPU that programmers would not normally encounter.

3.3.1 Compile-Time Guest Setup. Since we are dealing with a system that has no MMU, we are required to compile all guests with the hypervisor into a single runtime binary. The practical challenge is that, for symmetric guests (more than one instance of a single guest OS), we must change the name of each function and variable in order to avoid linker errors. This can be mildly annoying because it makes the RTOS code harder to read. We have written a script to perform this task automatically.

3.3.2 Imprecise Bus Fault Exceptions. The ARM Cortex M line of CPUs throws bus fault exceptions for accesses to privileged memory regions that are mapped to certain control registers. Some of these exceptions can be imprecise, meaning that the CPU does not record the exact instruction that caused the exception. Instead, it will throw a bus fault as soon as possible (in our experience 2-10 instructions past the faulting instruction). This makes the job of the Hermes exception handler difficult since it does not know which privileged memory access needs to be emulated. The only thing we can be sure of is that the faulting instruction occurs earlier in the instruction stream than where the exception was thrown.

We solve the problem of imprecise bus fault exceptions by tracing back through the instruction stream to look for a privileged instruction with the correct effective address that is likely to have caused the imprecise exception. Starting at the address of the instruction that caused the exception, we trace back through the last five instructions in order of memory address. We decode each instruction and compare its effective address to the address that caused the bus fault. If the instruction's effective address matches the offending effective address, then we assume a match and emulate that instruction.

If we do not find a bus fault address match in the last five instructions before the exception was thrown, we examine the same five instructions again, this time looking for any instruction that could have caused a bus fault. We do so by examining the effective address of each instruction and determining if it corresponds to a privileged memory region. We assume the most recently executed instruction that accessed a privileged memory region was the one that caused the bus fault.

Instruction	Function
`mrs`, `msr`	Read and write to special-purpose registers that control the execution state of the CPU. These instructions are used by the OS to set the supervisor and user stacks, change between supervisor and user mode, etc.
`wfe`, `wfi`	Put CPU to sleep and wait for interrupt.
`cpsie`, `cpsid`	Enable/disable interrupts.

Table 1: List of privileged instructions that do not cause privilege violations when executed in unprivileged mode.

Clearly, there are some pathological cases that could cause this approach to fail. For instance, consider the following instruction sequence for updating the value of a privileged hardware I/O register, executed in unprivileged mode:

```
  ld   r1,=privilegedAddress
* ld   r0, [r1]
  add  r0, r0, #4
* st   r0, [r1]
```

In this instruction sequence, the `ld` and `st` instructions marked with a * will each generate a bus fault because they access a privileged memory region. The bus fault may be imprecise, and it may be delayed by up to ten instructions past the offending instruction. This means that the CPU may not take an exception for the `ld` instruction until several instructions past the `st`. Since both instructions have the same effective address, we won't be able to trace back through the instruction stream and distinguish the two instructions based on effective address only. Furthermore, by the time the exception is taken, the value in register `r1` may have been changed by subsequent instructions, making it impossible to identify the effective address of the offending instructions.

Unfortunately, this instruction pattern is fairly common in code that configures or modifies the settings of peripherals or I/O devices. Our solution to this problem is to add pipeline synchronization instructions after each privileged memory access, which forces all in-flight instructions to complete before proceeding. This forces any exceptions caused by in-flight instructions to be taken before proceeding, making it possible for Hermes to identify the exact instruction responsible for the privileged memory access. Hand-patching OS kernels is a manual and painful process. A pre-patched version of FreeRTOS is available on the Hermes website. We have not yet developed any automated tools to search for privileged memory accesses in the OS source code and automatically add pipeline synchronization instructions in the right places, although it would in principle be possible to do so.

So far, we have not encountered any code in a guest that causes this approach to fail to emulate the guest.

3.3.3 Some Privileged Instructions Don't Cause Exceptions When Executed in Unprivileged Mode. Some privileged instructions on the ARM Cortex M7 do not cause privilege violation exceptions when executed by the guest. A list of some privileged instructions that we know do not cause exceptions when executed in unprivileged mode is shown in Table 1. This is a challenge of not having hardware support for a hypervisor. For example, the `mrs` and `msr` instructions are classified as privileged instructions, but when they are executed by code running in unprivileged mode, they fail silently: the register write is not committed, and the processor continues normal execution. Oddly, whether or not we are allowed to execute `mrs` and `msr` instructions seems to depend not on the privilege mode of execution, but on the stack pointer we are using. If we are using the Master Stack Pointer (MSP), we can execute `mrs` and `msr`, but if we are using the Process Stack Pointer (PSP) we cannot. Usually (but not always) the MSP is used when executing privileged code and the PSP is used when executing unprivileged code.

