A Virtualized Separation Kernel for Mixed Criticality Systems

Ye Li Richard West Eric Missimer

Computer Science Department
Boston University
Boston, MA 02215, USA
{liye,richwest,missimer}@cs.bu.edu

Abstract

Multi- and many-core processors are becoming increasingly popular in embedded systems. Many of these processors now feature hardware virtualization capabilities, such as the ARM Cortex A15, and x86 processors with Intel VT-x or AMD-V support. Hardware virtualization offers opportunities to partition physical resources, including processor cores, memory and I/O devices amongst guest virtual machines. Mixed criticality systems and services can then coexist on the same platform in separate virtual machines. However, traditional virtual machine systems are too expensive because of the costs of trapping into hypervisors to multiplex and manage machine physical resources on behalf of separate guests. For example, hypervisors are needed to schedule separate VMs on physical processor cores. In this paper, we discuss the design of the Quest-V separation kernel, which partitions services of different criticalities in separate virtual machines, or *sandboxes*. Each sandbox encapsulates a subset of machine physical resources that it manages without requiring intervention of a hypervisor. Moreover, a hypervisor is not needed for normal operation, except to bootstrap the system and establish communication channels between sandboxes.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

Keywords Separation kernel, chip-level distributed system

1. Introduction

Embedded systems are increasingly featuring multi- and many-core processors, due in part to their power, performance and price benefits. These processors offer new opportunities for an increasingly significant class of mixed criticality systems. In mixed criticality systems, there is a combination of application and system components with different safety and timing requirements. For example, in an avionics system, the in-flight entertainment system is considered less critical than that of the flight control system. Similarly, in an automotive system, infotainment services (navigation, audio and so forth) would be considered less timing and safety critical than the vehicle management sub-systems for anti-lock brakes and traction control.

A major challenge to mixed criticality systems is the safe isolation of separate components with different levels of criticality. Isolation has traditionally been achieved by partitioning components across distributed modules, which communicate over a network such as a CAN bus. For example, Integrated Modular Avionics (IMA) [1] is used to describe a distributed real-time computer network capable of supporting applications of differing criticality levels aboard an aircraft. To implement such concepts on a multicore platform, a software architecture that enforces the safe isolation of system components is required.

Hardware-assisted virtualization provides an opportunity to efficiently separate system components with different levels of safety, security and criticality. Back in 2006, Intel and AMD introduced their VT-x and AMD-V processors, respectively, with support for hardware virtualization. More recently, the ARM Cortex A15 was introduced with hardware virtualization capabilities, for use in portable tablet devices. Similarly, some Intel Atom chips now have VT-x capabilities for use in automobile In-Vehicle Infotainment (IVI) systems, and other embedded systems.

While modern hypervisor solutions such as Xen [2] and Linux-KVM [3] leverage hardware virtualization to isolate their guest systems, they are still required for CPU, memory, and I/O resource management. Traps into the hypervisor occur every time a guest system needs to be scheduled, when a remapping of guest-to-machine physical memory is needed, or when an I/O device interrupt is delivered to a guest. This is both unnecessary and potentially too costly for mixed criticality systems with real-time requirements.

In this paper we present an entirely new operating system that uses hardware-assisted virtualization as an extra *ring*

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2764-0 /14/03. . . $15.00.
http://dx.doi.org/10.1145/2576195.2576206

of protection, to achieve efficient resource partitioning and performance isolation for subsystem components. Our system, called Quest-V, is a separation kernel [4] design, effectively operating as a distributed system on a chip. The system avoids traps into a hypervisor (a.k.a. virtual machine monitor, or VMM) when making scheduling and I/O management decisions. Instead, all resources are partitioned at boot-time amongst system components that are capable of scheduling themselves on available processor cores. Similarly, system components are granted access to specific subsets of I/O devices and memory so that devices can be managed without involvement of a hypervisor.

Experiments show how Quest-V is able to make efficient use of CPU, memory and I/O partitioning, using hardware virtualization. We show how a Linux front-end (guest) system can be supported with minimal modifications to its source code. An *mplayer* benchmark for video decoding and playback running on a Linux guest in Quest-V achieves almost identical performance compared to running on a non-virtualized Linux system. Similarly, *netperf* running on a Linux guest in Quest-V achieves better network bandwidth performance than when running on Xen, for large packet sizes. Quest-V guest services are able to maintain functionality in the presence of faults in other sandboxed guests, and are also able to communicate with remote guest services using tunable bandwidth guarantees.

The next section briefly describes the rationale for our system. The architecture is then explained in Section 3. Section 4 details a series of experiments to evaluate the costs and performance of using hardware virtualization for resource partitioning in Quest-V. An overview of related work is provided in Section 5. Finally, conclusions and future work are discussed in Section 6.

2. Design Rationale

Quest-V is centered around three main goals: safety, predictability and efficiency. Of particular interest is support for safety-critical applications, where equipment and/or lives are dependant on the operation of the underlying system. With recent advances in fields such as cyber-physical systems, more sophisticated OSes beyond those traditionally found in real-time and embedded computing are now required. Consider, for example, an automotive system with services for engine, body, chassis, transmission, safety and infotainment. These could be consolidated on the same multicore platform, with space-time partitioning to ensure malfunctions do not propagate across services. Virtualization technology can be used to separate different groups of services, depending on their criticality (or importance) to overall system functionality.

Quest-V uses hardware virtualization technology to partition resources amongst separate *sandboxes*, each responsible for a subset of processor cores, memory regions, and I/O devices. This leads to the following benefits:

(1) Improved Efficiency and Predictability – the separation of resources and services eliminates, or reduces, resource contention. This is similar to the *share-nothing* principle of multi-kernels such as Barrelfish [5]. As system resources are effectively distributed across cores, and each core is managed separately, there is no need to have shared structures such as a global scheduler queue. This, in turn, can improve predictability by eliminating undue blocking delays due to synchronization.

(2) Fault Isolation and Mixed Criticality Services – virtualization provides a way to separate services and prevent functional components from being adversely affected by those that are faulty. This, in turn, increases system availability when there are partial system failures. Similarly, services of different criticalities can be isolated from one another, and in some cases may be replicated to guarantee their operation.

(3) Highest Safe Privilege – Rather than adopting a principle of *least* privilege for software services, as is done in micro-kernels, a virtualized system can support the *highest* safe privilege for different services. Virtualization provides an extra logical "ring of protection" that allows *guests* to think they are working directly on the hardware. Thus, virtualized services can be written with traditional kernel privileges, yet still be isolated from other equally privileged services in other guest domains. This avoids the communication costs typically associated with micro-kernels, to request services in different protection domains.

(4) Minimal Trusted Code Base – A micro-kernel attempts to provide a minimal trusted code base for the services it supports. However, it must still be accessed as part of inter-process communication, and basic operations such as coarse-grained memory management. Monitors form a trusted code base in the Quest-V separation kernel. Access to these can be *avoided almost entirely*, except to bootstrap (guest) sandbox kernels, handle faults and manage guest-to-machine physical memory mappings. This enables sandboxes to operate, for the most part, independently of any other code base that requires trust. In turn, the trusted monitors can be limited to a small memory footprint.

