2015

Leveraging virtualization technologies for resource partitioning in mixed criticality systems

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

LEVERAGING VIRTUALIZATION TECHNOLOGIES FOR RESOURCE PARTITIONING IN MIXED CRITICALITY SYSTEMS

by

YE LI

B.E., Wuhan University, 2009

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2015

Approved by

First Reader

Richard West, PhD
Associate Professor of Computer Science

Second Reader

Abraham Matta, PhD
Professor of Computer Science

Third Reader

Jonathan Appavoo, PhD
Assistant Professor of Computer Science

Acknowledgments

First and foremost, I wish to thank my advisor, professor Richard West. He has been supportive, patient, and offered tremendous help and guidance throughout the 6 years of both my personal and academic journey. I can only feel immense gratitude for the invaluable knowledge and experience I learned from him. He constantly pushed me to jump out of my comfort zone and challenged me with philosophical questions that made me a better researcher. Professor West gave me the freedom to expand upon the Quest kernel platform he developed in the early 2000s. Without his trust and encouragement, this dissertation would not have come to be. Finally, I have to thank Rich for not only being my advisor, but also a great friend. I will never be able to forget the countless hours we spent in the local coffee shop discussing research, cars, sports, and everything else in the whole universe.

I would also like to thank my committee members, professor Abraham Matta, professor Jonathan Appavoo, professor Hongwei Xi, and professor Mark Crovella. They provided invaluable comments, feedbacks, and guidance, which lead to the completion of this dissertation. I cannot thank them enough for their sincerity, patience, and support all these years.

Finally, I would like to thank all the people that were or are currently involved in the Quest platform development. Gary Wong first joined Rich at the very beginning of the Quest project. I thank him for helping Rich on building this fantastic platform for research and education. I would also like to thank my colleagues and friends, Matthew Danish, Eric Missimer, Ying Ye, Zhuoqun Cheng, and Katherine Zhao. They all worked on the Quest project at various times. Along with all the other members of the BOSS group, they made my 6 years at BU an enjoyable experience.

LEVERAGING VIRTUALIZATION TECHNOLOGIES FOR RESOURCE PARTITIONING IN MIXED CRITICALITY SYSTEMS

(Order No.)

YE LI

Boston University, Graduate School of Arts and Sciences, 2015

Major Professor: Richard West, Associate Professor of Computer Science

ABSTRACT

Multi- and many-core processors are becoming increasingly popular in embedded systems. Many of these processors now feature hardware virtualization capabilities, such as the ARM Cortex A15, and x86 processors with Intel VT-x or AMD-V support. Hardware virtualization offers opportunities to partition physical resources, including processor cores, memory and I/O devices amongst guest virtual machines. Mixed criticality systems and services can then co-exist on the same platform in separate virtual machines. However, traditional virtual machine systems are too expensive because of the costs of trapping into hypervisors to multiplex and manage machine physical resources on behalf of separate guests. For example, hypervisors are needed to schedule separate VMs on physical processor cores. Additionally, traditional hypervisors have memory footprints that are often too large for many embedded computing systems. This dissertation presents the design of the Quest-V separation kernel, which partitions services of different criticality levels across separate virtual machines, or *sandboxes*. Each sandbox encapsulates a subset of machine physical resources that it manages without requiring intervention of a hypervisor.

In Quest-V, a hypervisor is not needed for normal operation, except to bootstrap the system and establish communication channels between sandboxes. This approach not only reduces the memory footprint of the most privileged protection domain, it removes it from the control path during normal system operation, thereby heightening security.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
CAN	Controller Area Network
DMA	Direct Memory Access
ECU	Electronic Control Unit
EPT	Extended Page Table
GPU	Graphics Processor Unit
IDT	Interrupt Descriptor Table
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
MMU	Memory Management Unit
NUMA	Non-Uniform Memory Access
OS	Operating System
PIT	Programmable Interval Timer
RTOS	Real Time Operating System
SMP	Symmetric Multiprocessing
TLB	Translation Lookaside Buffer
VGA	Video Graphics Array
VM	Virtual Machine
VMM	Virtual Machine Monitor

Chapter 1

Introduction

1.1 Motivation

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

A major challenge to mixed criticality systems is the safe isolation of separate components with different levels of criticality. Isolation has traditionally been achieved by partitioning components across distributed modules, which communicate over a network such as a CAN bus. For example, Integrated Modular Avionics (IMA) [Wat06] is used to describe a distributed real-time computer network capable of supporting applications of differing criticality levels aboard an aircraft. To implement such concepts on a multicore

platform, a software architecture that enforces the safe isolation of system components is required.

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

While modern hypervisor solutions such as Xen [BDF+03] and Linux-KVM [Hab08] leverage hardware virtualization to isolate their guest systems, they are still required for CPU, memory, and I/O resource management. Traps into the hypervisor occur every time a guest system needs to be scheduled, when a remapping of guest-to-machine physical memory is needed, or when an I/O device interrupt is delivered to a guest. This is both unnecessary and potentially too costly for mixed criticality systems with real-time requirements. Additionally, existing hypervisor solutions do not provide predictable communication and service migration facilities to guest virtual machines. This also renders them less effective for real-time mixed criticality applications. The challenges of deploying mixed criticality systems on multi- and many-core platforms along with the limitations of current hypervisor based solutions motivated us to come up with the design and implementation of a new partitioning system from the ground up.

1.2 Quest-V Separation Kernel

The goal of this research is to design and implement an operating system that uses hardware-assisted virtualization as an extra *ring of protection*, to achieve efficient resource partition-

ing and performance isolation for subsystem components. The system, called Quest-V, is a separation kernel [Rus81] design, effectively operating as a distributed system on a chip. Quest-V is centered around three main goals: safety, predictability and efficiency. Of particular interest is support for safety-critical applications, where equipment and/or lives are dependant on the operation of the underlying system. With recent advances in fields such as cyber-physical systems, more sophisticated OSes beyond those traditionally found in real-time and embedded computing are now required. Consider, for example, an automotive system with services for engine, body, chassis, transmission, safety and infotainment. These could be consolidated on the same multicore platform, with space-time partitioning to ensure malfunctions do not propagate across services. Virtualization technology can be used to separate different groups of services, depending on their criticality (or importance) to overall system functionality.

However, unlike traditional virtualization solutions, Quest-V treats hardware-assisted virtualization features as hardware-assisted resource partitioning capabilities. Hardware resources are partitioned amongst different systems components in Quest-V. This avoids traps into a hypervisor (a.k.a. virtual machine monitor, or VMM) when making scheduling and I/O management decisions. System components are capable of scheduling themselves on available processor cores and are granted access to specific subsets of I/O devices and memory. This design leads to the following observations:

1. Since hardware resources are partitioned, resource multiplexing between subsystem components in Quest-V is not necessary. The Quest-V separation kernel is only responsible for granting access rights and leaves resource management responsibilities to the components themselves. The elimination of the resource management logic reduces the complexity and footprint of the code running in the most privileged domain of the system.

2. The most privileged domain of the system does not need to be involved in the service requests from subsystem components. It is only needed to initialize components, handle faults, and setup communication channels. This makes the subsystem components more predictable and efficient since no additional code from the separation kernel is executed during their normal execution.

3. Because of its simplicity, small footprint, and elimination from subsystem component control flow, the Quest-V separation kernel heightens system security and safety. The separation kernel with its resource partitioning logic constitutes a minimal trusted code base and small attack surface. This is essentially more effective than a micro-kernel, which needs to support the concept of an address space, threads, inter-process communication, and naming. The micro-kernel needs to be accessed for all the service requests involving these concepts and operations.

In addition to the above benefits, predictable communication and service migration between high criticality components in Quest-V are guaranteed by a real-time VCPU scheduling framework. Experiments show that worst case execution times for message passing and migration between isolated critical subsystem components can be guaranteed. Third party systems such as a Linux front-end can be supported with minimal modification and near bare-metal performance. Preliminary results show that Quest-V is able to make efficient use of CPU, memory and I/O partitioning, using hardware virtualization.

1.3 Thesis Statement

Thesis: *A separation kernel that offers efficient resource partitioning, predictable communication and performance isolation for mixed criticality systems on a multi-core platform is implementable with hardware-assisted virtualization, and software-based scheduling*

and communication.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces the details of mixed criticality systems and reviews some of the potential existing solutions to the multi-core mixed criticality system challenges. Chapter 3 describes the architecture of the Quest-V separation kernel and its resource partitioning capabilities. Chapter 4 details Quest sandbox support in Quest-V and explains how predictable communication and service migration are implemented between Quest sandboxes. Fault recovery and device sharing between Quest sandboxes are also discussed in this chapter. Chapter 5 introduces third party sandbox support in Quest-V. This includes Linux and OSEK/AUTOSAR OS sandbox support. Finally, Chapter 6 and 7 provide overall conclusions and discuss possible future work.

Chapter 2

Background and Related Work

2.1 Mixed Criticality Systems

Mixed criticality systems are often seen in the automotive and avionics industry, where the failure of a safety critical system component will potentially lead to catastrophic consequences including human life losses. Traditionally, the separation of subsystems with different criticality levels had been considered absolute and was always accomplished by means of physical isolation. In traditional Unmanned Aerial Vehicle (UAV) systems, for example, well defined divisions are created to group subsystems of different criticality levels [BBB+09]. Each system division is then assigned its own hardware and software and interfaces with each other via external buses and I/O. However, as the complexity of mixed criticality systems increases, this approach becomes increasingly inefficient mostly due to communication overhead and increased power consumption. For instance, it was estimated that by 2014, the amount of cables on a passenger aircraft varies from 200 km to 600 km depending on the aircraft model [NEX]. And a typical luxury vehicle in the early 2000s already have more than 2km of wire in the harness, 2000 terminals, 350 connectors, and nearly 1500 different circuits [KWMH96]. The electronics of a typical vehicle today consumes an average of above 2000 Watts of power. The peak power load can reach more than 12kW [NH00].

The inherent inefficiency of physical isolation soon led to the idea of consolidating sub-

systems of different criticality levels onto a single software system with shared hardware resources. However, this approach introduced the challenge of maintaining the isolation between different system components. One solution is to use process isolation. To help design and verify systems adopting this approach, industrial standards such as ARINC 653 [ARI08] and DO-178B [Aut92] had been developed for avionics systems. ARINC 653 is a software specification for space and time partitioning in safety critical avionics real-time operating systems. DO-178B is a certification document that uses assurance levels to guide the design of reliable software in certain airborne systems. In addition to spatial isolation provided by process address spaces, the temporal isolation between processes/partitions in these systems relies on the scheduling policy of the operating system. In ARINC 653, dedicated time slots can be allocated to partitions through the APEX API. Various real-time scheduling policies for mixed criticality systems [BD14] had also been proposed in recent years to provide criticality aware CPU resource management. One example system developed under these standards is the INTEGRITY-178B RTOS [INTb]. INTEGRITY-178B RTOS is a certified operating system that uses a hardware memory management unit (MMU) for memory protection and to isolate system components. Its secure partitions are designed for Ada, C, and Embedded C++ programs.

Even though the process based approach can provide sufficient spatial isolation between subsystem components for consolidation of a mixed criticality system, it still has two major limitations. First, a process is a logical abstraction that does not qualify how resources are managed in real-time; the operating system kernel is needed to multiplex resources amongst processes. Moreover, the process environment only exposes limited hardware capabilities to applications or tasks; it is difficult to support certain legacy services that require direct hardware access without significant engineering effort and visible performance overhead [HHL+97]. These limitations have since led to the increasing pop-

ularity of the use of virtualization in mixed criticality embedded systems.

2.2 Virtualization

Control Program/67 operating system [MS70], or CP/67, was the first software product to provide a virtual machine capability in order to offer time-sharing for multiple users. CP/67 gave each user a virtual machine in which the single-user Conversational Monitor System (CMS) operating system could be run to provide command processing and information management functions. With the success of CP/67, IBM announced VM/370 [Cre81], which is also designed to offer multiprocessing capabilities. And it soon became one of the most popular operating systems offering good interactive computing facilities and the capability to operate guest operating systems in virtual machines.

The use of virtualization to offer time-sharing capability was soon superseded by time-sharing operating systems such as UNIX. However, the concept of software virtualization emerged again in the 90s as a solution to address software scalability issues and to consolidate workloads on multi- and many-core hardware platforms. Disco [BDR97] is a hypervisor designed to run commodity OSes on scalable multiprocessors (e.g. Stanford FLASH). Virtual CPUs in Disco emulate MIPS R10000 instructions. Guest operating system kernels need to be paravirtualized in their Hardware Abstraction Layer (HAL) and driver framework to run on the Disco hypervisor. Xen [BDF+03] and Linux-KVM [Hab08] are virtual machine monitors designed for commodity hardware platforms today. They both leverage Linux kernel services to host different guest operating systems on a single physical machine. Desktop virtualization systems such as VMware [VMW] Workstation and VirtualBox [VIR] offer similar functionality.

Most of these software virtualization systems are designed to offer transparent and efficient hardware resource multiplexing. This conventional virtual machine monitor de-

sign mainly focuses on scalability and maximum resource utilization rather than resource partitioning and predictability as required by mixed criticality systems. This makes it necessary for a hypervisor to constantly interrupt virtual machine execution for resource management such as virtual machine scheduling and I/O device sharing. On some architectures (e.g. x86) with *non-privileged sensitive instructions* [PG74], inefficient software techniques such as *binary translation* are required to support unmodified guest virtual machines. This further increases the level of hypervisor interference. Additionally, most of these systems provide a virtual or emulated device interface for guest operating systems by default. By not allowing guests to access I/O devices directly, it is easier to manage hardware resources. However, a virtual or emulated device interface requires hypervisor intervention in all I/O related operations in the guest. Even though PCI passthrough is supported in recent versions of Xen and Linux-KVM, guest virtual machines can only directly access device registers. The hypervisor is still responsible for initial interrupt handling and interrupt acknowledgment. This potentially forces two hypervisor traps for each interrupt. ELI [GAH+12] is a software approach for handling a subset of device interrupts within guest virtual machines directly with shadow IDTs. In order to achieve direct interrupt delivery and handling, the virtual machine scheduling in the hypervisor has to be limited.

Because of the constant hypervisor intervention during guest virtual machine execution, it is difficult for conventional virtual machine monitors to provide the temporal isolation and efficiency required by mixed criticality systems. Some other systems, such as XtratuM [CRM10], Wind River Hypervisor [WIN], INTEGRITY multivisor [MUL], and Mentor Graphics Embedded Hypervisor [MEN] target and are optimized for embedded applications, but still feature the traditional hypervisor design and suffer from the same fundamental problems.

To help alleviate the performance overhead of virtual machine monitors, hardware vir-

tualization features had been added to many modern processor architectures. Hardware-assisted virtualization provides special execution environment and hardware resource management capabilities to facilitate the construction of hypervisors. The concept of hardware-assisted virtualization was first introduced in the IBM System/370 to offer physical partitioning of hardware in multiprocessor systems in the 60s. It essentially divides a dual processor system into two sides. Each side is a separate machine that can be operated independently. Logical Partitioning (LPAR) [BHR89] was later introduced in the IBM 3090E and ES/3090S processors. Similar to physical partitions, LPARs operate independently and are isolated from one another. The only interaction between partitions is via I/O operations. However, unlike physical partitions, the number of LPARs can exceed the number of processors (up to 6). An LPAR dispatcher manages and multiplexes hardware resources amongst different partitions. In 2006 Intel and AMD introduced their VT-x and AMD-V processors, respectively, with support for hardware virtualization. These new hardware virtualization features are designed as extensions to the widely used x86 architectures and provide hypervisor software with efficient hardware resource management capabilities in a virtual machine environment. Most of the popular hypervisors on the x86 platform now takes advantage of these hardware features to reduce virtualization overheads. However, as mentioned earlier, the traditional virtual machine design still renders them less effective in mixed criticality, safety critical applications.