The problem is that if a guest OS tries to modify the processor state with one of these privileged instructions, that state modification cannot be registered by Hermes since it does not cause an exception. The privileged instruction will complete like a `nop` instruction without modifying the CPU state. Critical CPU state changes like disabling interrupts will not work as intended.

3.3.4 FreeRTOS. FreeRTOS is a popular (if not the most popular) real-time operating system for embedded and IoT computers. It has been ported to CPUs manufactured by 20+ manufacturers representing every commonly used architecture (ARM, x86, etc.). FreeRTOS implements a rate monotonic scheduler in which each task has a fixed priority, and the highest priority task that is ready to run is executed first.

FreeRTOS and others like it impose strict rules about how user mode software can call the RTOS API in order to avoid deadlocks such as in (1). For example, no RTOS API calls can be made within high-priority ISRs, presumably to maintain responsiveness for the rest of the system. This further limits the scope of what we can do with the RTOS.

3.3.5 ARM Exception Handling. The ARM Cortex-M CPUs include an interrupt controller that allows software to prioritize interrupts. Each interrupt can be assigned a priority between 0 and 255 (higher priority values are associated with higher priority). ISRs associated with low-priority interrupts can be preempted by ISRs for high-priority interrupts.

Software running on the ARM device can also mask exceptions below a desired priority value by setting a core register called `BASEPRI`. When an interrupt occurs, its priority is compared with the value in `BASEPRI`, and its ISR is only executed if the interrupt priority is greater than the `BASEPRI`. Otherwise, the ISR is delayed until the value in `BASEPRI` is reduced below the interrupt's assigned priority. Changing the value in BASEPRI is done by executing a privileged `msr` instruction.

We circumvent this problem by patching the OS kernel, adding an undefined instruction immediately following an `mrs` or `msr`. When the hypervisor encounters an undefined instruction exception, it will search backward in the instruction stream for an `mrs` or `msr` instruction and emulate it. If we run an unpatched kernel inside the hypervisor, it will crash because the intended CPU state modifications will not happen as intended.

In FreeRTOS, there is a total of 38 assembly language instructions that need to be added in order to make the OS run as a guest in the hypervisor. All of the additional instructions are in FreeRTOS's chip-specific code that deals with timer interrupts and scheduling. We have posted a patched version of the FreeRTOS v9.0.0 kernel on the project website.

3.3.6 Online Instruction Decode. Because some privileged memory accesses cause imprecise exceptions, it was necessary for us to identify the likely source of an imprecise bus fault by sequentially decoding instructions immediately prior to the bus error exception and searching for one with a matching effective address. Our instruction decoding code, works to correctly execute a guest running FreeRTOS v9.0.0.

However, this approach is slow, and there is probably a better solution.

3.3.7 We Invalidate I$ for Every Privileged Instruction. The way Hermes handles un-emulated privileged instructions is by copying the instruction from the guest's instruction stream into a subroutine that is called by Hermes. Before we call the subroutine, we must invalidate the CPU's instruction cache because it contains stale data from before the instruction was copied to it.

The ARM Cortex M7 core does not allow us to selectively invalidate lines in the instruction cache. In order to clear the stale data from the instruction cache, we need to invalidate the entire cache, which causes a huge slowdown in execution.

Privileged instruction execution is mostly associated with I/O events, and most privileged instructions appear at boot time when peripherals are being initialized. This could be a problem if we try to emulate an I/O device that requires a lot of reads and writes to privileged memory regions, as illustrated by the study of I/O performance in Table 4 in Section 4.

4 EVALUATION

4.1 Problem Validation

Using performance counters on the ARM Cortex M7 CPU, we measure the ISR-user space latency—the time between beginning of ISR execution to beginning of userspace data processing. This is a metric of how long it takes to respond to an I/O event. Ideally, for time sensitive I/O this time should be short and deterministic, meaning the same for each I/O event. We find that in the FreeRTOS environment, the ISR-user space latency is **less deterministic** under high I/O load, as expected. We have also experienced this problem when developing other systems, but we did not study it as carefully [17].