3. Quest-V Separation Kernel Architecture

A high-level overview of the Quest-V architecture is shown in Figure 1. The current implementation works on Intel VT-x platforms but plans are underway to port Quest-V to the AMD-V and ARM architectures.

The system is partitioned into separate *sandboxes*, each responsible for a subset of machine physical memory, I/O devices and processor cores. Trusted monitor code is used to launch *guest* services, which may include their own kernels and user space programs. A monitor is responsible for managing special *extended page tables* (EPTs) that translate guest physical addresses (GPAs) to host physical addresses (HPAs), as described later in Figure 2.

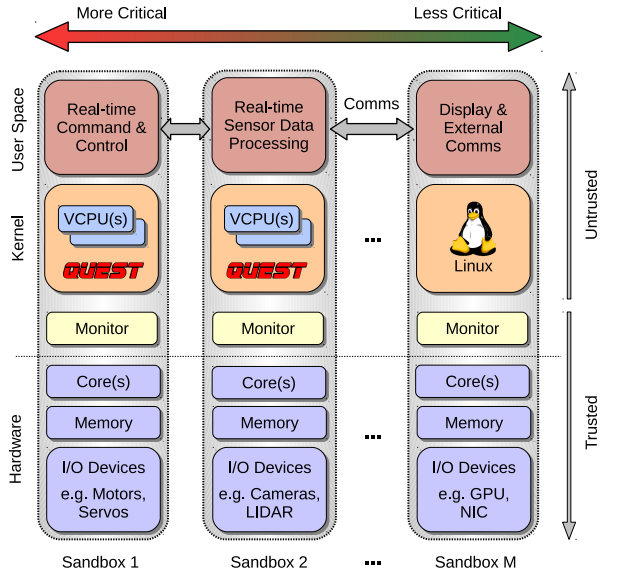

Figure 1. Example Quest-V Architecture Overview

We chose to have a separate monitor for each sandbox, so that it manages only one set of EPT memory mappings for a single guest environment. The amount of added overhead of doing this is small, as each monitor's code fits within $4KB$ [1]. However, the benefits are that monitors are made much simpler, since they know which sandbox they are serving rather than having to determine at runtime the guest that needs their service. Typically, guests do not need intervention of monitors, except to establish shared memory communication channels with other sandboxes, which requires updating EPTs. The monitor code needed after system initialization is about 400 lines.

Mixed-Criticality Example – Figure 1 shows an example of three sandboxes, where two are configured with Quest-native safety-critical services for command, control and sensor data processing. These services might be appropriate for a future automotive system that assists in vehicle control. Other less critical services could be assigned to vehicle infotainment services, which are partitioned in a sandbox that has access to a local display device. A non-real-time Linux system could be used in this case, perhaps also managing a network interface (NIC) to communicate with other vehicles or the surrounding environment, via a vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communication link.

3.1 Resource Partitioning

Quest-V supports configurable partitioning of CPU, memory and I/O resources amongst guests. Resource partitioning is mostly static, taking place at boot-time, with the exception of some memory allocation at run-time for dynamically created communication channels between sandboxes.

[1] The EPTs take additional data space, but 12KB is enough for a 1GB sandbox address space.

CPU Partitioning – In Quest-V, scheduling is performed within each sandbox. Since processor cores are statically allocated to sandboxes, there is no need for monitors to perform sandbox scheduling as is typically required with traditional hypervisors. This approach eliminates the monitor traps otherwise necessary for sandbox context switches. It also means there is no notion of a global scheduler to manage the allocation of processor cores amongst guests. Each sandbox's local scheduler is free to implement its own policy, simplifying resource management. This approach also distributes contention amongst separate scheduling queues, without requiring synchronization on one global queue.

Memory Partitioning – Quest-V relies on hardware assisted virtualization support to perform memory partitioning. Figure 2 shows how address translation works for Quest-V sandboxes using Intel's extended page tables. Each sandbox kernel uses its own internal paging structures to translate guest virtual addresses to guest physical addresses. EPT structures are then walked by the hardware to complete the translation to host physical addresses.

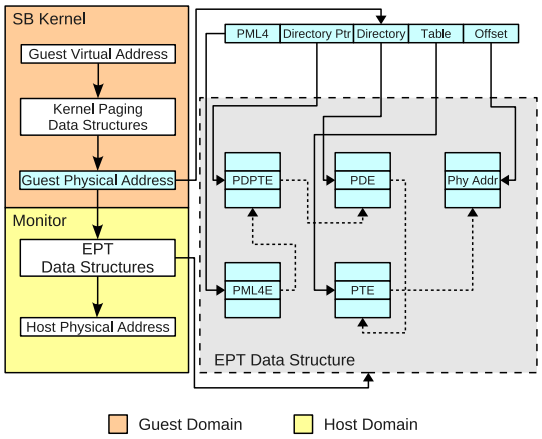

Figure 2. Extended Page Table Mapping

On modern Intel x86 processors with EPT support, address mappings can be manipulated at 4KB page granularity. For each 4KB page we have the ability to set read, write and even execute permissions. Consequently, attempts by one sandbox to access illegitimate memory regions of another sandbox will incur an EPT violation, causing a trap to the local monitor. The EPT data structures are, themselves, restricted to access by the monitors, thereby preventing tampering by sandbox kernels.

EPT mappings are cached by hardware TLBs, expediting the cost of address translation. Only on returning to a guest after trapping into a monitor are these TLBs flushed. Consequently, by avoiding exits into monitor code, each sandbox operates with similar performance to that of systems with conventional page-based virtual address spaces.

Cache Partitioning – Microarchitectural resources such as caches and memory buses provide a source of contention on multicore platforms. Using hardware performance coun-

ters we are able to establish cache occupancies for different sandboxes [6]. Also, memory page coloring can be used to partition shared caches [7] between sandboxes. Most of these features are under active development in Quest-V.

I/O Partitioning – In Quest-V, device management is performed within each sandbox directly. Device interrupts are delivered to a sandbox kernel without monitor intervention. This differs from the "split driver" model of systems such as Xen, which have a special domain to handle interrupts before they are directed into a guest. Allowing sandboxes to have direct access to I/O devices avoids the overhead of monitor traps to handle interrupts.

To partition I/O devices, Quest-V first has to restrict access to device specific hardware registers. Device registers are usually either memory mapped or accessed through a special I/O address space (e.g. I/O ports). For the x86, both approaches are used. For memory mapped registers, EPTs are used to prevent their accesses from unauthorized sandboxes. For port-addressed registers, special hardware support is necessary. On Intel processors with VT-x, all variants of `in` and `out` instructions can be configured to cause a monitor trap if access to a certain port address is attempted. As a result, an I/O bitmap can be used to partition the whole I/O address space amongst different sandboxes. Unauthorized access to a certain register can thus be ignored or trigger a fault recovery event.

