A Virtualized Separation Kernel for Mixed-Criticality Systems

RICHARD WEST, YE LI, ERIC MISSIMER, and MATTHEW DANISH, Boston University

Multi- and many-core processors are becoming increasingly popular in embedded systems. Many of these processors now feature hardware virtualization capabilities, as found on the ARM Cortex A15 and x86 architectures with Intel VT-x or AMD-V support. Hardware virtualization provides a way to partition physical resources, including processor cores, memory, and I/O devices, among guest virtual machines (VMs). Each VM is then able to host tasks of a specific criticality level, as part of a mixed-criticality system with different timing and safety requirements. However, traditional virtual machine systems are inappropriate for mixed-criticality computing. They use hypervisors to schedule separate VMs on physical processor cores. The costs of trapping into hypervisors to multiplex and manage machine physical resources on behalf of separate guests are too expensive for many time-critical tasks. Additionally, traditional hypervisors have memory footprints that are often too large for many embedded computing systems. In this article, we discuss the design of the Quest-V separation kernel, which partitions services of different criticality levels across separate VMs, or *sandboxes*. Each sandbox encapsulates a subset of machine physical resources that it manages without requiring intervention from a hypervisor. In Quest-V, a hypervisor is only needed to bootstrap the system, recover from certain faults, and establish communication channels between sandboxes. This not only reduces the memory footprint of the most privileged protection domain but also removes it from the control path during normal system operation, thereby heightening security.

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

General Terms: Design, Performance

Additional Key Words and Phrases: Separation kernel, chip-level distributed system, mixed criticality

ACM Reference Format:
Richard West, Ye Li, Eric Missimer, and Matthew Danish. 2016. A virtualized separation kernel for mixed-criticality systems. ACM Trans. Comput. Syst. 34, 3, Article 8 (June 2016), 41 pages.
DOI: http://dx.doi.org/10.1145/2935748

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 emerging class of mixed-criticality systems, which involve tasks with different safety and timing requirements. For example, in avionics, the in-flight entertainment services are considered less critical than the flight control subsystem, because the consequences of failure are less severe. Similarly, in an automotive system, infotainment services for navigation and audio are less timing and safety critical than the management of antilock brakes and traction control.

A major challenge to mixed-criticality systems is the isolation of separate components with different criticality levels. This is the basis for software system standards

Authors' addresses: R. West, Y. Li, E. Missimer, and M. Danish, Computer Science Department, 111 Cummington Mall, Boston University, MA 02215, USA; emails: {richwest, liye, missimer, md}@cs.bu.edu.

DOI: http://dx.doi.org/10.1145/2935748

such as ARINC 653 [ARINC 2008] and the Multiple Independent Levels of Security (MILS) [Alves-Foss et al. 2006] architecture. Isolation has traditionally been achieved by partitioning components across distributed modules, which communicate over a network. For example, CAN bus and FlexRay networks are commonly found in automotive systems to connect upward of a hundred electronic control units running different services. Modern multicore processors provide an opportunity to replace networks of control units with a single integrated solution that consolidates services of different criticality levels onto separate cores. However, mixed-criticality systems for multicore processors still need to isolate components, so that timing and safety requirements are ensured, even in the presence of failures.

Hardware-assisted virtualization, found in many mainstream processors, provides a means to isolate separate components of a system. Intel VT-x, AMD-V, and various ARM Cortex A-series (e.g., A15) processors all support hardware virtualization. The Intel Atom E3800-series system-on-chip has VT-x capabilities and is found in single-board computers such as the Edison and Minnowboard MAX, as well as automobile In-Vehicle Infotainment (IVI) systems. However, virtualization has mostly been applied to server-class computing, with hypervisors designed to logically share a single hardware platform among a set of guests. It is still questionable whether virtualization is too inefficient and too unpredictable for use in embedded mixed-criticality systems with diverse timing and safety requirements.

Modern hypervisors such as Xen [Barham et al. 2003] and Linux-KVM [Habib 2008] provide management of CPUs, memory, and I/O devices on behalf of their guests. Traps into the hypervisor occur every time a guest needs to be scheduled, when a remapping of guest-to-machine physical memory is needed, or when an I/O device generates an interrupt. These traps are both unnecessary and in conflict with the timing requirements in mixed-criticality systems. For example, a guest system might implement a real-time task scheduling policy that is compromised by the lack of predictability within the hypervisor. Rather than having a single hypervisor manage machine resources for all its guests, it would be better to allow separate guests with different criticality levels to manage their own physical resources.

This article presents our Quest-V separation kernel [Rushby 1981], which uses hardware-assisted virtualization to achieve efficient resource partitioning and performance isolation for subsystem components. Quest-V avoids traps into a hypervisor when making scheduling, memory, and I/O management decisions. Instead, all machine resources are partitioned at boot time among system components that are capable of directly managing their assigned hardware resources.

A key point of this article is to show that hardware virtualization offers a way to divide machine resources into subsets that are managed independently by their guest software with almost no additional virtualization costs. To the best of our knowledge, Quest-V is the first real-time chip-level separation kernel that uses hardware virtualization for space and time isolation of guest services, without the runtime overheads of a hypervisor. Quest-V leverages our earlier work on the Quest real-time operating system, to provide timing guarantees on thread execution, interrupt handling, and communication between separate guest services. Quest-V integrates component services in different guest domains into a tightly coupled, real-time distributed system.

Experiments show how Quest-V is able to make efficient use of CPU, memory, and I/O partitioning. We show how a Linux front-end (guest) system is 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 to when it is run on a nonvirtualized Linux system. Similarly, *netperf* running on a Linux guest in Quest-V achieves better network bandwidth performance than when running on Xen. Quest-V guest services are able to maintain functionality in the presence of faults

in other 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. Section 5 proposes potential architecture improvements to facilitate the construction of an efficient and predictable separation kernel. An overview of related work is provided in Section 6. Finally, conclusions and future work are discussed in Section 7.

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 OSs 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 provides a way to separate different groups of services, depending on their criticality (or importance) to overall system functionality.

Quest-V uses hardware virtualization to establish a collection of *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 multikernels such as Barrelfish [Baumann et al. 2009]. 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, improves 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 criticality levels are able to be isolated from one another, or even replicated to guarantee their operation.

(3) Highest Safe Privilege – Rather than adopting a principle of *least* privilege for software services, as is done in microkernels, a virtualized system supports 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 written with traditional kernel privileges are isolated from equally privileged services in other guest domains. This avoids the communication costs typically associated with microkernels, to request services in different protection domains. For example, a sandbox does not need a separation of guest kernel and user code. Instead, a sandbox might encapsulate code within a single privilege level, avoiding the cost of system calls to execute machine instructions that require more privilege. Virtualization allows components of a system to be granted the highest safe privilege level without being able to compromise the entire system spanning multiple guest domains.

(4) Minimal Trusted Code Base – A microkernel attempts to provide a minimal trusted code base for the services it supports. However, it must still be accessed as part of interprocess communication and basic operations such as coarse-grained memory management. Monitors form a trusted code base in the Quest-V separation kernel.

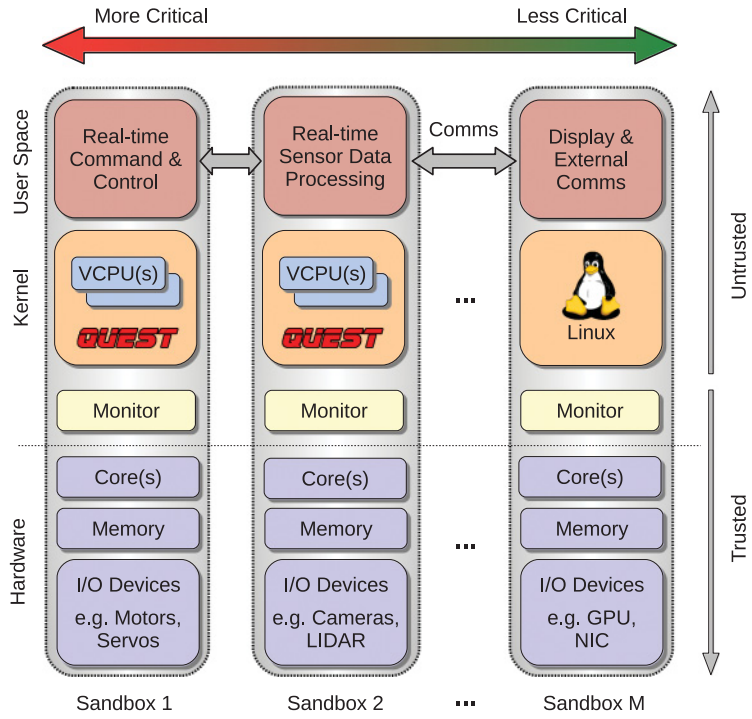

Fig. 1. Example Quest-V architecture overview.

Access to these is *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 are limited to a small memory footprint.

It should be noted that systems supporting process address spaces provide a way to logically isolate resources. One process cannot directly access the address space of another process without mediation by a trusted kernel. Similarly, a process is granted capabilities to access devices using abstractions such as file descriptors, and is restricted access to CPU resources by a system scheduler. However, a process differs from a sandbox in our system by requiring the runtime support of a more privileged kernel to request resources it cannot directly access. A sandbox, in contrast, provides an environment for code contained within to directly access resources that have been made accessible by the virtualization logic. Once the scope of system resources has been established for a sandbox, it is free to access those resources without requiring additional, more privileged, software.

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. A separate monitor within each sandbox 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.

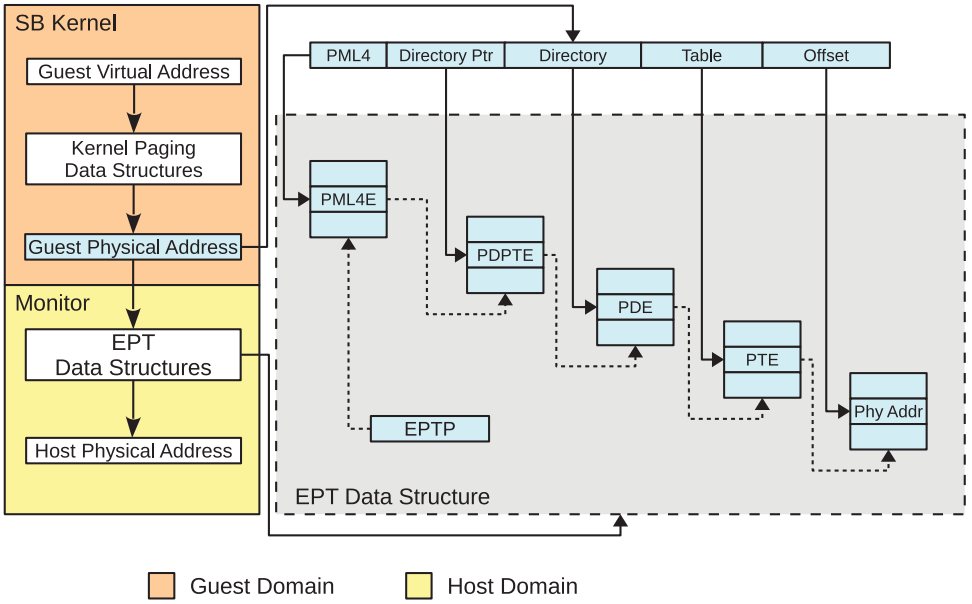

Fig. 2. Extended page table mapping.

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 vehicle infotainment services are partitioned in a sandbox that has access to a local display device. A non-real-time Linux system is used in this case, to manage a network interface (NIC) that communicates with other vehicles and the surrounding environment, via a vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communication link.

3.1. Distributed Monitors

Traditional virtual machine systems use a single hypervisor to multiplex physical resources among a set of guests. Quest-V replaces a central hypervisor with a separate trusted monitor for each sandbox. Monitors are event-driven, passive identities that do not interfere with the normal operations of the guests they service. The distributed monitor design leads to the following benefits:

(1) Efficiency and Predictability – Shared access to the code and data of a single monitor for all guests is avoided. This improves system predictability by eliminating variable synchronization delays. It also reduces resource contention and increases system efficiency. As each monitor only services one sandbox, the runtime overhead to determine which guest needs its service is also avoided.

(2) Functional Diversity – Monitors in Quest-V are customizable for the needs of a specific guest. The implementation of one monitor may differ from that of another. This functional diversity makes it possible to optimize the performance, or alter the capability, of a specific monitor without increasing the complexity of others. The simplicity and functional diversity of monitors make it both easier to formally verify their correctness and harder to exploit the same security vulnerability. A weakness in one monitor might not be present in another monitor whose implementation is different. This makes it difficult for an attacker to compromise every monitor in the system.

(3) Fault Tolerance – Since all the monitors operate at the same hardware privilege level, the failure of a single monitor threatens the integrity of all others. However, a distributed monitor design enables functionality to be replicated so that system availability is maintained, even if faults occur [Avižienis 1967, 1975, 1985; Parnas et al. 1990].