4.1.1 Experimental Setup. We measured the ISR-userspace latency for a serial port receive in FreeRTOS and Hermes. In FreeRTOS, we used an OS queue to transfer the data from an ISR to a user-mode task. In Hermes, we ran a FreeRTOS guest alongside a bare metal guest that transferred data between an ISR and userspace code using a memory buffer. In both runtime environments, we had two other periodic FreeRTOS tasks running alongside the latency test. In both cases, we ran the latency test in isolation as well as in the presence

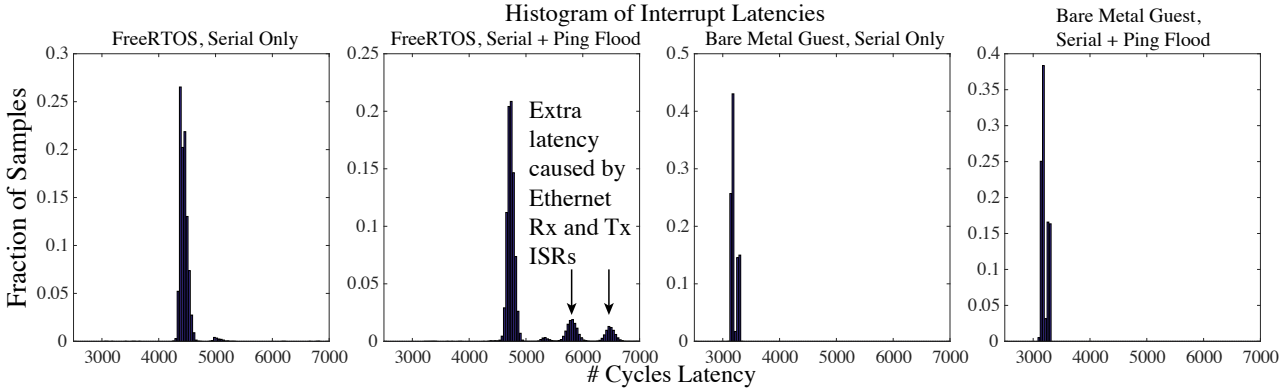

Figure 4: ISR-userspace latency histograms. Latency is measured as the number of cycles elapsed between executing the serial port receive ISR and beginning of userspace processing.

Software Environment	Entropy of Latency
FreeRTOS, Serial Only	2.73
FreeRTOS, Serial + Ping Flood	3.65
Bare Metal Guest, Serial Only	1.94
Bare Metal Guest, Serial + Ping Flood	2.08

Table 2: Entropy of the distributions of latency measurements (distributions shown in Figure 4). Low values of entropy are more deterministic. The bare metal guest running in Hermes has much more predictable latency than tasks in FreeRTOS. Under Hermes, latency is still highly deterministic under high I/O load.

of high I/O load (a ping flood) to test how well each software environment could provide a deterministic runtime environment. The networking software that responded to the pings was implemented as a low-priority task in FreeRTOS for both environments.

4.1.2 Results. Figure 4 shows the results of our latency tests. Each subplot is a histogram of ISR-userspace latencies. Ideally, we would want these plots to have only one bar—a single response time for every I/O event. Figure 4 (a) and (b) show latency in FreeRTOS only, under low and high I/O load respectively. Under high I/O load, the latency histogram is more spread out because exceptions raised by unrelated I/O events delay execution of the user mode code in an unpredictable way. This happens when a serial port exception and a network port exception occur close in time. Both exceptions must be processed before the user mode code to handle the serial port receive can begin executing. We get shorter and more deterministic response times when the serial port exception occurs in isolation. If the network port exception occurs near the same time as the serial port exception, the network port ISR will have to execute before the CPU can return to user mode, delaying the response time. This is an inherent disadvantage of running multiple unrelated programs on a single processor which we are trying to correct with Hermes.

Figure 4 (c) and (d) show the latency of the same I/O operation running as a bare-metal guest inside Hermes. Determinism is higher for histograms that are more clustered around a single value and lower for histograms that are more spread out.

4.1.3 Discussion. The reason that ISR-userspace latency is more deterministic in Hermes under high I/O load is that by design, Hermes can enable or disable different interrupt sources depending on which guest is active. In this test, we disabled the network port exception when the bare-metal serial port guest was running. This makes it impossible for the network port ISR to interrupt the user-mode code that handles the serial port receive. Operating systems in general do not support changing processor state for different threads[4], presumably because I/O transactions are assumed to be the domain of the operating system and mostly independent of user-mode software. That assumption was generally valid for early PCs and servers, whose job was primarily batch-mode processing with very little user interaction. Mobile and IoT devices have completely different set of requirements: they need to serve as a responsive user interface in which software works closely with I/O.

Uncertainty in scheduling can create real problems for these kinds of systems. For instance, if the same timing uncertainty in Figure 4 were imposed on ADC sampling in an IoT device, it could cause several decibels of harmonic distortion [3]. It is easy to imagine many situations in which timing errors could result in degraded system performance on mobile platforms.