2.3 Multikernel

The multikernel design is based on the observation of two evident trends in the architecture of future computers: rising core count and increasing hardware diversity, both between cores within a machine, and between systems with varying interconnect topologies and performance trade-offs. These trends impose challenges on current general-purpose

operating systems designed for ccNUMA and SMP machines.

General-purpose OSes today do not scale to the increasing core count of future architecture due to their shared memory kernel design with data structures protected by locks. The increasingly diverse hardware design also makes optimization of general OSes complicated and effective only to specific platforms. Moreover, current OS designs do not consider the core heterogeneity that is becoming more common in modern hardware architectures. To solve these problems, the multikernel [BBD+09] model was proposed. The multikernel design takes advantage of the networked nature of the machine to rethink OS architecture using ideas from distributed systems. It is guided by three principals: (1) make all inter-core communication explicit, (2) make OS structure hardware-neutral, and (3) view state as replicated instead of shared. Barrelfish [BBD+09] is a multikernel that replicates rather than shares system state, to avoid the costs of synchronization and management of shared data structures. Barrelfish runs a small kernel on each core in the system, and the OS is built as a set of cooperating processes, each running on one of these kernels, sharing no memory, and communicating via message passing.

The distributed design of multikernel can be used to logically partition system components in mixed criticality systems. However, isolation between different kernels in a multikernel is not enforced; all the kernels execute at the same and lowest privilege levels of the underlying hardware architecture. This is insufficient for the spatial isolation requirement imposed by mixed criticality applications.

Popcorn [POP], Twin-Linux [JPN+10], Mint Linux [NSN+11], and SHIMOS [SMI08] are recent efforts that combine the multikernel philosophy and traditional OS designs such as Linux. These systems boot multiple Linux kernels on a multi-core processor and partition hardware between them cooperatively in software. The Linux kernels are modified to share hardware resources, but also without isolation provided by hardware.

2.4 Separation Kernel

Computer security problems and solutions started drawing large interests since the late 1960s due to the increasingly wide adoption of computer systems in the government and military facilities. Conventional operating systems at the time all suffered from the problem of completeness and are highly susceptible to hostile penetration. If even one error in an operating system allows a user to write a program that subverts the operating system's access controls, hundreds of other errors may have been corrected to no avail [LWS+74]. This had led to the development of the security kernel approach, which represents an attempt to find an alternative to the futile and never-ending cycle of conducting penetration tests and correcting errors.

A security kernel is defined to be the hardware/software component that implements the concept of a reference monitor [And72], an abstract mechanism that controls the flow of information within a computer system by mediating every attempt by a subject (active systems element) to access an object (information container). The basis of the security kernel idea is that a small central portion of an operating system can be designed in such a way as to control all the rest of the system and in so doing make sure that the system functions according to some principle of good behavior [Ame81]. To verify the correctness of a security kernel, *flow analysis* [Mil76] is used to prove a formal specification of the kernel conforms to a simple mathematical state machine model [BL74] that the kernel design is based upon. However, after the construction of several kernelized systems [GLSS77][MD79][PKK+79], it is noted that the simplified mathematical model forms a sufficient but not necessary basis for preventing system compromises but it is insufficient in satisfying all service needs. For instance, in a printer spooler [Rus81], if the spooler and its spool files are at the highest security level, then the users of lower security levels cannot inspect their own spool files to monitor the progress of their jobs. This vio-

lates the *Simple Security Property*, which states that a subject at a given security level may not read an object at a higher security level. However, if the spool files are classified at the level of their owners while the spooler continues to run at the highest level so that it may read all spool files, then the spooler cannot delete spool files after they have been printed. This violates the **-Property*, which states that a subject at a given security level must not write to any object at a lower security level. To implement system services similar to the spooler, we have to create exceptions to the basic model and ensure that they do not destroy the security provided by the basic model. In practice, these exceptions quickly complicate the security model and make the centralized security kernel approach ineffective.

A very simple and natural model for a computer system where security does not rely upon a central mechanism is a functionally distributed system. In such a system, various functions are provided by specialized individual subsystems which are physically separated from each other and provided with only limited channels for communication with one another. With this system structure, a lot of security problems can be avoided or considerably simplified due to the separation of concerns. Of course, the components of a system usually interact with each other and cannot be studied independently in some cases. However, this challenge is no less formidable in a conventional security kernel. The same interactions and dependencies are also present in a centralized design, only more difficult to handle because of the lack of visibility [Rus81].

These observations had led to the development of the concept of a separation kernel [Rus81]. Similar to a virtual machine monitor, a separation kernel provides isolated *regimes* that resemble virtual machines for each component of a system. However, a separation kernel differs from a VMM in that there is no requirement for it to provide VMs, which are exact copies of the base hardware. A separation kernel should support explicit communication channels between regimes, while enforcing absolute isolation otherwise.

There exists commercial separation kernels such as LynxSecure [LYN], PikeOS [PIK], and INTEGRITY-178B RTOS [INTb]. Very few details of these systems are available in the public domain. LynxSecure separation kernel targeted at safety-critical real-time systems; it resembles a typical hypervisor with support of multiple guest operating systems. PikeOS is a separation micro-kernel [KEH+09] that supports multiple guest VMs, and targets safety-critical domains such as IMA. The micro-kernel supports a virtualization layer that is required to manage the spatial and temporal partitioning of resources amongst all guests. INTEGRITY-178B RTOS is a certified separation kernel that uses the MMU for memory protection. Its secure partitions are designed for Ada, C, and Embedded C++ programs instead of exposing maximum hardware capabilities.

Muen [MUE] is an open source prototype separation kernel written in the SPARK programming language. It is claimed to have been formally proven to contain no run-time 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, like traditional hypervisor, traps into the Muen separation kernel are necessary to handle external interrupts and to schedule subjects. A subject in Muen is automatically preempted by the separation kernel through the Intel VMX (Virtual Machine eXtensions) preemption timer when its alloted time slice is over [BR13].

Jailhouse [JAI] is a partitioning hypervisor that can create asymmetric multiprocessing setups on Linux based systems. It is able to run bare-metal real-time applications alongside Linux in separate *Cells* isolated using virtualization. Hardware resources such as processor cores and devices are statically partitioned for each Cell in Jailhouse. However, Jailhouse is currently still in prototype stage and there is no support for device assignment nor interrupt access control. Additionally, predictable inter-cell communication is currently also not available. In Jailhouse, a Linux kernel has to be bootstrapped first before

Cells can be created. This could potentially limit its application for platforms with strict resource limitations.

Even though the separation kernel design with isolated regimes is ideal for the deployment of mixed criticality applications, a straightforward implementation on the basis of existing hypervisor or conventional hypervisor design suffers from the lack of temporal isolation and efficiency as mentioned earlier. In order to respect the differences between separation kernels and VMMs and to take advantage of modern multi-/many-core platforms, an appropriate separation kernel design and implementation for mixed criticality systems should aim for providing predictability and efficiency by focusing on resource isolation instead of multiplexing.

2.5 Other Partitioning Systems

Operating system level virtualization is an operating system kernel approach to allow resource isolation between different user space instances. These instances, or containers, are usually collections of processes with private filesystem namespaces and various resource limits. Example implementations of this approach include the Linux Containers (LXC) [LXC], Docker [DOC], Linux-VServer [SPF+07], and Solaris Zones [SOL]. LXC, for instance, takes advantage of the Linux cgroup feature to account and isolate resource usage (e.g. CPU, memory, disk I/O, network) for a collection of processes in a container. cgroup also provides namespace isolation so that containers in LXC are allowed to have their own view of the operating environment including process trees, networking, user IDs, and mounted filesystems. As compared to virtual machines, containers essentially traded isolation for efficiency. The spatial isolation between containers is still based on process address space isolation. Consequently, the hardware capabilities exposed by a container is similar to that of a normal Linux process and all the containers still share a large amount

of kernel states which makes the isolation much weaker than VMs. For mixed criticality systems, the isolation provided by containers is not strong enough to host tasks of different criticality levels; the compromise of the operating system kernel threatens the integrity of all the containers in an operating system level virtualized system.

NoHype [SKLR11] 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 traps into the hypervisor. Hypervisor traps are treated as errors and will terminate the guest. For safety-critical applications it is necessary to handle faults without simply terminating guests. Additionally, NoHype focuses on security instead of mixed criticality and embedded applications. Predictability is basically not a concern for NoHype. However, the idea of avoiding hypervisor traps is similar to the Quest-V separation kernel philosophy.

Dune [BBM+12] uses hardware virtualization to create a sandbox for safe user-level program execution. By allowing user-level access to privileged CPU features, certain applications (e.g. garbage collection) can be made more efficient. However, most system services are still redirected to the Linux kernel running in hypervisor mode. VirtuOS [NB13] uses virtualization to partition existing operating system kernels into service domains, each providing a subset of system calls. Exceptionless system calls are used to request services from remote domains. The system is built on top of Xen and relies on both the shared memory facilities and event channels provided by the Xen VMM to facilitate communication between different domains. The PCI passthrough capability provided by the Xen VMM is also used to partition devices amongst service domains. However, interrupt handling and VM scheduling still requires VMM intervention.

Other systems that partition resources on many-core architectures include Factored OS [WA09], Corey [BWCC+08], Hive [CRD+95] and Disco [BDR97]. Unlike Quest-V,

these systems are focused on scalability rather than isolation and predictability.

Chapter 3

Quest-V Separation Kernel Architecture

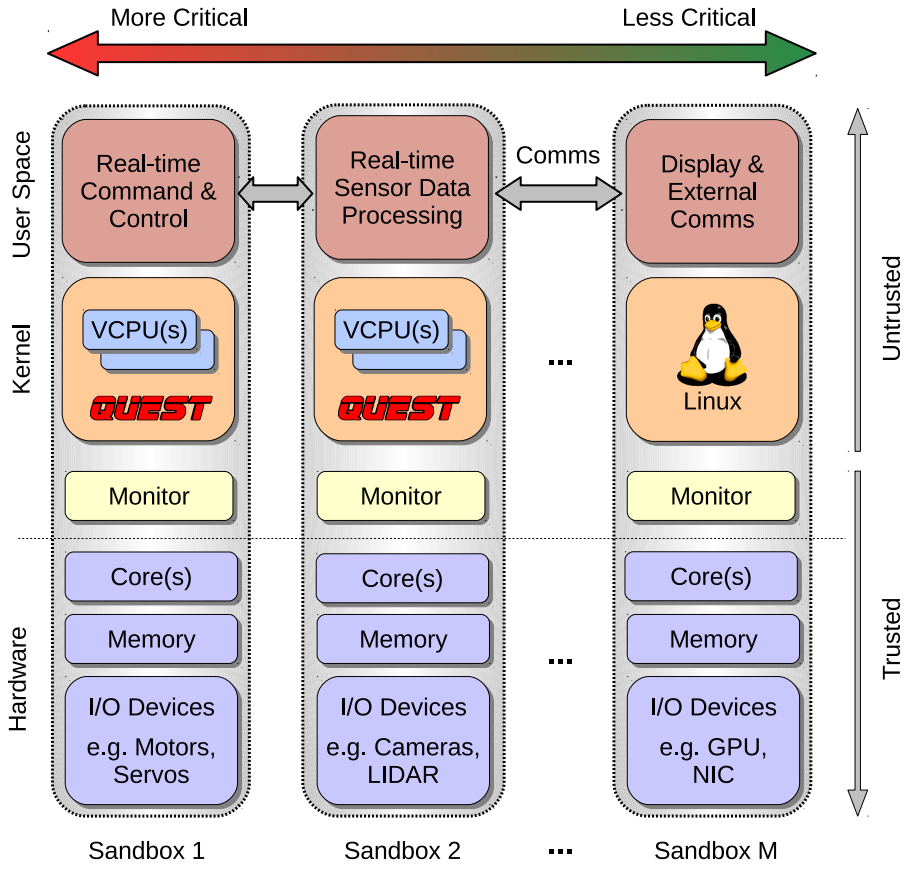

Figure 3.1: Example Quest-V Architecture Overview

One of the primary goals of Quest-V is to achieve efficient resource partitioning and performance isolation for subsystem components with different criticality levels. Quest-V uses hardware virtualization as an extra ring of protection and operates as a distributed

system on a chip. A high-level overview of the Quest-V architecture is shown in Figure 3.1. The current implementation works on Intel VT-x platforms, although conceptually it should work on any architecture with virtualization support, including AMD-V and ARMv7-A.

The system is partitioned into separate *sandboxes*, each responsible for a subset of machine physical memory, I/O devices and processor cores. Trusted monitor code is used to launch *guest* services, which may include their own kernels and user space programs. A monitor is responsible for managing special *extended page tables* (EPTs) that translate guest physical addresses (GPAs) to host physical addresses (HPAs), as described later in Figure 3.2.

Figure 3.1 shows an example of three sandboxes, where two are configured with Quest-native safety-critical services for command, control and sensor data processing. These services might be appropriate for a future automotive system that assists in vehicle control. Other less critical services could be assigned to vehicle infotainment services, which are partitioned in a sandbox that has access to a local display device. A non-real-time Linux system could be used in this case, perhaps also managing a network interface to communicate with other vehicles or the surrounding environment, via a vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communication link.

3.1 Distributed Monitors

Unlike traditional hypervisor design in which a single monitor is responsible for multiplexing resources amongst multiple virtual machine instances, Quest-V features a distributed monitor design, where a separate monitor exists for each sandbox. A monitor manages resources for and handles requests from only a single guest environment. Quest-V 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* – With distributed monitors, there is no need to have implicit shared data structures amongst monitors in Quest-V. This reduces resource contention and increases efficiency at the monitor level. Additionally, since each monitor only services one sandbox, there is no need for a monitor to determine at runtime the guest that needs its service. This feature, in turn, improves predictability by eliminating unnecessary synchronization.

2. *Functional Diversity* – Monitors in Quest-V can be customized to satisfy the needs of a specific guest. This functional diversity makes it possible to optimize the performance or enhance the capability of a specific monitor without increasing the complexity of the others.

3. *Fault Tolerance* – Since all the monitors operate at the same hardware privilege level, the compromise of a single monitor threatens the integrities of all the others. However, in the presence of inadvertent misbehaviors and hardware soft errors, a distributed monitor design with both duplication and functional diversity increases the reliability and availability of the overall system [Avi85] [Avi67] [Avi75] [PvSK90]. Because of their simplicity and functional diversity, it is also easier to formally verify the correctness and harder to exploit the potential security vulnerabilities of the Quest-V monitors.

Despite these benefits, the duplication of functionalities in the distributed monitor design inevitably increases the total memory footprint of the 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.

3.2 Resource Partitioning

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

3.2.1 CPU Partitioning

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

3.2.2 Memory Partitioning

Quest-V relies on hardware assisted virtualization support to perform memory partitioning. Figure 3.2 shows how address translation works for Quest-V sandboxes using Intel's extended page tables. Each sandbox kernel uses its own internal paging structures to translate guest virtual addresses to guest physical addresses. EPT structures are then walked by the hardware to complete the translation to host physical addresses.

