Predictable Virtualization on Memory Protection Unit-based Microcontrollers *

Runyu Pan, Gregor Peach, Yuxin Ren, Gabriel Parmer
The George Washington University
Washington, DC
{panrunyu,peachg,ryx,gparmer}@gwu.edu

***Abstract*—With the increasing penetration of embedded systems into the consumer market there is a pressure to have all of inexpensiveness, predictability, reliability, and security. As these systems are often attached to networks and execute complex code from varying sources, reliability and security become essential. To maintain low price and small power budgets, many systems use small microcontrollers with limited memory (on the order of 128KB of SRAM). Unfortunately, the isolation and protection facilities of these systems are often lackluster, making a principled treatment of reliability and security difficult.**

This paper details a system that provides isolation along the three dimensions of CPU, memory, and I/O on small microcontrollers. A key challenge is providing a effective means of harnessing the limited hardware memory protection facilities of microcontrollers. This is achieved through a combination of a static analysis to make the most of limited hardware protection facilities, and a run-time based on our Composite OS. On this foundation, we build a virtualization infrastructure to execute multiple embedded real-time operating systems predictably. We show that VMs based on FreeRTOS achieve reasonable efficiency and predictability, while easily enabling scaling up to 8 VMs in 512 KB SRAM.

I. INTRODUCTION

With the growing interest in embedded computation and the Internet of Things (IoT), it is important that the computational infrastructure underlying these systems is reliable and secure while maintaining the necessary predictability properties. The increasing adoption of these systems in consumer devices is driving the omnipresent network connectivity that characterizes IoT devices. With a network connection, come the security concerns associated with providing malicious adversaries the ability to interact with the system. The customizable filtering and processing of sensor data are also moving these systems away from the traditional static code-bases. Further, we see increasing decentralization in the development of these systems. Different code bases from different clients and system designers that are possibly at different assurance levels, must execute on the shared hardware. This *multi-tenancy* and *mixed-criticality* [1] additionally motivate strict isolation.

Outside of embedded systems, traditional means to bolster system dependability and security often rely on increasing isolation. For example, to prevent the unconstrained propagation of a fault or compromise, hardware facilities such as page-tables are used to segregate memory. However, due to cost, Size, Weight, and Power (SWaP) constraints, the hardware facilities for isolation in embedded and IoT domains are often quite constrained. Especially for the least expensive and most power efficient variants of these systems, an important question is if we can provide both real-time predictable execution *and* the isolation facilities required.

A Memory Protection Unit (MPU) is a simple hardware memory isolation capability found in many microcontrollers. As opposed to Memory Management Units (MMUs), MPUs do not provide a *translation* between page-granular ranges of virtual and physical addresses. Instead, MPUs allow controlling access to regions of physical addresses. MPUs are popular on systems that require low power consumption and predictability with an emphasis on inexpensiveness and simplicity. These systems often avoid the use of data-caches or use very small caches as the limited amount of on-chip SRAM executes at a rate commensurate with the processor. MPUs do *not* use an in-memory structure to track memory accessibility, and there is little variability in memory access times. In contrast, MMUs often use page-tables paired with Translation Lookaside Buffer (TLB) caches. TLB misses require page-table walks and add significant jitter to memory access times. Thus, MPUs are valuable for systems with less variability in an application's memory demand, especially when the TLB size is much smaller than the total number of pages required by the application. In these situations, MPU-based systems benefit from the increased predictability from significantly decreased memory access jitter. Given the wide deployment of MPUs in microcontrollers, this paper introduces an adaptation of the Composite μ-kernel operating system [2] to these processors to provide increased dependability and security in these systems.

Different microcontrollers have very different MPU configurations that vary in terms of the size, alignment, and number of regions of memory that can be protected. These varying constraints mean that most embedded OSes provide hardware-specific APIs for programming the MPU. In contrast, in this paper, we use an existing OS (Composite) with a strong access control model, and *generalize* its memory protection facilities to both MMUs and MPUs. This enables the use of a general API based on Path-Compressed radix Tries (PCTries) that generalize both hardware protection mechanisms, while maintaining the system's predictability properties. This system abstracts protection domains as *components* that encapsulate user-level code, data, and thread execution. A component is a light-weight abstraction over a hardware memory protection context, and a capability table [3], both of which limit the component's access to system resources. Components implement system resource management policies including scheduling, memory mapping/access control (based

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675, ONR Award No. N00014-14-1-0386, and ONR STTR N00014-15-P-1182 and N68335-17-C-0153. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

on PCTries), and I/O, thus providing isolation not only for applications, but also among system services.

Composite, augmented with PCTries to support MPUs, provides the foundation for microcontroller memory isolation. However, to provide the strong isolation between different tenants, between code with varying origins and assurance levels, or when increased fault tolerance is desirable, CPU and I/O isolation are also required. Thus, this paper combines PCTrie support with hierarchical scheduling [4] and a component that abstracts away and restricts I/O accesses to provide both CPU and I/O isolation. Implemented at user-level, the system scheduler [5], [6] and I/O manager components together implement a Virtual Machine Monitor (VMM) [7] that enables the predictable and isolated execution of Virtual Machines (VMs), each of which contains a traditional Real-Time Operating System (RTOS). To the best of our knowledge, this is the first virtualization system for MPU-based microcontrollers.

Using MPUs effectively is not straightforward as they are limited in the number of contiguous regions they can protect, and in the alignment and configuration of those regions. Especially when memory is shared between different components and VMs, the MPU protection of a large number of the shared memory ranges can surpass the capacity of the MPU. Additionally, unaligned components and shared memory can also require more MPU resources than are available. As the MPU is a very limited resource, we must lay out the memory of each component and of their shared memory regions in such a way to that we stay within the configuration space of the MPU. Thus, a core component of this research is a *static memory layout* algorithm that enables the VMM's *dynamic memory mapping* to best use the MPU. This static memory layout for all memory that each component can *possibly* access is a key aspect of enabling MPUs to be used for effective VM isolation.

Contributions. We begin by discussing a background on MPUs (§II), then focus on this paper's contributions:

- An algorithm to enable the effective use of a limited number of MPU resources by intelligently laying out component memory, and potential shared mappings (§III).
- The design (§IV) and implementation (§V) of memory access control and kernel interactions based on PCTries, and the adaptation of an existing microkernel to use them.
- A resource-constrained paravirtualization infrastructure to provide resource multiplexing and isolation for CPU, memory, and I/O (§VI).
- Finally, we provide an evaluation of Composite based on PCTries and its virtualization infrastructure compared to existing (not virtualized) RTOSes (§VII).

II. Embedded System Hardware and Memory Protection

To better understand the challenges in well-utilizing MPUs for the complex isolation configurations of a virtualization infrastructure, we review MMUs and MPU configuration details.

A. Page Tables and Memory Management Units

Page tables are pervasive in higher-end CPUs, and they provide both (1) page-granularity protection, and (2) virtualization of memory accesses by translating between virtual and physical addresses. This virtualization enables the decoupling of physical memory layout and executable memory, as well as the overlap of different component's virtual addresses. A common architecture for MMUs is based on page-tables implemented as radix tries. Hardware page-table walking logic activates in response to a load or store to translate between virtual and physical addresses. To avoid the significant overhead of this walk on each memory access, the Translation Lookaside Buffer (TLB) caches a limited number of translations. The impact of TLB misses can be significant on performance and on predictability. This has motivated TLB partitioning via coloring [8] to increase isolation within the translation cache. Additionally, MMUs complicate fine-grained isolation, and can induce significant overhead [9]. To increase the number of virtual addresses indexed by the TLB, the use of super-pages is common [10].

B. Memory Protection Units

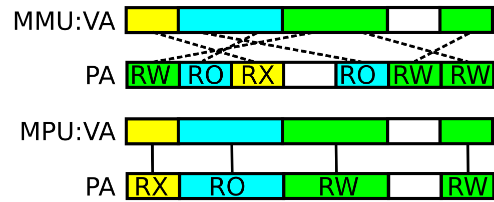

Fig. 1: MMU and MPU differences. MMUs do page-based translation, which is denoted by dotted lines; MPUs instead do region-based protection, which is denoted by vertical solid lines. RX:read-only and executable; RO:read-only; RW:read-write.

