Quest-V: A Virtualized Multikernel for High-Confidence Embedded Systems

Ye Li, Richard West, Eric Missimer, Matthew Danish

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

Abstract

This paper outlines the design of 'Quest-V', which is implemented as a collection of separate kernels operating together as a distributed system on a chip. Quest-V uses virtualization techniques to isolate kernels and prevent local faults from affecting remote kernels. A virtual machine monitor for each kernel keeps track of extended page table mappings that control immutable memory access capabilities. This leads to a high-confidence multikernel approach, where failures of system sub-components do not render the entire system inoperable. Communication is supported between kernels using both inter-processor interrupts (IPIs) and shared memory regions for message passing. Similarly, device driver data structures are shareable between kernels to avoid the need for complex I/O virtualization, or communication with a dedicated kernel responsible for I/O. In Quest-V, device interrupts are delivered directly to a kernel, rather than via a monitor that determines the destination. Apart from bootstrapping each kernel, handling faults and managing extended page tables, the monitors are not needed. This differs from conventional virtual machine systems in which a central monitor, or hypervisor, is responsible for scheduling and management of host resources amongst a set of guest kernels. In this paper we show how Quest-V can support online fault isolation and recovery techniques that are not possible with conventional systems. We also show how memory virtualization and I/O management do not add undue overheads to the overall system performance.

1 Introduction

Multicore processors are now ubiquitous in today's microprocessor and microcontroller industry. It is common to see two to four cores per package in embedded and desktop platforms, and higher core counts in server-class machines. This increase in on-chip core count is driven in part by trade-offs in power and computational demands. Many of these multicore processors also feature hardware virtualization technology (e.g., Intel VT and AMD-V CPUs). Virtualization has re-emerged in the last decade as a way to consolidate workloads on servers, thereby providing an effective means to increase resource utilization while still ensuring logical isolation between guest virtual machines.

Hardware advances with respect to multicore technology have not been met by software developments. In particular, multicore processors pose significant challenges to operating system design [8, 6, 36]. Not only is it difficult to design software systems that scale to large numbers of processing cores, there are numerous micro-architectural factors that affect software execution, leading to reduced efficiency and unpredictability. Shared on-chip caches [20, 37], memory bus bandwidth contention [44], hardware interrupts [43], instruction pipelines, hardware prefetchers, amongst other factors, all contribute to variability in task execution times. This is particularly problematic for real-time and embedded systems, where task deadlines must be met.

Coupled with the challenges posed by multicore processors are the inherent complexities in modern operating systems. Such complex interactions between software components inevitably lead to program faults and potential compromises to system integrity. Various faults may occur due to memory violations (e.g., stack and buffer overflows, null pointer dereferences and jumps or stores to out of range addresses [28, 14]), CPU violations (e.g., starvation and deadlocks), and I/O violations (e.g., mismanagement of access rights to files and devices). Device

drivers, in particular, are a known source of potential dangers to operating systems, as they are typically written by third party sources and usually execute with kernel privileges. To address this, various researchers have devised techniques to verify the correctness of drivers, or to sandbox them from the rest of the kernel [33, 34].

In this paper, we present a new system design that uses both virtualization capabilities and the redundancy offered by multiple processing cores, to develop a real-time system that is resilient to software faults. Our system, called 'Quest-V' is designed as a multikernel [6], or distributed system on a chip. Extended page tables (EPTs) [1] isolate separate kernel images in physical memory. These page tables map each kernel's 'guest' physical memory to host (or machine) physical memory. Changes to protection bits within EPTs can only be performed by a trusted monitor associated with the kernel on the corresponding core. This ensures any illegal memory accesses (e.g., write attempts on read-only pages) within a kernel are caught by the corresponding monitor. Our system has similarities to the Barrelfish multikernel, while also using virtualization similar to systems such as Xen [5]. We differ from traditional virtualized systems [9] by avoiding monitor intervention where possible, except for updating EPTs and handling faults.

We show how Quest-V does not incur significant operational overheads compared to a non-virtualized version of our system, simply called Quest, designed for SMP platforms. We observe that communication, interrupt handling, thread scheduling and system call costs are on par with the costs of conventional SMP systems, with the advantage that Quest-V can tolerate system component failures without the need for reboots.

We show how Quest-V can recover from component failure, using a web server in the presence of a misbehaving network device driver. Both local and remote kernel recovery strategies are described. This serves as an example of the 'self-healing' characteristics of Quest-V, with online fault recovery being useful in situations where high-confidence (or high availability) is important. This is typically the case with many real-time and embedded safety-critical systems found in healthcare, avionics, factory automation and automotive systems, for example.

In the following section we describe the rationale for the design of Quest-V. This is followed by a description of the architecture in Section 3. An experimental evaluation of the system is provided in Section 4. Here, we show the overheads of online device driver recovery for a network device, along with the costs of using hardware virtualization to isolate kernels and system components. Section 5 describes related work, while conclusions and future work are discussed in Section 6.

[1] Intel uses the term "EPT", while AMD refers to them as Nested Page Tables (NPTs). We use the term EPT for consistency.

2 Design Rationale

Quest-V is centered around three main goals: safety, predictability and efficiency. Quest-V is intended for safety-critical application domains, requiring high confidence in their operation [16]. Target applications include those emerging in healthcare, avionics, automotive systems, factory automation, robotics and space exploration. In such cases, the system requires real-time responsiveness to time-critical events, to prevent potential loss of lives or equipment. Similarly, advances in fields such as cyber-physical systems means that more sophisticated OSes beyond those traditionally found in real-time and embedded computing are now required. With the emergence of off-the-shelf and low-power processors now supporting multiple cores and hardware virtualization, it seems appropriate that these will become commonplace within this class of systems. In fact, the ARM Cortex A15 is expected to feature virtualization capabilities, on a processing core typically designed for embedded systems.