The results in Figure 4 are an improvement to [15] which had an incomplete implementation of guest context switching in response to interrupts. At boot, we initialize an array of data structures containing (1) an interrupt priority and (2) a pointer to a guest that owns the interrupt. The interrupt number is implied by the position in the array. We also configure the ARM's interrupt controller with the same interrupt priorities as the array. When an interrupt occurs, it will be masked by the interrupt controller unless the BASEPRI register has a smaller mask value than the interrupt's priority. Interrupts with low priority are associated with less time-critical tasks.

If the interrupt has a higher priority than the BASEPRI, its priority is inserted into the BASEPRI so its handler cannot be preempted by lower-priority interrupts. The associated guest is then given control of the CPU, and the guest's ISR is executed. This method is

[4]For example, we are not aware of any RTOS that allows the programmer to enable or disable different drivers while certain threads are running.

a generalization of the one presented in [15], but it is more flexible for multiple exceptions.

In this method, when a high-priority exception occurs, the corresponding guest will immediately be given access to the CPU, regardless of what other guests or ISRs are currently executing. The user-mode code for that guest will have the opportunity to respond to the interrupt without waiting for other lower priority exceptions to finish.

The main goal of the Hermes hypervisor is to provide a thinner layer between hardware and software than is possible with an RTOS. There are three general techniques for virtualizing I/O:

4.2 Mobile Device Use Case

To demonstrate how Hermes would behave in a real deployment scenario, we built a prototype handheld device based on the SAME70 Xplained development board, and LCD touchscreen, and a GPS receiver, and camera. Our demonstration platform is meant to mimic a smart watch or other low-power mobile device.

Using the GPS receiver and LCD screen, we developed an app that tracks the user's location in real time and computes the speed based on successive measurements. We developed a simple service to display the current speed on the LCD and display a moving sprite-based background image. The moving background image is intended to mimic the user interface (such as a moving map) that such an app might display. We also implemented an app that handles network services to mimic a WiFi or Bluetooth interface that may be needed to send or receive application data (emails, map updates, etc.).

We also developed a video app that gathers frames from the video camera, postprocesses each frame in real time, and displays the frame to the user. Our postprocessing consists only of simple color correction since the embedded CPU is not capable of running sophisticated workloads like face recognition in real time. Still, the real-time color correction is near the limit of the CPU's capacity, and we found that additional workload such as a large volume of incoming network traffic put the system over its capacity, creating performance problems in an RTOS environment.

We ported these apps to the FreeRTOS and Hermes runtime environments to compare their performance. We focus on evaluating how real-time data from the GPS module is handled in both environments and how I/O latencies can affect the GPS apps can affect computation of instantaneous speed in the presence of a realistic mobile device workload.

4.2.1 Video App. The board we use has a VGA image sensor interface that can accept images in 16-bit RGB color format at a rate of about 10 frames per second. Our video camera app has three major components: (1) a VGA interrupt service routine, (2) image data processing userland code (3) display ISR. All components are connected by 12-frame queues. Since the image processing code is a CPU hog, we do not want to run it in the VGA ISR. However, it must be run at high priority since it is in the critical path for the user interface. During the time that the video app is open—which we imagine would be relatively infrequent in our mobile device—the camera should be very responsive. The video app is a CPU-intensive real-time app that runs near our device's compute capacity.

We implemented the app inside FreeRTOS and Hermes. In each runtime environment, we tested the app in two scenarios—under low I/O load and under a ping flood to simulate a high volume of network traffic. In the FreeRTOS implementation, we were not able to use the standard queue API to pass data between user code and interrupt code because high-priority ISRs are prohibited from making API calls. Instead, we implemented a custom queue outside of the FreeRTOS API. Normally the RTOS would schedule the userland image processing thread if there was data in the queue. But since our queue was outside the RTOS, we could not just block the task pending incoming data. Instead, we put the task to sleep for a short amount of time and poll the queue periodically. If the queue fills up rapidly before the userland code can poll, it can fill up and drop a lot of frames. This happens in Figure 5. To allow the video app to catch up, we had to manually kill the ping flood and allow the userland code to catch up.

We used the same queue in the Hermes implementation without frame loss. In the Hermes implementation, when the VGA ISR starts, the whole video app takes control of the CPU, including the userland code. As soon as the ISR returns, the userland code begins reading data from the queue without interruption from the networking app. In Hermes, related ISRs and user code are logically grouped together into virtual machines that run as independent units.