Any sandbox attempting access to a PCI device must use memory-mapped or port-based registers identified in a special PCI *configuration space* [8]. Quest-V intercepts access to this configuration space, which is accessed via both an address (`0xCF8`) and data (`0xCFC`) I/O port. A trap to the local sandbox monitor occurs when there is a PCI data port access. The monitor then determines which device's configuration space is to be accessed by the trapped instruction. A device *blacklist* for each sandbox containing the *Bus, Device* and *Function* numbers of restricted PCI devices is used by the monitor to control actual device access.

A simplified control flow of the handling of PCI configuration space protection in a Quest-V monitor is given in Figure 3. Notice that simply allowing access to a PCI data port is not sufficient because we only want to allow the single I/O instruction that caused the monitor trap, and which passed the monitor check, to be correctly executed. Once this is done, the monitor should immediately restrict access to the PCI data port again. This behavior is achieved by setting the *trap flag* (TF) bit in the sandbox kernel system flags to cause a single step debug exception after it executes the next instruction. By configuring the processor to generate a monitor trap on debug exception, the system can immediately return to the monitor after executing the I/O instruction. After this, the monitor is able to mask the PCI data port again for the sandbox kernel, thereby mediating future device access.

In addition to direct access to device registers, interrupts from I/O devices also need to be partitioned amongst sandboxes. In modern multicore platforms, an external interrupt controller is almost always present to allow configuration of interrupt delivery behaviors. On modern Intel x86 processors, this is done through an I/O Advanced Programmable Interrupt Controller (IOAPIC). Each IOAPIC has an I/O *redirection table* that can be programmed to deliver device interrupts to all, or a subset of, sandboxes. Each entry in the I/O redirection table corresponds to a certain interrupt request from an I/O device on the PCI bus.

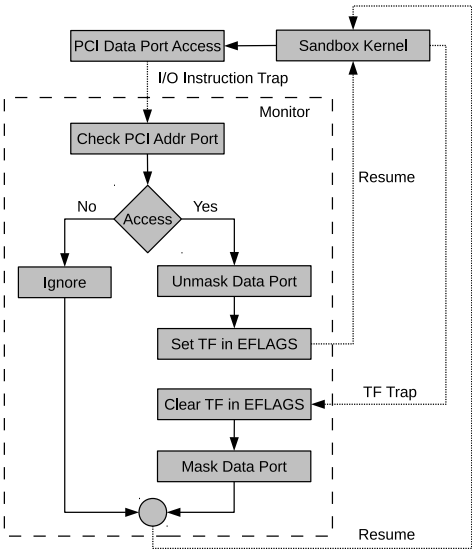

Figure 3. PCI Configuration Space Protection

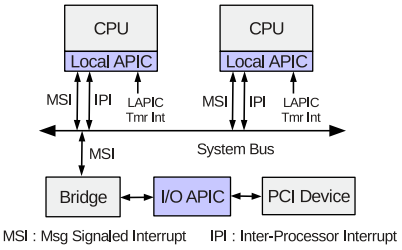

Figure 4. APIC Configuration

Figure 4 shows the hardware APIC configuration. Quest-V uses EPT entries to restrict access to memory regions used to access IOAPIC registers. Though IOAPIC registers are memory mapped, two special registers are programmed to access other registers similar to that of PCI configuration space access. As a result, an approach similar to the one shown in Figure 3 is used in the Quest-V monitor code for access control. Attempts by a sandbox to access the IOAPIC space cause a trap to the local monitor as a result of an EPT violation. The monitor then checks to see if the sandbox has authorization to update the table before allowing any changes to be made. Consequently, device interrupts are safely partitioned amongst sandboxes.

This approach is efficient because device management and interrupt handling are all carried out in the sandbox kernel with direct access to hardware. The monitor traps necessary for the partitioning strategy are only needed for device enumeration during system initialization.

3.2 Native Quest Sandbox Support

We have developed a native Quest kernel for real-time and embedded systems. The kernel code has been implemented from scratch for the IA-32 architecture, and is approximately 10,000 lines of C and assembly, discounting drivers and network stack. Each monitor is given access to a native Quest kernel address space so that direct manipulation of kernel objects during monitor traps are possible.

Real-Time VCPU Scheduling. Native Quest kernels feature a novel virtual CPU (VCPU) scheduling framework, to guarantee that one task, or thread, does not interfere with the timely execution of others [9]. VCPUs form the fundamental abstraction for scheduling and temporal isolation of threads. The concept of a VCPU is similar to that in traditional virtual machines [2, 10], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [2] represented as VCPUs to each of the guests. VCPUs exist as kernel rather than monitor abstractions, to simplify the management of resource budgets for potentially many software threads. We use a hierarchical approach in which VCPUs are scheduled on PCPUs and threads are scheduled on VCPUs.

A VCPU acts as a resource container [11] for scheduling and accounting decisions on behalf of software threads. It serves no other purpose to virtualize the underlying physical CPUs, since our sandbox kernels and their applications execute directly on the hardware. In particular, a VCPU does not need to act as a container for cached instruction blocks that have been generated to emulate the effects of guest code, as in some trap-and-emulate virtualized systems.

In common with *bandwidth preserving* servers [12][13][14], each VCPU, V, has a maximum compute time budget, C_V, available in a time period, T_V. V is constrained to use no more than the fraction $U_V = \frac{C_V}{T_V}$ of a physical processor (PCPU) in any window of real-time, T_V, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

A native Quest kernel defines two classes of VCPUs as shown in Figure 5: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts from I/O devices to be scheduled as threads, which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. The whole approach enables I/O VCPUs to be specified for certain devices, or for certain tasks that issue I/O requests, thereby allowing interrupts to be handled at different priorities and with different CPU shares than conventional tasks associated with Main VCPUs.

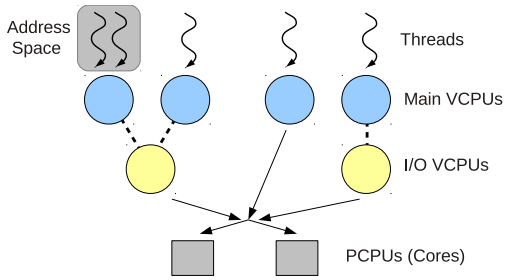

Figure 5. VCPU Scheduling Hierarchy

By default, Main VCPUs act like Sporadic Servers [15, 16], while each I/O VCPU acts as a bandwidth preserving server with a dynamically-calculated period, T_{IO}, and budget, C_{IO} [9]. Each I/O VCPU is specified a certain utilization factor, U_{IO}, to limit its bandwidth. When a device interrupt requires handling by an I/O VCPU, the system determines the thread τ associated with a corresponding I/O request [3]. All events, including those related to I/O processing are associated with threads running on Main VCPUs. In this framework, C_{IO} is calculated as $T_V \cdot U_{IO}$, while T_{IO} is set to T_V for a Main VCPU, V, associated with τ.