Embedded systems are historically designed for a specific task, thus enabling code specialization and avoiding the need for a feature-rich, general-purpose system. This enables the use of slower processors (ARM Cortex-M processors often span between 10-300 MHz), and smaller amounts of memory (from 16-1024 KB). Restricted memory sizes mean that the memory is often SRAM, co-located with chip logic, and executing at a rate commensurate with that logic. This avoids the need for large caches in-between the processor's logic and memory thus further simplifying chip design, significantly reducing power requirements, and increasing the predictability of software execution. In this environment, the memory requirements of page-tables, the internal fragmentation due to page-granularities, and the increased variance in memory accesses due to TLB accesses, are not desirable. In contrast, MPU access control state is maintained directly in registers that are programmable only in kernel-level, and have more flexible granularity control. Unfortunately, as MPUs do *not* provide virtualization of addresses, all memory must be non-overlapping, thus implementing a Single Address Space Operating System (SASOS) [11].

MPUs are widely provided by many embedded system architectures include ARM Cortex-M, MIPS, PowerPC and AVR32. Though there are subtle differences between MPU implementations, we present a generalization here. MPUs

provide a number of *regions* $\{r_0, r_1, \ldots\} = \mathcal{R}$. A region, r_i, can be configured to protect a range of memory starting at an address a_i, of a size s_i. Each region, r_i, has a number of S *subregion*s that each cover an extent of memory of size s_i/S. s_i is constrained by having a minimal value $s_i \geq \check{s}$, and may be required to be a power of two. Each region and subregion has some set of access permissions including $\{\text{readable}, \text{writable}\}$. By default, memory is not accessible while in user-level code, and regions and subregions define the ranges of physical memory that are accessible for which types of memory access. Regions often have an alignment constraint $\mathcal{A}$ which can include constant alignments (*i.e.* on word boundaries, $\mathcal{A} = (a_i \bmod 4 = 0)$), or size-aligned boundaries (*i.e.* $\mathcal{A} = (a_i \bmod s_i = 0)$).

MPU examples. Many ARM Cortex-M microprocessors, like the Cortex-M7, have MPUs configured as $|\mathcal{R}| = 1$ *to* 16, $S = 8$, $\mathcal{A} = (a_i \bmod s_i = 0)$, $\check{s} = 32$, and s_i must be power-of-two. Each region has access permissions in $\{\text{readable}, \text{writable}\}$, while each subregion inherits the permission from the region, and additionally has a bit for accessibility. MIPS M14k is simpler, $|\mathcal{R}| = 1$ *to* 16, $S = 1$, $\mathcal{A} = (a_i \bmod 4 = 0)$, and $\check{s} = 4$. In this research, we focus on ARM Cortex-M microprocessors as they are widely deployed and popular, but also because they have some of the most complicated MPU constraints.

C. I/O and Non-Volatile Memory

Many microcontrollers support *execute in place* read-only memory from flash at fixed virtual addresses. This enables the limited SRAM to be populated only with writable memory. Similarly, most I/O devices are memory mapped and accessed directly with loads and stores. To restrict different applications' access to these regions, MPU regions and sub-regions must be used appropriately. When a MPU region's alignment or granularity constraints prevent protecting individual devices' address ranges, access to them must be mediated by a service (as in [12]).

D. Embedded System Use of MPUs

FreeRTOS and other embedded RTOSes commonly provide architecture-specific APIs to explicitly program MPU regions (often avoiding subregions or exposing them as they are). FreeRTOS is optionally configured to use the MPU. To avoid modifying the structure of the system in which the kernel is compiled as a library with applications, instead of having a firm user-level/kernel separation, system call instructions (SVC) and the eventual handlers for them are inlined into code to enable the execution of sensitive MPU modification instructions. Though this effectively enables applications to constrain memory accesses, possibly increasing reliability, it does not provide security as the protection is discretionary. Safer Sloth [13] combines inline traps and MPU programming with static checks to ensure that the traps are used only where expected by the kernel. As Safer Sloth applications are written in C, this does not prevent security attacks that hijack control flow (for example, self-modifying code and buffer overflows). These systems provide increased reliability for non-malicious

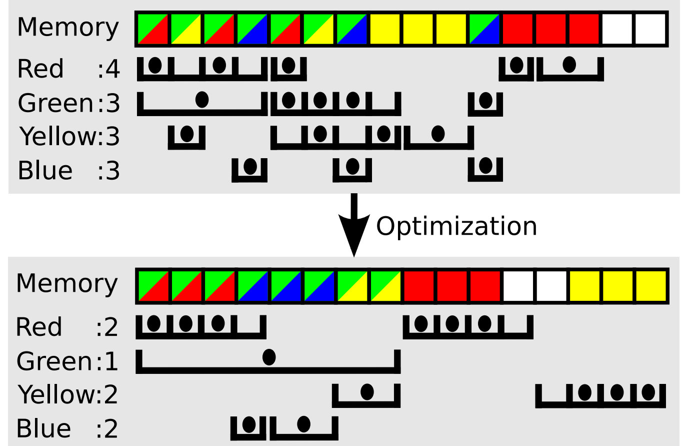

Fig. 2: Memory layout optimization. Each color in the figure represents a different component, and each block in the memory layout indicates a block of memory. The memory block size is some power-of-two. If two colors are present in the same block, then the block is shared between the two components. The MPU has $|\mathcal{R}| = 2$, $S = 4$, $\mathcal{A} = (a_i \bmod s_i = 0)$, and s_i must be power-of-two. The lines below the memory layout show the MPU region and subregion configuration for each component, and the number following the ":" is how many regions are required for that component. If a subregion is active, then it is marked with a solid oval.

applications, but not the strong isolation for security, mixed-criticality, and multi-tenancy.

In contrast, Tock [14] and the I/O virtualization work of [12] confine application-accessible memory using MPU regions. They employ a more traditional (non-inlined) separation of user- and kernel-level where the kernel abstracts away MPU programming and simply scopes memory access to the contiguous regions for code and data for each application. These systems, though simple, do not support the dynamic sharing relationships and complex MPU region layouts required for shared memory and virtualization.

III. Effective MPU Usage via Memory Layout

A. Memory Layout's Impact on MPU Usage

As systems require an increasing number of memory protection barriers between different components at different criticalities and from different sources, they require more complicated management of the memory protection hardware. For example, shared memory regions must be created between VMs and the VMM, and stream processing and publisher/-subscriber systems must create shared regions based on the communication patterns. This is in contrast to traditional embedded systems where components explicitly program MPU regions to protect a small number of memory regions, often one or two per component (see §II-D).

A subtle issue when providing finer-grained memory protection is that the MPU constraints around alignment, subregions, and sizing make the memory layout of all code and data in the system significantly impact how many regions are required for each application. Figure 2 depicts two different memory layouts of applications and their shared memory (for simplicity, we ignore kernel and VMM components in this example). In the first layout, memory is laid out naively and

the components require more regions than what the platform can provide (two regions). Some shared memory regions require separate MPU regions, and the non-power-of-two aligned regions require multiple regions. The red component requires four regions, and other components require three regions. In the second layout, memory is laid out in a manner to minimize the number of regions required to provide the necessary protection. After the optimization, no component uses more than two regions, and the green component requires only one. Note that to achieve this optimized layout, a gap of unused memory appears between the red and yellow memory ranges. This represents memory *overhead* due to internal fragmentation in the regions, the extent of which we study in §VII. In current systems, designers are required to lay out memory manually, or system images must be simple enough (single applications) that region placement is trivial.

To enable MPU-based systems to provide increased protection, we investigate algorithms that both attempt to minimize memory consumption while using a bounded number of regions.

B. Memory Placement Algorithm

Our first approach to making a memory assignment was to adapt the MPU constraints and component memory ranges into a Satisfiability Modulo Theory (SMT) formulation[1] Though the resulting solution is *exact* – if a memory layout exists that uses regions and subregions within the hardware constraints, it will find it – it took an impractical amount of computation time. For simple MPU configurations (four regions, four subregions, three components), solutions require more than 24 hours. Therefore, we introduce a greedy heuristic.