We used frame jitter and loss rate as metrics of performance. In both cases, we saw no frame losses under low I/O load. Figure 5 shows the number of frames dropped every second during a 276-second test of the FreeRTOS video app. **Under high I/O load, Hermes lost no frames, and the jitter did not increase** because the hypervisor prioritizes userland video frame processing over lower-priority networking interrupts. In fact, our video processing software represents a relatively mild test case—we were able to cause much more dramatic frame jitter and loss rate in FreeRTOS by increasing the complexity of the video processing algorithm. Table 3 shows a performance comparison between Hermes and FreeRTOS for the video app under low and high I/O load conditions. Figure 6 shows histograms of the inter-frame timing for the same test cases.

The real-time app performs much better in the Hermes runtime environment than in the RTOS, but we can't get something for nothing. The interesting feature of the video app is that it has real time requirements and generates high CPU load. When Hermes prioritizes the video app, it starves the networking app of CPU time, resulting in extremely high (97%) packet loss rates for the ping flood (lower-speed pings were still lossless). Hermes gives programmers a knob to turn to trade real-time performance for fairness. We can dynamically adjust the VM priorities in real time to achieve the desired middle ground at run time.

4.2.2 GPS App. The GPS module generates location estimates approximately once per second. To compute speed, the GPS app computes the distance between successive GPS coordinates and divides by one second. In our app, we only use the GPS to generate location estimates. We do not rely on its clock in our speed calculation. Instead, we use a clock internal to the ARM microcontroller to keep track of time. We do not want to treat the GPS module as an independent off-chip coprocessor. One of the goals of Hermes is

	Avg Frame Loss Rate	Inter-Frame Jitter (σ)
FreeRTOS, Low I/O Load	0 frames/sec	7.03 ms
FreeRTOS, High I/O Load	4.2 FPS	47.91 ms
Hermes, Low I/O Load	0 FPS	4.85 ms
Hermes, High I/O Load	0 FPS	4.86 ms

Table 3: Performance comparison of Hermesand FreeRTOS in the video app. Low frame loss rate and jitter is better.

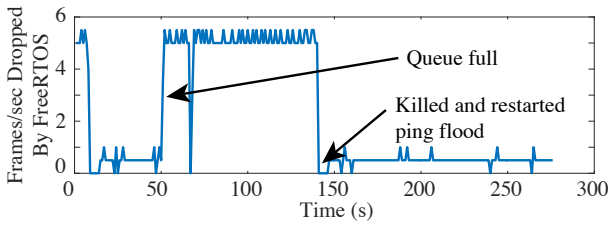

Figure 5: Number of frames dropped by FreeRTOS per second as a function of time while camera was running in the presence of a ping flood.

to allow us to put as much of the computational load as possible on a single centralized CPU rather than dispatching sub-tasks (like speed computations) to single-purpose off-chip devices.

In the speed computation, there are three main sources of error: GPS location error, cruise control error, and I/O latency errors in the ARM CPU. We model each of these as additive white Gaussian noise. Since we did not have access to a digital readout of the car's speedometer, we did not have access to a reliable ground truth to compare our results. Instead, we will compare the variance of speed estimates.

To benchmark the two different runtime environments, we set our mobile device up in a car and drove down a flat, straight stretch of highway with the cruise control set to about 67 miles per hour. Our mobile app's speed computations were logged to a file via the board's serial port. Histograms of our app's speed computations in the FreeRTOS and Hermes runtime environments are plotted in Figure 7. For comparison, we have also plotted speed estimates using the GPS's internal timebase.

We are mostly interested in the variance of speed estimates, which is caused by uncertainty in the I/O event arrival time for the GPS data and the internal clock. Like the problem validation in Section 4.1, uncertainty is caused by competing I/O events from other apps running on the system—in this case from a more realistic workload. Each of the trials show in Figure 7 was acquired separately, which resulted in a slightly different average speed for each trial.

The GPS-only plot tells us about how much of the speed error is coming from errors in the GPS and the cruise control. The remainder of the variance in the FreeRTOS and Hermes histograms comes from errors in I/O event synchronization and jitter in the ARM CPU. Hermes has about 46% less variance than FreeRTOS after controlling for errors from the GPS.

Bare Metal (no emulation)	0.1 ms
Bridged (partial emulation)	0.3 ms
Passthrough	1 ms

Table 4: Comparison of ping round trip times in Hermes for three Ethernet driver implementations on the ARM device.

Bare Metal (no emulation)	1675 kbytes/sec
Passthrough	12 kbytes/sec

Table 5: Comparison of SD card write throughput for three driver implementations on the ARM device.