Our native Quest kernel is designed for mission-critical tasks in mixed-criticality systems. By sandboxing these tasks in their own virtual machines, they are isolated from the effects of other less critical tasks. By default, all Quest-V sandboxes use native Quest kernels. Even for booting third party systems such as Linux in a Quest-V sandbox, a native Quest sandbox has to be started first.

3.3 Linux Sandbox Support

In addition to native Quest kernels, Quest-V is also designed to support other third party sandbox systems such as Linux and AUTOSAR OS [17]. Currently, we have successfully ported a Puppy Linux [18] distribution with Linux 3.8.0 kernel to serve as our system front-end, providing a window manager and graphical user interface. In Quest-V, a Linux sandbox can only be bootstrapped by a native Quest kernel. This means a native Quest sandbox needs to be initialized first and Linux will be started in the same sandbox via a bootloader kernel thread. To simplify the monitor logic, we paravirtualized the Linux kernel by patching the source code. Quest-V exposes the maximum possible privileges of hardware access to sandbox kernels. From Linux sandbox's perspective, all processor capabilities are exposed except hard-

[2] We define a PCPU to be either a conventional CPU, a processing core, or a hardware thread.

[3] E.g., τ may have issued a prior `read()` request that caused it to block on its Main VCPU, but which ultimately led to a device performing an I/O operation.

ware virtualization support. On Intel VT-x processors, this means a Linux sandbox does not see `EPT` or `VMX` features when displaying `/proc/cpuinfo`. Consequently, the actual changes made to the original Linux 3.8.0 kernel are less than 50 lines. These changes are mainly focused on limiting Linux's view of available physical memory and handling I/O device DMA offsets caused by memory virtualization.

An example memory layout of Quest-V with a Linux sandbox on a 4-core processor is shown in Figure 6. Even though the Linux kernel's view of (guest) physical memory is contiguous from address 0x0, the kernel is actually loaded after all native Quest kernels in machine physical memory. Since Quest-V does not require hardware IOMMU support, we patched the Linux kernel DMA layer to make it aware of this offset between guest physical and machine physical memory addresses during I/O device DMA.

In the current implementation, we limit Linux to manage the last logical processor or core. As this is not the bootstrap processing core, the Linux code that initializes a legacy 8253 Programmable Interval Timer (PIT) has to be removed. The 8253 PIT assumes interrupts are delivered to the bootstrap processor but instead we program the IOAPIC to control which interrupts are delivered to the Linux sandbox. In general, our implementation can be extended to support Linux running on a subset of cores (potentially more than one), with access to a controlled and specific subset of devices. Right now, the entire Linux sandbox runs in 512MB RAM, including space for the root filesystem. This makes it useful in situations where we want to prevent Linux having access to persistent disk storage.

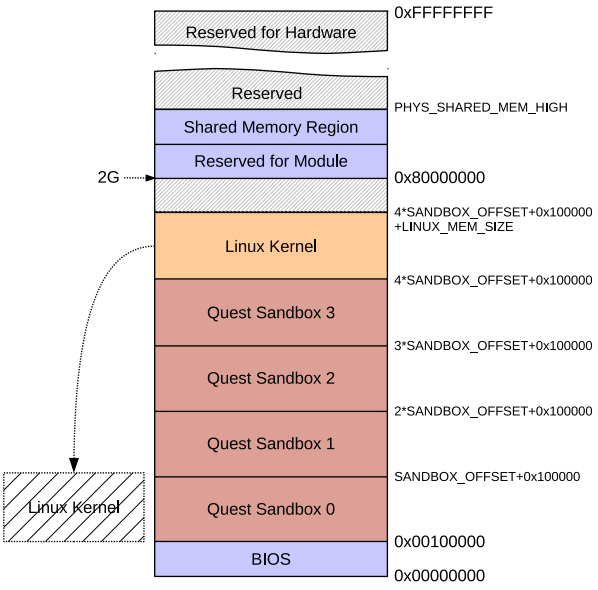

Figure 6. Quest-V Physical Memory Layout with Linux

Whenever a Linux sandbox is present, the VGA frame buffer and GPU hardware are always assigned to it for exclusive access. All the other sandboxes will have their default terminal I/O tunneled through shared memory channels to virtual terminals in the Linux front-end. We developed libraries, user space applications and a kernel module to support this redirection in Linux.

3.4 Shared Memory Communication Channels

Inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory, and asynchronous event notification mechanisms using Inter-processor Interrupts (IPIs). Monitors update EPT mappings as necessary to establish message passing channels between specific sandboxes. Only those sandboxes with mapped shared pages are able to communicate with one another.

A *mailbox* data structure is set up within shared memory by each end of a communication channel. By default, Quest-V supports asynchronous communication by polling a mailbox status bit, instead of using IPIs, to determine message arrival. Message passing threads are bound to VCPUs with specific parameters to control the rate of exchange of information in native Quest sandboxes. Likewise, sending and receiving threads are assigned to higher priority VCPUs to reduce the latency of transfer of information across a communication channel. This way, shared memory channels can be prioritized and granted higher or lower throughput as needed, while ensuring information is communicated in a predictable manner. Quest-V supports real-time communication between native Quest sandboxes without compromising the CPU shares allocated to non-communicating tasks.

A similar library is under development for communication between processes in Linux and native Quest sandboxes. In the current implementation, a Linux process can only request a memory channel shared with all native Quest sandboxes for non-critical communication.

4. Experimental Evaluation

We conducted a series of experiments to investigate the performance of the Quest-V resource partitioning scheme. For all the experiments, we ran Quest-V on a mini-ITX machine with a Core i5-2500K 4-core processor, featuring 4GB RAM and a Realtek 8111e NIC. In all the network experiments where both a server and a client are required, we also used a Dell PowerEdge T410 with an Intel Xeon E5506 2.13GHz 4-core processor, featuring 4GB RAM and a Broadcom NetXtreme II NIC. For all the experiments involving a Xen hypervisor, Xen 4.2.3 was used with a Fedora 18 64-bit domain 0 and Linux 3.6.0 kernel.

Monitor Intervention. To see the extent to which a monitor was involved in system operation, we recorded the number of monitor traps during Quest-V Linux sandbox initialization and normal operation. During normal operation, we observed only one monitor trap every 3 to 5 minutes caused by `cpuid`. In the x86 architecture, if a `cpuid` instruction is executed within a guest it forces a trap (i.e., VM-exit or hypercall) to the monitor. Table 1 shows the monitor traps

	Exception	CPUID	VMCALL	I/O Inst	EPT Violation	XSETBV
No I/O Partitioning	0	502	2	0	0	1
I/O Partitioning	10157	502	2	9769	388	1
I/O Partitioning (Block COM and NIC)	9785	497	2	11412	388	1

Table 1. Monitor Trap Count During Linux Sandbox Initialization

recorded during Linux sandbox initialization under three different configurations: (1) a Linux sandbox with control over *all* I/O devices but with no I/O partitioning logic, (2) a Linux sandbox with control over all I/O devices and support for I/O partitioning logic, and (3) a Linux sandbox with control over all devices except the serial port and network interface card, while also supporting I/O partitioning logic. However, again, during normal operation, no monitor traps were observed other than by the occasional `cpuid` instruction.