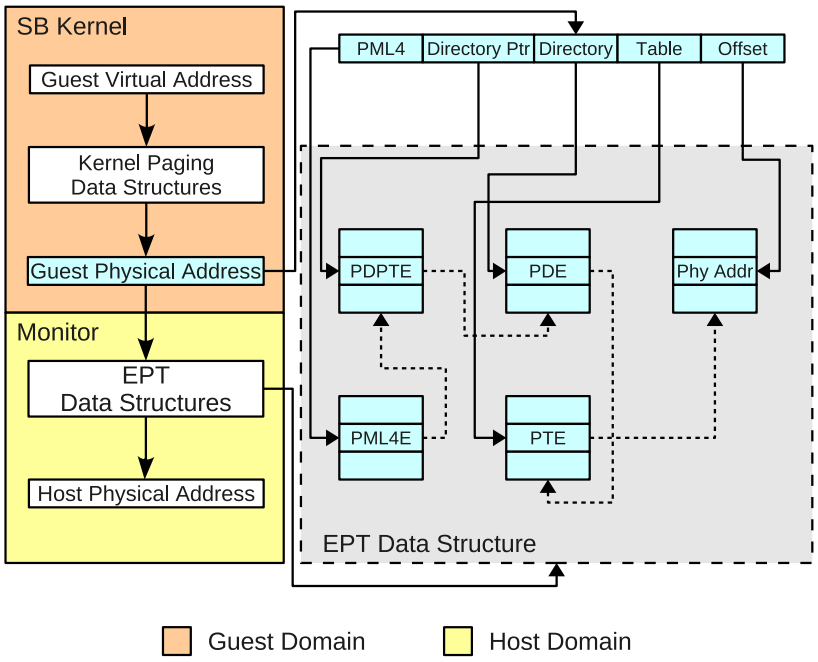

Figure 3.2: Extended Page Table Mapping

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

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

3.2.3 Cache Partitioning

Microarchitectural resources such as caches and memory buses provide a source of contention on multi-core platforms. While partitioning these resources is important to achieve true temporal and spatial isolation, it is out of the scope of this dissertation. Our group has been working on cache occupancy prediction and page coloring techniques to solve the microarchitectural resource partitioning problem. Quest-V uses hardware performance counters to establish cache occupancies for different sandboxes [WZWZ13, WZWZ10, WZW+08]. Dynamic page coloring techniques as described in our COLORIS [YWCL14] system are then able to partition shared caches between sandboxes [LHH97, Alb99, CS07, DSN06, Iye04, KCS04, LSK04, RLT06, RAJ00, SKI08, SRD04]. The implementation of COLORIS in Quest-V is currently being actively pursued.

Additional work is ongoing to account for contention on other micro-architectural resources, including on-chip buses and interconnects. Without hardware support, software techniques such as MemGuard [CPS+13] is being considered.

3.2.4 I/O Partitioning

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

To partition I/O devices, Quest-V first has to restrict access to device specific hardware registers. Device registers are usually either memory mapped or accessed through a special I/O address space (e.g. I/O ports). For the x86, both approaches are used. For memory mapped registers, EPTs are used to prevent their accesses from unauthorized sandboxes.

For port-addressed registers, special hardware support is necessary. On Intel processors with VT-x, all variants of `in` and `out` instructions can be configured to cause a monitor trap if access to a certain port address is attempted. As a result, an I/O bitmap can be used to partition the whole I/O address space amongst different sandboxes. Unauthorized access to a certain register can thus be ignored or trigger a fault recovery event.

On platforms featuring the PCI peripheral bus, any sandbox attempting access to a PCI device must use memory-mapped or port-based registers identified in a special PCI *configuration space* [PCI]. Most operating systems will enumerate all the devices in this configuration space using PCI *Bus*, *Device*, and *Function* numbers. Quest-V intercepts access to this configuration space, which is accessed via both an address (`0xCF8`) and data (`0xCFC`) I/O port on the x86. 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. The device blacklist can be configured statically by user before system initialization.

A simplified control flow of the handling of PCI configuration space protection in a Quest-V monitor is given in Figure 3.3. Notice that simply allowing access to a PCI data port when access to a legitimate device is detected is not sufficient because we only want to allow the single I/O instruction that caused the monitor trap, and which passed the monitor check, to be correctly executed. Once this is done, the monitor should immediately restrict access to the PCI data port again. This behavior is achieved by setting the *trap flag* (TF) bit in the sandbox kernel system flags to cause a single step debug exception after it executes the next instruction. By configuring the processor to generate a monitor trap on debug exception, the system can immediately return to the monitor after executing the I/O

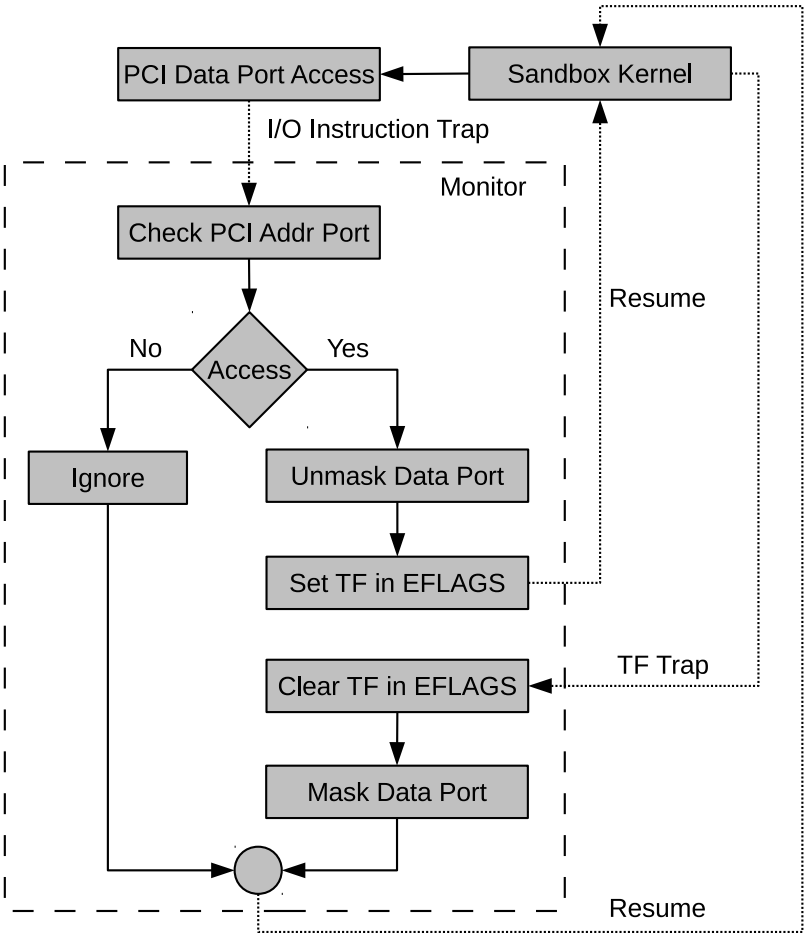

Figure 3.3: PCI Configuration Space Protection

instruction. After this, the monitor is able to mask the PCI data port again for the sandbox kernel, thereby mediating future device access.

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

I/O redirection table corresponds to a certain interrupt request from an I/O device.

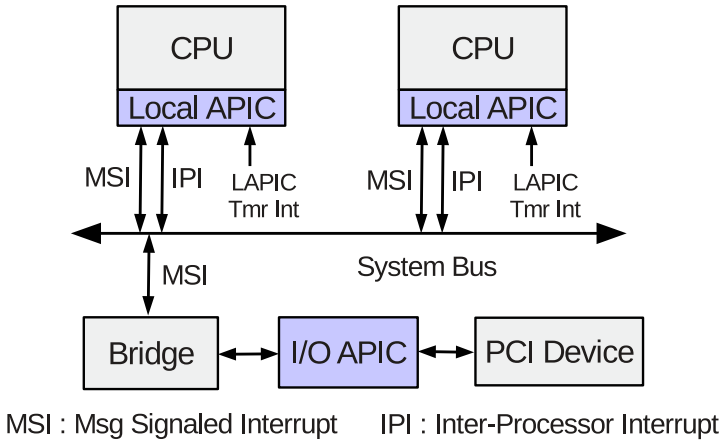

Figure 3.4: APIC Configuration

Figure 3.4 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 3.3 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 entry before allowing any changes to be made. Consequently, device interrupts are safely partitioned amongst sandboxes. It is worth mentioning that this technique is feasible for IRQ partitioning in Quest-V because processor cores are statically allocated to guest sandboxes. In traditional hypervisors with virtual machine scheduling, I/O interrupts are always intercepted by the hypervisor before being injected back into the guests since the virtual machine and processor core bindings are dynamic.

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.

Chapter 4

Quest Sandbox Support

Quest is a kernel our group developed for real-time and embedded systems. It currently operates on 32-bit x86 architectures and leverages hardware MMU support to provide page-based memory protection to processes and threads. As with UNIX-like systems, segmentation is used to separate the kernel from user-space. Quest is an SMP system, operating on multicore platforms. It has support for kernel threads, and a network protocol stack based on "lightweight IP" (lwIP) [LWI]. The kernel code has been implemented from scratch and is approximately 10,000 lines of C and assembly, discounting drivers and network stack.

In Quest-V, Quest serves as the default kernel for each sandbox during initialization. It is also designed for mission critical and safety critical tasks in a mixed criticality application. Additionally, each monitor in Quest-V is given access to a Quest kernel address space so that direct manipulation of kernel objects during monitor traps is possible. Since Quest-V currently only supports single processor sandboxes, Quest kernel will always operate in uni-processor mode when running in a Quest-V sandbox.

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 [AA06, BDF+03], where

a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [1] 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 [BDM99] for scheduling and accounting decisions on behalf of software threads. It serves no other purpose to virtualize the underlying physical CPUs, since our sandbox kernels and their applications execute directly on the hardware. In particular, a VCPU does not need to act as a container for cached instruction blocks that have been generated to emulate the effects of guest code, as in some trap-and-emulate virtualized systems.

In common with bandwidth preserving servers [AB98, DLS97, SB96], each VCPU, V, has a maximum compute time budget, C_V, available in a time period, T_V. V is constrained to use no more than the fraction $U_V = \frac{C_V}{T_V}$ of a physical processor (PCPU) in any window of real-time, T_V, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below 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 4.1: (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 [ZW06], which may be deferred execution when threads associated with higher

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

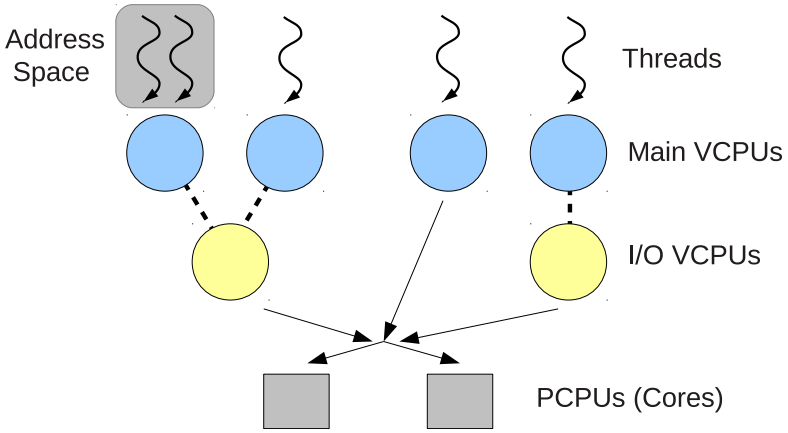

Figure 4.1: VCPU Scheduling Hierarchy

priority VCPUs having available budgets are runnable. The flexibility of Quest 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.

Even though the Quest kernel and its VCPU scheduling framework are not part of the contribution of this dissertation, they are the foundations for the predictable communication and service migration framework which will be described in detail later in this chapter.

4.1 Main and I/O VCPU Scheduling

By default, VCPUs act like Sporadic Servers [SSL89]. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [LL73]. 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 therefore be applied, to ensure each VCPU is guaranteed its share of CPU time, U_V, in finite windows of

real-time.

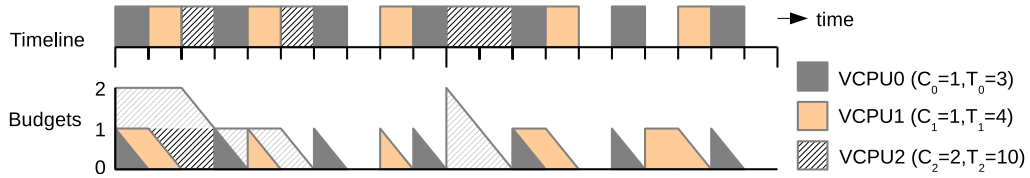

Figure 4.2: Example VCPU Schedule

An example schedule is provided in Figure 4.2 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.

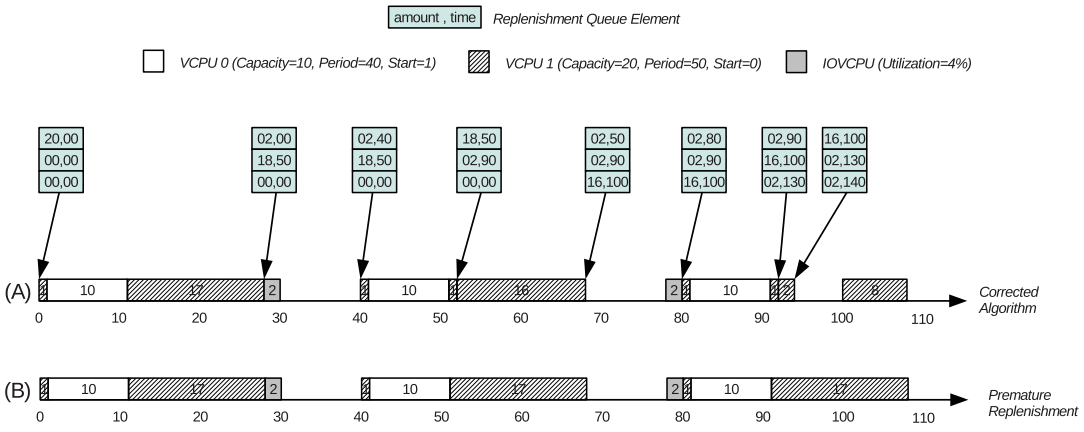

Figure 4.3: Sporadic Server Replenishment List Management

In Quest there is no notion of a periodic timer interrupt for updating 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 [SBWH10] to correct for early replenishment and budget amplification in the POSIX specification.

Figure 4.3 shows an example schedule for two Main VCPUs and one I/O VCPU for a certain device such as a gigabit Ethernet card. In this example, Schedule (A) avoids premature replenishments, while Schedule (B) is implemented according to the POSIX specification. In (B), VCPU1 is scheduled at $t = 0$, only to be preempted by higher priority VCPU0 at $t = 1, 41, 81$, etc. By $t = 28$, VCPU1 has amassed a total of 18 units of execution time and then blocks until $t = 40$. Similarly, VCPU1 blocks in the interval $[t = 68, 80]$. By $t = 68$, Schedule (B) combines the service time chunks for VCPU1 in the intervals $[t = 0, 28]$ and $[t = 40, 68]$ to post future replenishments of 18 units at $t = 50$ and $t = 90$, respectively. This means that over the first 100 time units, VCPU1 actually receives 46 time units, when it should be limited to 40%. Schedule (A) ensures that over the same 100 time units, VCPU1 is limited to the correct amount. 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. Hence, although VCPU1 actually receives 21 time units in the interval $[t = 50, 100]$ it never exceeds more than its 40% share of CPU time between blocking periods and over the first 100 time units it meets its bandwidth limit.