While safety is a key goal, we assume that users of our system are mostly trusted. That is, they are not expected to subject the system to malicious attacks, with the intent of breaching security barriers. Instead, our focus on safety is concerned with the prevention of software faults. While others have used techniques such as software fault isolation [14, 28], type-safe languages [25, 24, 19, 7], and hardware features such as segmentation [11, 35], Quest-V uses virtualization techniques to provide fault isolation. Notably, Quest-V relies on EPTs to separate system software components operating as a collection of services in a distributed system on a chip.

3 Quest-V Architecture

A high-level overview of the Quest-V architecture is shown in Figure 1. A single hypervisor is replaced by a separate trusted monitor for each sandbox. Quest-V uses memory virtualization as an integral design feature, to separate sub-system components into distinct sandboxes.

The Quest-V architecture supports sandbox kernels that have both replicated and complementary services. That is, some sandboxes may have identical kernel functionality, while others partition various system components to form an asymmetric configuration. The extent to which functionality is separated across kernels is somewhat con-

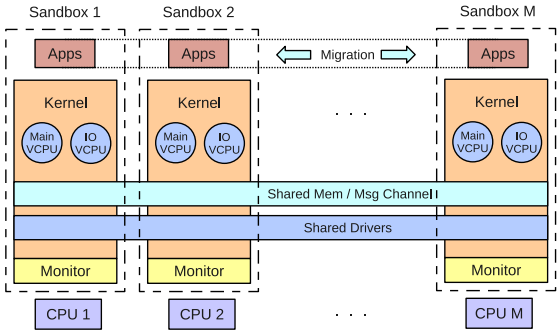

Figure 1: Quest-V Architecture Overview

figurable in the Quest-V design. In our initial implementation, each sandbox kernel replicates most functionality, offering a private version of the corresponding services to its local application threads. Certain functionality is, however, shared across system components. In particular, we share certain driver data structures across sandboxes [2], to allow I/O requests and responses to be handled locally.

Quest-V allows *any* sandbox to be configured for corresponding device interrupts, rather than have a dedicated sandbox be responsible for all communication with that device. This greatly reduces the communication and control paths necessary for I/O requests from applications in Quest-V. It also differs from the split-driver approach taken by systems such as Xen, that require all device interrupts to be channeled through a special driver domain.

Sandboxes that do not require access to shared devices are isolated from unnecessary drivers and associated services. Moreover, a sandbox can be provided with its own private set of devices and drivers, so if a software failure occurs in one driver, it will not necessarily affect all other sandboxes. In fact, if a driver experiences a fault then its effects are limited to the local sandbox and the data structures shared with other sandboxes. Outside these shared data structures, remote sandboxes (including all monitors) are protected by extended page tables.

Quest-V allows each sandbox kernel to be configured to operate on a chosen subset of CPUs, or *cores*. This is similar to how Corey partitions resources amongst applications [8]. In our current approach, we assume each sandbox kernel is associated with one physical core since that simplifies local (sandbox) scheduling and allows for relatively easy enforcement of service guarantees using a variant of rate-monotonic scheduling [22]. Notwithstanding, application threads can be migrated between sandboxes as part of a load balancing strategy. Similarly, multi-threaded applications can be distributed across sandboxes to allow parallel thread execution.

Application and system services in distinct sandbox kernels can communicate via shared memory channels. These channels are established by EPT mappings setup by the corresponding monitors. Messages are passed across these channels similar to the approach in Barrelfish [6].

Main and I/O VCPUs are used for real-time management of CPU cycles, to enforce *temporal isolation*. Application and system threads are bound to VCPUs, which in turn are assigned to underlying physical CPUs. We will elaborate on this aspect of the system in Section 3.1.3.

3.1 System Implementation

Quest-V is currently implemented as a 32-bit x86 system, targeting embedded rather than server domains. We plan to port Quest-V to the ARM Cortex A15 when it becomes available. Using EPTs, each sandbox virtual address space is mapped to its own host memory region. Only the BIOS, certain driver data structures, and communication channels are shared across sandboxes, while all other functionality is privately mapped.

Each sandbox kernel image is mapped to physical memory after the region reserved for the system BIOS, beginning from the low 1MB. While sandbox kernels can share devices and corresponding driver data structures, a device can be dedicated to a sandbox for added safety.

By default, Quest-V allows delivery of interrupts directly to sandbox kernels, where drivers are implemented. Only if heightened security is needed are drivers mapped to monitors. We are still investigating the implications of this in terms of performance costs.

Just as hardware devices can be shared between sandbox kernels, a process that does not require strict memory protection can be loaded into a user space region accessible across sandboxes. This reduces the cost of process migration and inter-process communication. However, in the current Quest-V system, we do not support shared user-spaces for application processes, instead isolating them within the local sandbox. While this makes process migration more cumbersome, it prevents kernel faults in one sandbox from corrupting processes in others.

3.1.1 Hardware Virtualization Support

Quest-V utilizes the hardware virtualization support available in most of the current x86 and the next generation ARM processors to encapsulate each sandbox in a separate virtual machine. As with conventional hypervisors, Quest-V treats a guest VM domain as an extra ring of memory protection in addition to the traditional kernel and user privilege levels. However, instead of having one

[2] Only for those drivers that have been mapped as shared between specific sandboxes.

hypervisor for the whole system, Quest-V has one monitor running in the host domain for each sandbox as shown earlier in Figure 1. Each sandbox kernel performs its own local scheduling and I/O handling without the cost of VM-Exits into a monitor. VM-Exits are only needed to handle software faults and update EPTs.