Microbenchmarks. We evaluated the performance of Quest-V using a series of microbenchmarks. The first, *findprimes*, finds prime numbers in the set of integers from 1 to 10^6. CPU cycle times for *findprimes* are shown in Figure 7, for the configurations in Table 2. All Linux configurations were limited to 512MB RAM. For `Xen HVM` and `Xen PVM`, we pinned the Linux virtual machine (VM) to a single core that differed from the one used by Xen's Dom0. For all `4VM` configurations of Xen, we allowed Dom0 to make scheduling decisions without pinning VMs to specific cores.

Configuration	Description
Linux	Standalone Linux (no virtualization)
Quest-V Linux	One Linux sandbox hosted by Quest-V
Quest-V Linux 4SB	One Linux sandbox co-existing with three native Quest sandboxes
Xen HVM	One Linux guest on Xen with hardware virtualization
Xen HVM 4VM	One Linux guest co-existing with three native Quest guests
Xen PVM	One paravirtualized Linux guest on Xen
Xen PVM 4VM	One paravirtualized Linux guest co-existing with three native Quest guests

Table 2. System Configurations

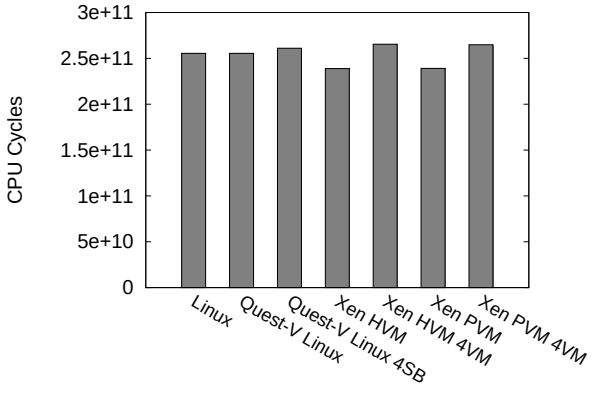

Figure 7. findprimes CPU Benchmark

As can be seen in the figure, `Quest-V Linux` shows no overhead compared to standalone Linux. `Xen HVM` and `Xen PVM` actually outperform standalone Linux, and this seems to be attributed to the way Xen virtualizes devices and reduces the impact of events such as interrupts on thread execution. The results show approximately 2% overhead when running *findprimes* in a Linux sandbox on Quest-V, in the presence of three native Quest sandboxes. We believe this overhead is mostly due to memory bus and shared cache contention. For the `4VM` Xen configurations, the performance degradation is slightly worse. This appears to be because of the overheads of multiplexing 5 VMs (one being Dom0) onto 4 cores.

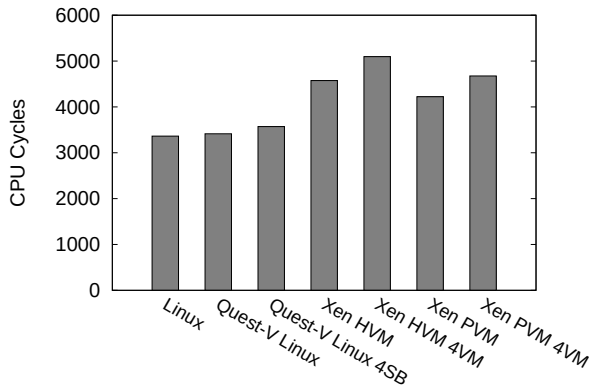

Figure 8. Page Fault Exception Handling Overhead

We evaluated the exception handling overheads for the configurations in Table 2, by measuring the average CPU cycles spent by Linux to handle a single user level page fault. For the measurement, we developed a user program that intentionally triggered a page fault and then skipped the faulting instruction in the `SIGSEGV` signal handler. The average cycle times were derived from 10^8 contiguous page faults. The results in Figure 8 show that exception handling in `Quest-V Linux` is much more efficient than Xen. This is mainly because the monitor is not required for handling almost all exceptions and interrupts in a Quest-V sandbox.

The last microbenchmark measures the CPU cycles spent by Linux to perform a million *fork-exec-wait* system calls. A test program forks and waits for a child while the child calls `execve()` and exits immediately. The results are shown in Figure 9. `Quest-V Linux` is almost as good as native Linux and more than twice as fast as any Xen configuration.

mplayer HD Video Benchmark. We next evaluated the performance of application benchmarks that focused on I/O and memory usage. First, we ran *mplayer* with an x264 MPEG2 HD video clip at 1920x1080 resolution. The video was about 2 minutes long and 102MB in file size. By invok-

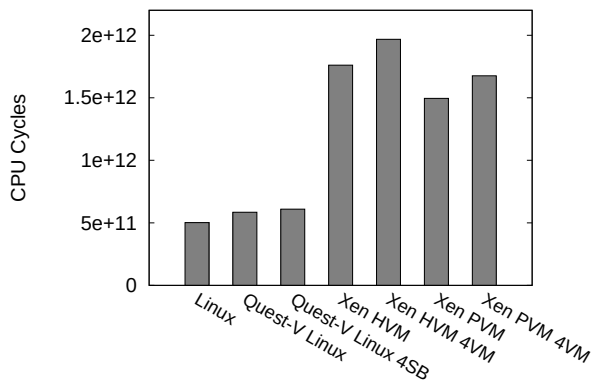

Figure 9. Fork-Exec-Wait Micro Benchmark

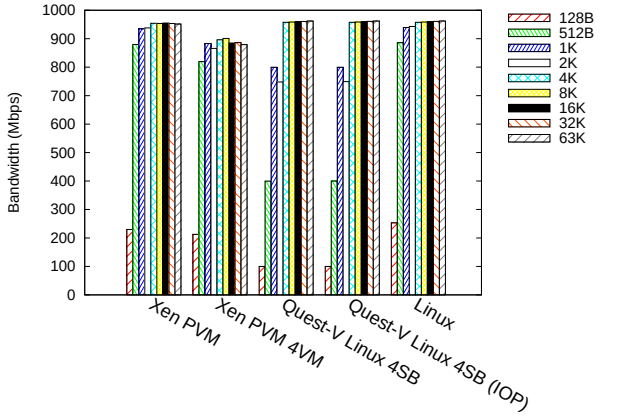

Figure 11. netserver UDP Receive

ing *mplayer* with `-benchmark` and `-nosound`, *mplayer* decodes and displays each frame as fast as possible. With the extra `-vo=null` argument, *mplayer* will further skip the video output and try to decode as fast as possible. The real-times spent in seconds in the video codec (VC) and video output (VO) stages are shown in Table 3 for three different configurations. In Quest-V, the Linux sandbox was given exclusive control over an integrated HD Graphics 3000 GPU. The results show that Quest-V incurs negligible overhead for HD video decoding and playback in Linux. We also observed (not shown) the same playback frame rate for all three configurations.