For completeness, Schedule (A) shows the list of replenishments and how they are updated at specific times, according to scheduling events in Quest-V. The invariant is that the sum of replenishment amounts for all list items must 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 $t + R$ can occur before $t + T_V$.

When VCPU1 first blocks at $t = 28$ it still has 2 units of budget remaining, with a further 18 due for replenishment at $t = 50$. At this point, the schedule shows the execution of the I/O VCPU for 2 time units. In Quest-V, threads running on Main VCPUs block (causing the VCPU to block if there are no more runnable threads), while waiting for

I/O requests to complete. All I/O operations in response to device interrupts are handled as threads on specific I/O VCPUs. Each I/O VCPU supports threaded interrupt handling at a priority inherited from the Main VCPU associated with the blocked thread. In this example, the I/O VCPU runs at the priority of VCPU1. The I/O VCPU's budget capacity is calculated as the product of it bandwidth specification (here, $U_{IO} = 4\%$) and the period, T_V, of the corresponding Main VCPU for which it is performing service. Hence, the I/O VCPU receives a budget of $U_{IO} \cdot T_V = 2$ time units, and through bandwidth preservation, will be eligible to execute again at $t_e = t + C_{actual}/U_{IO}$, where t is the start time of the I/O VCPU and $C_{actual} \mid 0 \leq C_{actual} \leq U_{IO} \cdot T_V$ is how much of its budget capacity it really used.

In Schedule (A), VCPU1 resumes execution after unblocking at times, $t = 40$ and 80. In the first case, the I/O VCPU has already completed the I/O request for VCPU1 but some other delay, such as accessing a shared resource guarded by a semaphore (not shown) could be the cause of the added delay. Time $t = 78$ marks the next eligible time for the I/O VCPU after it services the blocked VCPU1, which can then immediately resume. Further details about VCPU scheduling in Quest-V can be found in our accompanying paper for Quest [DLW11], a non-virtualized version of the system that does not support sandboxed service isolation.

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 can ultimately be migrated to remote sandboxes along with their VCPUs, according to load constraints or affinity settings of the target VCPU [LWCM14]. 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 [BDF+03] that manage multiple

separate guest OSes.

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 sandbox can determine 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. U_j is the utilization factor of I/O VCPU, V_j [DLW11].

4.2 Predictable Communication

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). 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. Briefly speaking, channel establishment requires the use of a unique key by each communicating endpoint, similar to the POSIX shared memory utilities (i.e. *shmget()*).

A *mailbox* data structure is set up within shared memory by each end of a commu-

nication channel. By default, Quest-V currently supports asynchronous communication by polling a status bit in each relevant mailbox to determine message arrival. Real-time communication is only available between sandboxes running Quest services, and featuring VCPU scheduling as described above. 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.

The lack of both a global clock and global scheduler for all sandboxes creates challenges for a system requiring strict timing guarantees. In the rest of this chapter, we elaborate on two such challenges, relating to predictable communication and address space migration.

For the purposes of predictable communication, we consider a system model as follows:

- A communication channel between a pair of endpoints in separate sandboxes is *half duplex* and has a *single slot*. A single slot has a configurable capacity, B, but is 4KB by default.

- One endpoint acting as a *sender* places up to one full slot of data in the channel when it detects the channel is *empty*.

- A second endpoint acting as a *receiver* consumes one slot of data from the channel when it is *full*.

- A transaction on a channel comprises the exchange of one or more slots of data. A sender sets a *start* flag to initiate a transaction. When the final unit of data is submitted to the channel, the sender sets an *end* of transaction flag.

- Each endpoint executes a thread mapped to a communication VCPU. The sender VCPU, V_s has parameters C_s and T_s, for its budget and period, respectively. Similarly, the receiver VCPU, V_r has parameters C_r and T_r. Both endpoints poll for the arrival of data when not sending, unless a special *out-of-band* signal is used, such as an interprocessor interrupt (IPI).

Consider a sending thread, τ_s, associated with a VCPU, V_s, which wishes to communicate with a receiving thread, τ_r, bound to V_r in a remote sandbox. Suppose τ_s sends a message of N bytes at a cost of δ_s time units per byte. Similarly, suppose τ_r replies with an M byte message at a cost of δ_r time units per byte. Before replying, let τ_r consume K units of processing time to service the communication request. The worst-case round-trip communication delay, Δ_{WC}, between τ_s and τ_r can now be calculated.

Case 1: All messages fit in one channel slot. In this case $N, M \leq B$. To calculate Δ_{WC}, we need to account for the time to send a request, process it, and wait for the reply. Let $S(N)$ be the total time taken by τ_s to send a request message of size N. That is:

$$S(N) = \lfloor \frac{N \cdot \delta_s}{C_s} \rfloor \cdot T_s + (N \cdot \delta_s) \bmod C_s$$

This accounts for multiple periods of V_s to send N bytes. At the receiver, we calculate the time, $R(N, M)$, to consume a request of size N, process the request and send a reply of size M as:

$$R(N,M) = \lfloor \frac{[N+M]\cdot\delta_r + K}{C_r} \rfloor \cdot T_r + ([N+M]\cdot\delta_r + K) \bmod C_r \tag{4.1}$$

The last stage of a communication transaction is consuming a response at the sender, which takes $S(M)$ time units.

Finally, we need to factor the shifts in time between when V_s and V_r are scheduled in their respective sandboxes. In the worst-case, a message is about to be sent when V_s uses up its current budget. This causes a delay of $T_s - C_s$ time units until its budget is replenished. The same situation might happen when V_s tries to consume the response. Similarly, a message arrives at the receiver when V_r has completed its current budget, so it will not be processed for another $T_r - C_r$ time. Consequently, the worst-case round-trip communication delay is:

$$\Delta_{WC}(N,M) = S(N) + (T_s - C_s) + R(N,M) + (T_r - C_r) + S(M) + (T_s - C_s) \tag{4.2}$$

Case 2: Messages take multiple slots. In this case, $N > B$ and M is arbitrary. For cases where the request messages take more than one slot, we need consider Equation 4.2 each time a request-response channel slot is used. Hence, the multi-slot worst-case response time, Δ'_{WC}, becomes:

$$\Delta'_{WC} = \lceil \frac{N}{B} \rceil \cdot \Delta_{WC}(B, min(M,B)) \tag{4.3}$$

For the special case where communication is only one-way (e.g., to migrate an address

space) of size N, Δ'_{WC} reduces to:

$$\Delta'_{WC} = \lceil \frac{N}{B} \rceil \cdot (S(B) + (T_s - C_s) + R(B, 0) + (T_r - C_r)) \tag{4.4}$$

4.3 Predictable Migration

Quest-V supports the migration of VCPUs and associated address spaces between Quest sandboxes for several reasons: (1) to balance loads across sandboxes, (2) to guarantee the schedulability of VCPUs and threads, and (3) for closer proximity to needed resources such as I/O devices that would otherwise have to be accessed by remote procedure calls.

Migration is initiated using the *vcpu_migration* interface shown in Listing 4.1. The *flag* is either 0, `MIG_STRICT` or `MIG_RELAX`. A *time* in milliseconds is used to specify either a deadline or timeout, depending on *flag*. The *dest* argument specifies the sandbox ID of a specific destination, or `DEST_ANY` if the caller wishes to allow the sandbox kernel to pick an acceptable destination.

The migration function is non-blocking. It returns `TRUE` only if the migration request is accepted, and the caller can resume its normal operation. The actual migration will happen at a later time decided by the local sandbox kernel. The calling thread can use a flag in its task structure, or retrieve its current sandbox ID, to check whether the migration succeeded or failed.

Listing 4.1: Predictable Migration User Interface

```
bool vcpu_migration(uint32_t time, int dest, int flag);
```

When *flag* is set to `MIG_STRICT`, the calling thread and its VCPU will be migrated to the destination with the restriction that the migrating VCPU's utilization cannot be

affected. The local sandbox kernel must find a suitable time to perform migration to make this guarantee. An optional timeout specified using the *time* argument can be used to avoid indefinite delays before migration can occur. A *time* of 0 disables the timeout deadline.

When *flag* is set to 0, *time* is used to specify a migration deadline. A sandbox kernel will try to migrate the calling thread and its VCPU to the specified destination within the deadline. Unlike the case with `MIG_STRICT` flag, the calling thread's VCPU utilization can potentially be affected during migration. The worst-case down time for the migrating VCPU would be from the time of the request to the specified deadline.

Finally, when *flag* is set to `MIG_RELAX`, the calling thread and its VCPU will be migrated to the destination no matter how long it takes. As with 0 flag, calling *vcpu_migration* with `MIG_RELAX` will potentially affect the migrating VCPU's utilization. However, the VCPU down time is not bounded as with 0 flag.

Notice that the use of different flags in *vcpu_migration* only affects the behavior of the migrating thread and its VCPU. All the other VCPUs running in both the source and the destination sandbox should not be affected. Additionally, if blocking is necessary for *vcpu_migration* with 0 or `MIG_RELAX` flag, a *vcpu_migration_blocking* function can easily be implemented based on the non-blocking *vcpu_migration* interface.

The pseudo-code for *vcpu_migration()* and its integration into the local scheduler are shown in the appendix, in Listings 4.2 and 4.3.

Listing 4.2: vcpu_migration Pseudo Code

```
bool vcpu_migration (uint32_t time, int dest, int flag) {
  if(!valid (dest) || !(valid (flag)))
    return FALSE;

  if(flag == MIG_STRICT) {
    if(time)
```

```
      cur_task.mig_timeout = now + time;
    else
      cur_task.mig_timeout = 0;
  } else if(flag == MIG_RELAX) {
    cur_task.mig_dl = MAX_DEADLINE;
  } else {
    if(time)
      cur_task.mig_dl = now + time;
    else
      return FALSE;
  }

  cur_task.mig_status = 0;
  cur_task.mig_flag = flag;
  cur_task.affinity = dest;

  return TRUE;
}
```

Listing 4.3: Scheduler Pseudo Code

```
void schedule (void) {
  ...
  /* Check migration request when de-scheduled */
  if(next_task != cur_task) {
    if(cur_task.affinity != cur_sandbox) {
      if(cur_task.affinity == DEST_ANY)
        cur_task.affinity =
          find_destination();

      /* Lock both source and destination */
      if(try_lock(cur_sandbox, cur_task.affinity)) {
        if(!check_utilization_bound()) {
```

```
            cur_task.mig_flag = FAIL;
            cur_task.affinity = cur_sandbox;
            goto release;
          }

          /* MIG_RELAX, migrate right now */
          if(cur_task.mig_flag == MIG_RELAX) {
            cur_task.mig_status = SUCCESS;
            do_migration(cur_task);
            goto release;
          }

          /* Calculate migration cost */
          WCET = calculate_wcet(cur_task);

          if(cur_task.mig_flag == MIG_STRICT) {
            /* next_event () returns Es */
            cur_task.mig_dl = next_event()+now;
          }

          if((now+WCET) > cur_task.mig_dl) {
            cur_task.mig_status = FAIL;
            cur_task.affinity = cur_sandbox;
          } else {
            cur_task.mig_status = SUCCESS;
            do_migration(cur_task);
          }
  release:
          unlock(cur_sandbox, cur_task.affinity);
        } else {
          /* Destination is busy or we are migrating another task */
```

```
          if(flag == MIG_STRICT) {
            if((now+next_event()) >= cur_task.mig_timeout) {
              cur_task.mig_status = FAIL;
              cur_task.affinity = cur_sandbox;
            }
          } else {
            if((now+next_event()) >= cur_task.mig_dl) {
              cur_task.mig_status = FAIL;
              cur_task.affinity = cur_sandbox;
            }
          }
        }
      }
    }
  resume_schedule:
    ...
  }
```

The major challenges in the implementation of this interface are: (1) accurately accounting for migration overheads, and (2) accurately estimating a the worst-case migration cost under all circumstances.

4.3.1 Predictable Migration Strategy

Threads in Quest sandbox have corresponding address spaces and VCPUs. The current design limits one, possibly multi-threaded, address space to be associated with a single VCPU. This restriction avoids the problem of migrating VCPUs and multiple address spaces between sandboxes, which could lead to arbitrary delays in copying memory. Migration from one sandbox's private memory requires a copy of an address space and all thread data structures to the destination. Each thread is associated with a `quest_tss`

structure that stores the execution context and VCPU state.

Dedicated migration threads and corresponding VCPUs are established within each sandbox at system initialization. A migration thread is responsible for the actual VCPU migration operation. An inter-processor interrupt (IPI) is used by the local sandbox kernel to notify a remote migration thread of a migration request. In our current implementation, only one migration thread and VCPU can be configured for each sandbox. If multiple migration requests occur at the same time, they will be processed serially.

Migration using Message Passing. This approach transfers a thread's state, including its address space and VCPU information, using a series of messages that are passed over a communication channel. The advantage of this approach is that it generalizes across different communication links, including those where shared memory is not available (e.g., Ethernet).

To initiate migration, an IPI is first sent to the migration thread in the destination sandbox. The destination then waits for data on a specific channel. Since the default communication channel size is 4KB, a stream of messages are needed to migrate an address space, along with its thread and VCPU state. This resembles the communication scenario described in *Case 2* of Section 4.2. The destination re-assembles the address space and state information before adding the migrated VCPU to its scheduler queue. An IPI or acknowledgement message from the destination to the source is now needed to signal the completion of migration. If successful, the migration thread in the source sandbox will be able to reclaim the memory of the migrated address space. Otherwise, the migrating VCPU will be put back into the run queue in the source sandbox.

Before a VCPU is migrated, admission control is performed at the destination. This is used to verify the schedulability of the migrating VCPU and all existing VCPUs in the

destination. If admission control fails, a migration request is rejected.

At boot time, Quest-V establishes base costs for copying memory pages without caches enabled [2]. These costs are used to determine various parameters used for worst-case execution time estimation.

An estimate of the worst-case migration cost requires: (1) the cost of serializing the migrated state into a sequence of messages (Δ_s), (2) the communication delay to send the messages (Δ_t), and (3) the cost of re-assembling the transferred state at the destination (Δ_a). We assume one migration thread is associated with a sender VCPU, V_s, and another is associated with a receiver VCPU, V_r.

$$\Delta_s = \lfloor \frac{\delta_s}{C_s} \rfloor \cdot T_s + \delta_s \bmod C_s + T_s - C_s \tag{4.5}$$

Here, δ_s is the execution time of a migration thread to produce a sequence of messages, assuming caches are disabled. Similarly, given δ_a, is the execution time to re-assemble a VCPU and address space:

$$\Delta_a = \lfloor \frac{\delta_a}{C_r} \rfloor \cdot T_r + \delta_a \bmod C_r + T_r - C_r \tag{4.6}$$

In this case, Δ_t is identical to Δ'_{WC} in Equation 4.3. Hence, the the worst-case migration cost when message passing is used is:

$$\Delta_{mig} = \Delta_s + \Delta'_{WC} + \Delta_a \tag{4.7}$$

Migration with Direct Memory Copy. As shown in Equation 4.3, the worst-case time

[2] We do not consider memory bus contention issues, which could make worst-case estimations even larger.

to transfer a large amount of state between two sandboxes can span numerous migration VCPU periods. This makes it difficult to satisfy a VCPU migration request using message passing, with the `MIG_STRICT` flag set. Fortunately, for Quest sandboxes that communicate via shared memory, it is possible to dramatically reduce the migration overhead. Quest-V monitors can be involved in the migration process to directly copy the target address space from source to destination sandbox. However, while the monitor involvement dramatically increases the efficiency of the migration process, it also potentially reduces the safety and security of the system. Although allowing the monitor of one sandbox to directly access the memory region of another sandbox violates the principle of the separation kernel design, it is worth noting that thread migration in a mixed criticality application should only occur between sandboxes with the same criticality level [3]. This limits the safety and security impact of migration with direct memory copy. We decided to provide direct memory copy migration capability in Quest-V and allow the user to decide whether the trade-off is worth making.