(4) Opportunities for New Security Models – With multiple monitors, one can observe suspicious behavior in another monitor, including attempts to compromise it or other parts of the system. Techniques such as Triple Modular Redundancy (TMR) [Lyons and Vanderkulk 1962] enable one monitor to check the state of another, to identify possible inconsistencies. As long as one monitor is functioning correctly, it is possible to intercept any security breaches by an adversary, who must compromise all monitors before the system is subverted.

Despite these benefits, the duplication of functionality in the distributed monitor design inevitably increases the total memory footprint of all Quest-V monitors. However, the amount of added memory overhead is small, as each monitor's code fits within 4KB. The monitor code needed after system initialization is about 400 lines to support both Linux and Quest sandboxes. The EPTs take additional data space, but 12KB is enough for a 1GB sandbox address space, and these data structures have to be allocated for each sandbox in any case. Adding support for new security models will undoubtedly increase the code complexity of monitors but offers potential benefits in terms of system availability. This tradeoff will be investigated in future work.

3.2. Resource Partitioning

Quest-V supports configurable partitioning of CPU, memory, and I/O resources among guests. Resource partitioning is mostly static, taking place at boot time, with the exception of some memory allocation at runtime for dynamically created communication channels between sandboxes. Static resource partitioning eliminates the need for a complex hypervisor to multiplex and allocate machine resources among guests at runtime. Instead, simpler logic is all that is needed to partition resources at boot time. The monitors in Quest-V are therefore removed from the control path in normal system operation. This has the potential to heighten system security.

CPU Partitioning – In Quest-V, scheduling is performed within each sandbox on a dedicated set of processor cores. 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 among guests. Contention on a single global queue is avoided, as each sandbox manages its own local scheduling queue. Similarly, each sandbox is free to implement its own scheduling policy, simplifying resource management.

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 that are walked by hardware to translate guest virtual addresses to guest physical addresses. EPT structures are then additionally walked by hardware to complete the translation to host physical addresses.

The base physical address of an EPT PML4 table is stored in the EPTP VM-Execution Control field of a *virtual machine control structure* (VMCS). The EPTP refers to a four-level paging structure, discounting the final page offset. Each entry in a guest's paging structure is a *guest physical address*, requiring five memory accesses to walk the EPT and obtain a corresponding host physical address. Quest-V sandboxes are

currently limited to 32-bit address spaces, which means guests use a two-level paging scheme (discounting the byte offset in the guest physical page) for guest-virtual-to-guest-physical-address translation. Consequently, a total of 3×5 memory accesses are required in the worst case to translate a guest virtual to host physical address.

On modern Intel x86 processors with EPT support, pages are minimally set to 4KB in size. 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 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.

Guest-virtual-to-machine-physical-address translations are cached by hardware TLBs. A VM-Exit causes a TLB flush when not using 64-bit guests with VM identifiers. By avoiding exits into monitor code, each sandbox operates with similar performance to that of non-VM systems with conventional page-based virtual address spaces. For a TLB hit, the added costs of walking EPTs are eliminated.

Cache Partitioning – Microarchitectural resources such as caches and memory buses provide a source of contention on multicore platforms. Quest-V uses hardware performance counters to establish cache occupancies for different sandboxes [West et al. 2008, 2010, 2013]. For each software thread, m_l represents the last-level cache misses on the local core in an interval when the thread is executing on the processor, while m_o represents the misses caused by all other threads on remote cores in the same time. Quest-V also uses h_l and h_o to measure local and other hits in the same time window for enhanced occupancy prediction on set associative caches. Using just misses, the updated occupancy estimate of thread τ_i at time t' is

$$E_i(t') = E_i(t) + [1 - E_i(t)/C] \cdot m_l - [E_i(t)/C] \cdot m_o,$$

where C is the number of cache lines, and $[t, t']$ represents a scheduling interval. Using additional hit information, Quest-V is able to enhance shared cache occupancy estimates to accommodate for set associative caches with least-recently-used line replacement schemes, among others [West et al. 2010, 2013]. For Intel processors predating Westmere and Sandy Bridge, there are insufficient per-core performance counters to achieve enhanced hit-based cache occupancy predictions. However, recent processors provide the ability to capture per-core and per-chip (socket) last-level cache hits and misses. On the Westmere and Sandy Bridge processors, there are two off-core response-model-specific registers that provide sufficient capability to acquire per-core misses and hits. These are in addition to performance counters to acquire global misses and hits across all cores.

Cache occupancy estimates are then used by dynamic page coloring techniques as described in our work on COLORIS [Ye et al. 2014] to partition shared caches between sandboxes [Liedtke et al. 1997; Albonesi 1999; Chang and Sohi 2007; Dybdahl et al. 2006; Iyer 2004; Kim et al. 2004; Liu et al. 2004; Rafique et al. 2006; Ranganathan et al. 2000; Srikantaiah et al. 2008; Suh et al. 2004]. Additional work is ongoing to account for contention on other microarchitectural resources, including on-chip buses and interconnects.

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.

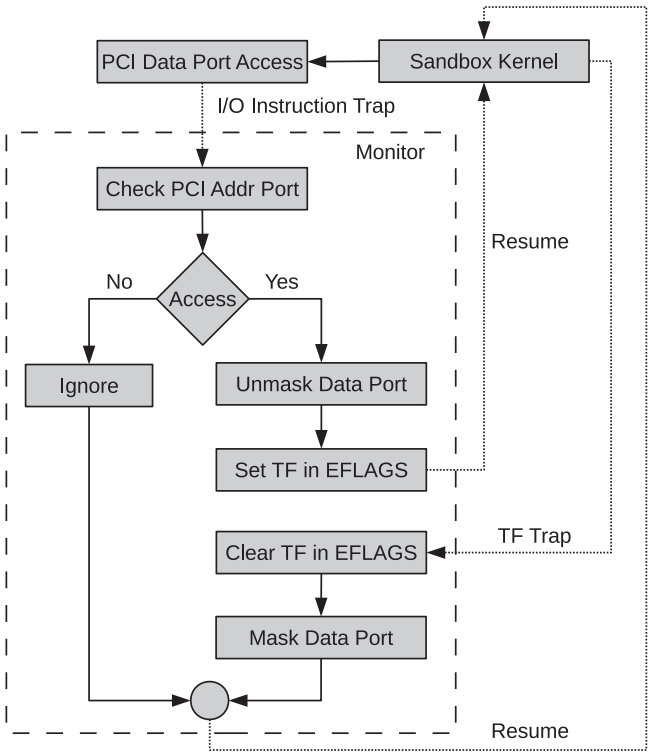

Fig. 3. PCI configuration space protection.

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 among 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* [PCI-SIG 2015]. 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

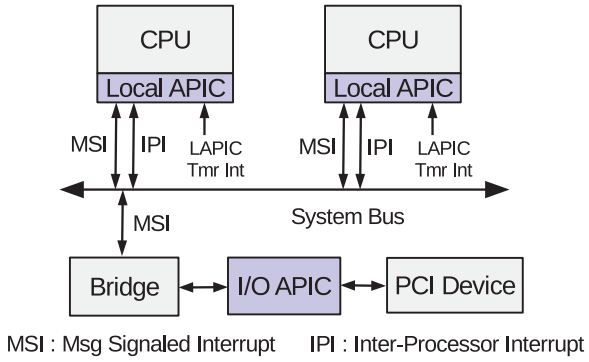

Fig. 4. APIC configuration.

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 among 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. Quest-V establishes a series of VMCSs to manage the operation of each sandbox. For Intel VT-x processors, the VM-Execution Control fields of a VMCS are configured so that a VM-Exit does not occur when an interrupt is delivered from the IOAPIC to the Local APIC of the corresponding core. Instead, interrupts are delivered through the guest sandbox's interrupt-descriptor table (IDT).

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 among 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.

3.3. Quest Sandbox Support

Quest-V is built upon the Quest real-time operating system for multicore processors. The configuration of a Quest-V separation kernel, including the number of sandboxes, the mapping of machine resources to sandboxes, and the software environment within each sandbox, is established at build time. Once configured, a Quest-V system first boots

up a separate Quest instance for each sandbox. This requires all application processors (i.e., separate cores from the bootstrap processor core) to fork both an instance of Quest and a monitor into a dedicated region of machine physical memory. At this point, each monitor establishes a VMCS for each core in its corresponding sandbox. The sandboxes are then launched with a separate Quest kernel initially running in each guest domain. Quest then acts as a boot loader within a sandbox that is configured to run another OS, such as Linux.

Although, by default, Quest-V forks copies of the same monitor code into every sandbox, it is possible to implement different monitor functionalities according to specific sandbox needs. A Linux guest, for example, needs a different boot loader from that of a native Quest sandbox. It also needs logic that traps service requests that do not exist in Quest. As mentioned earlier, for security, functional diversity avoids duplicating the same exploitable weakness in every monitor. For these reasons, it is possible to compile different monitor instances into object code that is loaded when a sandbox is bootstrapped.

A Quest-V separation kernel consists of the code to run multiple sandboxed guests, while Quest provides the boot logic and default execution environment for each sandbox. Moreover, as will be seen later, Quest's real-time features provide a means to support timing guarantees within specific sandboxes, as well as predictable intersandbox communication among sandboxes running Quest services.

We originally developed the Quest kernel for real-time and embedded systems on multicore processors to study the combined scheduling of tasks and interrupts, and the predictable management of shared microarchitectural resources such as last-level caches. 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.[1] Each Quest-V monitor is given access to a Quest kernel address space so that direct manipulation of kernel objects during monitor traps are possible.

Virtual CPUs – In Quest, *virtual CPUs* (VCPUs) form the fundamental abstraction for scheduling and temporal isolation of the system. Here, temporal isolation means that each VCPU is guaranteed its share of CPU cycles without interference from other VCPUs.

The concept of a VCPU is similar to that in virtual machines [Adams and Agesen 2006; Barham et al. 2003], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs)[2] represented as VCPUs to each of the guest virtual machines. VCPUs exist as kernel 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 [Banga et al. 1999] 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 common with bandwidth-preserving servers [Abeni and Buttazzo 1998; Deng et al. 1997; Spuri and Buttazzo 1996], 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

[1]Device drivers, support for a TCP/IP network stack, and ACPI functionality add several hundred thousand lines of code.

[2]We define a PCPU to be either a conventional CPU, a processing core, or a hardware thread in a simultaneous multithreaded (SMT) system.

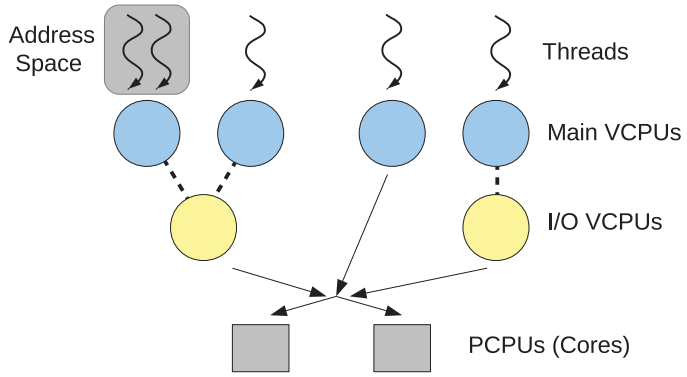

Fig. 5. VCPU scheduling hierarchy.

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 those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

A 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 [Zhang and West 2006], which may have deferred execution when threads associated with higher-priority VCPUs having available budgets are runnable. The flexibility of Quest-V allows 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.

VCPU API – VCPUs form the basis for managing time as a first-class resource: VCPUs are specified time bounds for the execution of corresponding threads. Stated another way, every executable control path in Quest is mapped to a VCPU that controls scheduling and time accounting for that path. The basic API for VCPU management is described next. It is assumed this interface is managed only by a user with special privileges.

(1) *int vcpu_create(struct vcpu_param *param)* – Creates and initializes a new Main or I/O VCPU. The function returns an identifier for later reference to the new VCPU. If the `param` argument is `NULL`, the VCPU assumes its default parameters. The current default is a Main VCPU using a `SCHED_SPORADIC` policy [Stanovich et al. 2010; Sprunt et al. 1989]. The `param` argument points to a structure that is initialized with the following fields:

```
struct vcpu_param {
   int vcpuid; // Identifier
   int policy; // SCHED_SPORADIC or SCHED_PIBS
   int mask; // PCPU affinity bit-mask
   int C; // Budget capacity
   int T; // Period
}
```

The `policy` is `SCHED_SPORADIC` for Main VCPUs and `SCHED_PIBS` for I/O VCPUs. `SCHED_PIBS` is a *priority-inheritance bandwidth-preserving* policy, which is

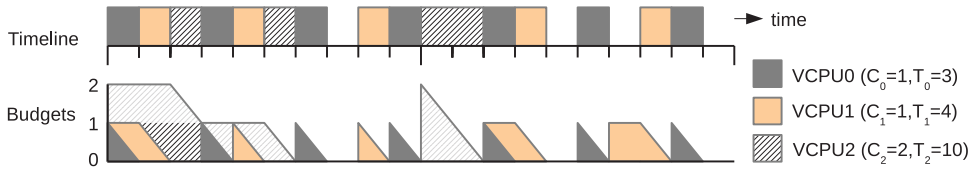

Fig. 6. Example VCPU schedule.

described in further detail later. The `mask` is a bitwise collection of processing cores available to the VCPU. It restricts the cores on which the VCPU can be assigned and to which the VCPU can be later migrated. The remaining VCPU parameters control the budget and period of a sporadic server, or the equivalent bandwidth utilization for a PIBS server. In the latter case, the ratio of C and T is all that matters, not their individual values.