3.1.2 Hardware-Assisted Memory Isolation

The isolation provided by memory virtualization requires additional steps to translate guest virtual addresses to host physical addresses. Modern processors with hardware support avoid the need for software managed shadow page tables, and they also support TLBs to cache various intermediate translation stages.

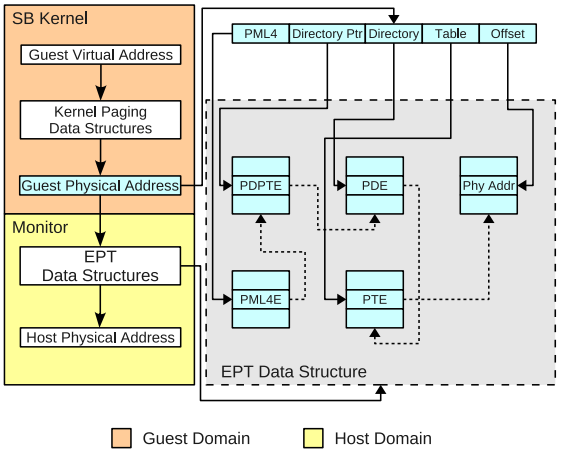

Figure 2: Extended Page Table Mapping

Figure 2 shows how address translation works for Quest-V guests (i.e., sandboxes) using Intel's extended page tables. Specifically, each sandbox kernel uses its own internal paging structures to translate guest virtual addresses to guest physical addresses (GPAs). EPT structures are then walked by the hardware to complete the translation to host physical addresses (HPAs).

On modern Intel x86 processors with EPT support, address mappings can be manipulated at 4KB page granularity. This gives us a fine grained approach to isolate sandbox kernels and enforce memory protection. 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 will incur an EPT violation, causing a trap to the local monitor. The EPT data structures are, themselves, restricted to access by the monitors, thereby preventing tampering by sandbox kernels.

EPT support alone is actually insufficient to prevent faulty device drivers from corrupting the system. It is still possible for a malicious driver or a faulty device to DMA into arbitrary physical memory. This can be prevented with technologies such as Intel's VT-d, which restrict the regions into which DMAs can occur using IOMMUs. However, this is still insufficient to address other more insidious security vulnerabilities such as "white rabbit" attacks [40]. For example, a PCIe device can be configured to generate a Message Signaled Interrupt (MSI) with arbitrary vector and delivery mode by writing to local APIC memory. Such malicious attacks can be addressed using hardware techniques such as Interrupt Remapping (IR). Having said this, the focus of our work is predominantly on fault isolation and safety in trusted application domains, rather than security in untrusted systems.

3.1.3 VCPU Scheduling

As stated earlier, Quest-V's goals are not only to ensure system safety, but also predictability. For use in real-time systems, the system must perform certain tasks by their deadlines. Quest-V does not require tasks to specify deadlines but instead ensures that the execution of one task does not interfere with the timely execution of others. For example, Quest-V is capable of scheduling interrupt handlers as threads, so they do not unduly interfere with the execution of higher-priority tasks. While Quest-V's scheduling framework is described elsewhere [42], we briefly explain how it provides temporal isolation between tasks and system events. This is the basis for real-time tasks with specific resource requirements to be executed in bounded time, while allowing non-real-time tasks to execute with specific priorities.

In Quest-V, *virtual CPUs* (VCPUs) form the fundamental abstraction for scheduling and temporal isolation of the system. The concept of a VCPU is similar to that in virtual machines [3, 5], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [3] 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 [4] for scheduling and accounting decisions on behalf of software threads. It serves no other purpose to virtualize the underlying physical CPUs, since our sandbox kernels and their applications execute directly on the hardware. In particular, a

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

VCPU does not need to act as a container for cached instruction blocks that have been generated to emulate the effects of guest code, as in some trap-and-emulate virtualized systems.

In common with bandwidth preserving servers [2, 12, 31], each VCPU, V, has a maximum compute time budget, C_{max}, available in a time period, V_T. V is constrained to use no more than the fraction $V_U = \frac{C_{max}}{V_T}$ of a physical processor (PCPU) in any window of real-time, V_T, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

Quest-V defines two classes of VCPUs: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts from I/O devices to be scheduled as threads, which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. The 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.

By default, VCPUs act like Sporadic Servers [30]. Local APIC timers are programmed to replenish VCPU budgets as they are consumed during thread execution. We use the algorithm by Stanovich et al [32] to correct for early replenishment and budget amplification in the POSIX specification. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [22]. This is significant, given I/O events can 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 can be applied, to ensure each VCPU is guaranteed its share of CPU time, V_U, in finite windows of real-time.

An example schedule is provided in Figure 3 for three 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 physical CPU.

3.1.4 Inter-Sandbox Communication

Inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory, and asynchronous event notification mechanisms using Interprocessor Interrupts (IPIs). IPIs are currently used to communicate with remote sandboxes to assist in fault recovery, and can also be used to notify the arrival of messages exchanged via shared memory channels. Monitors update extended page table 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. All other sandboxes are isolated from these memory regions.

A *mailbox* data structure is set up within shared memory by each end of a communication channel. By default, Quest-V currently supports asynchronous communication by polling a status bit in each relevant mailbox to determine message arrival. Message passing threads are bound to VCPUs with specific parameters to control the rate of exchange of information. 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. Thus, Quest-V supports real-time communication between sandboxes without compromising the CPU shares allocated to non-communicating tasks.