Figure 4.4 shows the general migration strategy when direct memory copy is used. An IPI is first sent to the destination sandbox, to initiate migration. The migration thread handles the IPI in the destination, generating a trap into its monitor that has access to machine physical memory of all sandboxes. The migrating address space in the source sandbox is temporarily mapped into the destination. The address space and associated `quest_tss` thread structures are then copied to the target sandbox's memory. At this point, the page mappings in the source sandbox can be removed by the destination monitor.

Similar to the message passing approach, an IPI from the destination to the source sandbox is needed to signal the completion of migration. All IPIs are handled in the sandbox kernels, with interrupts disabled while in monitor mode. The migration thread in

[3] Migration could also happen between group of sandboxes with the appropriate capabilities and access rights. In this work, it is tied in with criticality levels.

the destination can now exit its monitor and return to the sandbox kernel. The migrated address space is attached to its VCPU and added to the local schedule. At this point, the migration threads in source and destination sandboxes are able to yield execution to other VCPUs and, hence, threads.

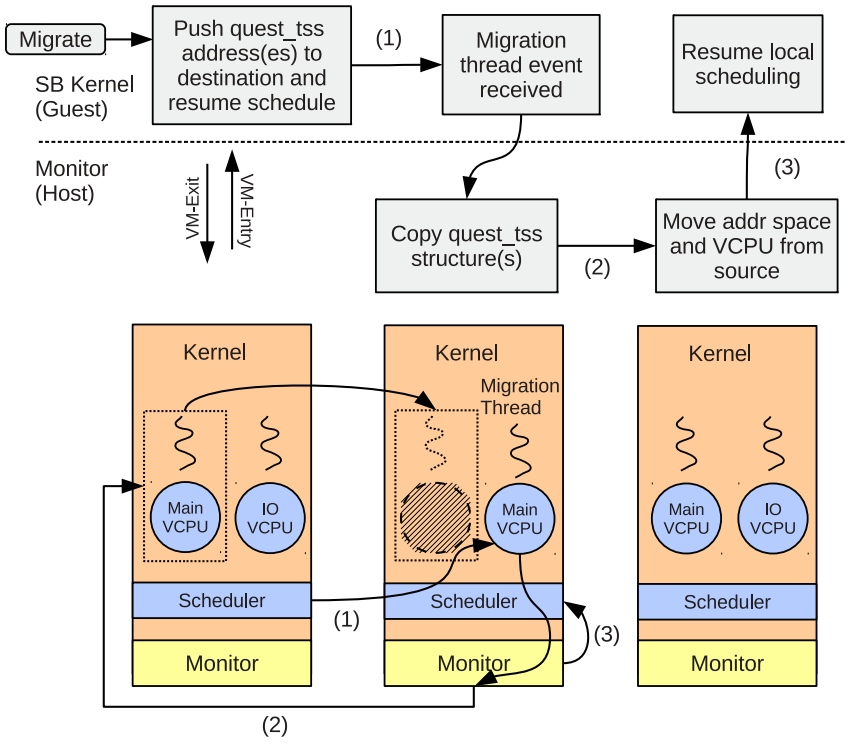

Figure 4.4: Migration Strategy

With direct memory copy, the worst-case migration cost can simply be defined as:

$$\Delta_{mig} = \lfloor \frac{\delta_m}{C_r} \rfloor \cdot T_r + \delta_m \bmod C_r + T_r - C_r \tag{4.8}$$

Here, C_r and T_r are the budget and period of the migration thread's VCPU in destination sandbox, and δ_m is the execution time to copy an address space and its $quest\_tss$ data structures to the destination.

Migration Thread Preemption. The migration thread in each sandbox is bound to a VCPU. If the VCPU depletes its budget or a higher priority VCPU is ready to run, the migration thread should be preempted. However, if direct memory copy is used, migration thread preemption is complicated by the fact that the thread spends most of its time inside the sandbox monitor, and each sandbox scheduler runs within the local kernel (outside the monitor).

Migration thread preemption, in this case, requires a domain switch between a sandbox monitor and its kernel, to access the local scheduler. This results in costly VM-Exit and VM-Entry operations that flush the TLB of the processor core. To avoid this cost, we limited migration thread preemption to specific preemption points. Additionally, we associated each migration thread with a highest priority VCPU, ensuring it would run until either migration was completed or the VCPU budget expired. Bookkeeping is limited to tracking budget usage at each preemption point. Thus, within one period, a migration thread needs only one call into its local monitor.

Preemption points are currently located: (1) immediately after copying each `quest_tss` structure, (2) between processing each Page Directory Entry during address space cloning, and (3) right before binding the migrated address space to its VCPU, for re-scheduling. In the case of a budget overrun, the next budget replenishment is adjusted according to the corrected POSIX Sporadic Server algorithm [SBWH10]. Figure 4.5 describes the migration control flow.

Clock Synchronization. One extra challenge to be considered during migration is clock synchronization between different sandboxes in Quest-V. Quest-V schedulers use Local APIC Timers and Time Stamp Counters (TSCs) in each core as the source for all time-

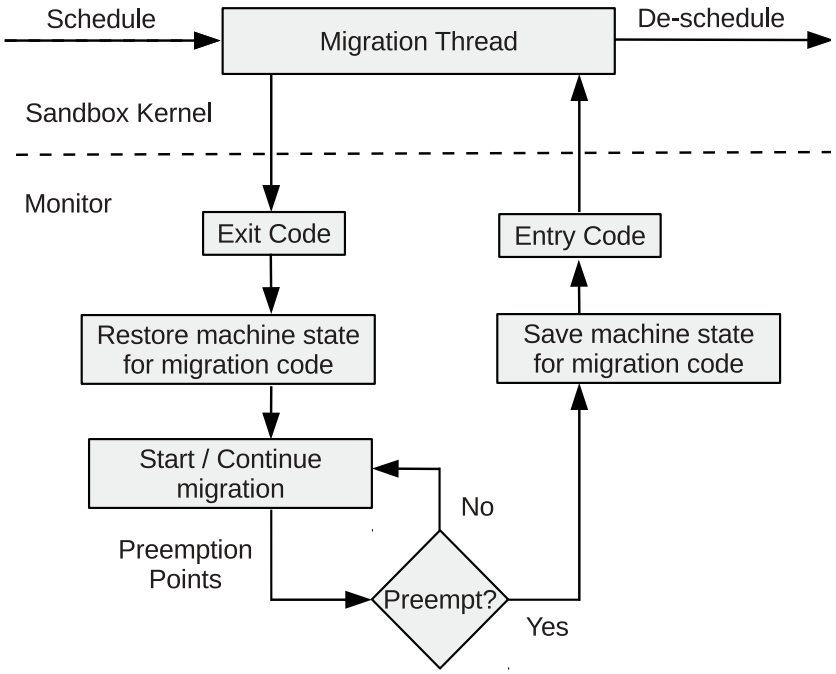

Figure 4.5: Migration Framework Control Flow

related activities in the system, and these are not guaranteed to be synchronized by hardware. Consequently, Quest-V adjusts time for each migrating address space to compensate for clock skew. This is necessary when updating budget replenishment and wakeup time events for a migrating VCPU that is sleeping on an I/O request, or which is not yet runnable.

The source sandbox places its current TSC value in shared memory immediately before sending a IPI migration request. This value is compared with the destination TSC when the IPI is received. A time-adjustment, δ_{ADJ}, for the migrating VCPU is calculated as follows:

$$\delta_{ADJ} = TSC_d - TSC_s - 2 * RDTSC_{cost} - IPI_{cost} \tag{4.9}$$

TSC_d and TSC_s are the destination and source TSCs, while $RDTSC_{cost}$ and IPI_{cost} are the average costs of reading a TSC and sending an IPI, respectively. δ_{ADJ} is then added to all future budget replenishment and wakeup time events for the migrating VCPU in the destination sandbox. In equation 4.9, IPI_{cost} is assumed to be predictable and relies on IPI delivery having a bounded worst case delay. For architectures with shared IPI bus, the worst case should be measured under load or by considering the bus arbitration protocol.

4.3.2 Migration Criteria

Quest-V restricts migrate-able address spaces to those associated with VCPUs that either: (1) have currently expired budgets, or (2) are waiting in a sleep queue. In the former case, the VCPU is not runnable at its foreground priority until its next budget replenishment. In the latter case, a VCPU is blocked until a wakeup event occurs (e.g., due to an I/O request completion or a resource becoming available). Together, these two cases prevent migrating a VCPU when it is runnable, as the migration delay could impact the VCPU's utilization.

For VCPU, V_m, associated with a migrating address space, we define E_m to be the *relative time* [4] of the next event, which is either a replenishment or wakeup.

If V_m issues a migration request with `MIG_STRICT` flag, for the utilization of V_m to be unaffected by migration, the following must hold:

$$E_m \geq \Delta_{mig} \tag{4.10}$$

Where Δ_{mig} can be calculated by either Equation 4.7 or 4.8. Quest-V makes sure that the migrating thread will not be woken up by asynchronous events until the migration is finished. The system imposes the restriction that threads waiting on I/O events cannot be migrated. Similarly, the migration deadline can be compared with Δ_{mig} to make migration

[4] i.e., Relative to current time.

decisions when `flag`=0.

4.4 Fault Recovery

The temporal and spatial isolation between Quest-V sandboxes contains software faults inside a single sandbox. Fault detection is needed before a sandbox can be recovered from faults. Fault detection is a topic worthy of separate discussion and not in the scope of this dissertation. Currently, 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 investigated in the future development.

Once the fault detection event has triggered a trap into a monitor, the fault recovery procedure is initiated. 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.

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 perhaps more rigorously tested, may be used in recovery. Such component replacements can lead to system robustness through

functional or implementation diversity [WHD+09]. 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 – The design of Quest-V also allows 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 off-line while it is being upgraded or repaired.

In Quest-V, remote fault recovery involves the local monitor identifying a target sandbox. The local monitor then informs the target sandbox via an IPI to pass control to the remote monitor, which performs the fault recovery. Ideally, information needs to be exchanged between monitors about the actions necessary for fault recovery and what threads, if any, need to be migrated. However, 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. When device driver is involved, we also redirect the device interrupt to the target sandbox and update the device blacklist if necessary. Currently, the target sandbox is chosen randomly. Software components to be recovered are also configured statically for each recovery routine.

Even though the sandbox isolation provided by the Quest-V separation kernel offers fault isolation and serves as a platform to construct various fault recovery mechanisms, current Quest-V implementation lacks the policies and mechanisms to support generic fault recovery routines. Existing fault recovery implementations in Quest-V are mostly application specific. A flexible fault detection and faulting component identification interface for applications and drivers is not available. In the future development of Quest-V, we will consider approaches that generalize the fault recovery process for different applications and device drivers in order to provide convenient APIs for developers.

4.5 Configurable Device Sharing

Even though any form of implicit sharing of hardware resources violates the principle of isolation in a separation kernel, we decided to expose the capability of sharing an I/O device between multiple Quest sandboxes under user discretion. Two or more Quest sandboxes with the same criticality level might wish to access the same device managed by a common driver. The driver need not be replicated in separate sandboxes but can instead be mapped to shared memory. This avoids the need for a specific sandbox to operate as a special device manager on behalf of other sandboxes. While it has the potential to improve efficiency it means that a failed shared driver affects multiple sandboxes. If a sandbox wishes to have exclusive access to a device it can still choose to do so using a private driver instead.

Quest-V uses the I/O APIC found on modern x86 platforms to multicast hardware interrupts to *all sandboxes sharing a corresponding device*. We expect the number of sandboxes sharing a device to be relatively low 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 4.6 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.

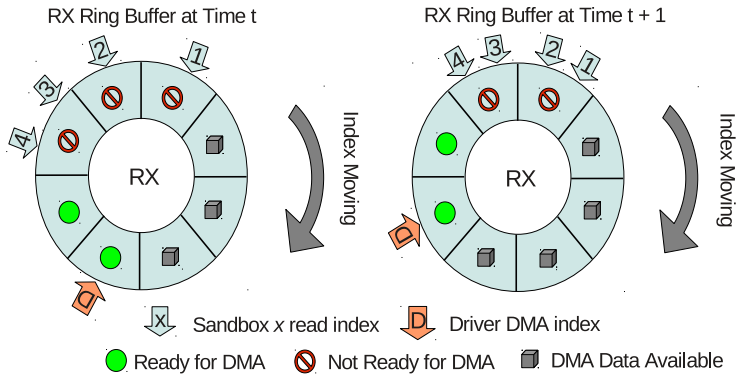

Figure 4.6: Example NIC RX Ring Buffer Sharing

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 [SRI] are now emerging, although not commonplace in embedded systems. Without hardware sup-

port, 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 [GAH+12], they do not simplify device sharing.

With device sharing, some of the I/O related communication and device proximity based thread migration between Quest sandboxes can be avoided in Quest-V. It is essentially a form of communication based on implicit shared memory and I/O device states. However, as mentioned earlier, device sharing allows fault in the device driver to affect all the sandboxes sharing the device. Consequently, the use of device sharing is not recommended for mixed criticality applications in general. In some special cases, it might be acceptible for low criticality Quest sandboxes with the same criticality levels to share certain devices for efficiency and resource utilization. Moreover, with the extra synchronization required in the driver, device sharing in Quest-V will cause scalability issues. However, since Quest sandboxes sharing a same device still have direct access to hardware registers and can handle physical interrupts, shared devices in Quest-V are more efficient than virtual devices in traditional hypervisors.

4.6 Experimental Evaluation

A series of experiments have been conducted to evaluate the effectiveness of predictable inter-sandbox communication and service migration frameworks and the performance of the device sharing mechanism in Quest-V. Examples of fault isolation and recovery are also presented. All the experiments in this section are conducted on a Gigabyte Mini-ITX machine with an Intel Core i5-2500K 3.3GHz 4-core processor, 8GB RAM and a Realtek 8111e NIC.

4.6.1 Predictable Communication

We first ran 5 different experiments to predict the worst-case round-trip communication time using Equation 4.2. The VCPU settings of the sender and receiver, spanning two different sandboxes, are shown in Table 4.1.

Case #	Sender VCPU	Receiver VCPU
Case 1	20/100	2/10
Case 2	20/100	20/100
Case 3	20/100	20/130
Case 4	20/100	20/200
Case 5	20/100	20/230

Table 4.1: VCPU Parameters

We calculated the values of δ_s and δ_r by setting the message size to 4KB for both sender and receiver (i.e. $M = N = 4$KB) and *disabling caching* of the shared memory communication channel on the test platform. The message processing time K has essentially been ignored because the receiver immediately sends the response after receiving the message from the sender.