	VC (VO=NULL)	VC	VO
Linux	16.593s	29.853s	13.373s
Quest-V Linux	16.705s	29.915s	13.457s
Quest-V Linux 4SB	16.815s	29.986s	13.474s

Table 3. mplayer HD Video Benchmark

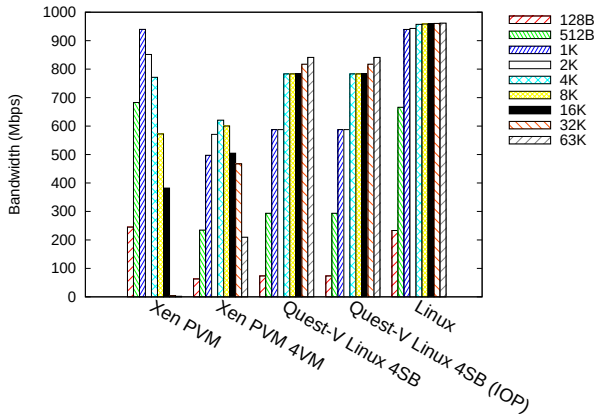

Figure 10. netperf UDP Send with Different Packet Sizes

netperf UDP Bandwidth Benchmark. We next investigated the networking performance of Quest-V, using the *netperf* UDP benchmark. The measured bandwidths of separate UDP send (running *netperf*) and receive (running *netserver*) experiments, on the mini-ITX machine, are shown in Figures 10 and 11, respectively.

We have omitted the results for `Xen HVM`, since it did not perform as well as `Xen PVM`. For `Xen PVM` and `Xen PVM 4VM`, `virtio` [19] is enabled. It can be seen that this helps dramatically improve the UDP bandwidth for small size UDP packets. With 512B packet size, `Xen PVM` outperforms standalone Linux. In most other cases, Quest-V outperforms Xen with bigger packet sizes and multiple VMs.

I/O Partitioning. We also tested the potential overhead of the I/O partitioning strategy in Quest-V. For the group of bars labelled as `Quest-V Linux 4SB (IOP)`, we enabled I/O partitioning logic in Quest-V and allowed all devices except the serial port to be accessible to the Linux sandbox. Notice that even though no PCI device has been placed in the blacklist for the Linux sandbox, the logic that traps PCI configuration space and IOAPIC access is still in place. The results show that the I/O partitioning does not impose any extra performance overhead on normal sandbox execution. I/O resource partitioning-related monitor traps only happen during system initialization and faults.

However, Quest-V does incur a network performance penalty compared to standalone Linux. This is especially noticeable for small size packets. To determine the cause of this behavior, we ran the same experiments with the server and client running on the same machine for standalone Linux and Quest-V Linux. This eliminated the potential influence from hardware device access and DMA. The results shown in Figure 12 demonstrate that at least part of the overhead is related to memory management rather than just I/O.

We believe that this overhead is caused by multiple factors, including the usage of shared caches and TLBs [20]. For instance, the fact that some of the virtual machine related data structures (e.g. EPT tables) are cacheable could increase the cache contention in a virtualized environment. Further studies are needed to more precisely identify the overheads of virtualization in Quest-V.

TLB Performance. We ran a series of experiments to measure the effects of address translation using EPTs. A

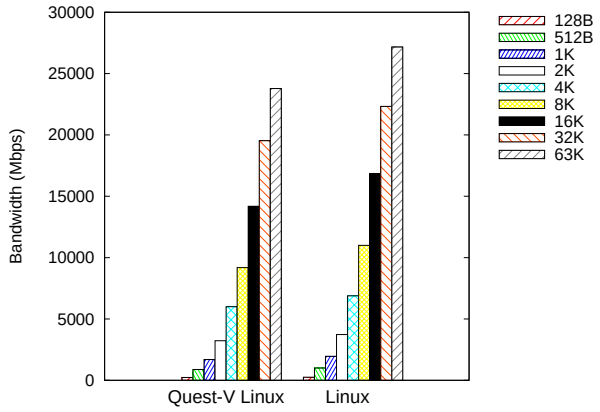

Figure 12. netperf UDP Local Host

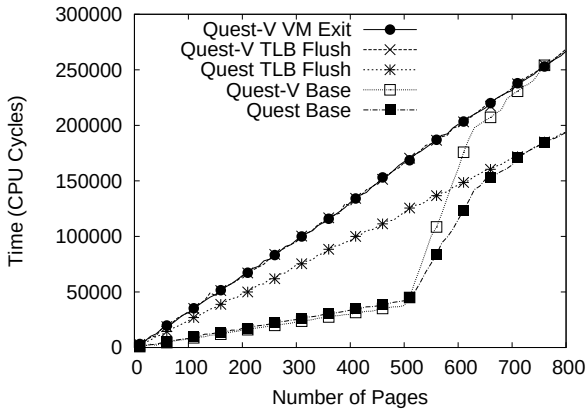

Figure 13. Data TLB Performance

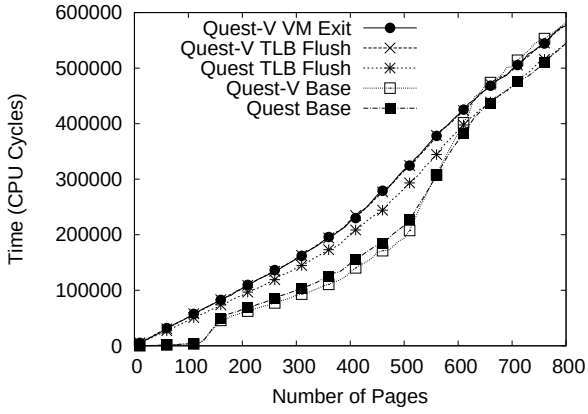

Figure 14. Instruction TLB Performance

TLB-walking thread in a native Quest kernel was bound to a Main VCPU with a 45ms budget and 50ms period. This thread made a series of instruction and data references to consecutive 4KB memory pages, at 4160 bytes offsets to avoid cache aliasing effects. The average time for the thread to complete access to a working set of pages was measured over 10 million iterations.

Figures 13 and 14 compare the performance of a native Quest kernel running in a virtual machine (i.e., sandbox) to when the same kernel code is running without virtualization. Results prefixed with `Quest` do not use virtualization, whereas the rest use EPTs to assist address translation. Experiments involving a `VM Exit` or a TLB `Flush` performed a trap into the monitor, or a TLB flush, respectively, at the end of accessing the number of pages on the x-axis. All other `Base` cases operated without involving a monitor or performing a TLB flush.