Algorithm 1: Memory Address Assignment Heuristic

Input: C: Set of components, A: Set of SRAM arenas
1 **while** $|\{c \in C \mid \texttt{num_allocated_regions}(c) > |\mathcal{R}| - 1\}| > 0$ **do**
2 $\quad collapsable = \{c \in C \mid \exists_{a \in A} \texttt{is_enabled}(a, c)\}$
3 $\quad$ **if** $|collapsable| = 0$ **then**
4 $\quad\quad$ **return** *None*
5 $\quad c = \arg\max_{c \in collapsable} \texttt{num_allocated_regions}\ (c)$
6 $\quad a_0 = \arg\max_{a \in \texttt{accessible_enabled_arenas}(c)} |\texttt{users}(a)|$
7 $\quad \mathcal{O} = \texttt{accessible_enabled_arenas}\ (c) \setminus a_0$
8 $\quad partners = \{a \in \mathcal{O} \mid \texttt{subregions}(a) + \texttt{subregions}(a_0) \leq S\}$
9 $\quad$ **if** $|partners| = 0$ **then**
10 $\quad\quad \texttt{disable_arena}\ (c, a_0)$
11 $\quad\quad$ **continue**
12 $\quad a_1 = \arg\max_{a \in partners} |\texttt{users}(a_0) \cap \texttt{users}(a)|$
13 $\quad a_{merged} = \texttt{merge_arenas}\ (a_0, a_1)$
14 $\quad A = (A - \{a_0, a_1\}) \cup \{a_{merged}\}$
15 $\quad \texttt{reenable_all_arenas}\ ()$
16 **end**
17 **return** `assign_addresses` *(A)*

Algorithm 1 shows the Memory Address Assignment Algorithm that takes a set of ranges of memory for each component, and determines a layout that can provide the necessary protection within the system's MPU configuration limitations. We define an *arena* as a range of memory with a specific access permission that is accessible by a component (*e.g.* readable, or read-writable). Arenas include component code and read-only data, writable data, shared memory regions between different components, and system heaps to respond to dynamic memory allocation. Importantly, these arenas denote *potential* mappings for each component that are established dynamically via requests to the run-time.

If all components have a number of arenas less than $|\mathcal{R}|$, then there is a trivial assignment of regions to arenas. Otherwise we try to merge two region's arenas into one region, and use separate subregions for each arena to maintain isolation. To effectively use the MPUs introduced in II-B, multiple arenas must use separate subregions in the same region as this lowers the number of regions required for components. If a component requires too many regions, we must decide which of its regions to combine with others. Thus, we first choose a component that has the most regions (line 5). Within that component we choose the arena that is shared with the highest number of components (`users`) (line 6). Intuitively, removing the need for this region will decrease the region requirements for the largest number of components. To find the second region (the *partner*) to merge with, all other regions of that component are considered. However, it is possible that merging some of regions requires more than S sub-regions, in which case the search must start again (line 9-11). By maintaining a boolean *enabled* flag for component arena pairs, we ensure that the algorithm will never try to merge the same two arenas repeatedly. Last, we select the region to merge as the one with the largest intersection of components that use it with the first region (a_0). This process is repeated until all component's memory can be protected by sub-regions.

At this point, the algorithm has a set of assignments of arenas to regions. The `assign_addresses` subroutine sorts the regions by their size (from largest to smallest), then lays them out in memory in that order while abiding by their power-of-two alignment requirement. One notable optimization here is putting the beginning of the next region at the end of the *last used* subregion of the previous region. While simple, we find that this optimization improves memory utilization by up to $\sim 4\times$.

Heuristic sub-optimalities. We have qualitatively compared the solutions output by the SMT solver with those of Algorithm 1 and identified a number of ways the heuristic is sub-optimal. In contrast to the heuristic, the SMT solutions often: (1) use extra regions that are laid out contiguously immediately after a region of a larger size to generate a sub-region of a non-power-of-two; (2) places arenas of various sizes into contiguous sub-regions to minimize internal fragmentation; and (3) enable arenas to span multiple contiguous regions, possibly with varying sub-region size. These sub-optimalities can lead to system configurations that *could possibly* fit within a limited number of MPU regions/subregions and amount of memory via intelligent memory layout (*e.g.* if the SMT solver is used), but where Algorithm 1 fails to find them. Thus, in §VII-D we study the impact of our greedy heuristic on effectively utilizing memory.

Workflow with memory placement. Figure 3 depicts how

[1]Our implementation uses the Python API for the Z3 constraint solver (https://github.com/Z3Prover/z3).

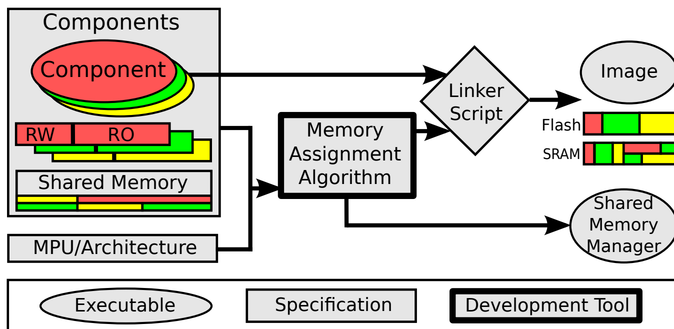

Fig. 3: Memory and region layout toolkit. The toolkit will take the component memory specifications and MPU/architecture information, then turn that into a linker script with the memory assignment algorithm. The final executable binary will be generated from the linker script. The memory assignment algorithm also generates the information needed by the shared memory manager.

the specifications for each component, for the possible shared memory arenas, and for the hardware constraints are integrated into Algorithm 1. The algorithm's outputs are fed into both a linker script to generate the system's image, and to the memory management service in the system to guide dynamic requests.

IV. Generalizing OS Support for Hardware Memory Protection

To provide a strong form of isolation, we adapt the access control abstractions of an existing OS that is based on capability-based security [15], instead of adapting the ad-hoc interfaces of an existing embedded RTOSes. The core security properties of capability-based systems include that system resources are referenced only through *unforgeable tokens* called capabilities, and these capabilities are only accessible to a component if *delegated* from another component that already has a capability to the resource. All system memory and kernel resources are accessible through capabilities, and they can be used to provide strong isolation through *confinement* [16]. Capabilities have proven a good fit for simple systems that focus on moving most resource management functionality to user-level [2], [17], [18], [19].

This research uses a mechanism based on PCTries to *generalize* the capability-based memory access control in Composite – previously specific to MMUs – to control both MPUs and MMUs. This requires updating both the capability-tracking structures within the kernel, and the *retyping* infrastructure. To simplify the kernel and move dynamic kernel memory allocation to user-level Composite uses memory retyping to enable the *user-level* management of *kernel memory* [2] which is heavily inspired by seL4 [18].

A. Composite Kernel Abstractions

The resource management and isolation policies in a Composite system are implemented in user-level components that access and control system resources through capabilities. In our system, the system scheduler (that also controls shared memory), and the I/O manager form the conceptual Virtual Machine Monitor that manage resources to ensure isolation between RTOS VMs. They delegate system resources in response to requests from VMs by mapping memory, creating threads, and creating communication end-points.

Kernel objects and resources. The Composite kernel has a small number of kernel objects: components, threads, TCaps [5], communication end-points (synchronous and asynchronous), and two different types of *resource tables* [2]. Resource tables are indexed by a capability, and map to the referenced kernel resource. Composite has two types of resource tables: page-tables which track memory access permissions, and kernel resource capability tables which track access to kernel resources (including other resource tables). Page-tables view load/store memory accesses as capability references which (via hardware-accelerated MMUs) resolve to the kernel resource of a specific word of physical memory. Kernel resource capability table operations require system calls to perform operations on their resources. Each component is essentially just a collection of two resource tables (one of each type).