3.1.5 Interrupt Distribution and I/O Management

By default, Quest-V allows interrupts to be delivered directly to sandbox kernels. Hardware interrupts are delivered to *all* sandbox kernels with access to the corresponding device. This avoids the need for interrupt handling to be performed in the context of a monitor as is typically done with conventional virtual machine approaches. Quest-V does not need to do this since complex I/O virtualization is not required. Instead, early demultiplexing in the sandboxed device drivers determines if subsequent interrupt handling should be processed locally. If that is not the case, the local sandbox simply discards the interrupt. We believe this to be less expensive than going through a dedicated coordinator as is done in Xen [5] and others.

Quest-V uses the I/O APIC found on modern x86 platforms to multicast hardware interrupts to *all sandboxes sharing a corresponding device*. The I/O APIC is reprogrammed as necessary to re-route interrupts as part of fault recovery, when a new sandbox is required to continue or restore a service. We expect the number of sandboxes sharing a device to be relatively low (around 2-4

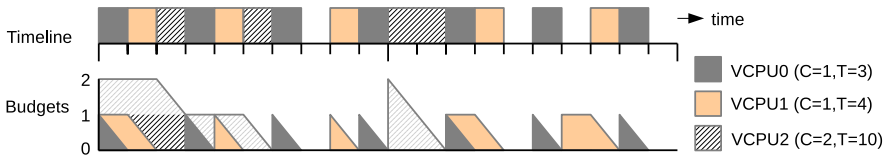

Figure 3: Example VCPU Schedule

cores) so multicasting interrupts should not be an issue.

Aside from interrupt handling, device drivers need to be written to support inter-sandbox sharing. Certain data structures have to be duplicated for each sandbox kernel, while others are shared and protected by synchronization primitives. For example, with a NIC driver, we duplicate indices into the receive (RX) ring buffer, while sharing both the transmit (TX) and RX buffers between sandboxes. Synchronization is used to read and update RX and TX descriptors in the respective ring buffers. Figure 4 shows an RX ring buffer shared between 4 sandboxes, with separate indices. Between t and $t + 1$, sandboxes 2, 3, and 4 all handle interrupts and advance their indices. The driver needs to be written so that a slot in the buffer only becomes ready for DMA data when it is not referenced by *any* index. Any of the 4 sandboxes can examine indexes to see if one is lagging above a *threshold* behind the others, as might be the case for a faulty sandbox. A functioning sandbox can then correct this by advancing indexes as necessary, or triggering fault recovery.

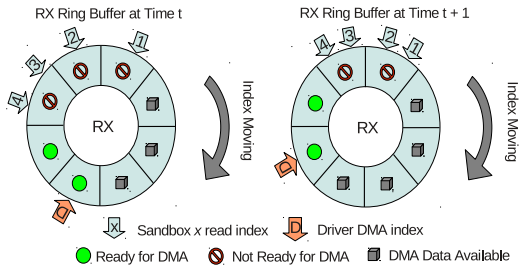

Figure 4: Example NIC RX Ring Buffer Sharing

The duplication of certain driver data structures, and synchronization on shared data may impact the performance of hardware devices multiplexed between sandboxes. However, I/O virtualization technologies to support device sharing such as SR-IOV [18] are now emerging, although not commonplace in embedded systems. Without hardware support, Quest-V's software-based shared driver approach is arguably more flexible than having devices assigned to single sandboxes. While technologies such as VT-d support I/O passthrough, they do not allow device sharing.

It is possible that faulty device drivers could issue DMA transfers to local APIC memory-mapped pages, to trigger arbitrary interrupts. A "storm" of IPIs could then be dispatched to remote cores, potentially flooding the system bus. In our tests to generate as many IPIs as quickly as possible we did not observe this as a problem, since there appears to be a limit on the number of interrupts on the bus itself. Moreover, Quest-V runs all interrupt handlers as threads bound to time-budgeted VCPUs, so a burst of interrupts cannot cause denial-of-service. A VCPU is placed into a background (low) priority class until its budget is replenished.

3.2 Fault Recovery

Quest-V is designed to be robust against software faults that could potentially compromise a system kernel. As long as the integrity of one sandbox is maintained it is theoretically possible to build a Quest-V multikernel capable of recovering service functionality online. This contrasts with a traditional system approach, which may require a full system reboot if the kernel is compromised by faulty software such as a device driver.

In this paper, we assume the existence of techniques to identify faults. Although fault detection mechanisms are not necessarily straightforward, faults are easily detected in Quest-V if they generate EPT violations. EPT violations transfer control to a corresponding monitor where they may be handled. More elaborate schemes for identifying faults will be covered in our future work. Here, we explain the details of how fault recovery is performed without requiring a full system reboot.

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. We assume monitors are trusted and have a minimal code base. If the fault does not automatically trigger a VM-Exit, it can be forced by a fault handler issuing an appropriate instruction. [4] 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

[4] For example, on the x86, the `cpuid` instruction forces a VM-Exit.

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 within a sandbox kernel 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* [1]. 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 in our future work.

Assuming that a fault detection event has either 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 re-initialize 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 more rigorously tested, may be used in recovery. Such component replacements can lead to system robustness through functional or implementation *diversity* [39]. That is, a component suffering a fault or compromising attack may be immune to the same fault or compromised 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 re-initialization. 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 paper, information needs to be exchanged between monitors about the actions necessary for fault recovery and what threads, if any, need to be migrated.