Upon success, a `vcpuid` is returned for a new VCPU. An admission controller must check that the addition of the new VCPU meets system schedulability requirements; otherwise, the VCPU is not created and an error is returned.

(2) *int vcpu_destroy (int vcpuid, int force)* – Destroys and cleans up state associated with a VCPU. The count of the number of threads associated with a VCPU must be 0 if the `force` flag is not set. Otherwise, destruction of the VCPU will force all associated threads to be terminated.

(3) *int vcpu_setparam (struct vcpu_param *param)* – Sets the parameters of the specified VCPU referred to by `param`. This allows an existing VCPU to have new parameters from those when it was first created.

(4) *int vcpu_getparam (struct vcpu_param *param)* – Gets the VCPU parameters for the next VCPU in a list for the caller's process. That is, each process has associated with it one or more VCPUs, since it also has at least one thread. Initially, this call returns the VCPU parameters at the head of a list of VCPUs for the calling thread's process. A subsequent call returns the parameters for the next VCPU in the list. The current position in this list is maintained on a per-thread basis. Once the list end is reached, a further call accesses the head of the list once again.

(5) *int vcpu_bind_task (int vcpuid)* – Binds the calling task, or thread, to a VCPU specified by `vcpuid`.

Functions *vcpu_destroy*, *vcpu_setparam*, *vcpu_getparam*, and *vcpu_bind_task* all return 0 on success, or an error value.

Main and I/O VCPU Scheduling – By default, VCPUs act like Sporadic Servers [Sprunt et al. 1989]. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [Liu and Layland 1973]. This is significant, given I/O events may occur at arbitrary (aperiodic) times, potentially triggering the wakeup of blocked tasks (again, at arbitrary times) having higher priority than those currently running. RMS analysis ensures that each VCPU is guaranteed its share of CPU time, U_V, in finite windows of real time.

An example schedule is provided in Figure 6 for three Main VCPUs, whose budgets are depleted when a corresponding thread is executed. Priorities are inversely proportional to periods. As can be seen, each VCPU is granted its real-time share of the underlying PCPU.

Our experience shows that Sporadic Servers work well for managing CPU bandwidth on behalf of tasks running on Main VCPUs. This is because tasks typically have time slices in milliseconds and, although they may block (e.g., on I/O) before exhausting their time slice, they tend to execute with sufficient granularity to avoid too

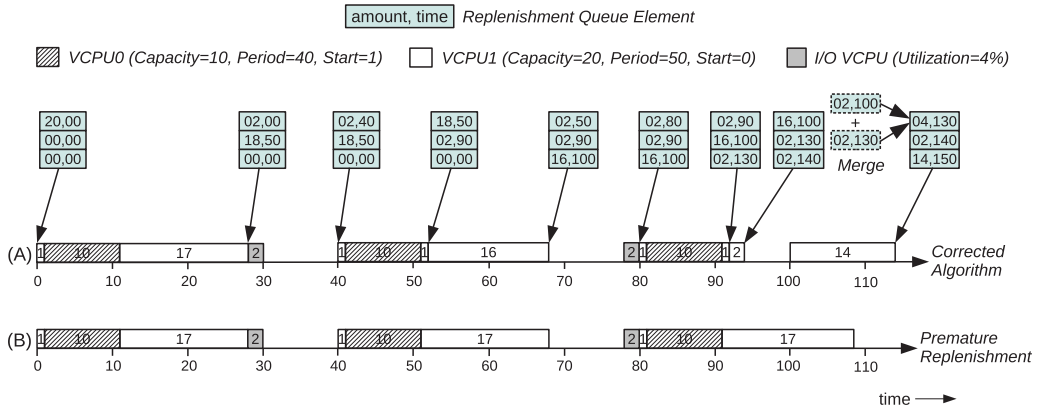

Fig. 7. Sporadic Server replenishment list management.

much fragmentation of their Sporadic Server budgets. In contrast, device interrupts often occur in bursts, and the time spent handling them is sufficiently short to heavily fragment the use of Sporadic Server budgets. The correct time management of Sporadic Servers requires the posting of budget replenishment events, and the maintenance of lists that track when a server has the capacity to execute on a physical CPU. The reprogramming of one-shot timers to post events when the budget becomes available is relatively expensive compared to the amount of time to handle interrupts. Consequently, replenishment list entries are often merged with later ones, to provide larger chunks of server budget and reduce the number of timing events. However, this delays the time when a server is eligible to execute at its foreground priority, and therefore reduces its effective CPU bandwidth.

To address the fragmentation problem with Sporadic Servers, I/O VCPUs use a different policy to handle interrupts. Our PIBS approach associates a utilization factor, $U_j \mid 0<U_j<1.0$, with an I/O VCPU V_j. If V_j is associated with a device requested by a thread running on a Main VCPU, V_i, then V_j inherits the priority of V_i. Essentially, this allows the I/O handling to be prioritized at the level of the requester. A thread may block on its Main VCPU until I/O processing (on the corresponding I/O VCPU) is complete. In this case, V_j is assigned a dynamic priority based on the period of V_i, and it guarantees a budget of $C_j{=}U_j{\times}T_i$ over a period of time T_i. Unlike a Sporadic Server that requires a *list* of budget replenishments, PIBS requires only one dynamically calculated budget update. If we assume V_j executes for $C_{actual}{\leq}C_j$ from time t when an interrupt occurs, then it receives a budget update of C_{actual} at time $t + C_{actual}/U_j$. This maintains the bandwidth constraint, U_j, for I/O VCPU V_j.

In Quest, there is no notion of a periodic timer interrupt to update the system clock time. Instead, the system is event driven, using per-processing core-local APIC timers to replenish VCPU budgets as they are consumed during thread execution. We use the algorithm proposed by Stanovich et al. [2010] to correct for early replenishment and budget amplification in the POSIX specification.

Figure 7 shows an example schedule for two Main VCPUs and one I/O VCPU for a device such as a gigabit Ethernet card. VCPU0 has a higher priority than VCPU1, because of its smaller period. The I/O VCPU has a dynamically calculated priority depending on the Main VCPU it serves. In this example, we assume the I/O VCPU only serves VCPU1, while VCPU0 is purely CPU bound.

Schedule (A) shows a hypothetical timeline in Quest, while Schedule (B) shows an equivalent execution timeline using Sporadic Server scheduling according to the POSIX

specification. The state of the budget replenishment list for VCPU1 is also shown at different times during its execution. Each item is shown as a tuple $(amout, time)$ and is originally set to the full capacity of VCPU1 at time $t = 0$. In both schedules, VCPU1 begins execution at $t = 0$, only to be preempted by higher-priority VCPU0 at $t = 1, 41, 81$, and so forth. By $t = 28$, VCPU1 has amassed a total of 18 units of execution time and then blocks on an I/O request. In this example, VCPU1 issues an I/O request every 18 time units of execution. For simplicity, we show the I/O VCPU immediately executing at $t = 28$ for its full budget, which is calculated from its utilization (here, 4%) and the period of the Main VCPU it serves (here, 50 time units). Thus, the full budget is calculated as $0.04 \times 50 = 2$. Due to the execution of other VCPUs (not shown) in the interval $t = [30, 40]$, VCPU1 is delayed resumption until $t = 40$. Similarly, the I/O VCPU is not eligible to execute again until $t = 28 + 2/0.04 = 78$. This is to maintain the CPU bandwidth of the I/O VCPU at 4%.

When VCPU1 blocks at $t = 28$, it splits its original budget into two parts: a remaining budget that has not been consumed and a future replenishment for the amount used. Thus, the replenishment item $(20, 00)$ is split into two parts: $(02, 00)$ and $(18, 50)$, with the latter being available one period after it is first consumed. The next adjustment to the list occurs at $t = 40$, when VCPU1 resumes. The available time of its first replenishment is simply updated to the current time, which is a reference for the next future replenishment. Budgets are only split when a VCPU blocks, while they are potentially merged when a VCPU wakes up. Two replenishments, $r_1 = (a_1, t_1)$ and $r_2 = (a_2, t_2) \,|\, t_2 > t_1$, are merged into one replenishment $r_3 = (a_1 + a_2, t_1)$ if $t_1 + a_1 \geq t_2$. Similarly, for finite replenishment lists that will otherwise overflow, one item is merged with a later item, since this will not allow a server to receive more resource bandwidth than it should. At preemption points, a VCPU does not split or merge replenishments, but simply keeps track of how long it has been executing to that point.

Following along Schedule (A), VCPU1 is preempted at $t = 41$ and on resumption at $t = 51$ observes it has consumed one time unit of its first replenishment $(02, 40)$. Thus, at $t = 52$, it removes the exhausted first replenishment from its list and adds a future replenishment of $(02, 90)$, which is one period later than when it is first consumed. Once again, VCPU1 blocks at $t = 68$ and its I/O request is served by the I/O VCPU at $t = 78$, when it is eligible to execute again.

With Schedule (A), VCPU1 receives the correct utilization of 40% in the first 100 time units. Although it actually receives 21 time units in the interval $t = [50, 100]$, the system ensures that it is safe to do this, to compensate for time it did not obtain in the interval $t = [0, 50]$ when it blocked before exhausting all its budget capacity. With Schedule (B), VCPU1 actually receives 46 time units over the first 100. The problem is triggered by the blocking delays of VCPU1. Schedule (A) ensures that when a VCPU blocks (e.g., on an I/O operation), on resumption of execution it effectively starts a new replenishment phase. This means the replenishment for $(18, 50)$ should be distinct from the available budget of two time units at $t = 40$. However, Schedule (B) incorrectly merges these two replenishments at $t = 51$ due to the preemption by VCPU0.

Scheduling within a Quest sandbox ensures that the sum of replenishment amounts for all list items does not exceed the budget capacity of the corresponding VCPU (here, 20, for VCPU1). Also, no future replenishment R for a VCPU, V, executing from t to $t + R$ is allowed to occur before $t + T_V$. For completeness, we show in Schedule (A) at $t = 114$ the merging of $(02, 100)$ and the next item $(02, 130)$ to yield a single item $(04, 130)$. This type of merging is needed if the replenishment lists cannot exceed a maximum length (here, three items). It causes Sporadic Servers to achieve an effective utilization lower than their desired value, which is why we use PIBS for I/O processing of short-lived interrupt handlers. Further details about VCPU scheduling in Quest are available in our accompanying paper [Danish et al. 2011].

Since each sandbox kernel in Quest-V supports local scheduling of its allocated resources, there is no notion of a global scheduling queue. Forked threads are by default managed in the local sandbox but are migratory to remote sandboxes along with their VCPUs, according to load constraints or affinity settings of the target VCPU [Li et al. 2014]. Although each sandbox is isolated in a special guest execution domain controlled by a corresponding monitor, the monitor is not needed for scheduling purposes. This avoids costly virtual machine exits and re-entries (i.e., VM-Exits and VM-Resumes) as would occur with hypervisors such as Xen [Barham et al. 2003] that manage multiple separate guest OSs.

Temporal Isolation – Quest provides temporal isolation of VCPUs assuming the total utilization of a set of Main and I/O VCPUs within each sandbox do not exceed specific limits. Each Quest sandbox determines the schedulability of its local VCPUs independently of all other sandboxes. For cases where a sandbox is associated with one PCPU, n Main VCPUs, and m I/O VCPUs, we have the following:

$$\sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n\left(\sqrt[n]{2} - 1\right).$$

Here, C_i and T_i are the budget capacity and period of Main VCPU, V_i, respectively. U_j is the utilization factor of I/O VCPU, V_j [Danish et al. 2011].

The utilization bound described previously is an extension of the rate-monotonic scheduling bound [Liu and Layland 1973], to include the demand for CPU resources by I/O VCPUs running the PIBS policy. Timing analysis across multiple cores is more complex, and for this reason we focus on the assignment of threads and VCPUs to per-core scheduling queues. If a sandbox encapsulates more than one core, the core-local schedulers negotiate the migration of threads and VCPUs to ensure (1) each core's utilization bound is not violated, and (2) other objectives are met. We currently support other objectives such as balancing utilization across cores or assigning threads and VCPUs so that corunners avoid heavy cache contention.