A thread is the basic unit of sequential execution, and each component can have zero or more threads. A component with a capability to a thread is able to schedule it. Communication end-points take two forms: (1) `sinv/ret` which provide *synchronous* communication between two components via thread migration [20], and (2) `asnd/rcv` which provide *asynchronous* event notification where a thread *sends* an event via `asnd` which activates a thread waiting on `rcv`. Operations on resource tables themselves include the ability to construct or destroy resource tables (*e.g.* to create a new component), and to copy a capability from one resource table to another. The latter provides *delegation* between components, and requires that a component have a capability to the resource tables of a client.

Only components that have a capability to resource table of a client can delegate to it. These components comprise the system's *resource managers*. To adapt Composite to control a microcontroller's MPU, we update the page-tables to provide a more flexible means of tracking memory access control, and update the memory management resource manager to use this updated abstraction.

B. PCTries as a Generalization of MMUs and MPUs

Path-compressed Radix Tries (PCTrie). In contrast to the radix tries underlying common page-table structures in which subsequent levels translate a fixed number of bits, guarded page-tables and general compressed radix tries [21] translate varying number of bits. This enables a more memory-efficient representation while supporting a sparse address space, and allowing "pages" of sizes corresponding to any power of two. The PCTries in our system map addresses to access permissions instead of physical addresses; as MPUs don't perform translation, the access rights are sufficient.

Radix tries (*i.e.* page-tables) pass address translation through a set of nodes with 2^n entries that reference the next nodes in the trie. Each node translates n bits of the address until a page is referenced (which might be a superpage). In contrast, PCTries enable the *omission* of internal nodes where all possible translations share the same bits for the omitted

nodes. In effect, this removes "chains" of page-table nodes, instead bypassing the need for more nodes by tracking the common prefix of the bits of all following addresses. This is particularly important when address spaces are sparse to save memory and lookup time. Memory in many microcontrollers *is* sparse: most addressable memory and I/O falls within a 4MB region. In such a case, the upper 10 bits are common to all addresses, thus omitting the top levels of nodes. Similar to superpages in page-tables, the least significant bits of an address don't require nodes, and instead result in addressing a page of a larger size.

MPU regions as PCTries. PCTries have many similarities with the MPU model introduced in §II-B. The ARM Cortex-M model uses size-aligned regions (similar to superpages), and requires power-of-two sized regions (thus sub-regions). PCTries also have the same constraints on their superpages, thus can be applied directly to generate MPU region configurations. The static memory layout from §III-B ensures that each PCTrie requires no more than $|\mathcal{R}| \times S$ protected spans of memory.

Page-tables as PCTries. Page-tables are trivially PCTries that omit no nodes to translate bits, and support multiple superpage sizes based on the number of bits at the omitted lower levels.

V. Implementation of Isolation in Constrained Embedded Systems

A. PCTrie Implementation

Composite normally supports page-tables consisting of separately allocated nodes (each 4096 bytes) that are connected together to form active page-tables on the x86. The goals of using PCTries are to (1) represent the sparse address space of small-memory systems with a small number of nodes, (2) to eliminate the separate kernel allocation of radix trie nodes, and, importantly, (3) to enable the efficient and predictable programming of the MPU. The first goal is met by the path-compressed nature of the PCTries, while the second is enabled by *inlining* the PCTrie node memory into the capability slots in a component's capability tables. The third is achieved by maintaining a "summary" of the regions and subregions represented by the entire PCTrie on each modification to the PCTrie.

Page-table nodes are dynamically allocated as separate pages that adhere to the hardware-mandated format. In contrast, PCTrie representations offer more flexibility as they are not directly bound to the hardware. Each PCTrie node contains four references either to memory regions, or to the next node in the PCTrie, thus translating two bits. Each of these references also includes either the number of omitted bits that are to be bypassed for subsequent addresses in the next PCTrie node, or the size of the referenced memory region.

The PCTrie node representation fits into 44 bytes. The capability slots in Composite's capability tables can be of multiple sizes, and the largest size (64 bytes) is used to fully contain PCTrie nodes. This enables components to avoid allocating (retype, activate, and manage) separate memory for each node [2]. Importantly, this means that memory management overheads are amortized into capability-table management, and significant memory is saved by removing the page-granularity representation. User-level libraries manage all of the details of page-table management (using the safe kernel APIs), and these libraries are adapted to understand PCTries.

Efficient MPU updates. The MPU on ARMv7-M processors is programmed through a set of memory-mapped registers, namely the Region Number Register (RNR), Region Base Address Register (RBAR) and Region Attribute and Size Register (RASR). The RNR controls which region is being programmed (though the RBAR has a region field that overrides RNR), while the RBAR and RASR control the start address of the region, the size of the region, and the region's subregion statuses. No hardware assistance is provided for loading these registers, so they must be filled manually.

To enable efficient programming of these registers, we maintain a *summary* of all regions described in a PCTrie in the head (top) capability. This summary is stored in the exact MPU region representation expected by the RBAR and RASR. When switching to a new component, the PCTrie head's summary is read into CPU registers and written into MPU registers without interpretation. This enables efficient MPU programming on the fast path, but relies on the summary containing all memory access information of the whole PCTrie. To provide this invariant, all PCTrie operations that change memory access (*e.g.* mapping and unmapping) *both* modify the PCTrie *and* the summary. To enable this, each PCTrie node contains back-pointers to the head, and enough local information to know which addresses it represents. This design does have one negative side-effect: the system does not support aliases *within* a PCTrie, instead only of memory itself. Semantic support for such in-PCTrie aliasing, if required, can be provided by the user-level library that manages the PCTries.

The RBAR/RASR are laid out contiguously in memory, and are directly followed by three aliases. Given this, we use the ARM Cortex-M7 processor's multi-register load/store (LDM/STM) instructions to accelerate MPU programming by programming *4 regions at a time*. §VII-A evaluates our use of these instructions versus the hardware abstraction layer code provided by the manufacturer.

VI. Small-Scale Virtualization

The memory management abstractions in Composite provide predictable protection for MPU-based microcontrollers. With hardware protection, native code runs safely on microcontrollers. Our goals in providing a virtualization infrastructure for resource constrained systems are to:

- enable *efficient and predictable execution* of each VM close to that on the bare metal;
- use *hierarchical scheduling* [4] to maintain processor isolation between VMs;
- use *protected I/O demultiplexing* through a component to mediate I/O access and isolation;

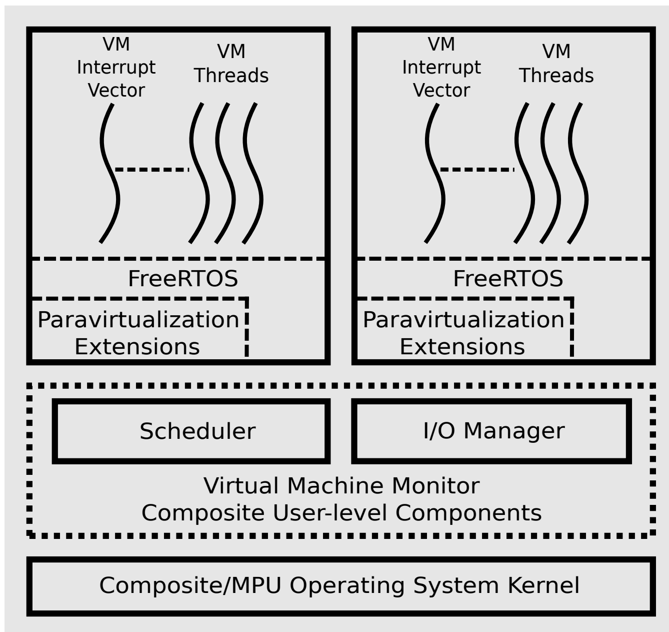

Fig. 4: Virtualization infrastructure. Memory isolation is denoted by the dark solid lines, logical separation is denoted by the dashed lines, and virtual machine monitor is denoted by the dotted lines.

- effectively use the memory layout algorithm output to enable *effective use of the MPU* and memory isolation; and
- provide facilities for *inter-VM communication*.