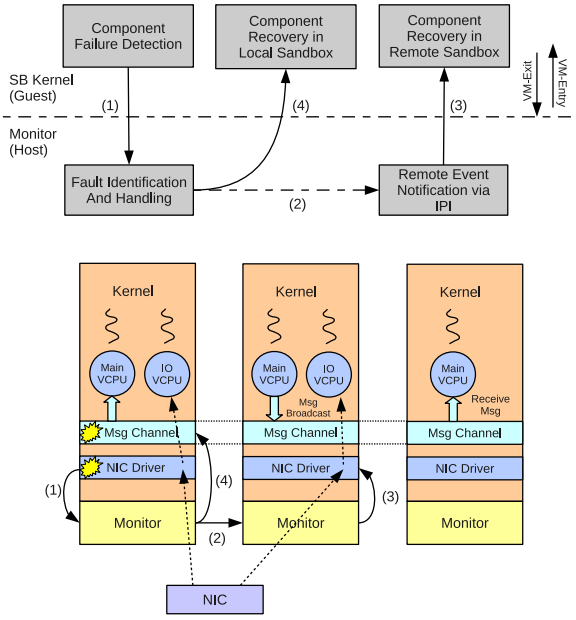

Figure 5: Example NIC Driver Recovery

An example of remote recovery involving a network interface card (NIC) driver is shown in Figure 5. Here, an IPI is issued from the faulting sandbox to the remote sandbox 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 is 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 driver's state is not recovered, but instead the driver is completely re-initialized. 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 5. 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 re-initializes 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 multicast 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, requiring the monitor itself to be responsible for 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 in step 4.

In the current implementation of Quest-V, we assume that all recovered services are re-initialized 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 recoverable state. Snapshots of such 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.

4 Experimental Evaluation

We conducted a series of experiments that compared Quest-V to both Linux and a non-virtualized Quest system. For network experiments, we ran Quest-V on a mini-ITX machine with a Core i5-2500K 4-core processor, featuring 8GB RAM and a Realtek 8111e NIC. In all other cases we used a Dell PowerEdge T410 server with an Intel Xeon E5506 2.13GHz 4-core processor, featuring 4GB RAM. Unless otherwise stated, all software threads were bound to Main VCPUs with 100% utilization.

4.1 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 HTTP 1.0-compliant single-threaded web server in user-space. Our simple web server was ported to a socket API that we implemented on top of lwIP. A remote Linux machine running `httperf` attempted to send requests at a rate of 120 per second during both the period of driver failure and normal operation of the web server. Request URLs referred to the Quest-V website, with a size of 17675 bytes.

Figure 6 shows the request and response rate over several seconds during which the server was affected by the faulting driver. The request and response rate recorded by `httperf` drops for a brief period while the NIC driver is re-initialized and the web server is restarted in another sandbox to the one that failed. Steady-state is reached in less than 0.5s of driver failure. This is significantly faster than a system reboot, which can take over a minute to restart the network service.

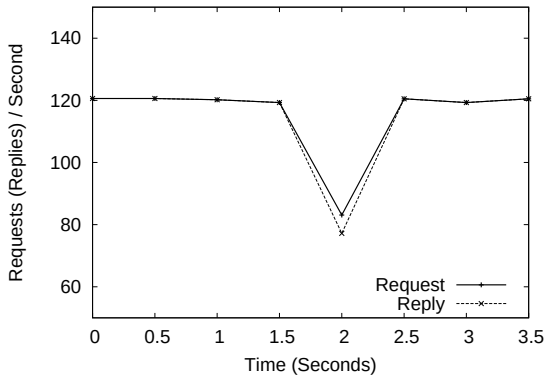

Figure 6: Web Server Recovery

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 re-initialized. This either involves re-initialization of 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 paper, we triggered the fault recovery event manually by assuming an error occurred. Aside from optional replacement of the faulting driver, and re-initialization, 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 1. The only added cost not shown is to restart the web server but Figure 6 shows this not to

be expensive. For most system components, we expect re-initialization to be the most significant recovery cost.

Phases	CPU Cycles	
	Local Recovery	Remote Recovery
VM-Exit	885	
Driver Replacement	10503	N/A
IPI Round Trip	N/A	4542
VM-Enter	663	
Driver Re-initialization	1.45E+07	
Network I/F Restart	78351	

Table 1: Overhead of Different Phases in Fault Recovery

4.2 Forkwait Microbenchmark

In Quest-V, sandboxes spend most of their life-time in guest mode, and system calls that trigger context switches will not induce VM-Exits to a monitor. Consequently, we tried to measure the overhead of hardware virtualization on normal system calls for Intel x86 processors. We chose the `forkwait` microbenchmark [3] because it involves two relatively sophisticated system calls, (`fork` and `waitpid`), involving both privilege level switches and memory operations.

40000 new processes were forked in each set of experiments and the total CPU cycles were recorded. We then compared the performance of Quest-V against a version of Quest without hardware virtualization enabled, as well as a Linux 2.6.32 kernel in both 32- and 64-bit configurations. Results in Table 2 suggest that hardware virtualization does not add any obvious overhead to Quest-V system calls. Moreover, both Quest and Quest-V took less time than Linux to complete their executions.

4.3 Address Translation Overhead

To show the costs of address translation as described in Figure 2, we measured the latency to access a number of data and instruction pages in a guest user-space process. Figures 7 and 8 show the execution time of a process bound to a Main VCPU with a 20ms budget every 100ms. Instruction and data references to consecutive pages are 4160 bytes apart to avoid cache aliasing effects. The results show the average cost to access working sets taken over 10 million iterations. In the cases where there is a TLB flush or a VM exit, these are performed each time the set of pages on the x-axis has been referenced.