3.4. Linux Sandbox Support

In addition to Quest real-time kernels, Quest-V is also designed to support other third-party sandbox systems such as Linux and AUTOSAR OS [AUTOSAR 2015]. Currently, we have successfully ported a Puppy Linux [Kauler 2015] 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 Quest kernel. This means a Quest sandbox needs to be initialized first and Linux is started in the same sandbox via a boot loader 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 hardware 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 fewer than 50 lines. These changes are mainly focused on limiting Linux's view of available physical memory and handling I/O device direct memory access (DMA) offsets caused by memory virtualization.

An example memory layout of Quest-V with a Linux sandbox on a four-core processor is shown in Figure 8. Even though the Linux kernel's view of (guest) physical memory is contiguous from address 0x0, the kernel is actually loaded after all Quest kernels in machine physical memory. Quest-V does not require a hardware I/O memory management unit (IOMMU). We therefore patched the Linux kernel DMA layer to make it

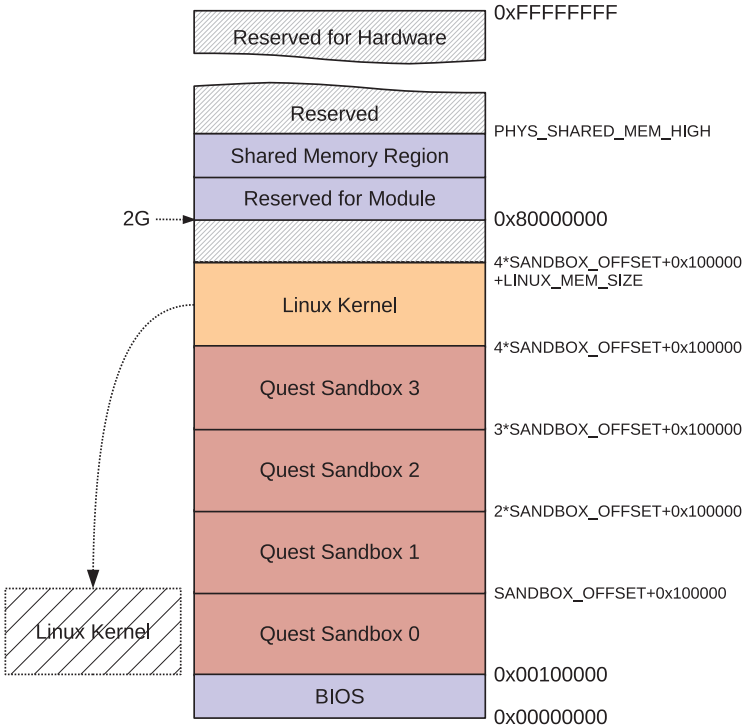

Fig. 8. Quest-V physical memory layout with Linux.

aware of this offset between guest physical and machine physical memory addresses during I/O device DMA. However, for more robust security, an IOMMU is required to restrict the DMA address range of an I/O device to a specific sandbox.

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.

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.

3.5. 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 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 Quest sandboxes without compromising the CPU shares allocated to noncommunicating tasks.

We developed a similar library for communication between processes in Linux and Quest sandboxes. However, in the current implementation, predictable communication between Linux and Quest sandboxes is not guaranteed. Memory channels shared between Linux and other Quest sandboxes are for non-time-critical communication only.

Inter-sandbox communication is necessary in Quest-V to access remote services and nonlocal resources. The communication mechanism is rich enough to support the migration of threads and their VCPUs. Configurable policies decide whether it is preferential to make a remote procedure call into another sandbox or simply migrate a thread and its VCPU to a destination where it will have local access to a required service or resource. Quest-V supports the migration of process address spaces, including threads and VCPUs, via shared-memory communication channels. However, for large address spaces, multiple message transfers are required via a shared-memory channel, potentially affecting the timing guarantees associated with a migrated VCPU. Quest-V provides an alternative scheme whereby an IPI is sent from a source to destination sandbox, to initiate a direct memory copy of an address space. This approach requires the destination monitor to temporarily map the source sandbox pages containing the migrating address space. Once a copy is made of a migrating address space, it is deleted from the source sandbox. Quest-V is able to successfully migrate address spaces without violating their corresponding VCPU timing requirements or the requirements of other VCPUs in the destination. Further details about predictable communication and migration in Quest-V are discussed in an accompanying paper [Li et al. 2014], which addresses the challenges of not having a global scheduler or common clock for all sandboxes.

3.6. Fault Recovery

Fault detection is, itself, a topic worthy of separate discussion. In this article, we assume the existence of techniques to identify faults. In Quest-V, faults are easily detected if they generate EPT violations, thereby triggering control transfer to a corresponding monitor. More elaborate schemes for identifying faults will be covered in our future work. In this section, we explain the details of how fault recovery is performed without requiring a full system reboot.

The distributed design adopted by Quest-V allows for fault recovery either in the local sandbox, where the fault occurred, or in a remote sandbox that is presumably unaffected. Upon detection of a fault, a method for passing control to the local monitor is required. If the fault does not automatically trigger a VM-Exit, it can be forced by a fault handler issuing an appropriate instruction. An astute reader might assume that carefully crafted malicious attacks to compromise a system might try to rewrite fault detection code within a sandbox, thereby preventing a monitor from ever gaining control. First, this should not be possible if the fault detection code is presumed to exist in

read-only memory, which should be the case for the sandbox kernel text segment. This segment cannot be made write accessible since any code executing with the sandbox will not have access to the EPT mappings controlling host memory access. However, it is still possible for malicious code to exist in writable regions of a sandbox, including parts of the data segment.

To guard against compromised sandboxes that lose the capability to pass control to their monitor as part of fault recovery, certain procedures can be adopted. One such approach would be to periodically force traps to the monitor using a preemption timeout [Intel Corporation 2015]. This way, the fault detection code could itself be within the monitor, thereby isolated from any possible tampering from a malicious attacker or faulty software component. Many of these techniques are still under development in Quest-V and will be considered as we progress with this work.

Assuming that either a fault detection event has triggered a trap into a monitor or the monitor itself is triggered via a preemption timeout and executes a fault detector, we now describe how the handling phase proceeds.

Local Fault Recovery – In the case of local recovery, the corresponding monitor is required to release the allocated memory for the faulting components. If insufficient information is available about the extent of system damage, the monitor may decide to reinitialize the entire local sandbox, as in the case of initial system launch. Any active communication channels with other sandboxes may be affected, but the remote sandboxes that are otherwise isolated will be able to proceed as normal. As part of local recovery, the monitor may decide to replace the faulting component, or components, with alternative implementations of the same services. For example, an older version of a device driver that is perhaps not as efficient as a recent update, but is perhaps more rigorously tested, may be used in recovery. Such component replacements can lead to system robustness through functional or implementation diversity [Williams et al. 2009]. That is, a component suffering a fault or compromised attack may be immune to the same fault or compromising behavior if implemented in an alternative way. The alternative implementation could, perhaps, enforce more stringent checks on argument types and ranges of values that a more efficient but less safe implementation might avoid. Observe that alternative representations of software components could be resident in host physical memory and activated via a monitor that adjusts EPT mappings for the sandboxed guest.

Remote Fault Recovery – Quest-V also supports the recovery of a faulty software component in an alternative sandbox. This may be more appropriate in situations where a replacement for the compromised service already exists, and which does not require a significant degree of reinitialization. While an alternative sandbox effectively resumes execution of a prior service request, possibly involving a user-level thread migration, the corrupted sandbox can be healed in the background. This is akin to a distributed system in which one of the nodes is taken offline while it is being upgraded or repaired.

In Quest-V, remote fault recovery involves the local monitor identifying a target sandbox. There are many possible policies for choosing a target sandbox that will resume an affected service request. However, one simple approach is to pick any available sandbox in random order, or according to a round-robin policy. In more complex decision-making situations, a sandbox may be chosen according to its current load. Either way, the local monitor informs the target sandbox via an IPI. Control is then passed to a remote monitor, which performs the fault recovery. Although out of the scope of this article, information needs to be exchanged between monitors about the actions necessary for fault recovery and what threads, if any, need to be migrated.

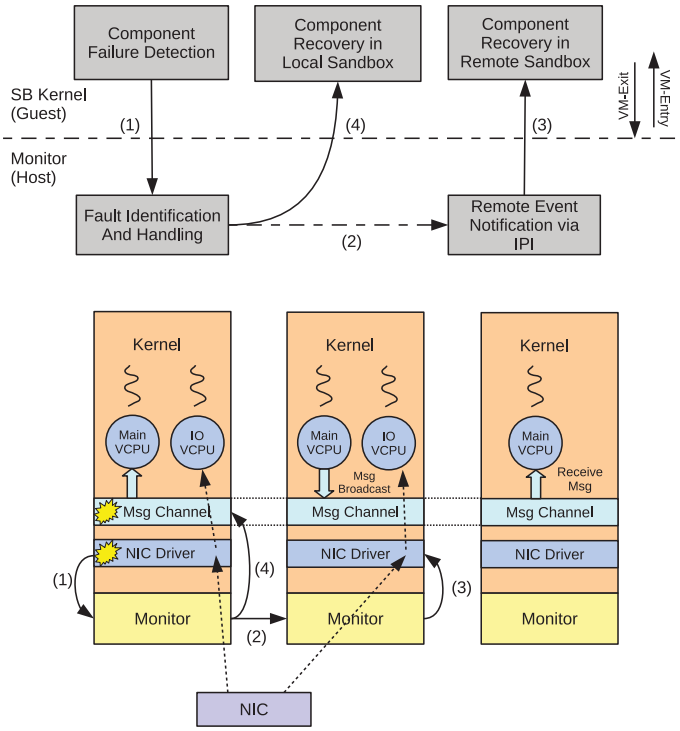

Fig. 9. NIC driver remote recovery.

An example of remote recovery involving a network interface cased (NIC) driver is shown in Figure 9. Here, an IPI is issued from the faulting sandbox kernel to the remote sandbox kernel via their respective monitors, in order to kick-start the recovery procedures after the fault has been detected. For the purposes of our implementation, an arbitrary target sandbox was chosen. The necessary state information needed to restore service is retrieved from shared memory using message passing if available. In our simple tests, we assume that the NIC drivers state is not recovered, but instead the driver is completely reinitialized. This means that any prior in-flight requests using the NIC driver will be discarded.

The major phases of remote recovery are listed in both the flow chart and diagram of Figure 9. In this example, the faulting NIC driver overwrites the message channel in the local sandbox kernel. After receiving an IPI, the remote monitor resumes its sandbox kernel at a point that reinitializes the NIC driver. The newly selected sandbox responsible for recovery then redirects network interrupts to itself. Observe that in general this may not be necessary because interrupts from the network may already be broadcast and, hence, received by the target sandbox. Likewise, in this example, the target sandbox is capable of influencing interrupt redirection via an I/O APIC because of established capabilities granted by its monitor. It may be the case that a monitor does not allow such capabilities to be given to its sandbox kernel, in which case the corresponding monitor would be responsible for the interrupt redirection.

When all the necessary kernel threads and user processes are restarted in the remote kernel, the network service will be brought up online. In our example, the local sandbox (with the help of its monitor) will identify the damaged message channel and try to restore it locally in step 4.

Table I. Monitor Trap Count During Linux Sandbox Initialization

	Exception	CPUID	VMCALL	I/O Inst	EPT Violation	XSETBV
No I/O Partitioning	0	502	2	0	0	1
I/O Partitioning	10,157	502	2	9,769	388	1
I/O Partitioning (Block COM and NIC)	9,785	497	2	11,412	388	1

In the current implementation of Quest-V, we assume that all recovered services are reinitialized and any outstanding requests are either discarded or can be resumed without problems. In general, many software components may require a specific state of operation to be restored for correct system resumption. In such cases, we would need a scheme similar to those adopted in transactional systems to periodically checkpoint the recoverable state. Snapshots of such a state can be captured by local monitors at periodic intervals, or other appropriate times, and stored in memory outside the scope of each sandbox kernel.

3.7. Quest Multikernel

On platforms without hardware virtualization support, a configuration option in Quest-V enables separate sandboxes to be replaced with multiple kernel instances. This differs from the traditional symmetric multiprocessing (SMP) approach, whereby one instance of a kernel image is shared across all processors. While hardware protection cannot be used to separate machine resources among the kernel instances, a *multikernel* is arguably more scalable on processors with high core counts [Baumann et al. 2009]. This arrangement still adheres to the distributed design of Quest-V, eliminating the need for monitor code, with the caveat that faults in one kernel can affect all other kernels.

According to the Quest-V design philosophy, hardware virtualization serves as an extra ring of protection for hardware resource partitioning and legacy service support. To avoid traps into the monitors during normal sandbox execution in Quest-V, a certain level of hardware support is required. Hardware virtualization features on the x86 architecture provide enough capabilities to achieve this goal, but they do incur some overheads. Moreover, they are tailored to consolidating guest services on a shared physical machine. We will elaborate more on this observation in Section 5, by proposing a list of hardware features that we feel are better suited to the construction of separation kernels.