As can be seen, the `Quest-V Base` case refers to the situation when the monitor is not involved. This yields address translation costs similar to when the TLB walker runs on a base system without virtualization (`Quest Base`) for working sets with less than 512 pages. We believe this is acceptable for safety-critical services found in embedded systems, as they are likely to have relatively small working sets. The cost of a VM-Exit is equivalent to a full TLB flush, but entries will not be flushed in Quest-V sandboxes if they are within the TLB reach. Note that without the use of TLBs to cache address translations, the EPTs require 5 memory accesses to perform a single guest-physical address (GPA) to host-physical address (HPA) translation. The kernels running the TLB walker use two-level paging for 32-bit virtual addresses, and in the worst-case this leads to 3 memory accesses for a GVA to GPA translation. However, with virtualization, this causes $3 \times 5 = 15$ memory accesses for a GVA to HPA translation.

Fault Isolation and Predictability. To demonstrate fault isolation in Quest-V, we created a scenario that includes both message passing and networking across 4 different native Quest sandboxes. Specifically, sandbox 1 has a kernel thread that sends messages through private message passing channels to sandbox 0, 2 and 3. Each private channel is shared only between the sender and specific receiver, and is guarded by EPTs. In addition, sandbox 0 also has a network service running that handles ICMP echo requests. After all the services are up and running, we manually break the NIC driver in sandbox 0, overwrite sandbox 0's message passing channel shared with sandbox 1, and try to corrupt the kernel memory of other sandboxes to simulate a driver fault. After the driver fault, sandbox 0 will try to recover the NIC driver along with both network and message passing services running in it. During the recovery, the whole system activity is plotted in terms of message reception rate and ICMP echo reply rate in all available sandboxes, and the results are shown in Figure 15.

In the experiment, sandbox 1 broadcasts messages to others (`SB0,2,3`) at 50 millisecond intervals. Sandbox 0, 2 and 3 receive at 100, 800 and 1000 millisecond intervals. Another machine sends ICMP echo requests at 500 millisecond intervals to sandbox 0 (`ICMP0`). All message passing threads are bound to Main VCPUs with 100ms periods and

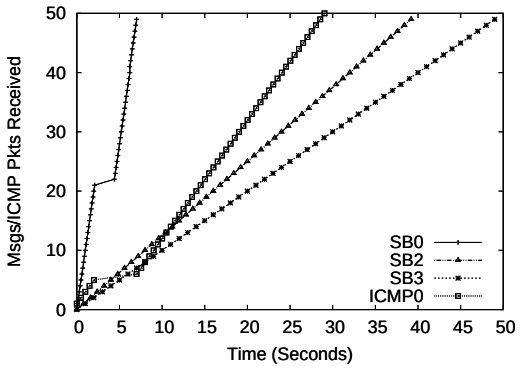

Figure 15. Sandbox Isolation

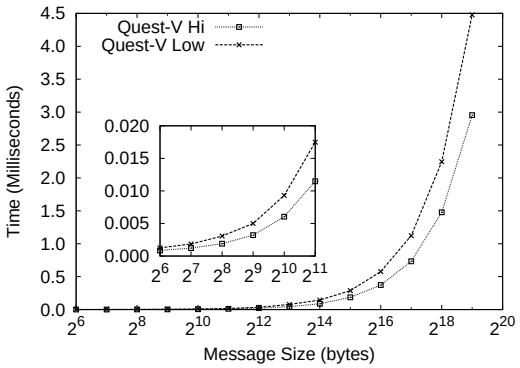

Figure 16. Message Passing Microbenchmark

20% utilization. The network driver thread is bound to an I/O VCPU with 10% utilization and 10ms period.

Results show that an interruption of both message passing and packet processing occurred in sandbox 0, but all the other sandboxes were unaffected. This is because of memory isolation between sandboxes, enforced by EPTs.

Inter-Sandbox Communication. The message passing mechanism in Quest-V is built on shared memory. Instead of focusing on memory and cache optimization, we tried to study the impact of scheduling on inter-sandbox communication in Quest-V.

We setup two kernel threads in two different sandbox kernels and assigned a VCPU to each of them. One kernel thread used a 4KB shared memory message passing channel to communicate with the other thread. In the first case, the two VCPUs were the highest priority with their respective sandbox kernels. In the second case, the two VCPUs were assigned lower utilizations and priorities, to identify the effects of VCPU parameters (and scheduling) on the message sending and receiving rates. In both cases, the time to transfer messages of various sizes across the communication channel was measured. Note that the VCPU scheduling framework ensures that all threads are guaranteed service as long as the total utilization of all VCPUs is bounded according to rate-monotonic theory [21]. Consequently, the impacts of message passing on overall system predictability can be controlled and isolated from the execution of other threads in the system.

Figure 16 shows the time spent exchanging messages of various sizes, plotted on a log scale. `Quest-V Hi` is the plot for message exchanges involving high-priority VCPUs having 100ms periods and 50% utilizations for both the sender and receiver. `Quest-V Low` is the plot for message exchanges involving low-priority VCPUs having 100ms periods and 40% utilizations for both the sender and receiver. In the latter case, a shell process was bound to a highest priority VCPU. As can be seen, VCPU parameter settings affect message transfer times.

In our experiments, the time spent for each size of message was averaged over a minimum of 5000 trials to normalize the scheduling overhead. The communication costs grow linearly with increasing message size, because they include the time to access memory.

5. Related Work

Xen [2], Linux-KVM [3], XtratuM [22], the Wind River Hypervisor, and Mentor Graphics Embedded Hypervisor all use virtualization technologies to logically isolate and multiplex guest virtual machines on a shared set of physical resources. LynxSecure [23] is another similar approach targeted at safety-critical real-time systems. PikeOS [24] is a separation micro-kernel [25] that supports multiple guest VMs, and targets safety-critical domains such as Integrated Modular Avionics. The micro-kernel supports a virtualization layer that is required to manage the spatial and temporal partitioning of resources amongst guests.

In contrast to the above systems, Quest-V statically partitions machine resources into separate sandboxes. Services of different criticalities can be mapped into separate sandboxes. Each sandbox manages its own resources independently of an underlying hypervisor. Quest-V also avoids the need for a split-driver model involving a special domain (e.g., `Dom0` in Xen) to handle device interrupts. Interrupts are delivered directly to the sandbox associated with the corresponding device, using I/O passthrough. Even though PCI passthrough is supported in recent versions of Xen and KVM, guest virtual machines can only directly access device registers. The hypervisor is still responsible for initial interrupt handling and interrupt acknowledgment. This potentially forces two hypervisor traps for each interrupt. ELI [26] is a software-only approach for handling interrupts within guest virtual machines directly with shadow IDTs. In combination with PCI passthrough, this is similar to the approach Quest-V uses to partition I/O resources. However, Quest-V allows a sandbox to use its own IDT and eliminates monitor intervention on all interrupts instead of only interrupts from a specific device. IOAPIC redirection table access control is used to prevent unauthorized interrupt redirection.