To meet these goals, we use *paravirtualization* [22] to avoid the complexity and overheads of emulating raw hardware accesses (*i.e.* I/O accesses, timer accesses, and interrupts). The Composite kernel essentially plays the role of the access monitor – appropriately constraining resource accesses – and we implement a *Virtual Machine Monitor* (VMM) as user-level scheduler and I/O manager components. We paravirtualize FreeRTOS by modifying the hardware-specific platform layer to utilize the services of the VMM. Figure 4 depicts the entire system.

A. CPU Virtualization

As the Composite kernel include a scheduling policy, each FreeRTOS virtual machine (FreeRTOS/VM) uses its own scheduler, while using the dispatching facilities of the Composite kernel. To control how each of these schedulers is multiplexed across the processor, we employ hierarchical scheduling. A single VMM scheduler component multiplexes the CPU across VMs. For simplicity, the root scheduler uses fixed-priority, preemptive scheduling, as does FreeRTOS.

Our infrastructure relies on Composite support for TCaps, or *temporal capabilities* that enables controlled delegation of time between different schedulers [5]. This infrastructure enables two key features that we leverage: (1) each VM's scheduler can manage its own timer interrupts over the window of time it has been delegated, and (2) hardware interrupts can safely be vectored directly to a VM if it is the sole VM interacting with the corresponding device. Please see [5] for details of how this is done safely, properly, and efficiently.

Fine-grained access to time By isolating system components, the user-level can no longer directly access privileged hardware resources. For example, native RTOSes often directly access `SYSTICK`, which gives a measure of the progress of time. Each user-level scheduler must maintain time, however, both for execution accounting, and properly delegation of execution time. We use our platform's (STM32F767IGT6) `TIM4` timer register to provide a cycle-accurate counter. Unfortunately, `TIM4` is only 16 bits and we require 64-bit timestamps, so we also must maintain a software word to track the "higher-order bits". In the actual implementation, we increase a 64-bit variable by 65536 when the `TIM4` overflow interrupt happens. When reading the timestamp counter, we read the 64-bit variable and add it to the current `TIM4` counter value to get the correct timestamp. The 64-bit timestamp value is readable by all components, but is only writable by the kernel.

Schedulers are able to control timer interrupts within their TCap's allocated slice of time. To support this, when schedulers dispatch to a thread, they are able to specify that a one-shot timer should occur at some time in the future. These timer interrupts are vectored to a "scheduler thread". To enable these one-shot programmed timers, we use our platform's `TIM3` for this as it features low-overhead programming times and cycle-accuracy.

FreeRTOS CPU paravirtualization. FreeRTOS itself is abstracted by its designers to enable portable implementation of the system on various architectures. We modify the hardware abstraction layer of FreeRTOS for it to execute as a VM, only relying on the VMM and kernel services. We provide Composite-specific implementations of some of the lowest-level functions for dispatching, timer control, and interrupts. Thread dispatch in FreeRTOS maps directly to Composite dispatch, and requires only capabilities to the threads that are provided to the VM, while timers are emulated with the one-shot timer support discussed previously. Interrupts are serviced by Composite threads that suspend on `rcv` end-points, and are activated in response to hardware interrupts.

B. I/O Virtualization

As discussed previously in § II-C, interaction with I/O devices is performed via memory mapped interfaces at platform-defined memory addresses. As the alignment and granularity of these I/O devices in memory preclude fine-grained isolation via MPU regions, we use an isolated I/O manager component to mediate device accesses (see Figure 4). The I/O manager is configured to allow each VM to access a subset of the devices. Each VM makes requests via protected component invocation to the I/O manager to read data from a device, or write to a device. We currently configure this statically, and more intelligent means are left as future work.

For I/O devices that have relatively low data rates, data is passed in registers associated with Composite synchronous invocations. For higher-rate devices, shared memory is used to transfer data. If multiple VMs read data from the same device, the I/O manager must multiplex the device. Thus

interrupts from the device are sent to the I/O manager, and it sends software interrupts (through `asnd` end-points) to the subscribing VMs. The priority of the interrupt thread in the I/O manager is set to the ceiling of the VMs to receive interrupts similar to [23].

FreeRTOS I/O virtualization. I/O routines within FreeRTOS are platform-specific, and we provide Composite-specific versions that invoke the I/O manager. Data and events due to I/O are exposed as FreeRTOS queues.

C. Memory Virtualization

As the system is a SASOS, it does *not* provide address vitualization. Thus, the VMM's memory management (implemented in the scheduler for simplicity) uses the memory layout derived is from the analysis in § III. The lack of address translation hardware is the determining factor guiding this design. The main limitation of requiring non-overlapping VMs is that VMs must be re-linked (thus must be provided as ELF objects with symbol information), and they cannot be written in a way that assumes any static addresses. The former likely limits the use of proprietary software, while the latter is mostly used for I/O accesses which we address via paravirtualization.

Position-independent VMs. To achieve some level of address virtualization, we looked into software support. Ideally, the OS component of each VM could be loaded to flash once, and reused as a library in each VM to save space in Flash (only a single image for N VMs). We initially investigated providing this facility using a VM-specific Global Offset Table (GOT) that maintains the address of each global variable. A register is used to reference the GOT, thus allowing a single image to be used across different VMs, with separate GOTs that translate to the symbols within that image. However, this proved unsatisfactory as (1) portable compilers (`gcc`) generate expensive translations for *each* variable, (2) RTOSes are often compile-time specialized to their applications which prevents using a single generic image, and (3) the execution cost of translation is not well traded for memory given the relatively large flash memories.

D. Inter-VM Communication

Past research has provided facilities for predictable inter-VM communication [24]. We avoid the typical approach of virtualizing inter-VM communication as network communication. This is due to the variety of networking stacks and protocols in embedded systems, and because not every VM wants (or should have to have) a networking stack that consumes a non-trivial amount of memory. Instead, we provide a communication abstraction that is closer to a point-to-point serial line. This is implemented using `sinv` synchronous invocation end-points, and data is pass in registers. The focus of this abstraction is on low-bandwidth data movement, and inter-VM event notification.

Shared memory regions are used for higher-bandwidth data movement between VMs. These regions are set up in accordance with the layout and potential MPU region solution as discussed in §III. The I/O manager is responsible for setting up both the communication end-points between VMs, and, where necessary, setting up the shared memory regions for higher-bandwidth data movement.

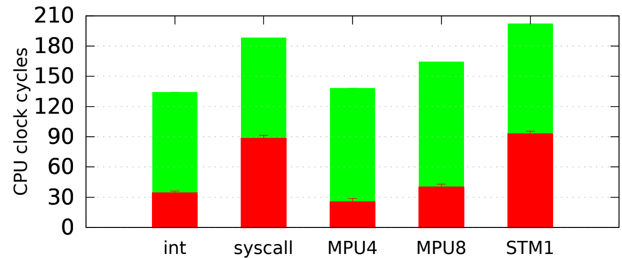

Fig. 5: Hardware overheads. `int` is interrupt overhead, `syscall` is system call overhead, `MPU4` and `MPU8` are for optimized MPU programming overheads for 4 and 8 regions, respectively, and `STM1` is the ST Microelectronics hardware abstraction layer library programming overhead for one MPU region.

FreeRTOS communication paravirtualization. As with the rest of I/O, communication is integrated into FreeRTOS in platform-specific code. Data and events from other VMs are exposed as FreeRTOS queues.

VII. Evaluation

Hardware configurations. For all evaluations, we use an ARM Cortex-M7 microcontroller running at 216MHz (STM32F767IGT6), with 512KB embedded SRAM and 1024KB embedded flash. The microcontroller has a 16KB instruction cache and 16KB data cache, which are always enabled. Also, to speedup instruction fetching from embedded flash, the flash prefetch accelerator is always enabled. The microcontroller also featured a double-precision FPU, which is always disabled. We use the `gcc` compiler version 5.4.1, with the -O2 optimization flag for all cases.

All measurements are repeated 10000 times, and the average, maximum, and standard deviation are calculated. All bar graphs in this section depict the average (the bottom darker bar), stdev (the error bar displayed on the average bar), and maximum (the lighter top bar) measurements.