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 four-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 four-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 sixty-four-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 I shows the monitor traps recorded during Linux sandbox initialization

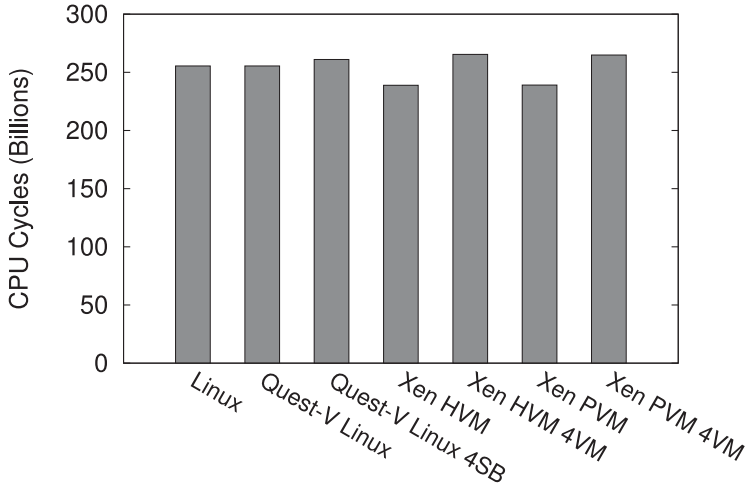

Fig. 10. *findprimes* CPU benchmark.

Table II. System Configurations

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

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 10 for the configurations in Table II. All Linux configurations were limited to 512MB RAM. For `Xen HVM` and `Xen PVM`, we pinned the Linux 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.

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 five VMs (one being Dom0) onto four cores.

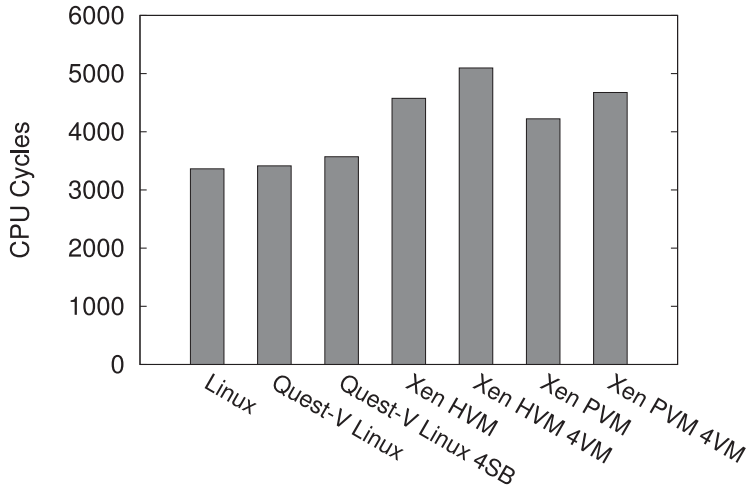

Fig. 11. Page fault exception handling overhead.

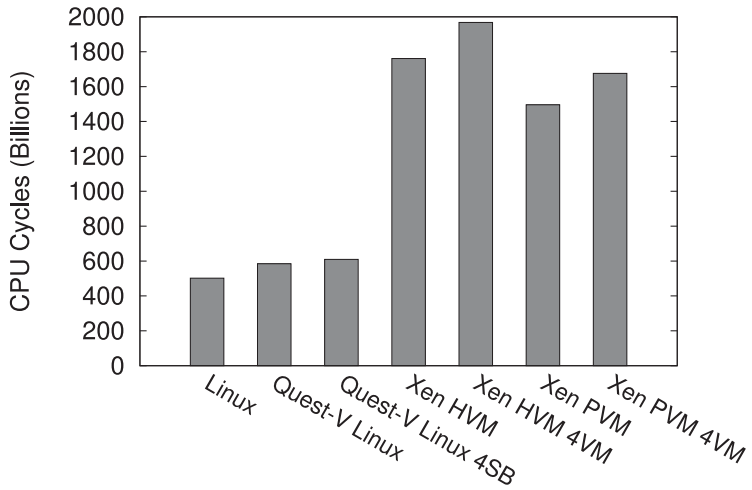

Fig. 12. *Fork-exec-wait* microbenchmark.

We evaluated the exception handling overheads for the configurations in Table II 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 11 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 12. `Quest-V Linux` is almost as good as native Linux and more than twice as fast as any Xen configuration.

Table III. *mplayer* HD Video Benchmark

	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

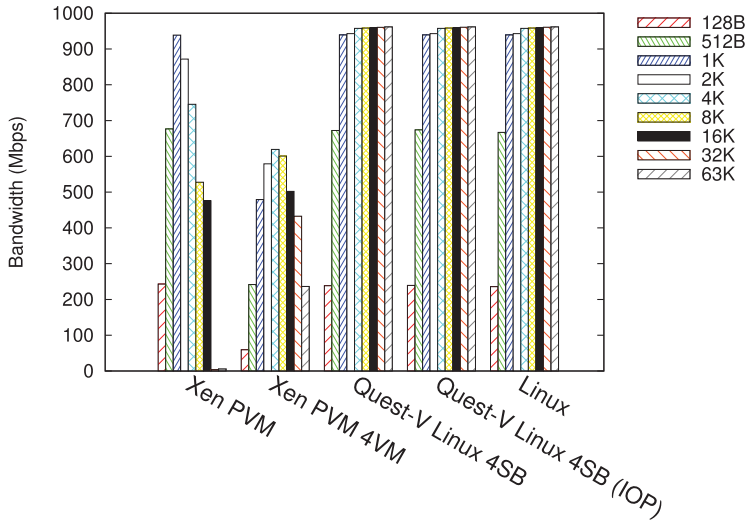

Fig. 13. *netperf* UDP send with different packet sizes.

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 invoking *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 III 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.

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 13 and 14, 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` [Russell 2008] is enabled. It can be seen that this helps dramatically improve the UDP bandwidth for small-size UDP packets. With a 512B packet size, `Xen PVM` outperforms standalone Linux. However, Quest-V Linux exhibits no visible overhead as compared to standalone Linux and 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 labeled 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

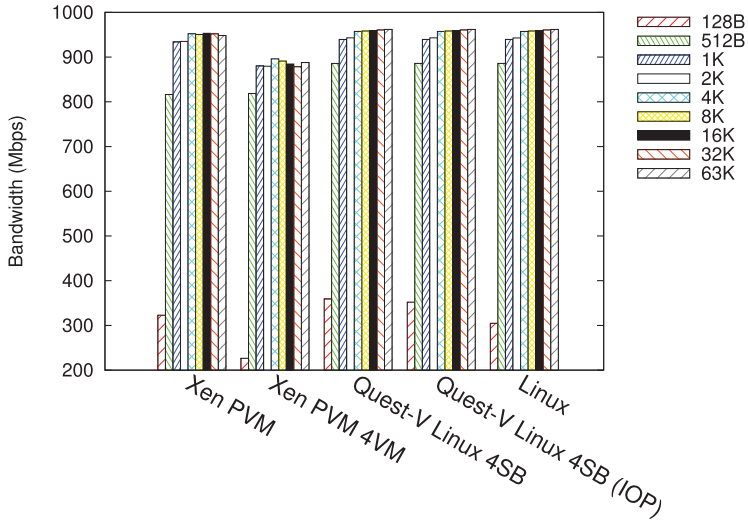

Fig. 14. *netserver* UDP receive.

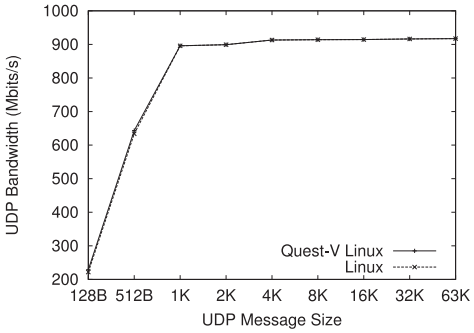

Fig. 15. UDP bandwidth.

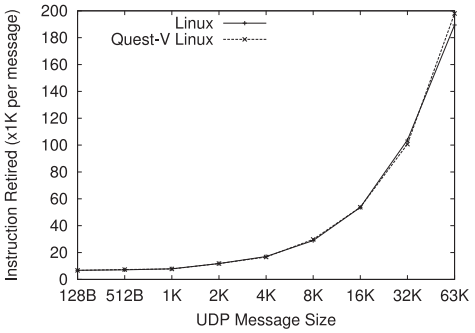

Fig. 16. Instructions retired.

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.

Partitioning Costs – We ran a set of experiments to investigate the costs of hardware partitioning in Quest-V. As part of our evaluation, we measured the last-level cache and TLB misses, as well as instructions retired for an instrumented UDP benchmark (similar to *netperf*) that collects hardware performance counter readings. The results are shown in Figures 15 through 20.

Figure 15 compares a standalone Linux against an equivalent sandboxed version of Linux in Quest-V. Again, from these results, we see no visible UDP bandwidth degradation caused by Quest-V. In Figure 16, we show the number of instructions retired over each UDP send operation. The results confirm that Quest-V does not interfere with the operation of a sandbox since no extra monitor instructions are executed. Figures 17

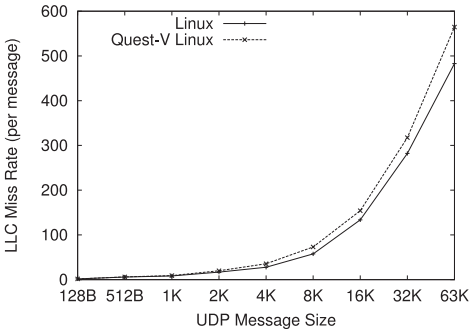

Fig. 17. Last-level cache misses.

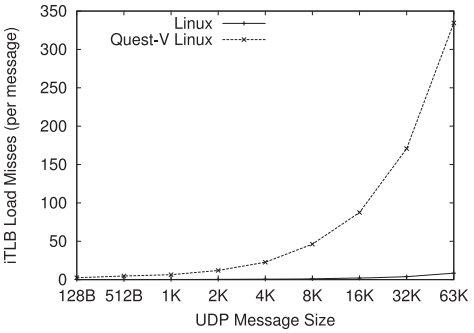

Fig. 18. iTLB load misses.

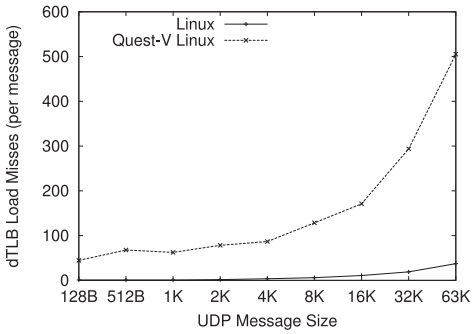

Fig. 19. dTLB load misses.

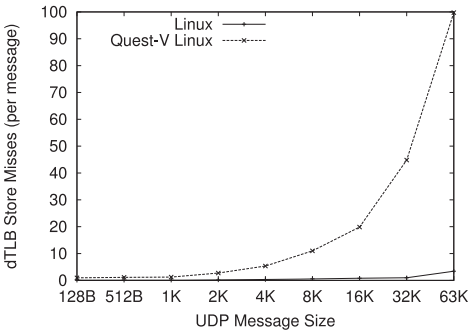

Fig. 20. dTLB store misses.

through 20 show the number of last-level cache misses, instruction TLB load misses, data TLB load misses, and data TLB store misses recorded during each UDP send operation, for different packet sizes in Linux and Quest-V Linux. The TLB load and store misses are events recorded during memory load and store instructions, respectively. As can be seen from these results, Quest-V does incur some increased cache and TLB miss overheads. The overheads are mainly due to the extra levels of address translation caused by EPTs. Quest-V experiences increased TLB contention, due to the caching of translated addresses for both EPTs and guest page tables. The extra last-level cache misses are potentially the consequence of added page table walks caused by EPT TLB misses. However, since the UDP benchmark is I/O bound, microarchitectural overheads do not affect the bandwidth results.

We ran the same experiments with the UDP server and client running on the same machine, for both standalone and Quest-V Linux. This was to investigate communication performance without I/O overheads associated with DMA and device interrupts. The bandwidth results shown in Figure 21 confirm that a Linux sandbox in Quest-V does incur a performance penalty compared to a standalone Linux system. This is irrespective of all experiments again showing the same number of instructions retired in each case, as seen in Figure 22. Figures 23 through 26 show results for last-level cache and TLB misses, respectively. In addition to the standalone and Quest-V Linux configurations, we added a new configuration labeled `Quest-V Linux LP`. This used 2MB large pages in the EPTs for the mapping of Linux sandbox memory, instead of the default 4KB pages. By increasing the page size, we removed one extra level of indirection in the EPTs. As can be seen from the results, this helped reduce both last-level cache and TLB

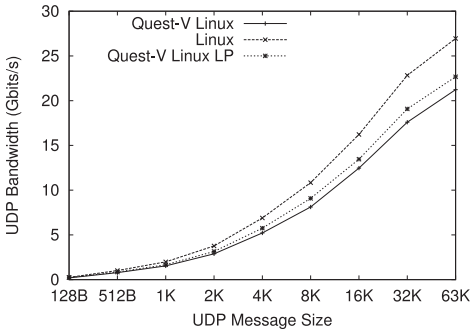

Fig. 21. Local UDP bandwidth.