For working sets less than 512 pages Quest-V (`Base` case) performs as well as a non-virtualized version of Quest. Extra levels of address translation with extended paging only incur costs above the two-level paging of a 32-bit Quest virtual memory system when address spaces are larger than 512 pages. For embedded systems, we do not see this as a limitation, as most applications will have smaller working sets. As can be seen, the costs of a VM-Exit are equivalent to a TLB flush, but Quest-V avoids this by operating more commonly in the `Quest-V base` case. Hence, extended paging does not incur significant overheads under normal circumstances, as the hardware TLBs are being used effectively.

	Quest	Quest-V	Linux32	Linux64
CPU Cycles	9.03E+09	9.20E+09	9.37E+09	1.29E+10

Table 2: Forkwait Microbenchmark

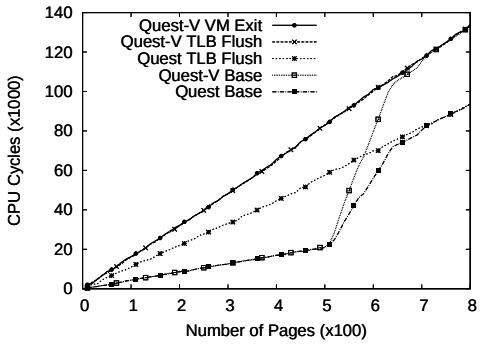

Figure 7: Data TLB Performance

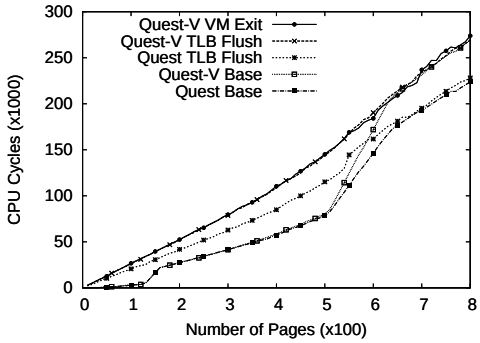

Figure 8: Instruction TLB Performance

4.4 Interrupt Distribution and Handling

Besides system calls, device interrupts also require control to be passed to a kernel. We therefore conducted a series of experiments to show the overheads of interrupt delivery and handling in Quest-V. For comparison, we recorded the number of interrupts that occurred and the total round trip time to process 30000 ping packets on both Quest and Quest-V machines. In this case, the ICMP requests were issued in 3 millisecond intervals from a remote machine. The results are shown in Table 3.

Notice that in Quest, all the network interrupts are directed to one core and in Quest-V, we deliver network in-

	Quest	Quest-V
# Interrupts	30004	30003
Round-trip time (ms)	5737	5742

Table 3: Interrupt Distribution and Handling Overhead

terrupts to all cores but only one core (i.e., one sandbox kernel) actually handles them. Each sandbox kernel in Quest-V performs early demultiplexing to identify the target for interrupt delivery, discontinuing the processing of interrupts that are not meant to be locally processed. Consequently, the overhead with Quest-V also includes dispatching of interrupts from the I/O APIC. However, we can see from the results that the performance difference between Quest and Quest-V is almost negligible, meaning neither hardware virtualization nor multicasting of interrupts is prohibitive. Here, Quest-V does not require intervention of a monitor to process interrupts. Instead, interrupts are directed to sandbox kernels according to rules setup in corresponding virtual machine control structures.

4.5 Inter-Sandbox Communication

The message passing mechanism in Quest-V is built on shared memory. While we will consider NUMA effects in the future, they are arguably less important for the embedded systems we are targeting. Instead of focusing on memory and cache optimization, we tried to study the impact of scheduling on message passing in Quest-V.

We setup two kernel threads in two different sandbox kernels and assigned a VCPU to each of them. One kernel thread used a 4KB shared memory message passing channel to communicate with the other thread. In the first case, the two VCPUs were the highest priority with their respective sandbox kernels. In the second case, the two VCPUs were assigned lower utilizations and priorities, to identify the effects of VCPU parameters (and scheduling) on the message sending and receiving rates. In both cases, the time to transfer messages of various sizes across the communication channel was measured. Note that the VCPU scheduling framework ensures that all threads are guaranteed service as long as the total utilization of all VCPUs is bounded according to rate-monotonic theory [22]. Consequently, the impacts of message passing on overall system performance can be controlled and isolated from the execution of other threads in the system.

Figure 9 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

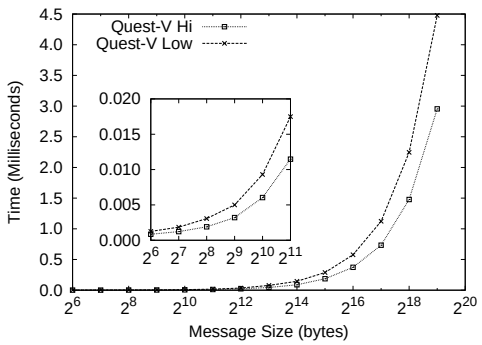

Figure 9: Message Passing Microbenchmark

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, the VCPU parameters have an effect on message transfer times.

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

4.6 Isolation

To demonstrate fault isolation in Quest-V, we created a scenario that includes both message passing and network service across 4 different sandboxes. Specifically, sandbox 1 has a kernel thread that sends messages through private message passing channels to sandbox 0, 2 and 3. Each private channel is shared only between the sender and specific receiver, and is guarded by EPTs. In addition, sandbox 0 also has a network service running that handles ICMP echo requests. After all the services are up and running, we manually break the NIC driver in sandbox 0, overwrite sandbox 0's message passing channel shared with sandbox 1, and try to wipe out 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 10.