4.2.3 Modifying the FreeRTOS Task for Hermes. API Calls need to be converted from FreeRTOS to Atmel Software Framework (ASF), which is a board support library of drivers for bare-metal programs. Different CPUs or boards will use different board support libraries. Our FreeRTOS implementation already was using the ASF library for most of the drivers since it does not provide chip-specific drivers. For the most part, ASF drivers could be used out of the box without modification.

One exception is `vTaskDelay`, which is equivalent to `sleep` in Linux. `vTaskDelay` causes the current task to become inactive for a period of time and allowing other tasks to run. It is tightly coupled to the OS's scheduler. Bare-metal apps—which run as guests in Hermes—use an ASF library function called `Wait` which puts the CPU into a low-power sleep state. When converting our app to run in Hermes, we need to change all calls from `vTaskDelay` to `Wait`. Also in cases where we modified ASF drivers to be compatible with FreeRTOS, we need to revert to the stock ASF driver.

4.3 I/O Benchmarking

We evaluated three techniques for virtualizing I/O operations:

- **Passthrough** uses interrupt and DMA remapping to give guests direct access to hardware resources.
- **Partial emulation** implements a reduced-function virtual hardware device with a custom device driver for the guest.
- **Full emulation** implements full emulation of the physical hardware device, including the full complement of registers, FIFOs, etc available on the hardware.

In this work, we studied passthrough and partial emulation, using the board's Ethernet and SD card interfaces as target I/O devices. The network driver is convenient because it is easy to benchmark using ICMP echoes (pings), and it's easy to compare to other virtualization platforms. We've benchmarked multiple I/O interfaces to see if there are any differences in performance.

4.3.1 Ethernet Interface Benchmarking. Table 4 shows a comparison of round trip times for three different Ethernet driver implementations. The bare metal implementation is the unmodified driver supplied by the chip manufacturer with no virtualization; it is our reference implementation.

The bridged implementation is a custom driver running in the guest. Ethernet device interrupts are handled by Hermes without being passed up to the guest. The hypervisor presents a virtualized network interface to the guest, and they hypervisor calls the chip

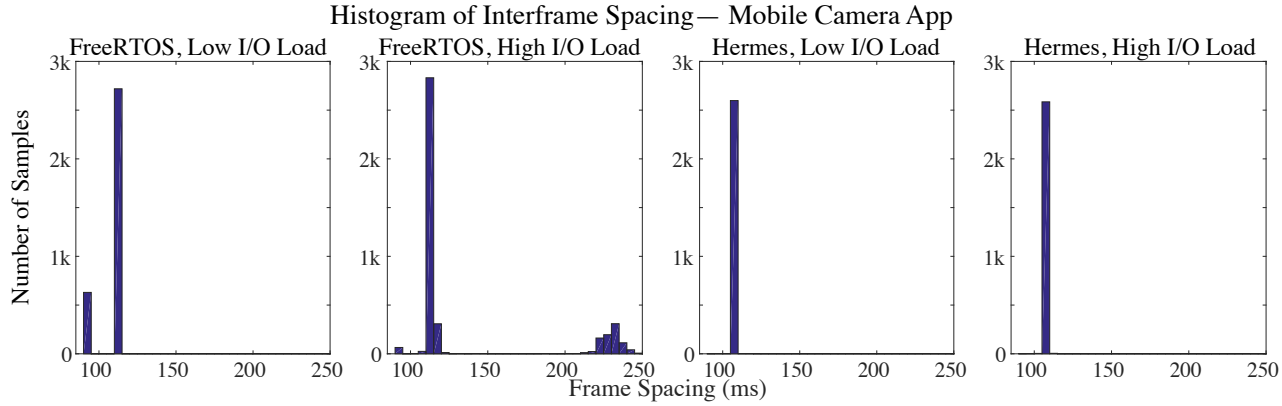

Figure 6: Histograms of inter-frame spacing for the video app. Standard deviation of these histograms are *jitter* in Table 3.

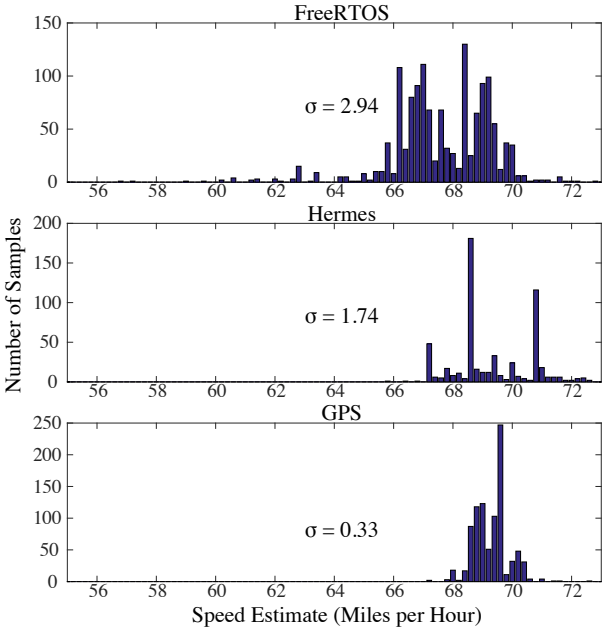

Figure 7: Histograms of speed estimates by our GPS app.