NoHype [27] is a secure system that uses a modified version of Xen to bootstrap and then partition a guest, which is granted dedicated access to a subset of hardware resources.

NoHype requires guests to be paravirtualized to avoid VM-Exits into the hypervisor. VM-Exits are treated as errors and will terminate the guest, whereas in Quest-V they are avoided under normal operation, except to recover from a fault or establish new communication channels. For safety-critical applications it is necessary to handle faults without simply terminating guests. Essentially Quest-V shares the ideas of NoHype, while extending them into a fault tolerant, mixed-criticality system on a chip.

Barrelfish[5] is a multi-kernel that replicates rather than shares system state, to avoid the costs of synchronization and management of shared data structures. As with Quest-V, communication between kernels is via explicit message passing, using shared memory channels to transfer cacheline-sized messages. In contrast to Barrelfish, Quest-V focuses on the use of virtualization techniques to efficiently partition resources for mixed criticality applications.

Dune [28] uses hardware virtualization to create a sandbox for safe user-level program execution. By allowing user-level access to privileged CPU features, certain applications (e.g. garbage collection) can be made more efficient. However, most system services are still redirected to the Linux kernel running in VMX root mode. VirtuOS [29] uses virtualization to partition existing operating system kernels into service domains, each providing a subset of system calls. Exceptionless system calls are used to request services from remote domains. The system is built on top of Xen and relies on both the shared memory facilities and event channels provided by the Xen VMM to facilitate communication between different domains. The PCI passthrough capability provided by the Xen VMM is also used to partition devices amongst service domains. However, interrupt handling and VM scheduling still requires VMM intervention.

Other systems that partition resources on many-core architectures include Factored OS [30], Corey [31], Hive [32] and Disco [33]. Unlike Quest-V, these systems are focused on scalability rather than isolation and predictability.

6. Conclusions and Future Work

This paper introduces Quest-V, which is an open-source separation kernel built from the ground up. It uses hardware virtualization to separate system components of different criticalities. Consequently, less important services can be isolated from those of higher criticality, and essential services can be replicated across different sandboxes to ensure availability in the presence of faults.

Quest-V avoids traditional costs associated with hypervisor systems, by statically partitioning machine resources across guest sandboxes, which perform their own scheduling, memory and I/O management. Sandboxes can communicate via shared memory channels that are mapped to extended page table (EPT) entries. Only trusted monitors are capable of changing entries in these EPTs, preventing guest access to arbitrary memory regions in remote sandboxes.

This paper shows how multiple native Quest-V sandboxes can be mapped to different cores of a multicore processor, while allowing a Linux front-end to co-exist and manage less safety-critical legacy applications. We describe the method by which resources are partitioned amongst sandboxes, including I/O devices. Allowing interrupts to be delivered directly to the sandbox guests rather than monitors reduces the overheads of I/O management. Similarly, allowing sandbox guest kernels to perform local scheduling without expensive hypercalls (VM-exits) to monitor code leads to more efficient CPU usage. Quest-V manages CPU usage using a novel hierarchy of VCPUs implemented as Sporadic Servers, to ensure temporal isolation amongst real-time, safety-critical threads. Since Quest-V attempts to avoid VM-exits as much as possible, except to update EPTs for communication channels, bootstrap the sandboxes and handle faults, the TLBs caching EPT mappings are rarely flushed. This benefit comes about due to the fact that multiple guests are not multiplexed onto the same processor core, and in the embedded systems we envision for this work, sandbox working sets will fit within the TLB reach (at least for critical services in native Quest-V sandboxes).

Quest-V requires system monitors to be trusted. Although these occupy a small memory footprint and are not involved in normal system operation, the system can be compromised if the monitors are corrupted. Future work will investigate real-time fault detection and recovery strategies similar to those in traditional distributed systems. We also plan to investigate additional hardware features to enforce safety and security. These include Intel's trusted execution technology (TXT) to enforce safety of monitor code, IOMMUs to restrict DMA memory ranges, and *Interrupt Remapping* (IR) [34] to prevent delivery of unauthorized interrupts to specific cores [35]. Protection of CPU model-specific registers (MSRs) will be similarly enforced using hardware-managed bitmaps.

Please see `www.questos.org` for more details.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work is supported in part by NSF grants #0615153 and #1117025.

References

[1] C. B. Watkins, "Integrated Modular Avionics: Managing the allocation of shared intersystem resources," in *Proceedings of the 25th Digital Avionics Systems Conference*, pp. 1–12, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, 2003.

[3] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[4] J. M. Rushby, "Design and verification of secure systems," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 12–21, 1981.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 29–44, 2009.

[6] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, *Multicore Technology: Architecture, Reconfiguration and Modeling*, ch. 8. CRC Press, ISBN-10: 1439880638, 2013.

[7] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

[8] PCI: http://wiki.osdev.org/PCI.

[9] M. Danish, Y. Li, and R. West, "Virtual-CPU scheduling in the Quest operating system," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*, pp. 169–179, 2011.

[10] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2006.

[11] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A new facility for resource management in server systems," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[12] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-time Systems Symposium*, pp. 4–13, 1998.

[13] Z. Deng, J. W. S. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[14] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, pp. 179–210, 1996.

[15] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[16] M. Stanovich, T. P. Baker, A. I. Wang, and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[17] AUTOSAR: AUTomotive Open System ARchitecture – http://www.autosar.org.

[18] "Puppy Linux." http://www.puppylinux.org.

[19] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[20] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pp. 13–23, 2005.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[22] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach.," in *the European Dependable Computing Conference*, pp. 67–72, 2010.

[23] "LynxSecure Embedded Hypervisor and Separation Kernel." http://www.lynuxworks.com/virtualization/hypervisor.php.

[24] "SYSGO PikeOS." http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept.

[25] 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 *the 22nd ACM Symposium on Operating Systems Principles*, pp. 207–220, 2009.

[26] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal performance for I/O virtualization," in *Proceedings of the 17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 411–422, 2012.

[27] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 401–412, 2011.

[28] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 335–348, 2012.

[29] R. Nikolaev and G. Back, "VirtuOS: An operating system with kernel virtualization," in *the 24th ACM Symposium on Operating Systems Principles*, pp. 116–132, 2013.

[30] D. Wentzlaff and A. Agarwal, "Factored operating systems (FOS): The case for a scalable operating system for multicores," *SIGOPS Operating Systems Review*, vol. 43, pp. 76–85, 2009.

[31] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. hua Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 43–57, 2008.

[32] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 12–25, 1995.

[33] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 143–156, 1997.

[34] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O," *Intel Technology Journal*, vol. 10, pp. 179–192, August 2006.

[35] R. Wojtczuk and J. Rutkowska, "Following the white rabbit: Software attacks against Intel VT-d technology," April 2011. Inivisible Things Lab.