In the experiment, sandbox 1 broadcasts messages to others at 50 millisecond intervals, while sandbox 0, 2 and 3 receive at 100, 800 and 1000 millisecond intervals. Also, another machine in the local network sends ICMP echo requests at 500 millisecond intervals to sandbox 0. All message passing threads are bound to Main VCPUs

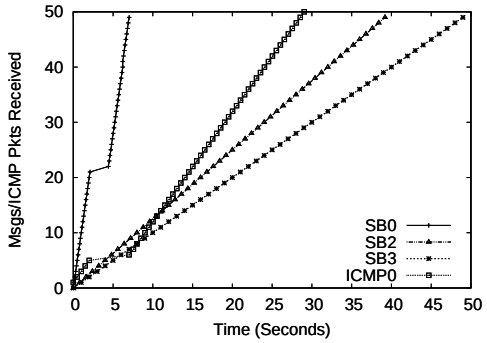

Figure 10: Sandbox Isolation

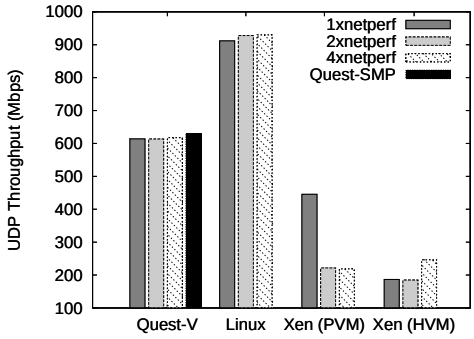

Figure 11: UDP Throughput

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 service happened for both message passing and network packet processing in sandbox 0, but all the other sandboxes were unaffected. This is because of memory isolation between sandboxes enforced by EPTs. When the "faulty" driver in sandbox 0 tries to overwrite memory of the other sandboxes, it simply traps into the local monitor because of a memory violation. Consequently, the only memory that the driver can wipe out is only the writable memory in sandbox 0. Hence all the monitors and other sandboxes will remain protected from this failure.

4.7 Shared Driver Performance

We implemented a shared driver in Quest-V for a single NIC device, providing a separate virtual interface for each sandbox requiring access. This allows for each sandbox to have its own IP address and even a virtual MAC address for the same physical NIC.

We compared the performance of our shared driver design to the I/O virtualization adopted by Xen 4.1.2, both para-virtualized (PVM) and hardware-virtualized (HVM). We used an x86_64 root-domain (Dom0) for Xen, based on Linux 3.1. For guests, and non-virtualization cases, we also used Ubuntu Linux 10.04 (32-bit kernel 2.6.32).

Figure 11 shows UDP throughput measurements using `netperf`, which was ported to the Quest-V and non-virtualized `Quest-SMP` systems. Up to 4 `netperf` clients were run in separate guest domains, or sandboxes, for the virtualized systems. For Xen, each guest had one VCPU that was free to run on any processor. Similarly, for non-virtualized cases, the clients ran as separate threads on arbitrary processors. Each client produced a stream of 16KB messages.

Quest-V shows better performance than other virtualized systems, although it is inferior to a non-virtualized Linux system for network throughput. We attribute this in part to the virtualization overheads but also to our system not yet being optimized. Future work will focus on performance tuning our system to reach throughput values closer to Linux, but initial results are encouraging. Note that the increases in throughput for all cases of increased `netperf` instances, except for paravirtualized Xen (`Xen (PVM)`), appear to be because of the increased traffic being generated by the clients. Xen is apparently sensitive to the VCPU utilization for its communicating threads [41, 23].

5 Related Work

The concept of a multikernel is featured in Barrelfish[6], which has greatly influenced our work. Barrelfish replicates system state rather than sharing it, 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 cache-line-sized messages. In contrast to Barrelfish, Quest-V uses virtualization mechanisms to partition separate kernel services as part of our goal to develop high-confidence systems.

Systems such as Hive [10] and Factored OS (FOS) [36] also take the view of designing a system as a distributed collection of kernels on a single chip. FOS is primarily designed for scalability on manycore systems with potentially 100s to 1000s of cores. Each OS service is factored into a set of communicating servers that collectively operate together. In FOS, kernel services are partitioned across spatially-distinct servers executing on separate cores, avoiding contention on hardware resources such as caches and TLBs. Quest-V differs from FOS in its primary focus, since the former is aimed at fault recovery and dependable computing. Moreover, Quest-V manages

resources across both space and time, providing real-time resource management that is not featured in the scalable collection of microkernels forming FOS.

Hive [10] is a standalone OS that targets features of the Stanford FLASH processor to assign groups of processing nodes to *cells*. Each cell represents a collection of kernels that communicate via message exchanges. The whole system is partitioned so that hardware and software faults are limited to the cells in which they occur. Such fault containment is similar to that provided by virtual machine sandboxing, which Quest-V relies upon. However, unlike Quest-V, Hive enforces isolation using special hardware *firewall* features on the FLASH architecture.

There have been several notable systems relying on virtualization techniques to enforce logical isolation and implement scalable resource management on multicore and multiprocessor platforms. Disco [9] is a virtual machine monitor (VMM) that was key to the revival in virtualization in the 1990s. It supports multiple guests on multiprocessor platforms. Memory overheads are reduced by transparently sharing data structures such as the filesystem buffer cache between virtual machines.