Software configurations. The Composite system is always run with the VMM including the I/O manager, and system scheduler components with enough untyped memory to satisfy all requests from other components. Two VMs are loaded for the measurements, and each VM runs one or more threads. We use FreeRTOS version 9.0.0, and we evaluate the system configured both with and without MPU support.

A. Microbenchmarks

Hardware overheads. Many system operations require interacting with hardware features such as privilege modes, the MPU, and interrupts. To understand the context for the operating system and virtualization overheads, we first investigate the hardware overheads for the relevant operations. These provide a *lower bound* on the performance of the software abstractions that use them. Figure 5 includes the hardware overheads for mode transitions, interrupt overhead, and MPU programming.

We measure interrupt overhead as the bare hardware overhead for an interrupt. This overhead is measured using the system call instruction with a system call handler that returns

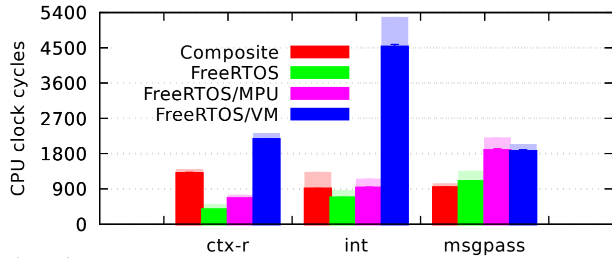

Fig. 6: Overheads of primary system operations, in Composite, FreeRTOS, and paravirtualized FreeRTOS. The Compositemeasurements used in this comparison are all intra-component measurements. `ctx-r` is the round-trip context switch time, `int` is interrupt latency, `msgpass` is message-passing time. For FreeRTOS, message-passing is `xQueueSend()` and `xQueueReceive()`; for Composite, message-passing is `asnd()` and `rcv()`.

immediately (using the `BX LR` instruction). In contrast, system call overhead is measured using the handler in Composite which is modified to return immediately. The difference in these costs includes the small amount of logic for system call routing in the kernel, and the saving and restoring additional registers.

The costs of programming the MPU involve updating the registers associated with the MPU as described in §V-A. The overheads in this operation include saving CPU registers to the stack that will be clobbered, programming the MPU registers, and restoring the clobbered registers. We compare the overheads for using the ST Microelectronics (STM)-provided Hardware Abstraction Layer (HAL) library for MPU programming, and our optimized version that uses ARM multi-register store operations. The optimized version programs four registers at a time, thus we report both four and eight region programming latencies.

Discussion. These results show that the average and worst-case latencies for the hardware operations that we use to provide increased isolation – including the programming of the MPU – are not prohibitive. Of note, it is important to use optimized routines for MPU programming as the implementation in the STM HAL library has unnecessary overhead. The variations in these measurements are generally small, but the maximum values double or even triple the average case. There are two reasons for this: (1) The system contains instruction and data caches and a flash prefetcher, which induce jitter when cache misses occurs, especially when the cache is cold during the first few runs of the benchmark. (2) The measurements are conducted with a 64-bit timestamp counter simulated with 16-bit counters, which causes overhead when its interrupt arrives. Thus, in realistic production systems, the maximum values of these will rarely be reached, and they may not be as large as shown.

Operating system operation overheads. To investigate the cost of various primitive system abstractions in different configurations of Composite and FreeRTOS, we compare: (1) Composite component execution which directly uses the system call API, (2) FreeRTOS which has *no protection facilities* thus represents the overhead of a lightweight RTOS, (3) FreeRTOS/MPU which is FreeRTOS configured to enable explicit programming support for the MPU, and

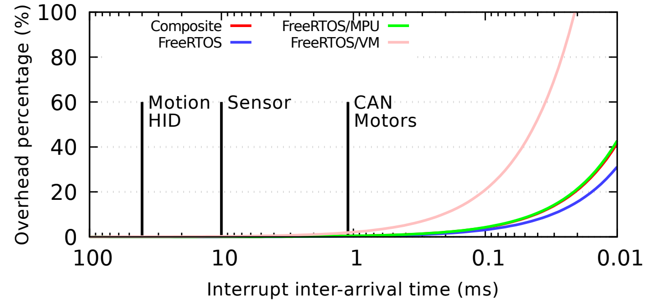

Fig. 7: Overheads of interrupt vs. interrupt inter-arrival time.

(4) FreeRTOS/VM which executes paravirtualized FreeRTOS on top of the VMM and Composite system. These are depicted in Figure 6. The core operations provided by these systems include interrupt handling, thread context switching, and inter-thread communication (which we call IPC).

All of these systems divide interrupt execution into "top half" Interrupt Service Routine (ISR) execution, and "bottom half" execution in the context of a thread. Interrupt latency is measured as the time interval between the activation of the ISR, and the start of the corresponding interrupt handler threads execution. In Composite, this is measured by activating a kernel "asynchronous send" (`asnd`) endpoint in the ISR, that then activates an interrupt thread blocked waiting on a "receive" (`rcv`) endpoint. In FreeRTOS, the measurement is done in a similar fashion: the ISR will use `xQueueSendFromISR()` to send to a queue, thus activating the interrupt thread. In all cases, the time interval between entering the ISR and beginning the interrupt thread's execution is measured. In FreeRTOS/VM, the latency is measured between the execution of Composite's hardware ISR, and when a paravirtualized FreeRTOS thread receives from its interrupt queue.

Discussion. In many cases, FreeRTOS represents a lower bound on the overheads we can expect. It includes no protection facilities, and includes highly optimized versions of many routines such as thread dispatch. We would expect at least the additional overheads from Figure 5 in any system that uses MPU-based protection. The results show that Composite's operations are generally on par with the FreeRTOS/MPU, with the exception of thread context switch costs. Context switching costs in both Composite and FreeRTOS/MPU exhibit overheads over FreeRTOS due to system call and MPU programming costs. FreeRTOS/VM exhibits overheads both from FreeRTOS (due to its scheduling logic), *and* from Composite (which performs the thread context switch).

Interestingly, Composite's message-passing operations are faster, as the Composite system is highly optimized for message passing. For interrupt handling latency, most of the overhead for FreeRTOS/VM comes from hierarchical scheduling overheads that are generally small on larger systems, but prevalent here. The largest component of this overhead is due to a current policy in the Composite scheduling libraries that always switch to a scheduler thread after an interrupt thread's execution. The resulting additional thread dispatch overheads increase interrupt latency. Though this design is not necessary, we have not yet investigated optimizing it for this system.

To better understand the real impact of these overheads, particularly those involving interrupts, we plot the percentage

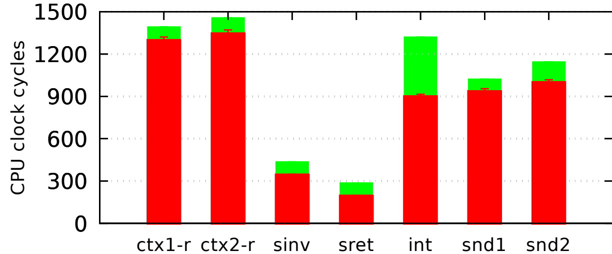

Fig. 8: Overheads of Composite system operations. `ctx1-r` and `ctx2-r` are intra- and inter- component round-trip context switch times, respectively, `sinv` is synchronous invocation enter time, `sret` is synchronous invocation return time, `int` is interrupt latency, `snd1` and `snd2` are intra- and inter- component asynchronous send/receive time, respectively.

of system overheads vs. interrupt inter-arrival time in Figure 7. Please note that the axis goes from a high interrupt inter-arrival rate (low frequency) to a low inter-arrival rate (high frequency). The labels denote typical interrupt inter-arrival times for real devices (from product sheets). `Motion` is for accelerometers for step and motion-detection applications, `HID` are human interface devices such as mouses and keyboards, `Sensor` includes sensors such as accelerometers for detecting finer-grained motion, `CAN` is a CAN bus, and `Motor` is for control of a typical electric stepping motor. A CAN bus at a 125-kbps with typical packet sizes, generates close to 1kHz interrupts. For the electric motor, we assume a common motor that runs at approximately 60 rpm, and the interrupts come from a coaxial 1000-step-per-rotation encoder. Note that the FreeRTOS/MPU and Composite lines are mostly overlapping.