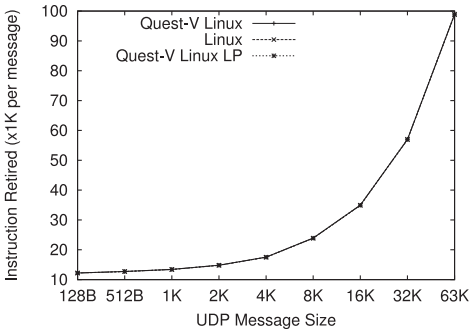

Fig. 22. Instructions retired.

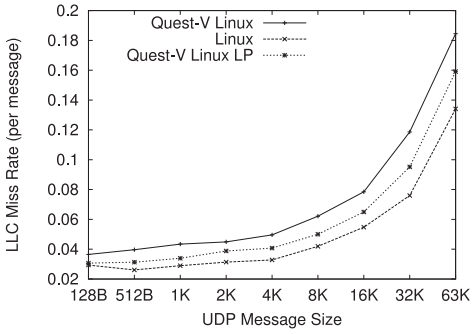

Fig. 23. Last-level cache misses.

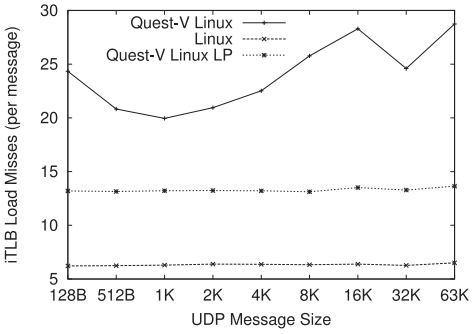

Fig. 24. iTLB load misses.

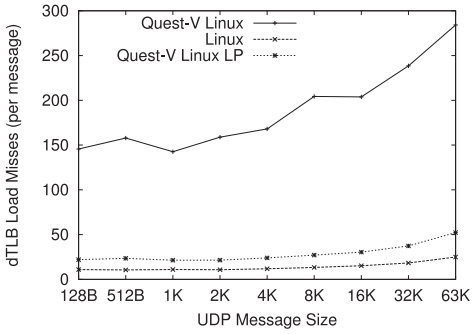

Fig. 25. dTLB load misses.

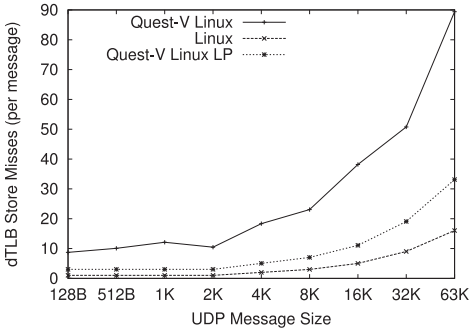

Fig. 26. dTLB store misses.

misses. The UDP bandwidth also improved under this configuration. The lesson to be learned is that EPT-based memory virtualization adds small but noticeable overhead, due primarily to TLB contention with guest page tables, as stated earlier.

TLB Performance – We ran a series of experiments to measure the effects of address translation using EPTs. A 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 4,160-byte offsets

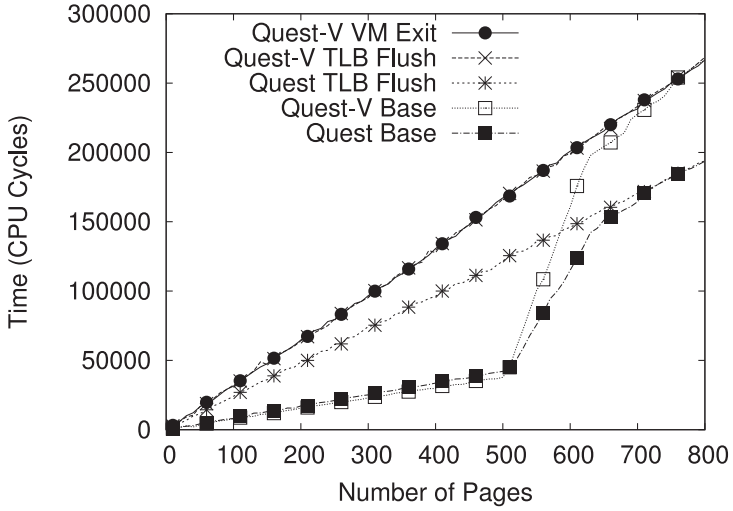

Fig. 27. Data TLB performance.

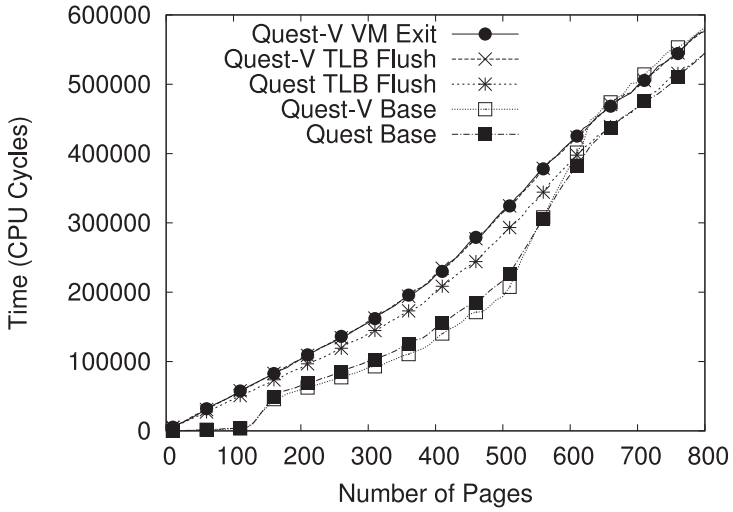

Fig. 28. Instruction TLB performance.

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 27 and 28 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 fewer than

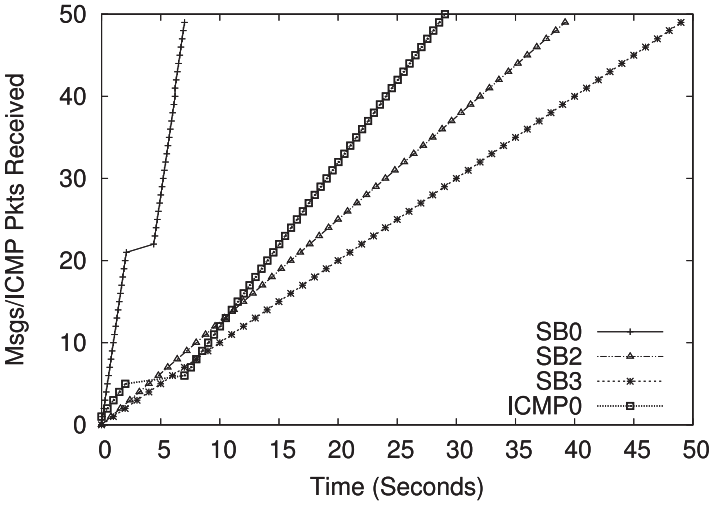

Fig. 29. Sandbox isolation.

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 five 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 three 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 four different native Quest sandboxes. Specifically, sandbox 1 has a kernel thread that sends messages through private message-passing channels to sandboxes 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 29.

In the experiment, sandbox 1 broadcasts messages to others (`SB0,2,3`) at 50-millisecond intervals. Sandboxes 0, 2, and 3 receive at 100-, 800-, and 1,000-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 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.

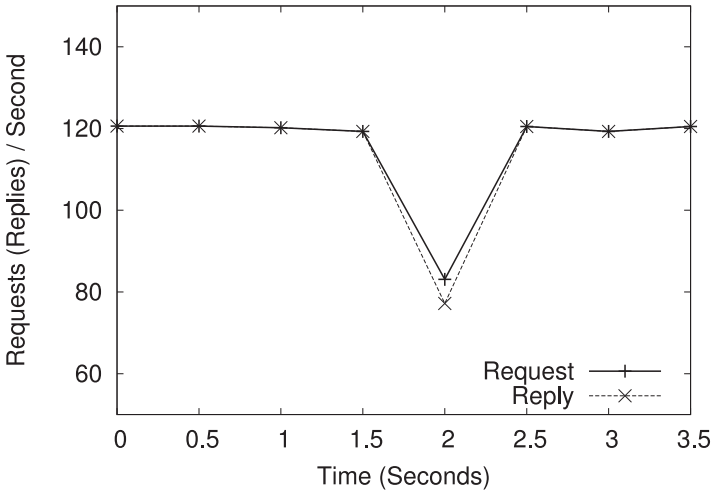

Fig. 30. Web server recovery.

Table IV. Overhead of Different Phases in Fault Recovery

	CPU Cycles	
Phases	**Local Recovery**	**Remote Recovery**
VM-Exit	885	
Driver Replacement	10,503	N/A
IPI Round Trip	N/A	4,542
VM-Enter	663	
Driver Reinitialization	1.45E+07	
Network I/F Restart	78,351	

Fault Recovery – To demonstrate the fault recovery mechanism of Quest-V, we intentionally corrupted the NIC driver on the mini-ITX machine while running a simple HTTP 1.0-compliant web server in user-space. Our web server was ported to a socket API that we implemented on top of lwIP [Dunkels 2015]. A remote Linux machine running `httperf` attempted to send 120 requests per second during both the period of driver failure and normal web server operation. Request URLs referred to the Quest-V website, with a size of 17,675 bytes.

Figure 30 shows the request and response rate at 0.5s sampling intervals. The driver failure occurred in the interval [1.5s,2s], after which recovery took place. Recovery involved reinitializing the NIC driver and restarting the web server in another sandbox, taking less than 0.5s. This is significantly faster than a system reboot, which can take close to a minute to restart the network service.

Fault recovery can occur locally or remotely. In this experiment, we saw little difference in the cost of either approach. Either way, the NIC driver needs to be reinitialized. This involves reinitialization of either the same driver that faulted in the first place or an alternative driver that is tried and tested. As fault detection is not in the scope of this article, we triggered the fault recovery event manually by assuming an error occurred. Aside from optional replacement of the faulting driver and reinitialization, the network interface needs to be restarted. This involves re-registering the driver with lwIP and assigning the interface an IP address.

The time for different phases of kernel-level recovery is shown in Table IV.

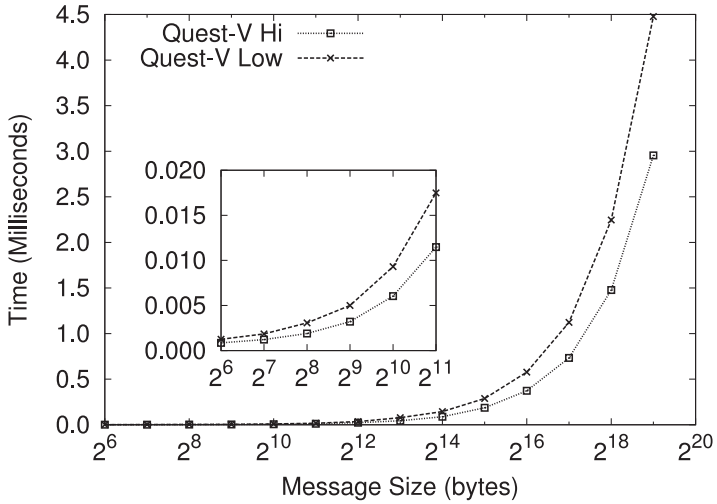

Fig. 31. Message-passing microbenchmark.

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 set up 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 [Liu and Layland 1973]. 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 31 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 5,000 trials to normalize the scheduling overhead. The communication costs grow linearly with increasing message size, because they include the time to access memory.

5. PROPOSED HARDWARE ARCHITECTURE

Our experiences with Quest-V show that hardware-based virtualization features on modern x86 architectures are *sufficient* for the construction of a separation kernel with minimal resource partitioning overheads. It is possible to build a separation kernel with a minimal trusted compute base (TCB) that is removed from the most frequently used control paths.

However, the performance analysis presented in Section 4 showed that with Intel VT-x, the memory translation overhead introduced by EPTs is nonnegligible for memory-intensive applications with relatively large working sets. Extended page table management at 4KB granularity may be necessary in conventional hypervisors, which must multiplex an overcommitted set of hardware resources among guest VMs. However, a simpler memory management scheme is preferred in a partitioned separation kernel, where resources are not overcommitted but, instead, fully committed. This has led us to consider the ideal hardware architecture for the construction of Quest-V-style separation kernels.

Hierarchical Protection Domains – A Quest-V-style separation kernel requires an extra ring of protection beyond that within any one sandbox. For example, a legacy Linux sandbox requires two rings of protection to separate kernel- and user-level protection domains; a separation kernel that supports legacy Linux services would therefore require a third ring of protection to securely partition resources among sandboxes. On the x86 architecture, sandboxed (guest) code has the ability to use up to four rings of protection (or privilege levels) in IA-32 mode. Almost no legacy OSs take advantage of more than two such privilege levels, meaning a simpler protection hierarchy could be adopted. If we forego support for legacy code, only two privilege levels are actually required: one for the monitor and another for a (guest) sandbox. Minimally, hardware protection would allow for the construction of library-based OS services within a sandbox and the idea of system calls to switch between user- and kernel-level privileges would not be necessary.