manufacturer's driver functions to send and receive packets. The bridged driver allows multiple guests to share the same network interface by multiplexing incoming packets to the guests based on MAC address.

In the passthrough implementation, the guest runs the manufacturer's driver in raw form, emulated by Hermes. Ethernet device interrupts are caught by Hermes and passed to the guest, so all exception handling code is done in guest mode. The Ethernet device is not shared among multiple guests in this configuration.

Surprisingly, we find that the bridged (hypervisor-assisted) Ethernet driver performs far better than the passthrough. Since the passthrough driver runs all driver code in guest mode, all privileged instructions must be emulated by Hermes. This causes a significant slowdown in packet handling because the Ethernet driver has to invalidate a lot of data cache lines each time a packet is received, which requires many privileged instructions and memory accesses. In the bridged driver, the majority of privileged memory accesses and privileged instructions are done by the hypervisor, so they don't need to be emulated.

4.3.2 SD Card Benchmarking. Table 5 shows a comparison of IO throughput different SD card driver implementations. The reason for the poor passthrough driver performance is that in the SD card write benchmark, the bottleneck of the implementation is in transferring a large volume of data through the ARM device's DMA interface. Each time we perform a block write to the SD card, the DMA interface invalidates about 1k lines in the data cache. The cache line invalidation operation is privileged, so it must be emulated by the hypervisor. If we do not invalidate those data cache lines during the block write, we can get a 3x performance improvement[5]. There are several other privileged bulk data transfer operations in the SD card write benchmark that create similar slowdowns. By comparison, the Ethernet driver needs to only transfer a small amount of data to respond to an ICMP echo.

5 RELATED WORK

Other authors have explored real-time schedulers in hypervisors, in particular for Linux running in Xen [14, 25, 26]. Nemesis [18], an operating system that provided soft real-time guarantees for video processing, was aimed to solve a similar problem under similar assumptions, but it did not assume hard deadlines. There has also been work done porting [11] and evaluating [10] the KVM hypervisor on the ARM Cortex-A core, which is an application processor with a full MMU. None that we know of have been implemented on MMUless machines—even early hypervisors ran on machines with memory management units [9].

6 CONCLUSION

We have demonstrated that Hermes can improve the responsiveness of real-time I/O operations by creating separate virtual execution environments for each task on a real-time system. We arrived at our implementation by thinking at a high level about what is required

[5] Of course, by skipping the data cache invalidation, the correct data will not be written to the SD card, but that does not matter since we are just writing random numbers for the benchmark.

to make the time-critical software as responsive as possible. In particular, we found that traditional real-time operating systems had an important shortcoming: they always prioritize interrupt handlers above user-mode processing, a vestige of early time-sharing mainframes that has the unintended consequence of occasionally running parts of a low-priority task before higher priority tasks. No amount of reconfiguration in the hardware interrupt controller can make this problem go away.

Instead, Hermes creates a fully-isolated virtual execution environment for each of its guests. The most notable difference between a hypervisor and and RTOS is that the hypervisor can create a consistent virtualized CPU state for each guest that can be safely modified by the guest without affecting the other guests. The hypervisor's virtual CPU state can include things like the configuration of the interrupt controller, peripherals, etc. By contrast, RTOSes only independently track the program counter and stack pointer for each task. Except for those two registers, RTOS tasks cannot modify CPU state without affecting other tasks. Doing so could cause the entire system to crash.

Our experiments revealed that I/O performance can be degraded—sometimes a little, sometimes a lot—by emulation in Hermes. The question of whether or not that reduced I/O performance is acceptable is largely application-dependent. In this work, we made a high-level evaluation of the performance implications for I/O devices, and we will leave a more detailed analysis for future work. Also left for future work is an analysis of how competition among multiple real-time tasks will affect determinism in event responses. In the mean time, we have made the source code for Hermes freely available for anyone to download.

7 ACKNOWLEDGMENTS

We would like to acknowledge the anonymous reviewers. The authors were supported in part by the US National Science Foundation through grants CNS-1719336, CNS-1647152, CNS-1629833, and CNS-1343363.