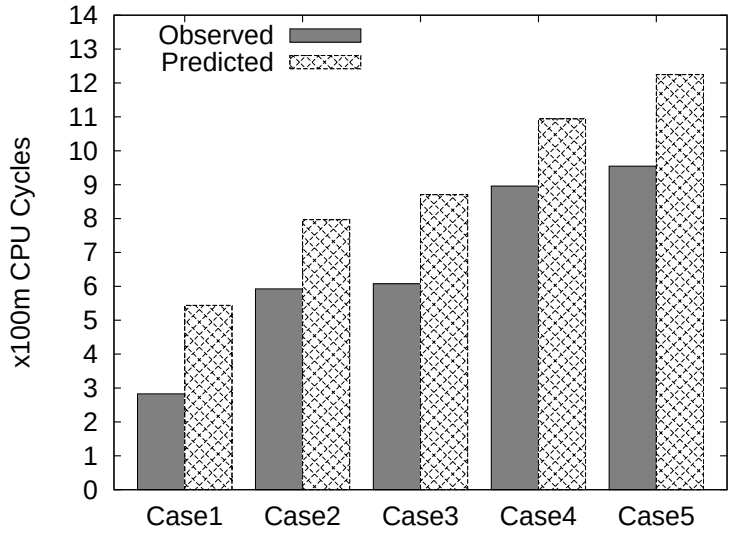

Figure 4.7: Worst-Case Round-trip Communication

Both sender and receiver threads running on VCPUs V_s and V_r, respectively, sleep for controlled time units, to influence phase shifts between their periods T_s and T_r. Similarly, the sender thread adds busy-wait delays before transmission, to affect the starting point of communication within its VCPU's available budget, C_s. Figure 4.7 shows results after 10000 message exchanges are performed for each of the 5 experiments. As can be seen, the observed value is always within the prediction bounds derived from Equation 4.2.

We next conducted a series of one-way communication experiments to send 4MB messages through a 4KB channel with different VCPU parameters as shown in Table 4.2. Figure 4.8 again shows that the observed communication times are within the bounds derived from our worst-case estimations. However, the bounds are not as tight as for round-trip communication. We believe this is due to the fact that we used a pessimistic worst-case estimation, which includes leftover VCPU budgets in each instance of the multi-slot communication. Estimation error is reduced when the difference between VCPU budgets and periods is smaller.

Case #	**Sender VCPU**	**Receiver VCPU**
Case 1	20/50	20/50
Case 2	10/100	10/100
Case 3	10/100	10/50
Case 4	10/100	10/200
Case 5	5/100	5/130
Case 6	10/200	10/200

Table 4.2: VCPU Parameters

4.6.2 Predictable Migration

To verify the predictability of the Quest-V migration framework, we constructed a task group consisting of 2 communicating threads and another CPU-intensive thread running a Canny edge detection algorithm on a stream of video frames. The frames were gathered

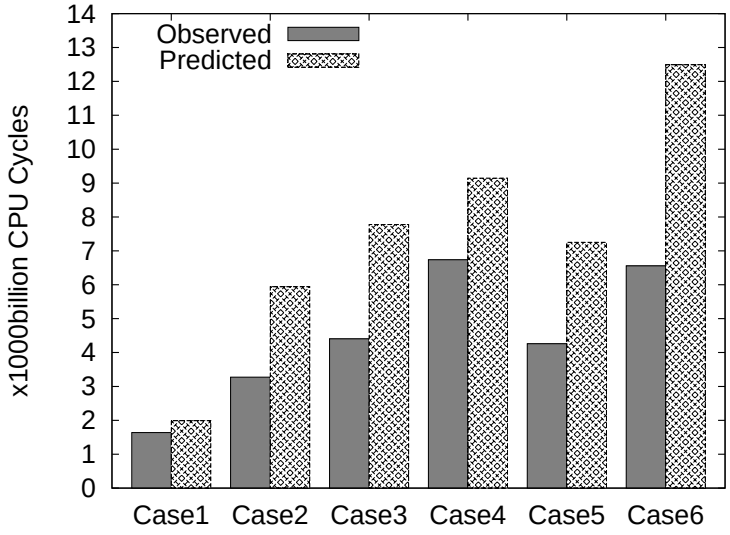

Figure 4.8: Worst-Case One-way Multi-slot Communication

from a LogiTech QuickCam Pro9000 camera mounted on our RacerX mobile robot, which traversed one lap of Boston University's indoor running track at Agganis Arena [5]. To avoid variable bit rate frames affecting the results of our experiments, we applied Canny repeatedly to the frame shown in Figure 4.9 rather than a live stream of the track. This way, we could determine the effects of migration on a Canny thread by observing changes in processing rate while the other threads communicated with one another.

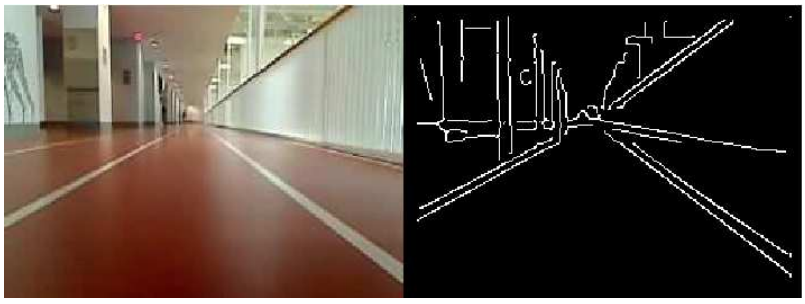

Figure 4.9: Track Image Processed by Canny

For all the experiments in this section, unless stated otherwise, we have two active

[5]RacerX is a real-time robot that runs Quest-V.

sandbox kernels each with 5 VCPUs. The setup is shown in Table 4.3. The Canny thread is the target for migration from sandbox 1 to sandbox 2 in all cases. Migration is always requested at time 5. A logger thread is used to collect the result of the experiment in a predictable manner. Data points are sampled and reported in a one second interval. For migration with message passing, a low priority migration VCPU (10/200) is used. In the case of direct memory copy, the migration thread is associated with the highest priority VCPU (10/50).

VCPU (C/T)	**Sandbox 1**	**Sandbox 2**
20/100	Shell	Shell
10/200 (10/50)	Migration Thread	Migration Thread
20/100	Canny	
20/100	Logger	Logger
10/100	Comms 1	Comms 2

Table 4.3: Migration Experiment VCPU Setup

Figure 4.10 shows the behavior of Canny as it is migrated using message passing in the presence of the two communicating threads. The y-axis shows both Canny frame rate (in frames-per-second, *fps*) and message passing throughput (in multiples of a 1000 Kilobytes-per-second). Canny requested migration with the `MIG_RELAX` flag, leading to a drop in frame rate during transfer to the remote sandbox. However, the two communicating threads were not affected.

Table 4.4 shows the estimated worst-case and actual migration cost. The worst-case is derived from Equation 4.7. Even though the actual migration cost is much smaller than the estimation, it is still larger than E_m, forbidding migration with the `MIG_STRICT` flag.

Variables	E_m	$\Delta_{mig}, worst$	$\Delta_{mig}, actual$
Time (ms)	79.8	243681.02	4021.18

Table 4.4: Message Passing Migration Condition

In Figure 4.11, the same experiment was conducted with direct memory copy and

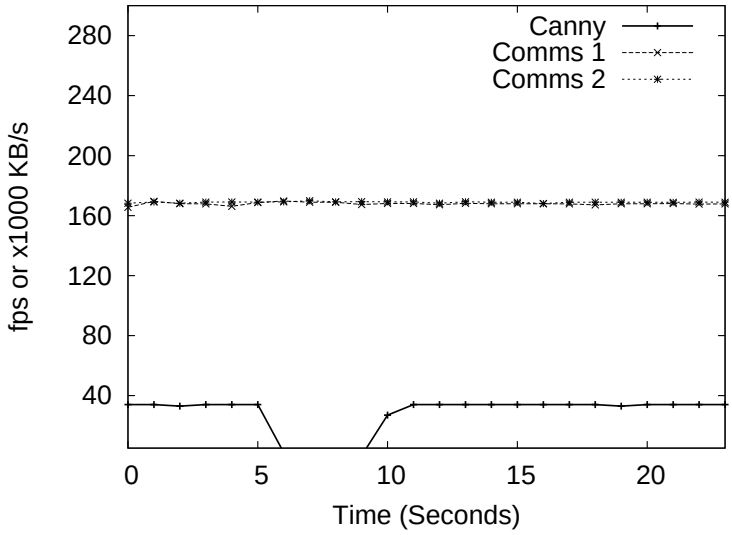

Figure 4.10: Migration using Message Passing

`flag=MIG_STRICT`. Since the migration thread was self-preempted, the right y-axis shows its actual CPU consumption in (millions of, *x1m*) cycles. We can see from this figure that none of the threads have been affected by migration. The sudden spike in migration thread CPU consumption occurs during the migration of the Canny thread.

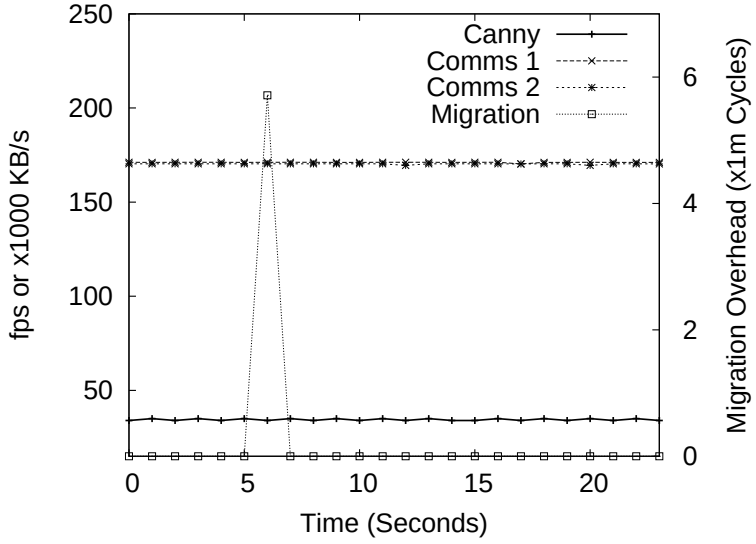

Figure 4.11: Migration using Direct Memory Copy

Table 4.5 shows the values of variables as defined in Equation 4.8 and 4.10. $\delta_m, worst$ is the worst-case time to copy a Canny address space with all caches disabled, including the overhead of walking its page directory. $\delta_m, actual$ is the actual migration thread budget consumption with caches enabled. Both worst-case and actual migration costs satisfy the constraints of Equation 4.10. This guarantees that all VCPUs remain unaffected in terms of their CPU utilization during migration.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	5.4	1.7	10	50

Table 4.5: Direct Memory Copy Migration Condition

In the next experiment, we switched back to `flag=MIG_RELAX` and manually increased the migration cost by adding a busy-wait of $800\mu s$ to the address space clone procedure for each processed Page Directory Entry (of which there were 1024 in total). This forced the migration cost to violate Equation 4.10. Similar to Figure 4.10, Figure 4.12 shows how the migration costs increase, with only the migrating thread being affected. Here, the preemption points within each sandbox monitor prevent excessive budget overruns that would otherwise impact VCPU schedulability.

Table 4.6 shows the migration parameters for this experiment. We also measured the budget utilization of the migration thread while it was active. Results are shown in Table 4.7 for the interval [6s,10s] of Figure 4.12. Migration thread budget consumption peaks at 91.5% rather than 100%, because of self-preemption and accounting overheads. We are currently working on optimizations to reduce these overheads.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	891.4	825.1	10	50

Table 4.6: Migration Condition With Added Overhead

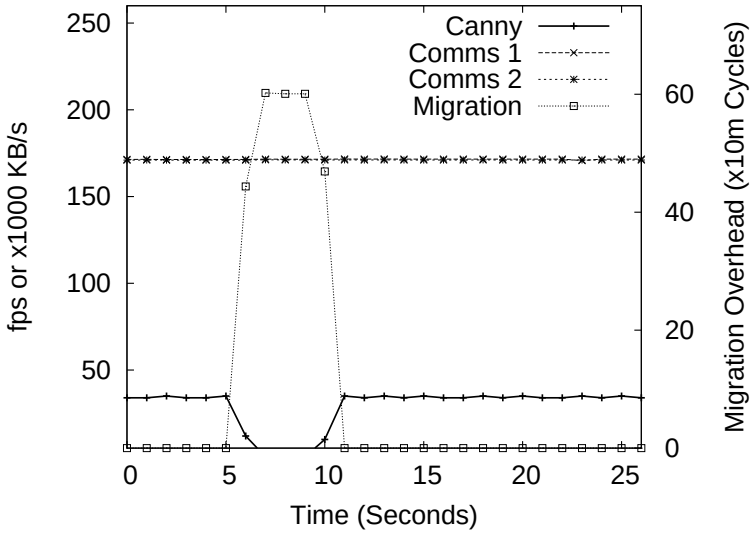

Figure 4.12: Migration With Added Overhead

Time (sec)	**6**	**7**	**8**	**9**	**10**
Utilization	67.5%	91.5%	91.5%	91.5%	71.5%

Table 4.7: Migration Thread Self-Preemption Budget Utilization

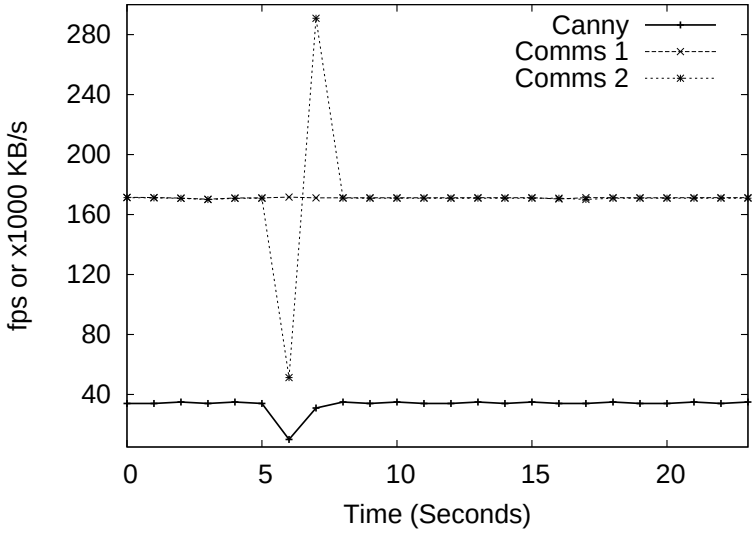

Figure 4.13: Migration Without a Dedicated Thread

The same experiment was repeated without a dedicated migration thread. Here, migration was instead handled in the context of an IPI handler that runs with interrupts subsequently disabled. Consequently, the handler delays all other threads and their VCPUs during its execution, as shown in Figure 4.13.