Discussion. From the graph we can conclude that even the relatively higher costs of the isolation provided by the VM infrastructure are reasonable at many of these interrupt rates, often imposing less than 1% overhead. Nonetheless, some applications may require lower interrupt response time, and hence more efficient interrupt response logic. In such cases, the Composite native functionality's overhead is on par with FreeRTOS/MPU and not much more than FreeRTOS, while providing strong isolation.

Composite microbenchmarks. To better understand these results, we perform a number of measurements on the base Composite system (Figure 8). In this part, we discuss the following operations of Composite: (1) intra-component thread switch, (2) inter-component thread switch, (3) synchronous invocation entering time, (4) synchronous invocation returning time, (5) interrupt latency, (6) intra-component asynchronous communication time, and (7) inter-component asynchronous communication time.

Intra-component thread switch (`ctx1-r`) time is the cost of a *round trip* switch between two threads using the Composite dispatch between threads in the same component, while inter-component thread switch (`ctx2-r`) time is between two threads in separate protection domains (as used by the VMM scheduler to switch between VMs). The difference between these is that inter-component thread switch will program the MPU twice (when crossing protection domains), while the intra-component thread switch does not program the MPU.

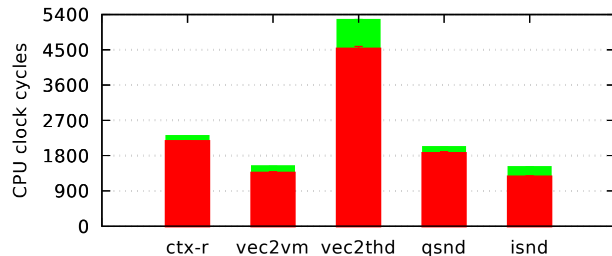

Fig. 9: Overheads of primary system operations in paravirtualized FreeRTOS(FreeRTOS/VM). `ctx-r` is round-trip context switch time, `vect2vm` is the time from Composite interrupt vector to FreeRTOS interrupt vector inside the VM, `vec2thd` is the total time from Composite interrupt vector to the FreeRTOS receiving thread, `qsnd` is FreeRTOS queue send/receive time, and `isnd` the cost of a send.

There are two types of IPC mechanisms in Composite: synchronous invocation (`sinv`/`sret`) via thread migration [20], and asynchronous send/receive (`asnd`/`rcv`). Synchronous invocation between two components involves switching between components (including MPU programming and system calls), and is the most common communication mechanism between memory-isolated components. `snd1` measures the latency between a `asnd`, and the activation of the thread that is waiting on `rcv` where both threads are in the same component. In contrast, `snd2` is between threads in different components, thus requires MPU programming. Interrupt latency `int` is replicated from Figure 6 and demonstrates the expected intuition that interrupt latency is very tied to asynchronous communication. This is unsurprising as these two operations share the same abstractions and code.

B. Virtualization Evaluation

The virtualization environment constructed in Composite along with the paravirtualization of FreeRTOS entail overheads on system operations. These are investigated in more detail in Figure 9. `ctx-r` displays a round-trip context switch via FreeRTOS's `taskYIELD()`. This operation involves both FreeRTOS scheduling, and the scheduling libraries (and dispatch) in Composite.

We focus on the evaluation of two aspects of paravirtualized interrupt latency. First, `vect2vm` is the time from the Composite kernel interrupt vector/ISR execution to the execution of the FreeRTOS ISR in the VM. Second, `vect2thd` is time from the Composite ISR, to the execution of the FreeRTOS receiving thread. We can see from this result, that the majority of the overhead comes from delivering the interrupt *within* FreeRTOS. This validates our previous analysis: the current implementation of the scheduler library always switches back to a "scheduler thread" after processing an interrupt, and the resulting context switch costs are significant. We will optimize this in future work, but as we have shown in Figure 7, current overheads are reasonable for many forms of I/O.

Lastly, we measure both the cost of communication between tasks in FreeRTOS/VM using `xQueueSend()` and `xQueueReceive()`, and between different VMs. Inter-VM communication is performed via virtual interrupts. A sending-VM initiates a synchronous invocation to the destination VM

Segment Name	.text	.data	.bss
Kernel	134727	158	36816
Scheduler	65012	8192	15612
I/O Manager	508	4	512
FreeRTOS	47920	2652	9196
FreeRTOS/MPU	58865	3012	8908
FreeRTOS/VM	38177	12740	17260

TABLE I: System memory overheads.

which executes `xQueueSendFromISR()` to emulate interrupt reception. The intra-VM communication measurement is conducted by measuring the time difference between entering of `xQueueSend()` in the sending thread and the exiting of `xQueueReceive()` in the receiving thread. The inter-VM communication measurement, measures the round-trip costs of the inter-VM invocation (via `sinv` and `ret`) wrapped in the FreeRTOS paravirtualization layers.

Discussion. These results confirm the source of the interrupt overheads of the virtualized environment seen in Figure 6 which are mainly due to excessive thread context switches. The difference between the `qsnd` and `isnd` communication mechanisms is interesting: inter-VM communication is significantly faster than FreeRTOS API-based communication within a VM. Note that `isnd` is a "round-trip" measurement, while `qsnd` is a "one-way" measurement. Thus, our inter-VM communication is more than twice as efficient as intra-VM communication. This is mostly due to the inter-VM communication's direct reliance on optimized Composite IPC mechanisms.

C. Paravirtualization Complexity

Paravirtualization relies on modifications to the lowest-levels of the virtualized OS. In FreeRTOS/VM, we port the hardware-specific platform layer of FreeRTOS to use Composite scheduling support, timer support, and I/O (via invocations to the I/O manager component). This platform implementation is 363 Source Lines of Code (SLOC). In contrast, the bare-metal platform layer for our microcontroller is 483 SLOC without MPU support, and 622 SLOC with MPU support. Though SLOC is not a perfect measure of complexity, we conclude that paravirtualizing FreeRTOS in Composite is not a major effort, and we anticipate that other RTOSes could be ported to the system.

D. Memory Virtualization Overheads

We have evaluated the ability of the system to provide increased isolation through virtualization with reasonable overhead while maintaining predictability. Another important aspect of the system is the memory overhead, and how effectively memory can be laid out to best use the MPU for protection (§III). Additionally, both the kernel, and the components constituting the VMM require memory. We compare against the `.text`, `.data` and `.bss` sections of a minimal FreeRTOS configuration as a baseline.

Kernel and virtualization overheads. The size of each software component is shown in the Table I. The kernel and VMM components impose a constant memory overhead, while the overhead for the changes in the size of FreeRTOS increase with the number of VMs. Paravirtualized FreeRTOS consumes more RAM mainly due to the Composite user-level scheduling library that statically allocates threads; however, its code size shrinks because architecture-dependent code is effectively moved to the Composite kernel. FreeRTOS does dynamic allocation of system resources (including threads and queues), so this table is not a perfect representation of memory requirements as FreeRTOS will require more at run-time. We believe that these overheads are reasonable for many systems.

Sharing Config.	System Only	System + Chaining	System + PubSub
VM Configuration 1	0.528	0.494	0.494
VM Configuration 2	0.600	0.564	0.564
VM Configuration 3	0.534	0.499	0.499
VM Configuration 4	0.400	0.370	0.370
VM Configuration 5	0.487	0.453	0.453

TABLE II: Memory assignment overheads measured as the fraction of used memory.

Memory Assignment Algorithm effectiveness.

Static memory layouts are determined by the algorithm in §III that attempts to best use the available MPU regions and subregions. However, these layouts are heavily influenced by the MPU alignment and sizing constraints, thus producing internal fragmentation.