REFERENCES

[1] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 21. http://dl.acm.org/citation.cfm?id=1267336.1267357

[2] Richard Berry. 2017. FreeRTOS. (2017). http://www.freertos.org.

[3] Brad Brannon and Allen Barlow. 2006. Aperture uncertainty and ADC system performance. *Application Note AN501* (2006).

[4] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, New York, NY, USA, 400–409. https://doi.org/10.1145/1653662.1653711

[5] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 565–578. https://www.usenix.org/conference/atc16/technical-sessions/presentation/cho

[6] Atmel Corporation. 2017. SAM E ARM Cortex-M7 Microcontrollers. (2017). http://www.atmel.com/products/microcontrollers/arm/sam-e.aspx.

[7] Atmel Corporation. 2017. SAM E70 Xplained Evaluation Kit. (2017). http://www.atmel.com/tools/atsame70-xpld.aspx.

[8] Andy Crabtree, Tom Lodge, James Colley, Chris Greenhalgh, Kevin Glover, Hamed Haddadi, Yousef Amar, Richard Mortier, Qi Li, John Moore, Liang Wang, Poonam Yadav, Jianxin Zhao, Anthony Brown, Lachlan Urquhart, and Derek McAuley. 2018. Building accountability into the Internet of Things: the IoT Databox model. *Journal of Reliable Intelligent Environments* 4, 1 (01 Apr 2018), 39–55. https://doi.org/10.1007/s40860-018-0054-5

[9] R. J. Creasy. 1981. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.* 25, 5 (Sept. 1981), 483–490. https://doi.org/10.1147/rd.255.0483

[10] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM Virtualization: Performance and Architectural Implications. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 304–316. https://doi.org/10.1145/3007787.3001169

[11] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 333–348. https://doi.org/10.1145/2541940.2541946

[12] Nigel Davies, Nina Taft, Mahadev Satyanarayanan, Sarah Clinch, and Brandon Amos. 2016. Privacy Mediators: Helping IoT Cross the Chasm. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile '16)*. ACM, New York, NY, USA, 39–44. https://doi.org/10.1145/2873587.2873600

[13] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. 2001. Building the IBM 4758 Secure Coprocessor. *Computer* 34, 10 (Oct. 2001), 57–66. https://doi.org/10.1109/2.955100

[14] Marisol GarcÃĆÂŋa-Valls, Tommaso Cucinotta, and Chenyang Lu. 2014. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture* 60, 9 (2014), 726 – 740. https://doi.org/10.1016/j.sysarc.2014.07.004

[15] Neil Klingensmith and Suman Banerjee. 2018. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications (HotMobile '18)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3032970.3032973

[16] Neil Klingensmith and Suman Banerjee. 2018. A Hypervisor-Based Privacy Agent for Mobile and IoT Systems. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications (HotMobile '19)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3301293.3302356

[17] Neil Klingensmith, Dale Willis, and Suman Banerjee. 2013. A Distributed Energy Monitoring and Analytics Platform and Its Use Cases. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings (BuildSys'13)*. ACM, New York, NY, USA, Article 36, 2 pages. https://doi.org/10.1145/2528282.2534156

[18] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J.Sel. A. Commun.* 14, 7 (Sept. 1996), 1280–1297. https://doi.org/10.1109/49.536480

[19] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[20] Fabien Le Mentec. 2014. Using the Beaglebone PRU to achieve realtime at low cost. *Embedded Related* (April 2014). https://www.embeddedrelated.com/showarticle/586.php.

[21] Carlos Moratelli, Sergio Johann, and Fabiano Hessel. 2016. Exploring Embedded Systems Virtualization Using MIPS Virtualization Module. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 214–221. https://doi.org/10.1145/2903150.2903179

[22] Bryan Parno, Jonathan M McCune, and Adrian Perrig. 2010. Bootstrapping trust in commodity computers. In *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 414–429.

[23] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 841–856. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj

[24] Dale F. Willis, Arkodeb Dasgupta, and Suman Banerjee. 2014. ParaDrop: A Multi-tenant Platform for Dynamically Installed Third Party Services on Home Gateways. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC '14)*. ACM, New York, NY, USA, 43–44. https://doi.org/10.1145/2627566.2627583

[25] Sisu Xi, Chong Li, Chenyang Lu, Christopher D Gill, Meng Xu, Linh TX Phan, Insup Lee, and Oleg Sokolsky. 2015. RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 179–186.

[26] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in xen. In *Embedded Software (EMSOFT), 2014 International Conference on*. IEEE, 1–10.