Privileged Memory Protection Mechanism – A memory protection mechanism should be provided that can only be manipulated in the most privileged protection domain. This is necessary to enforce memory partitioning between sandboxes. With the Intel VT-x and AMD-V, this is implemented using an extra set of page tables that translate guest physical to machine physical addresses. However, this fine-granularity memory protection mechanism brings relatively high overhead to memory accesses and is often unnecessary since memory partitions in Quest-V are mostly large contiguous memory regions. A segmentation-based approach that identifies a range of accessible memory addresses for a sandbox would both suffice and be more efficient. One could then build a separation kernel in which sandboxes are isolated in their own memory segments but which are able to communicate using special channels built in additional shared-memory segments. The establishment of shared-memory communication segments would be part of a capability mechanism under the control of a most privileged protection domain (e.g., for a monitor).

One problem with architectures that lack virtualization support is that memory protection schemes such as paging and segmentation are managed in the most privileged protection domain. Ideally, in a separation kernel, a scheme would allow for a sandboxed *guest* domain to have control over paging or segmentation to support legacy OSs with virtual memory. Such a scheme would restrict the translation of guest virtual addresses to machine physical memory that has been granted to the sandbox. Similarly, some mechanism is needed to provide guest (unprivileged) access to a replicated set of machine registers, where each set applies to a single sandbox. The number of sandboxes that can be supported will be limited to how many replicated register sets there are. Since we envision the construction of separation kernels that do not overcommit machine physical resources, we can realistically expect the total number of register sets to be limited by the number of processor cores or hardware threads. Along with register sets, hardware capabilities are needed to identify the subsets of I/O devices, cores, and memory regions accessible to sandboxes. In effect, this would be like a simplified Intel VT-x virtual machine control structure per sandbox, limited to a maximum number of partitions.

For security-critical domains, restrictions must be placed on information flow. For example, information may legitimately flow between a pair of components in two sandboxes, SB_1 and SB_2, but not subsequently from SB_2 to a component in sandbox SB_3. A tagged memory architecture [Zeldovich et al. 2008] would allow, at some appropriate granularity, the specification of read and write permissions on certain address ranges. Security policies for confidentiality [Bell and LaPadula 1976] and integrity [Biba 1975] could then be implemented using labeling techniques [Zeldovich et al. 2006; Efstathopoulos et al. 2005]. For example, confidentiality rules could be enforced, to prevent data being written from a high-security sandbox, or system component, to one of lower security via a shared channel. Similarly, a low-security sandbox or component could be disallowed from reading information in a sandbox, or component, at a higher security level.

Configurable Privileged I/O Protection Mechanism – Hardware is needed to enforce access control on device registers and mediate the delivery of interrupts to only those sandboxes with appropriate access rights. The Intel VT-x uses an I/O bitmap and EPTs for the protection of both port-based and memory-mapped device registers. In particular, I/O bitmaps can be established for individual sandboxes to restrict the devices they can directly access. However, due to the lack of a configuration mechanism for PCI device partitioning, a single-step debug exception approach is needed, as described in Section 3.2 for PCI device filtering. Even though this approach does not incur additional overhead once sandbox initialization is finished, it is possible that a misbehaving sandbox could repeatedly issue access requests to illegal ports, thereby causing VM-Exits to a monitor. Using multiple monitors as in Quest-V prevents this problem from affecting other sandboxes, however. That said, additional hardware support could help reduce sandbox initialization time by avoiding unnecessary VM-Exits to trap illegal register accesses. In particular, having hardware support to identify PCI blacklisted devices (by either a Vendor/Device ID pair or bus/device/function tuple) for each sandbox would be desired.

A hardware mechanism is necessary to control the target sandbox for an interrupt. In Quest-V, EPTs are used to protect the IOAPIC for device interrupt configuration and delivery management, as described in Section 3.2. Hardware support would alleviate the need to trap every access to an IOAPIC page, be it legitimate or not. A mechanism is also needed to prevent uncontrolled delivery of inter-processor interrupts (IPIs) between sandboxes. For example, a compromised device might have a DMA onto a local APIC memory page on the x86, which could then generate arbitrary IPIs to other sandboxes. To deal with this, the x86 virtualization features support interrupt remapping and IOMMU techniques to control where interrupts can be delivered and where DMA transfers occur. Rather than a full-blown IOMMU, a simpler offset assignment for each sandbox could restrict where DMA transfers could occur in each sandbox.

Deprivileged Interrupt Handling and Scheduling – A hardware mechanism that delivers interrupts directly to a sandboxed (guest, or non-root) domain without having to be interpreted by a more trusted monitor layer is essential for performance and predictability. The arrival of interrupts has the potential to affect the timing of a running thread, so a mechanism that combines the scheduling of threads and interrupts is desirable. For example, systems such as Linux allow interrupts to be handled with immediate effect (barring their dispatch latency), which might cause a high-criticality real-time thread to miss its deadline. A hardware approach that allows an interrupt to be assigned a (potentially dynamic) priority according to the entity that triggers its generation would be desirable. For example, an application thread might issue a read on a socket descriptor that triggers a network device to subsequently generate an interrupt when data is available. It is desirable to process the interrupt at the priority related

to the thread that issued the read request. To deal with this, it would be beneficial to have a hardware mechanism that allowed software threads issuing I/O requests to associate their priorities with devices. Then, when an interrupt from a device occurred, the processor could check the interrupt priority against that of the current thread and decide whether to defer interrupt handling or not. Deferred interrupt handling would be queued and managed like a *softIRQ* in a system such as Linux. Quest-V attempts to address this problem by using I/O VCPUs and specially written device drivers, but the problem would be simplified with added hardware support.

Real Time and Predictability – Finally, for real-time and safety-critical applications, it is essential that hardware provide highly predictable operations. For instance, caches on a multicore platform should adopt an open and predictable replacement policy, or at least allow software control. Even though we are currently working on a cache-partitioning strategy using page coloring in Quest-V [Ye et al. 2014], it would be more efficient if cache-partitioning support at the hardware level were available. Additionally, a predictable memory controller and system bus protocol should also be provided for memory bus contention management and predictable inter-processor communication.

6. RELATED WORK

Separation Kernels and Virtual Machine Systems. Conventional virtual machine monitors are designed to efficiently and transparently multiplex hardware resources among a set of guests. Their design tends to focus on scalability and maximizing resource utilization, rather than resource partitioning and predictability. This makes it necessary for a hypervisor to frequently interrupt virtual machine execution to perform resource management. Examples of such systems being used in embedded applications include Xen [Barham et al. 2003] and Linux-KVM [Habib 2008]. Some other systems, such as XtratuM [Crespo et al. 2010], Wind River Hypervisor [River 2014], INTEGRITY multivisor [Software 2015b], and Mentor Graphics Embedded Hypervisor [Graphics 2015], target and are optimized for embedded applications but still feature a traditional hypervisor design.

In contrast to these systems, Quest-V is a separation kernel that statically partitions machine resources 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 pass-through. Even though PCI pass-through 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 acknowledgment. This potentially forces two hypervisor traps for each interrupt. ELI [Gordon et al. 2012] is a software-based approach for handling a subset of interrupts within guest virtual machines using shadow IDTs. In combination with PCI pass-through, this is similar to the approach Quest-V uses to partition I/O resources. Fundamentally, ELI's goal is focused on improving I/O performance of select guests, while Quest-V is focused on using virtualization in a timing-predictable separation kernel.

There exist commercial separation kernels such as LynxSecure [Technologies 2015], PikeOS [AG 2015], and INTEGRITY 178B RTOS [Software 2015a]. Very few details of these systems are available in the public domain. The LynxSecure separation kernel is targeted at safety-critical real-time systems; it resembles a typical hypervisor with support for multiple guest operating systems. PikeOS is a separation microkernel [Klein et al. 2009] that supports multiple VMs and targets safety-critical domains such as Integrated Modular Avionics. The microkernel supports a virtualization layer

that is required to manage the spatial and temporal partitioning of resources among all guests. INTEGRITY 178B RTOS is a certified separation kernel that uses an MMU for memory protection. Its secure partitions are designed for Ada, C, and Embedded C++ programs rather than legacy operating systems.

Muen [Buerki and Rueegsegger 2015] is an open-source prototype separation kernel written in the SPARK programming language. It is claimed to have been formally proven to contain no runtime errors at the source code level. The Muen separation kernel also uses virtualization to separate the system into multiple *subjects*, which are equivalent to virtual machines. However, unlike in Quest-V, traps into the Muen separation kernel are necessary to handle external interrupts and to schedule subjects.

Jailhouse [Technology 2014] is a partitioning hypervisor that enables asymmetric multiprocessing on Linux-based systems. It is able to run bare-metal real-time applications alongside Linux, in separate *cells* that are isolated using virtualization. Similar to Quest-V, hardware resources such as processor cores and devices are statically partitioned for each cell in Jailhouse. However, early prototypes of Jailhouse neither enforce access control on device interrupts nor guarantee predictable intercell communication. Additionally, a Linux kernel has to be bootstrapped first before cells are created in Jailhouse. This could render it unsuitable for platforms with relatively limited resources.

NoHype [Szefer et al. 2011] 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 for chip multiprocessors.

Barrelfish [Baumann et al. 2009] is a multikernel that replicates 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 [Belay et al. 2012] 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) are made more efficient. However, most system services are still redirected to the Linux kernel running in VMX root mode. VirtuOS [Nikolaev and Back 2013] 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 virtual machine monitor (VMM) to facilitate communication between different domains. The PCI pass-through capability provided by the Xen VMM is also used to partition devices among service domains. However, interrupt handling and VM scheduling still require VMM intervention.

Arrakis [Peter et al. 2014] and IX [Belay et al. 2014] show how to use hardware I/O virtualization to bypass OS kernels in the common case, thereby allowing applications to take control of machine resources for their specific needs. A kernel is only needed to set up the execution environment and assign resources directly to an application. IX separates management and scheduling functions of the kernel (the control plane) from network processing (the data plane) to show how to implement efficient user-space network protocol stacks. Arrakis also shows how to use I/O virtualization to isolate the

performance of separate execution domains. Quest-V partitions all resources, not just I/O, among a distributed collection of domains with different criticalities.

Tessellation [Liu et al. 2009; Colmenares et al. 2013] shares many of the goals of Quest-V. It attempts to enforce space-time partitioning of resources among *cells*, which are similar to Quest-V's sandboxes. Cells export their assigned resources to user level, where scheduling and resource management policies are implemented. Applications compose cells via secure channels that support user-level asynchronous message-passing mechanisms. Tessellation allows resources to be dynamically allocated among cells. However, a kernel must interact with a broker to control resource allocation and mediate user-level resource management decisions within cells. Quest-V does not rely on a single kernel, or trusted layer, to broker sandbox resources at runtime. While Tessellation's cells are limited to single address spaces, Quest-V's sandboxes support traditional process address spaces, allowing integration of legacy services with new Quest real-time services.

Other systems that partition resources on many-core architectures include Factored OS [Wentzlaff and Agarwal 2009], Corey [Boyd-Wickizer et al. 2008], Hive [Chapin et al. 1995], and Disco [Bugnion et al. 1997]. Unlike Quest-V, these systems are focused on scalability rather than isolation and predictability.

Real-Time Systems. While many real-time operating systems exist today, it is less common to see systems with explicit support for temporal isolation using resource reservations or budgets. One such system that is built around the notion of resource reserves is Linux/RK [Oikawa and Rajkumar 1998]. The concept of a reserve in Linux/RK was derived and generalized from processor capacity reserves [Mercer et al. 1993] in RT-Mach. In more recent times, there has been similar work on resource containers to account for resource usage [Banga et al. 1999]. Each time-multiplexed reserve in the system has a budget C, interval T, and deadline D. Linux/RK requires a priori knowledge of application resource demands and relies on an admission control policy to guarantee a reasonable global reserve allocation. In contrast, Quest focuses on the temporal isolation between tasks and system events using a hierarchy [Shin and Lee 2003; Regehr 2001] of virtual servers, acting as either Main or I/O VCPUs.

Redline [Yang et al. 2008] is a system that focuses on predictability for interactive and multimedia applications. It also has the notion of budgets and replenishments, but the task scheduling model appears similar to that used in Deferrable Servers [Strosnider et al. 1995; Bernat and Burns 1999]. Given Redline's focus, the system differentiates interactive and best-effort tasks and optimistically accepts new tasks based on the actual usage of the system. In the presence of overload, a load monitor will select an interactive victim and downgrade it to best effort in order to fulfill response time requirements of other interactive tasks. Quest shares some of Redline's properties but addresses the dependency between tasks and system events triggered in response to I/O requests. In contrast to both Redline and Linux/RK, Quest allows I/O events to be processed at priorities inherited from virtual servers responsible for executing tasks, for whom I/O event processing is being performed.