Cellular Disco [15] extends the Disco VMM with support for hardware fault containment. As with Hive, the system is partitioned into cells, each containing a copy of the monitor code and all machine memory pages belonging to the cell's nodes. A failure of one cell only affects the VMs using resources in the cell. Quest-V does not focus explicitly on hardware fault containment but its system partitioning into separate kernels means that it is possible to support such features.

Xen[5] is a subsequent VMM that uses a special driver domain and (now optional) paravirtualization techniques [38] to support multiple guests. In contrast to VMMs such as Disco and Xen, Quest-V operates as a single system with sandbox kernels potentially implementing different services that are isolated using memory virtualization. Quest-V also avoids the need for a split-driver model involving a special domain (`Dom0` in Xen) to handle device interrupts.

Helios [26] is another system that adopts multiple *satellite kernels*, which execute on heterogeneous platforms, including graphics processing units, network interface cards, or specific NUMA nodes. Applications and services can be off-loaded to special purpose devices to reduce the load on a given CPU. Helios builds upon Singularity [17] and all satellite microkernels communicate via message channels. Device interrupts are directed to a *coordinator* kernel, which restricts the location of drivers.

Helios, Singularity, and the *Sealed Process Architecture* [17] enforce dependability and safety using language support based on C#. In Quest-V, virtualization techniques are used to isolate software components. While this may seem more expensive, we have seen on modern processors with hardware virtualization support that this is not the case.

In other work, Corey[8] is a library OS providing an interface similar to the Exokernel[13], and which attempts to address the bottlenecks of data sharing across modern multicore systems. Cores can be dedicated to applications which then communicate via shared memory IPC. Quest-V similarly partitions system resources amongst sandbox kernels, but in a manner that ensures isolation using memory virtualization.

Finally, Quest-V has similarities to systems that support self-healing, such as ASSURE [29] and Vigilant [27]. Such self-healing systems contrast with those that attempt to verify their functional correctness before deployment. seL4 [21] attempts to verify that faults will never occur at runtime, but as yet has not been developed for platforms supporting parallel execution of threads (e.g., multicore processors). Regardless, verification is only as good as the rules against which invariant properties are being judged, and as a last line of defense Quest-V is able to recover at runtime from unforeseen errors.

6 Conclusions and Future Work

This paper describes a virtualized multikernel, called Quest-V. Extended page tables are used to isolate sandbox kernels across different cores in a multicore system. This leads to a distributed system on a chip that is robust to software faults. While operational sandboxes proceed as normal, faulting sandboxes can be recovered online using either local or remote fault recovery techniques.

Experiments show that hardware virtualization does not add significant overheads in our design, as VM-Exits into monitor code are only needed to handle software faults and update extended page tables. Unlike conventional hypervisors that virtualize underlying hardware for use by multiple disparate guests, Quest-V assumes all sandboxes are operating together as one collective system. Each sandbox kernel is responsible for scheduling of its threads and VCPUs onto local hardware cores. Similarly, memory allocation and I/O management are handled within each sandbox without involvement of a monitor.

In this paper, we assume the existence of a fault detector that transfers control to a local monitor for each sandbox. While such transfers can be triggered by EPT violations, we will investigate more advanced techniques for fault detection. Similarly, we will investigate policies and mechanisms for online recovery of faults requiring the continuation of stateful tasks. Some method of check-

pointing and transactional recovery might be appropriate in such cases. Although our fault recovery schemes thus far require re-initialization of a service, we feel this is still better in many cases than a full system reboot.

Since Quest-V is a system built from scratch, it lacks the rich APIs and libraries found in modern systems. This limits our ability to draw comparisons with current OSes, as evidenced by our time spent porting `netperf` and a socket API to Quest-V. We will continue to add more extensive features, while investigating techniques to address security as well as safety violations. Similarly, more advanced multi-threaded applications will be developed, to study migration between sandbox kernels. Notwithstanding, we believe Quest-V's design could pave the way for future high-confidence systems, suitable for emerging applications in safety-critical, real-time and embedded domains. NB: The source code is available on request.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*. See www.intel.com.

[2] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2006.

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

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

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

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

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

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

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

[11] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, 1999.

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

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[14] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, November 6-8 2006.

[15] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, 1999.

[16] NITRD Working Group: IT Frontiers for a New Millenium: High Confidence Systems, April 1999. http://www.nitrd.gov/pubs/bluebooks/2000/hcs.html.

[17] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*, pages 341–354, 2007.

[18] PCI-SIG SR-IOV primer. www.intel.com.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[20] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architectures and Compilation Techniques (PACT '04)*, October 2004.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

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

[23] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, 2006.

[24] G. Morrisett, K. Crary, N. Glew, D. Grossman, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[25] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[26] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 221–234, 2009.

[27] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan. Vigilant: Out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 42:26–31, January 2008.

[28] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[29] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic software self-healing using rescue points. In *Proceedings of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[30] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[31] M. Spuri, G. Buttazzo, and S. S. S. Anna. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

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

[33] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[34] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

[35] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

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

[37] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4), December 2010. Special VMware Track.

[38] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation*, December 2002.

[39] D. Williams, W. Hu, J. Davidson, J. Hiser, J. Knight, and A. Nguyen-Tuong. Security through diversity. *Security & Privacy, IEEE*, 7:26–33, Jan 2009.

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

[41] *Xen Network Throughput and Performance Guide*. http://wiki.xen.org/wiki/Network_Throughput_-Guide.

[42] XXXX. Omitted for blind review.

[43] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE Real Time Systems Symposium*, December 2006.

[44] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *In Proceedings of the 15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 129–141, March 2010.