To evaluate the memory overheads of our system, we test five different configurations of VMs, with three different shared memory setups. These setups all involve VMs running MiBench [25] applications with memory requirements detailed in Table III. Each *small* VM contains one MiBench application. Each *big* VM contains all five MiBench applications that have distinct memory sizes (see Table III). Our five different configurations are as follows: (1) 3 small (`basicmath`, `dijkstra`, `gsm`), (2) 3 small + 1 big (`basicmath`, `dijkstra`, `gsm`, `big`), (3) 4 small + 1 big (`basicmath`, `dijkstra`, `gsm`, `pbmsrch`, `big`), (4) 5 small (`basicmath`, `dijkstra`, `gsm`, `pbmsrch`, `rijndael`), and (5) 5 small + 1 big (`basicmath`, `dijkstra`, `gsm`, `pbmsrch`, `rijndael`, `big`). Each of these setups also includes the kernel and VMM components. Each VM is compiled the applications and paravirtualized FreeRTOS.

As shared memory for inter-VM and VM/VMM communication is essential and complicates MPU region placement, we investigate three sharing configurations:

- Pairwise shared regions between each VM and the I/O manager.
- The same as the first, but also with shared memory connecting the VMs in a ring. This roughly mimics a sensor processing pipeline of filters and transformations.
- The same as the first, but with shared memory configured as a publisher-subscriber network, where each VM can write to a shared memory location that every other VM can read (pairwise sharing between each VM).

To determine the size of these regions, we observe that packet sizes of 256 bytes are common on these platforms[2]. A realistic ring buffer size – 16 – yields a shared region of 4096 bytes.

To evaluate the overhead in each configuration, we apply the memory assignment algorithm. We give the system a configurable amount of SRAM, and run the heuristic with ever smaller amounts of memory until it fails to find an assignment when given XKB. The total used memory for

[2]For example, IoT devices communicating using MQTT (http://mqtt.org/).

each configuration is YKB, and Table II includes the memory overhead ($(X-Y)/Y$).

Discussion. Internal fragmentation for power-of-two sized regions has an average overhead of 25% which we are exceeding at $\sim 50\%$. Though this overhead is reasonable (*e.g.* compared to page-table's memory, alignment, and granularity requirements), it is by no means optimal. In contrast, our initial design of this algorithm based on an SMT formulation with much simpler configurations tended to demonstrate overheads on the order of 10%. The $\sim 50\%$ overheads this heuristic exhibits represent the trade-off between a reasonable solution, and long runtimes.

VM scalability. With an increasing need for isolation, a single system will include multiple VMs. To evaluate the scalability of the system to an increasing number of VMs, we factor in the size of each VM, and the necessary MPU alignment considerations. Each FreeRTOS/VM (with no application) can fit into one 32KB memory block. If all virtual machines are empty, eight will fit into the memory of the STM32F767IGT6 (512KB SRAM) along with the kernel and the VMM components with 128KB left over for shared mappings and dynamic allocations. This represents a limit based solely on system overheads, and large or computationally hungry applications would practically force fewer VMs. Even for existing systems, this context is useful if a system with a large number of applications wished to isolate different subsets of them in different VMs. Regardless, this represents a surprising level of VM scalability for such a small system.

VM Name	.text	.data	.bss
basicmath	61961	12740	17260
dijkstra	64256	12744	57708
gsm	86545	12932	17260
pbmsrch	63264	12744	18284
rijndael	80576	12748	17260
big	193512	12948	59244

TABLE III: VM memory overheads.

To understand the scalability of the system *with* applications, Table III includes the memory requirements for different MiBench [25] applications compiled with paravirtualized FreeRTOS. In realistic settings, the applications inside a VM are comparable or smaller than the VM memory footprint. Thus, a mid-range microcontroller can easily fit four VMs, while the high-end microcontrollers with more (1MB or more) SRAM can easily fit 12 or more VMs.

VIII. Related Work

Avoiding hardware protection. Many memory-constrained systems provide isolation via language safety. These include TinyOS [26] and Tock [14]. These have the benefit that software bugs are confined by software checks based on type-safety which can lead to low overheads for many operations. However, this makes virtualization of separate code bases difficult. Other projects use safe languages on microcontrollers, but focus on programmability at the cost of performance and predictability [27], [28], [29], [30].

In contrast, other embedded OSes don't focus on safety and instead on memory usage and/or dynamic update [31], [32], [26], [33]. Though these dimensions are important, we focus on an infrastructure for predictabily providing isolation between multiple components or virtual machines. Of note, the static memory layout that enables adequate region-based protection is performed with full knowledge of all system components. This complicates dynamic update; the ability to update layouts at run-time is future work.

Some systems simplify protection domain interactions (IPC) down to function calls for performance, and use co-routines, or cooperative scheduling [32], [26], [14]. This design decision is appropriate when there is high confidence that all code adheres to low, bounded execution times, otherwise this non-preemptibility prohibits temporal isolation.

Explicitly managing the MPU. As discussed in section §II-D, many embedded OSes provide APIs to specifically program MPU regions. Such systems do not concentrate on providing a general interface for MPU programming, nor do they pair these facilities with a memory layout algorithm to efficiently use the MPU with sharing and virtualization.

MPU as cache. Previous versions of embedded Linux supported MPUs as a software-managed cache of accessible regions. More mappings could be created for a process than the number of regions on the system, and those not *active* cause faults when their memory is accessed. Software handlers (similar to software TLB misses) define how the regions are tracked, and the region eviction policy determines which region to drop when a new region needs to be activated. Though this generalizes MPU regions to an arbitrary number of regions, it imposes unpredictable overheads similar to those imposed by TLB misses and hardware page-table walks. For resource-constrained microcontrollers, this technique provides maximum compatibility with the MMU programming paradigm, but can lead to unpredictable execution and large overheads when memory accesses happen in a specific order. This work aims to maximally use the MPU by employing a static analysis while ensuring predictable, protection-fault-free execution of embedded code.

Virtualization support. To the best of our knowledge, this paper presents the first virtualization infrastructure for MPU-based microcontrollers. Similar to our I/O virtualization, [12] implements a FreeRTOS task that handles I/O, and other tasks indirect their I/O accesses through it. In contrast, our research focuses a virtualization infrastructure that enables isolation in all of CPU, memory, and I/O.

IX. Conclusions

This paper has introduced an infrastructure to support strong isolation along the CPU, memory, and I/O dimensions. We have shown how a somewhat limited MPU can be effectively used to provide fine-grained isolation through a combination of a static memory placement algorithm, and an efficient kernel representation that generalizes different hardware protection mechanisms. The VM infrastructure supports paravirtualized FreeRTOS, and can easily scale to 8 VM instances.

Acknowledgments. We'd like to thank the anonymous reviewers for their helpful feedback, and our shepherd for helping to significantly improve the clarity of this paper.

References

[1] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

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

[3] H. Levy, "Capability-based computer systems," 1984.

[4] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

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

[6] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[7] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010, pp. 209–222.

[8] S. A. Panchamukhi and F. Mueller, "Providing task isolation via TLB coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

[9] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, 2007.

[10] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Transparent operating system support for superpages," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002.

[11] J. S. Chase, M. Baker-Harvey, H. M. Levy, and E. D. Lazowska, "Opal: A single address space system for 64-bit architectures," *Operating Systems Review*, vol. 26, no. 2, p. 9, 1992. [Online]. Available: citeseer.ist.psu.edu/58003.html

[12] 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), Pittsburgh PA, USA, October 6, 2016.*, 2016.

[13] 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.

[14] 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.

[15] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 26, no. 1, pp. 29–35, 1983.

[16] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[17] "The Fiasco microkernel: http://l4re.org, retrieved 10/6/17."

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct 2009.

[19] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Symposium on Operating Systems Principles*, 1999, pp. 170–185. [Online]. Available: citeseer.ist.psu.edu/shapiro99eros.html

[20] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, 1994.

[21] J. Liedtke and K. Elphinstone, "Guarded page tables on Mips R4600 or an exercise in architecture-dependent micro optimization," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 1, pp. 4–15, 1996.

[22] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[23] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 191–201.

[24] C. Li, S. Xi, C. Lu, C. D. Gill, and R. Guerin, "Prioritizing soft real-time network traffic in virtualized hosts based on xen," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001.

[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

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

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

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

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

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

[32] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, 2005.

[33] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, 2004.