Next, table 4.8 shows the values of the variables used in Equation 4.8 and 4.10 when the migration overhead first starts to impact the Canny frame rate. In theory, the minimum δ_m that violates Equation 4.10 is any value greater than 10ms. However, because $\delta_m, worst$ is a worst-case estimation and the worst-case VCPU phase shift ($T_r - C_r$ in Equation 4.8) rarely happens in practice, the first visible frame rate drop happens at 26.4ms. At this time, the actual budget consumption of the migration thread is 19.2ms, which is greater than 10ms.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	26.4	19.2	10	50

Table 4.8: Migration Boundary Case Condition

Finally, as mentioned earlier in Section 4.3.1, if multiple migration requests to a destination sandbox are issued simultaneously, they will be processed serially. Currently, parallel migration is not supported. The source sandbox kernel has to essentially lock both its own migration thread and the migration thread in the destination before initiating migration. To demonstrate this effect, we conducted an experiment in which two sandboxes issued a migration request at the same time, to the same destination. The VCPU setup is shown in Table 4.9. In addition to Canny and the 2 communicating threads, we added another thread in sandbox 3 that repeatedly counts prime numbers from 1 to 2500 and increments a counter after each iteration. Canny and Prime attempt to migrate to sandbox 2 at the same time.

The results of the experiment are shown in Figure 4.14. The y-axis now also shows the

VCPU (C/T)	Sandbox 1	Sandbox 2	Sandbox 3
20/100	Shell	Shell	Shell
10/100	Mig Thread	Mig Thread	Mig Thread
20/100	Canny		
10/100	Logger	Logger	Logger
10/100	Comms 1	Comms 2	
10/100			Prime

Table 4.9: Migration Thread Contention Experiment VCPU Setup

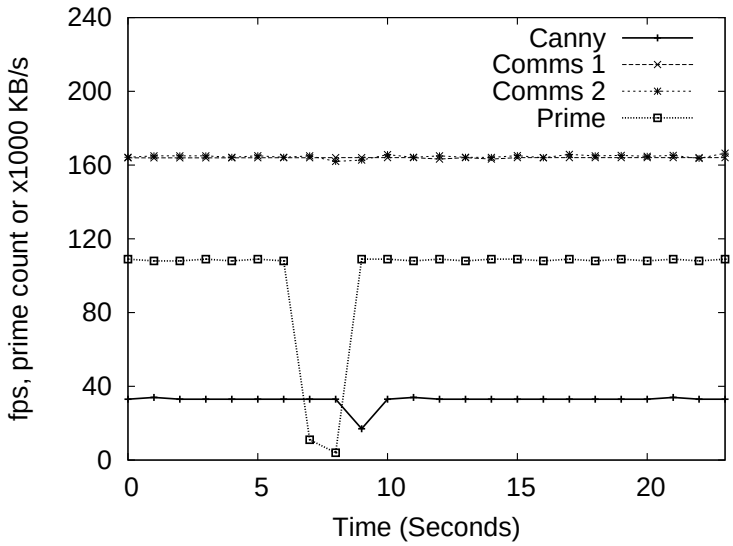

Figure 4.14: Migration Thread Contention

Prime count in addition to Canny frame rate and message passing throughput. Both Prime and Canny request migration to sandbox 2 at some time after 6 seconds. Prime acquires the locks first and starts migration immediately. The migration request of Canny is delayed since the *try_lock* function returns `FALSE` in Listing 4.3. Because both requests are issued with `flag=MIG_RELAX`, Canny is migrated soon after Prime finishes migration.

By setting the migration start time for Prime to $t_0 = 0$, Table 4.10 shows the relative time of: the start of data transfer of Prime (t_1), the end of data transfer of Prime (t_2), the end of Prime migration (t_3), the start of Canny migration (t_4), the start of data transfer of Canny (t_5), the end of data transfer of Canny (t_6) and the end of Canny migration (t_7).

t_0	t_1	t_2	t_3
0	3.15	1903.81	1913.47
t_4	t_5	t_6	t_7
1999.82	2003.67	2402.72	2412.98

Table 4.10: Migration Time Sequence (milliseconds)

4.6.3 Fault Isolation and Recovery

To demonstrate fault isolation in Quest-V, we created a scenario that includes both message passing and networking across 4 different Quest sandboxes. Specifically, sandbox 1 has a kernel thread that sends messages through private message passing channels to sandbox 0, 2 and 3. Each private channel is shared only between the sender and specific receiver, and is guarded by EPTs. In addition, sandbox 0 also has a network service running that handles ICMP echo requests. After all the services are up and running, we manually break the NIC driver in sandbox 0, overwrite sandbox 0's message passing channel shared with sandbox 1, and try to corrupt the kernel memory of other sandboxes to simulate a driver fault. After the driver fault, sandbox 0 will try to recover the NIC driver along with both network and message passing services running in it. During the recovery, the whole system activity is plotted in terms of message reception rate and ICMP echo reply rate in all available sandboxes, and the results are shown in Figure 4.15.

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

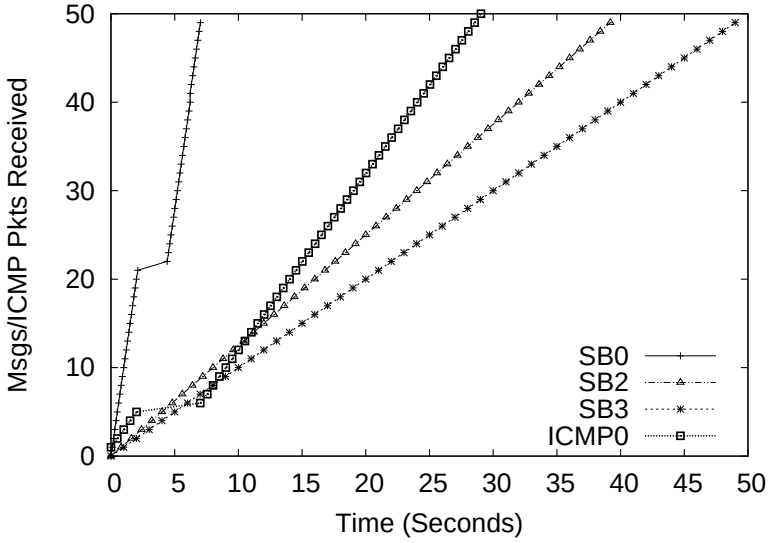

Figure 4.15: Sandbox Isolation

memory isolation between sandboxes, enforced by EPTs.

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. 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 17675 bytes.

Figure 4.16 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 re-initializing 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 re-initialized. This either involves re-initialization of the same driver that faulted in the first place, or an

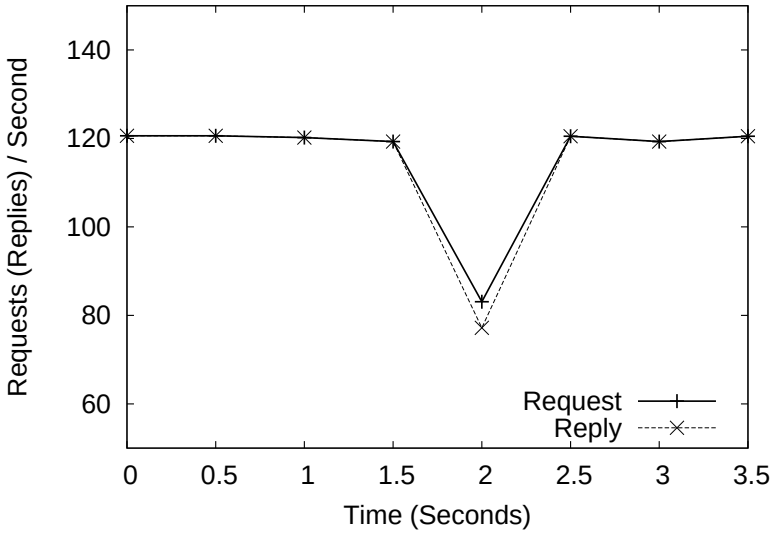

Figure 4.16: Web Server Recovery

alternative driver that is tried and tested. As fault detection is not in the scope of this dissertation, 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 both remote and local sandbox recovery are shown in Table 4.11. We can see from the table that Quest-V monitor layer introduced relatively small overhead to the overall recovery cost in both cases.

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 4.11: Overhead of Different Phases in Fault Recovery

4.6.4 Shared Device Performance

We implemented device sharing drivers in Quest sandboxes for a single NIC device, providing a separate virtual interface for each sandbox requiring access. This allows for each Quest sandbox to have its own IP address.

We compared the performance of our device sharing design to the performance of I/O virtualization adopted by Xen 4.1.2. Both para-virtualized (PVM) and hardware-virtualized (HVM) guests were evaluated. 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).

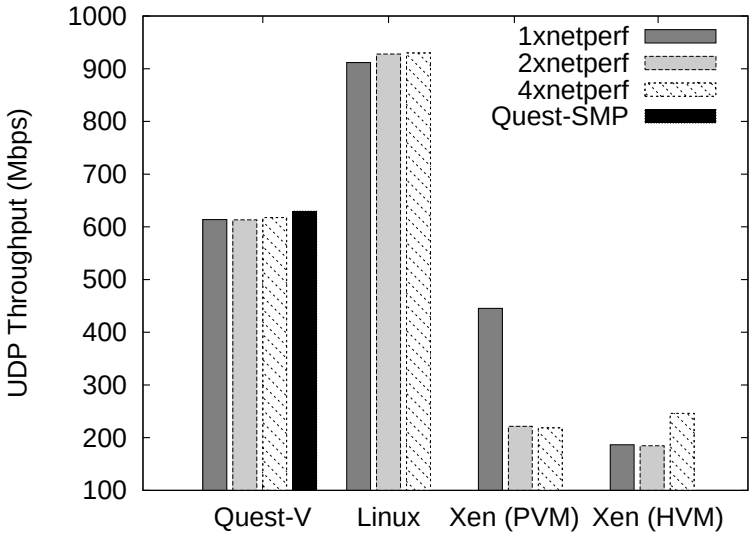

Figure 4.17: UDP Throughput

Figure 4.17 shows UDP throughput measurements using `netperf`, which was ported to Quest 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. For non-virtualized `Quest-SMP`,

only a single `netperf` instance was executed. 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 driver implementation 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 positive.

By comparing non-virtualized `Quest-SMP` and a single `netperf` in a Quest sandbox in Quest-V, we can see a small throughput overhead. This is caused by the extra logic in the driver as described in chapter 4.5. The device sharing logic exists even if only one Quest sandbox is using the device. With an identical standard driver, the overhead introduced by the Quest-V separation kernel would be negligible.

4.7 Conclusions

In this chapter, we introduced the Quest sandbox support in the Quest-V separation kernel. Quest kernel and its VCPU scheduling framework are designed for real-time and embedded applications. In Quest-V, Quest is the default sandbox kernel targeting high criticality tasks in a mixed criticality system.

This chapter focuses on predictable communication and migration in the Quest-V separation kernel. With the temporal isolation provided by the VCPU scheduling framework in the Quest kernel, we have shown how Quest-V is able to enforce predictable time bounds on the exchange of information between threads mapped to different Quest sandboxes. This lays the foundations for real-time communication in a distributed embedded system and helps making predictable service migration possible.

Quest-V allows threads to migrate between Quest sandboxes. This might be necessary

to ensure loads are balanced, and each sandbox can guarantee the schedulability of its virtual CPUs (VCPUs). In other cases, threads might need to be migrated to sandboxes that have direct access to I/O devices, thereby avoiding expensive inter-sandbox communication. We have shown how Quest-V's migration mechanism between separate sandboxes is able to ensure predictable VCPU and thread execution. Experiments show the ability of the Canny edge detector to maintain its frame processing rate while migrating from one sandbox to another.

The isolation between sandboxes in Quest-V offers software component fault containment and allows for the implementation of various fault recovery mechanisms such as local and remote recovery in Quest-V. Even though current Quest-V implementation offers limited fault recovery support for applications and drivers, experiments show that faults are contained in the faulting sandbox. Application specific fault recovery experiment demonstrates that software faults can be recovered in either local or remote Quest sandbox with the monitor introducing relatively small overhead.

Finally, even though not recommended, Quest-V does offer device sharing between Quest sandboxes. Experiments show that device sharing between Quest sandboxes offers better performance than I/O multiplexing in traditional VMMs. However, faults in the device driver of a shared device can potentially affect all the sandboxes sharing the same device. This makes device sharing acceptable most likely only between low criticality sandboxes with the same criticality level.

Chapter 5

Third Party Sandbox Support

In addition to Quest real-time kernels, Quest-V is also designed to support other third party sandbox systems such as Linux, OSEK [OSE] and AUTOSAR [AUT] OS. Support of these systems are necessary for reusing existing legacy services and potential Intellectual Property (IP) protection.

In order to integrate a mixed criticality application designed for a physically distributed platform onto a single multi-core platform, minimal or even no changes to the software components are always preferred. Sometimes the legacy services are so complicated that porting it to a new platform is almost impossible or too costly. For instance, the infotainment systems in modern automobiles are often developed on top of full blown operating systems such as Linux and Windows. To integrate an infotainment system with the body electronics management system onto a single multi-core platform would require the support of a Linux or Windows environment on the new platform. Additionally, some mixed criticality systems involve software components supplied by third party developers with no source code available for IP protection. This also makes legacy service support necessary for mixed criticality and embedded applications.

In this chapter, we describe how Linux and OSEK/AUTOSAR OSes can be supported in Quest-V sandboxes with a focus on the more complicated Linux kernel and its performance. Experiments show that the Linux distribution we ported to Quest-V achieves near

bare-metal performance and strong temporal isolation for most applications.

5.1 Linux Sandbox Support

Linux is an operating system that supports a vast variety of hardware devices, applications, and network protocols. Its large developer community and low cost contributed to the rapid growth of its adoption in the embedded applications. Due to its popularity, we made Linux sandbox support our primary objective for legacy service support in Quest-V.

Currently, we have successfully ported a Puppy Linux [PUP] distribution with Linux 3.8.0 kernel to serve as the front-end to Quest-V, 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 bootloader kernel thread. Theoretically, the Linux kernel can be supported in Quest-V directly without requiring any modifications to the kernel source code given the hardware virtualization features. However, to simplify the monitor logic and reduce its footprint, we paravirtualized the Linux kernel by patching the source code.

5.1.1 Kernel Paravirtualization

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 less than 50 lines. These changes are mainly focused on limiting Linux's view of available physical memory and handling I/O device DMA offsets caused by memory virtualization.

An example memory layout of Quest-V with a Linux sandbox on a 4-core processor

is shown in Figure 5.1. While 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. Since Quest-V does not require a hardware IOMMU, we have to patch the Linux kernel to make it aware of this offset between guest physical and machine physical memory addresses during I/O device DMA. This is necessary because I/O device is only aware of the machine physical memory and knows nothing about guest physical memory. Fortunately, the Linux kernel DMA layer is highly modularized and the changes required are minimal. We added a total of 4 lines of C code and some macro switches to support a configurable constant offset in the Linux device DMA subsystem.