The HARTIK kernel [Abeni and Buttazzo 1998] supports the coexistence of both soft and hard real-time tasks. To ensure temporal isolation between hard and soft real-time tasks, the soft real-time tasks are serviced using a Constant Bandwidth Server (CBS). A CBS has a current budget c_s and a bandwidth limited by the ratio Q_s/T_s, where Q_s is the maximum server budget available in the period T_s. When a server depletes all its budget, it is recharged to its maximum value. A corresponding server deadline is updated by a function of T_s, depending on the state of the server when a new job arrives for service or when the current budget expires. CBS guarantees a total utilization factor no greater than Q_s/T_s, even in overloads, by specifying a maximum

budget in a designated window of time. This contrasts with work on the Constant Utilization Server (CUS) [Deng et al. 1997] and Total Bandwidth Server (TBS) [Spuri and Buttazzo 1994], which ensures bandwidth limits only when actual job execution times are no more than specified worst-case values. CBS has bandwidth preservation properties similar to that of the Dynamic Sporadic Server (DSS) [Ghazalie and Baker 1995] but with better responsiveness.

CBS, CUS, TBS, and DSS all assume the existence of server deadlines. We chose not to assume the existence of deadlines for VCPUs in Quest, instead restricting VCPUs to fixed priorities. This avoids the added complexity of managing dynamic priorities of VCPUs as their deadlines change. Additionally, for cases when there are multiple tasks sharing a fixed-priority VCPU, the execution of one task will not change the importance of the VCPU for the other tasks. That said, PIBS has some similarities to CUS and TBS. It assigns priorities to I/O VCPUs based on the priority of the task (specifically, its Main VCPU) associated with the I/O event to be serviced. Eligibility times are then set in a manner similar to how deadlines are updated with CUS and TBS, except they denote when the I/O VCPU is allowed to resume usage of processor cycles without exceeding its bandwidth capacity. Observe that with PIBS, the next server eligibility time is not set until all the I/O VCPU budget is consumed or an I/O event completes within the allowed budget. In comparison, CUS and TBS determine deadlines *before* execution, assuming knowledge of WCET values.

The rationale for both Main and I/O VCPUs in Quest is based on our earlier work to integrate the scheduling of interrupts and tasks [Zhang and West 2006]. Others have proposed methods to unify task and interrupt scheduling [Leyva-del-Foyo et al. 2006] or have considered bandwidth constraints on device driver execution [Lewandowski et al. 2007]. However, Quest attempts to prioritize and budget I/O events according to the tasks that lead to their occurrence. The system integrates asynchronous event processing for both device interrupts and tasks waking up after the completion of blocking (e.g., I/O) operations.

7. CONCLUSIONS AND FUTURE WORK

This article introduces Quest-V, which is an open-source separation kernel built from the ground up. It uses hardware virtualization to separate system components into *sandboxes*. Sandboxes manage their own subsets of machine resources and perform scheduling, memory, and I/O management without the involvement of a hypervisor. Inter-sandbox communication is made possible by 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. The design of Quest-V enables less important services to be isolated from those of higher criticality, and essential services to be replicated across different sandboxes to ensure availability in the presence of faults.

Quest-V system monitors occupy a small memory footprint compared to traditional hypervisors. They are used only to partition resources at boot time, assist in fault recovery, and establish inter-sandbox communication channels. By comparison, traditional hypervisors need extra functionality to multiplex guest virtual machines at runtime onto a shared set of hardware resources. This means Quest-V monitors are rarely involved in normal system execution. Removing the most trusted software management layer from the most common control paths has the potential to heighten system security. If a security breach or fault does occur in a monitor, it is possible to use triple modular redundancy [Lyons and Vanderkulk 1962] or N-versioning [Avižienis 1985] techniques to maintain system operation.

This article shows how to build a separation kernel consisting of multiple instances of the Quest real-time system on a multicore processor, along with a Linux front end.

Quest is used as the boot loader for Linux guests, which require minimal paravirtualization to run directly on their hardware resources. The setup allows time-critical Quest services to coexist with legacy, less critical Linux services, and applications. We describe the method by which resources are partitioned among 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 and predictable CPU usage.

Quest-V is built on our Quest real-time operating system, which manages CPU usage using a novel hierarchy of bandwidth-preserving Main and I/O VCPUs. These form the basis for predictable scheduling of both tasks and interrupts. Quest-V avoids VM-Exits as much as possible, meaning the TLBs that cache EPT mappings are rarely flushed. This benefit comes about due to the fact that multiple guests are not multiplexed onto the same processor core. In the embedded systems we envision for this work, sandbox working sets are expected to fit within the TLB reach, at least for critical services in native Quest-V sandboxes.

The message to be taken from this work is that virtualization is applicable not only to server-based systems but also to mixed-criticality embedded systems. This work is not simply aimed at showing how static resource partitioning is more efficient than using a hypervisor to multiplex machine resources among a series of guests. Instead, this work serves to show how to rethink the use of hardware features in the design of new systems. We hope this will dispel counterarguments that machine virtualization is far too inefficient and unpredictable for use in mixed-criticality systems with safety and timing requirements.

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) [Abramson et al. 2006] to prevent delivery of unauthorized interrupts to specific cores [Wojtczuk and Rutkowska 2011]. Protection of CPU model-specific registers (MSRs) will be similarly enforced using hardware-managed bitmaps.

New security models using Quest-V's distributed monitors will be considered, as discussed in Section 3.1. This will, however, require execution of monitor code during normal system operation. We will compare the performance costs to the system security benefits. Finally, Quest-V has thus far focused on the static partitioning of resources among sandboxes. Inter-sandbox communication is required to access resources in remote sandboxes. Future work will study techniques to efficiently and predictably repartition sandboxes at runtime and identify when it is preferable to do so instead of paying the cost of inter-sandbox communication.

ACKNOWLEDGMENTS

We thank the reviewers for their help improving the quality of this article. This material is based on work supported by the National Science Foundation under Grant Nos. 1117025 and 1527050, and a gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Please see www.questos.org for more details.

REFERENCES

Luca Abeni and Giorgio Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. 4–13.

Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel virtualization technology for directed I/O. *Intel Technology Journal* 10, 3 (August 2006), 179–192.

Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2–13.

SYSGO AG. 2015. PikeOS Hypervisor. (2015). http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept.

David H. Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *ACM/IEEE International Symposium on Microarchitecture (MICRO'99)*. 248–259.

Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. 2006. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems* 2 (2006), 239–247.

ARINC. 2008. ARINC 653 - An Avionics Standard for Safe, Partitioned Systems. Wind River Systems/IEEE Seminar. (August 2008).

AUTOSAR. 2015. AUTomotive Open System ARchitecture. (2015). http://www.autosar.org.

Algirdas Avižienis. 1967. Design of fault-tolerant computers. In *Proceedings of the Fall Joint Computer Conference*. 733–743.

Algirdas Avižienis. 1975. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the International Conference on Reliable Software*. 458–464.

Algirdas Avižienis. 1985. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering* (1985), 1491–1501.

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. 164–177.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. 29–44.

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *the 10th USENIX Conference on Operating Systems Design and Implementation*. 335–348.

Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Broomfield, CO, 49–65.

David Elliott Bell and Leonard J. LaPadula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report ESD-TR-75-306. Mitre Corporation, Bedford, MA.

Guillem Bernat and Alan Burns. 1999. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*. 68–78.

Kenneth J. Biba. 1975. *Integrity Considerations for Secure Computer Systems*. Technical Report MTR-3153. Mitre Corporation.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue hua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An operating system for many cores. In *The 8th USENIX Symposium on Operating Systems Design and Implementation*. 43–57.

Reto Buerki and Adrian-Ken Rueegsegger. 2015. Muen Separation Kernel. (2015). http://muen.sk/.

Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 143–156.

Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *International Conference on Supercomputing*. 242–252.

John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 12–25.

Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. 2013. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *Design Automation Conference (DAC'13)*.

Alfons Crespo, Ismael Ripoll, and Miguel Masmano. 2010. Partitioned embedded architecture based on hypervisor: The xtratum approach. In *The European Dependable Computing Conference*. 67–72.

Matthew Danish, Ye Li, and Richard West. 2011. Virtual-CPU scheduling in the quest operating system. In *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*. 169–179.

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

Adam Dunkels. 2015. lwIP – A Lightweight TCP/IP Stack. (2015). http://savannah.nongnu.org/projects/lwip/.

Haakon Dybdahl, Per Stenström, and Lasse Natvig. 2006. A cache-partitioning aware replacement policy for chip multiprocessors. *High Performance Computing* 4297 (2006), 22–34.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazires, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 17–30. DOI: http://dx.doi.org/10.1145/1095810.1095813

T. M. Ghazalie and T. P. Baker. 1995. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems* 9, 1 (July 1995), 31–68.

Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. 411–422.

Mentor Graphics. 2015. Mentor Embedded Hypervisor. (2015). http://www.mentor.com/embedded-software/hypervisor/.

Irfan Habib. 2008. Virtualization with KVM. *Linux Journal* 2008, 166 (2008), 8.

Intel Corporation. 2015. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*. http://www.intel.com.

Ravi Iyer. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing*. 257–266.

Barry Kauler. 2015. Puppy Linux. (2015). http://www.puppylinux.org.

Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architectures and Compilation Techniques (PACT'04)*.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. 207–220.

M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. 2007. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*.

Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. 2006. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. 2014. Predictable communication and migration in the Quest-V separation kernel. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS'14)*. Rome, Italy.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*.

Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. 2004. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 176–185.

C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20, 1 (1973), 46–61.

Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: Space-time partitioning in a manycore client OS. In *1st USENIX Workshop on Hot Topics in Parallelism*.

Robert E. Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209.

Clifford Mercer, Stefan Savage, and Hideyuki Tokuda. 1993. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of the 4th Workshop on Workstation Operating Systems*. 129–134.

Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 116–132.

Shuichi Oikawa and Ragunathan Rajkumar. 1998. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*.

David L. Parnas, A. John van Schouwen, and Shu Po Kwan. 1990. Evaluation of safety-critical software. *Communications of the ACM* (June 1990), 636–648.

PCI-SIG. 2015. PCI Configuration Space. (2015). https://www.pcisig.com/.

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 1–16.

Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. 2006. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. 2–12.

Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 214–224.

John Regehr. 2001. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. 3–14.

Wind River. 2014. Wind River Hypervisor. (2014). http://www.windriver.com/products/hypervisor/.

John M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*. 12–21.

Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

Insik Shin and Insup Lee. 2003. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*. 2–13.

Green Hills Software. 2015a. INTEGRITY-178B RTOS. (2015). http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

Green Hills Software. 2015b. INTEGRITY Multivisor. (2015). http://www.ghs.com/products/rtos/integrity_virtualization.html.

Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal* 1, 1 (1989), 27–60.

M. Spuri and G. Buttazzo. 1994. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*.

Marco Spuri and Giorgio Buttazzo. 1996. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems* 10 (1996), 179–210.

Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive set pinning: Managing shared caches in CMPs. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Mark Stanovich, Theodore P. Baker, An I Wang, and Michael Gonzalez Harbour. 2010. 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*.

Jay K. Strosnider, John P. Lehoczky, and Lui Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers* 44, 1 (January 1995), 73–91.

G. Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *Journal of Supercomputing* 28, 1 (April 2004), 7–26.

Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. 401–412.

LYNX Software Technologies. 2015. LynxSecure Embedded Hypervisor and Separation Kernel. (2015). http://www.lynx.com/products/hypervisors/.

Siemens Corporate Technology. 2014. Jailhouse Partitioning Hypervisor. (October 2014). https://github.com/siemens/jailhouse.

David Wentzlaff and Anant Agarwal. 2009. Factored operating systems (FOS): The case for a scalable operating system for multicores. *SIGOPS Operating Systems Review* 43, 2 (2009), 76–85.

Richard West, Puneet Zaroo, Carl Waldspurger, Xiao Zhang, and Haoqiang Zheng. 2008. Online Computation of Cache Occupancy and Performance. Filed with the USPTO. (October 14, 2008). Related to United States Patent Number US 8,429,665 B2. April 23, 2013.

Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2010. Online cache modeling for commodity multicore processors. *Operating Systems Review* 44, 4 (December 2010). Special VMware Track.

Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2013. *Multicore Technology: Architecture, Reconfiguration and Modeling*. CRC Press, ISBN-10: 1439880638, Chapter 8.

Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. 2009. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy* 7, 1 (Jan. 2009), 26–33.

Rafal Wojtczuk and Joanna Rutkowska. 2011. Following the White Rabbit: Software Attacks Against Intel VT-d Technology. (April 2011). Inivisible Things Lab.

Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*.

Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. 2006. Making information flow explicit in histar. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*. 263–278.

Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*.

Yuting Zhang and Richard West. 2006. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*.

Received April 2015; revised March 2016; accepted May 2016