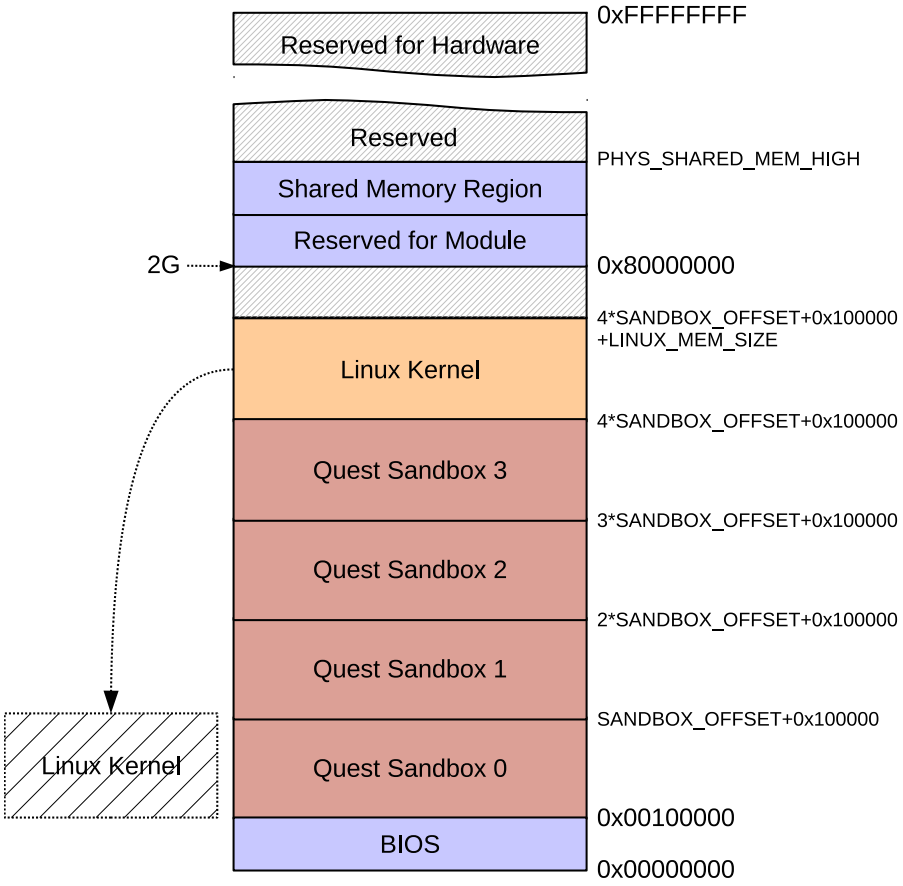

Figure 5.1: Quest-V Physical Memory Layout with Linux

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 from having access to persistent disk storage. In case memory resource is limited, the root filesystem RAM disk can be replaced by flash storage or hard drive. Additionally, the graphical interface can also be disabled to further reduce kernel image size and memory consumption. Since Quest-V allocates only a subset of the physical memory to the Linux sandbox, the monitor needs to prevent the Linux kernel from detecting the actual memory available in the system. To minimize the monitor, instead of emulating the BIOS, we modified the Linux kernel with a statically configured memory size. This limits the Linux memory management subsystem to only access the memory it is allocated to. In case a malfunctioned driver tries to access memory outside of those allocated to the Linux sandbox, a trap into the monitor will be triggered and fault recovery can be initiated.

5.1.2 Remote Sandbox Access

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. Each non-Linux sandbox is allocated a virtual VGA frame buffer. These sand-

boxes output all VGA data to their virtual frame buffer as if they are writing directly to the physical one. We developed a Linux character device driver kernel module that exposes each virtual VGA frame buffer as a character device. These devices can be accessed from the Linux device filesystem through basic file operations. A *read* from any of the device files will return the raw data in their corresponding virtual frame buffers. A *write* can be used to output VGA data to the virtual frame buffer and to send commands to the sandbox remote shell. We developed a Linux user space application called *quesh* to simulate a virtual terminal by reading and writing the virtual VGA frame buffer device file for each remote sandbox.

A screen shot of Quest-V after booting the Linux front-end sandbox is shown in Figure 5.2. Here, we show two virtual terminals connected to two different Quest sandboxes similar to the configuration shown in Figure 3.1. In this particular example, we allocated 512MB of memory to the Linux sandbox (including an in-memory root filesystem) and 256MB to each native Quest sandbox. The network interface card has been assigned to Quest sandbox 1, while the serial device has been assigned to Quest sandbox 2. The Linux sandbox is granted ownership of the USB host controller in addition to the GPU and VGA frame buffer. Observe that although the machine has four processor cores, the Linux kernel detects only one core. From the content of `/proc/cpuinfo` file shown in the bottom terminal window, we can see that Linux identifies all the CPU capabilities except `VMX` and `EPT`.

5.1.3 Inter-Sandbox Communication

The Linux CFS scheduler does not offer the level of predictability provided by the VCPU scheduling framework in the Quest kernel. Consequently, it is difficult to extend the predictable communication framework introduced in Chapter 4.2 to communications between

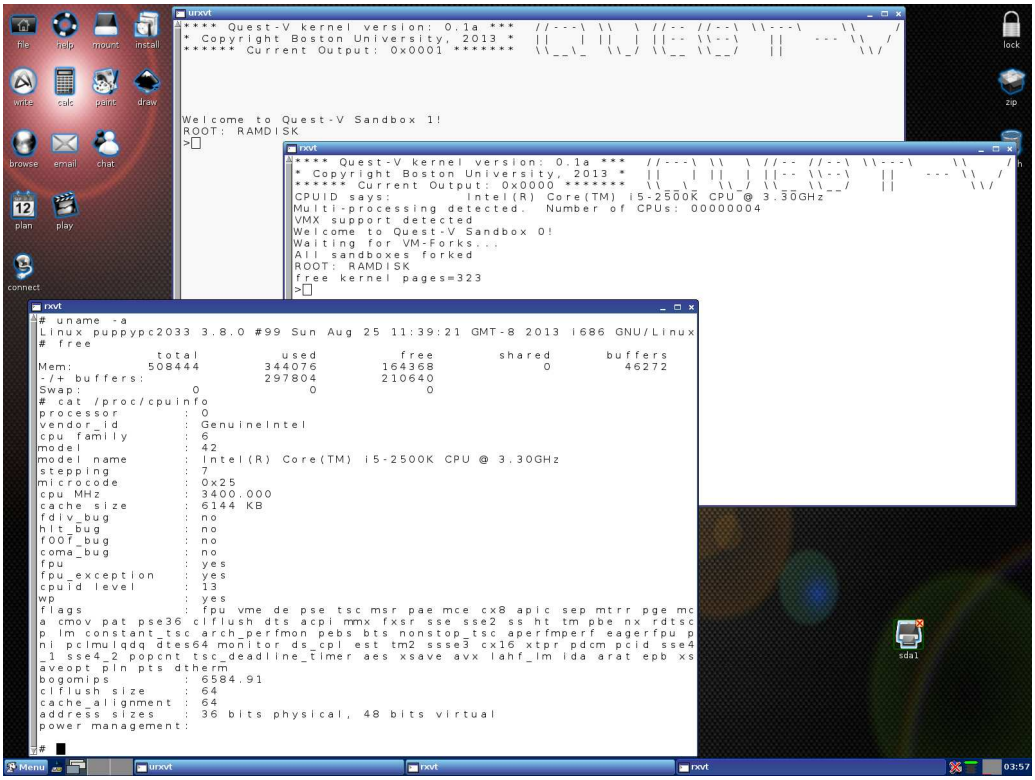

Figure 5.2: Quest-V with Linux Front-End

Linux and Quest sandboxes. However, the communication between Linux and Quest processes executing in different sandboxes is still necessary for mixed criticality applications. To enable this feature, we developed the kernel extension, user library, and daemon service required for the Linux front-end.

As mentioned in Chapter 4.2, inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory. Private message passing channel needs to be established before communication commences. The message passing channel establishment protocol is handled by a kernel thread running in each Quest sandbox. These threads are responsible for handling message passing channel establishment requests, sending acknowledgments, and setting up EPT mappings for the shared memory regions. We implemented the same message passing channel establishment protocol for the Linux sandbox

as a daemon process. However, in case a Quest sandbox initiates an establishment request, the channel will be allocated in machine physical memory regions outside of Linux sandbox. As mentioned earlier, Linux kernel memory management subsystem is only aware of the machine physical memory region allocated to the Linux sandbox. In order to minimize the effort required to paravirtualize Linux kernel for Quest-V, we developed a separate *out-of-band* physical memory manager for the Linux front-end as a kernel module. The daemon process manages EPT mappings of the message passing channels by interfacing the new physical memory manager which in turn traps into Quest-V monitor for actual EPT configurations.

The user APIs for shared memory channel management in the Linux front-end is provided through a library called *libshm*. This library simply communicates with the daemon process via POSIX shared memory utilities for message passing channel management. Additionally, this library also provides high level communication protocols such as four-slot [Sim90] and ring buffer on top of shared memory message passing channels.

5.2 OSEK/AUTOSAR OS Sandbox Support

In addition to the Linux sandbox, we decided to also support OSEK/AUTOSAR OSes in Quest-V for automotive applications. OSEK is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems. OSEK was designed to provide a standard software architecture for the various electronic control units (ECUs) throughout a car. The OSEK operating system is a single processor operating system designed to provide priority based scheduling, synchronization, alarms, and interrupt management for application tasks. It is usually implemented as a single address space operating system and is configured statically by the user before deployment.

As a standards body found in the early 90s, OSEK is now gradually replaced by the new AUTOSAR initiative. AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers, and tool developers. The AUTOSAR architecture is very ambitious and covers the standardization of both software and hardware components on various vehicle platforms. The large scale of the domain makes the standards overly complicated and difficult to implement comprehensively. However, the AUTOSAR OS specification reuses the OSEK OS specification. All the AUTOSAR OSes are backward compatible with OSEK standard with the optional enhancement of schedule tables [1], software timers, memory protection, and multi-core support, etc.

As a prototype, we ported the ERIKA Enterprise OS [ERI] to Quest-V. ERIKA Enterprise is an open source OSEK OS with several AUTOSAR extensions. It runs on the x86 platform as a single address space system by default. The system operates in the x86 protected mode with paging disabled. GRUB is required to bootstrap the multiboot compliant ERIKA OS image. As with the Linux sandbox, a Quest sandbox needs to be initialized before any OSEK/AUTOSAR OS can be loaded. We developed utilities in the Quest kernel and user space that enable a Quest sandbox to essentially boot any multiboot compliant OS. A user can use the *kexec* (kernel exec) command from within the Quest shell to boot a new kernel image in the sandbox. The kexec command requires the kernel image path to be specified. Theoretically, no changes are needed to the target operating system as long as the PIT is not used as the system timer. The reason that the PIT causes problems has been explained earlier in Chapter 5.1.1. Unfortunately, the ERIKA Enterprise OS does utilize PIT. Consequently, we had to modify the kernel source code and switch the system timer to a local APIC timer. In the future, we will investigate the support of emulated software

[1] schedule tables are recurring or single-shot task activation and event sequences configured by a user statically before system generation.

timers in Quest-V, especially for platforms with only one physical timer available. Since OSEK/AUTOSAR OSes do not have terminal I/O, the debug output is directed to a serial port by default. To grant serial port access to an OSEK/AUTOSAR sandbox, we simply assigned the serial device to the sandbox statically before system initialization.

Since OSEK/AUTOSAR OS support is a relatively recent addition to the Quest-V separation kernel, it is under active development. We currently do not have utilities for communication between OSEK/AUTOSAR OS sandbox and other sandboxes. A solution similar to that of the Linux sandbox is not sufficient since both OSEK and AUTOSAR define their own communication subsystems. Details of potential future work including an OSEK/AUTOSAR compliant communication framework will be discussed in Chapter 7.

5.3 Experimental Evaluations

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

5.3.1 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

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

Table 5.1: Monitor Trap Count During Linux Sandbox Initialization

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

5.3.2 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 5.3, for the configurations in Table 5.2. All Linux configurations were limited to 512MB RAM. For `Xen HVM` and `Xen PVM`, we pinned the Linux virtual machine (VM) to a single core that differed from the one used by Xen's Dom0. For all `4VM` configurations of Xen, we allowed Dom0 to make scheduling decisions without pinning VMs to specific cores.

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

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

Table 5.2: System Configurations

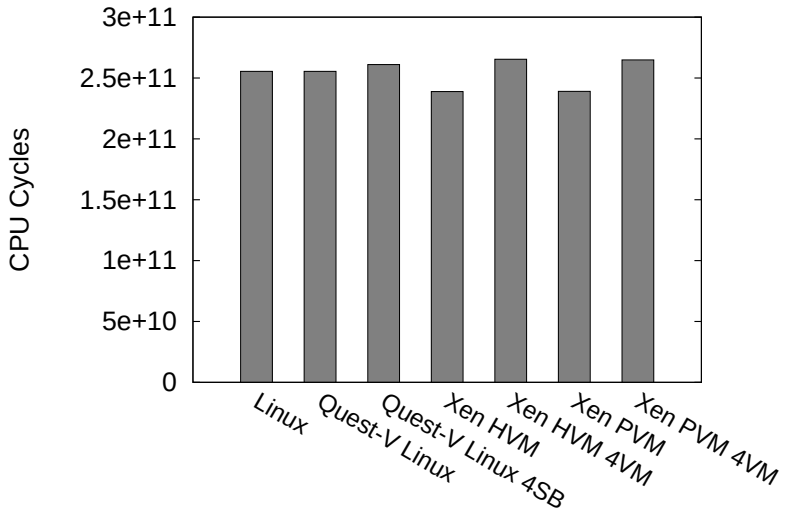

Figure 5.3: findprimes CPU Benchmark

seems to be attributed to the way Xen virtualizes devices and reduces the impact of events such as interrupts on thread execution. The results show approximately 2% overhead when running *findprimes* in a Linux sandbox on Quest-V, in the presence of three native Quest sandboxes. We believe this overhead is mostly due to memory bus and shared cache contention. For the `4VM` Xen configurations, the performance degradation is slightly worse. This appears to be because of the overheads of multiplexing 5 VMs (one being Dom0)

onto 4 cores.

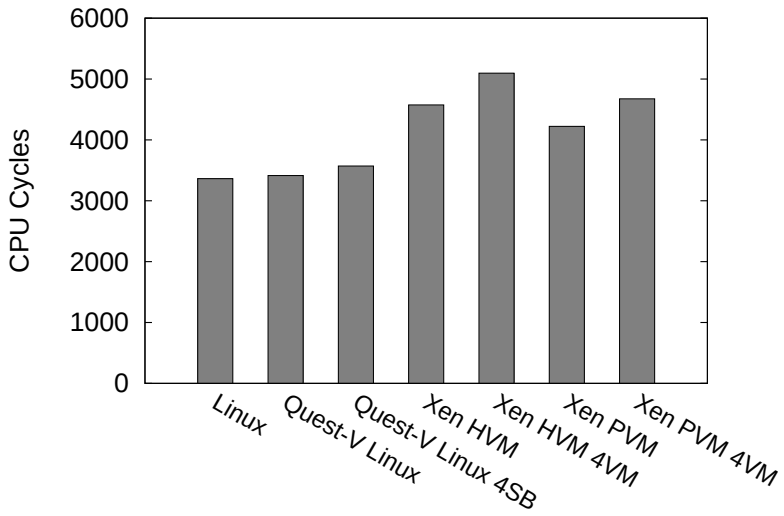

Figure 5.4: Page Fault Exception Handling Overhead

We evaluated the exception handling overheads for the configurations in Table 5.2, by measuring the average CPU cycles spent by Linux to handle a single user level page fault. For the measurement, we developed a user program that intentionally triggered a page fault and then skipped the faulting instruction in the `SIGSEGV` signal handler. The average cycle times were derived from 10^8 contiguous page faults. The results in Figure 5.4 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 5.5. `Quest-V Linux` is almost as good as native Linux and more than twice as fast as any Xen configuration.

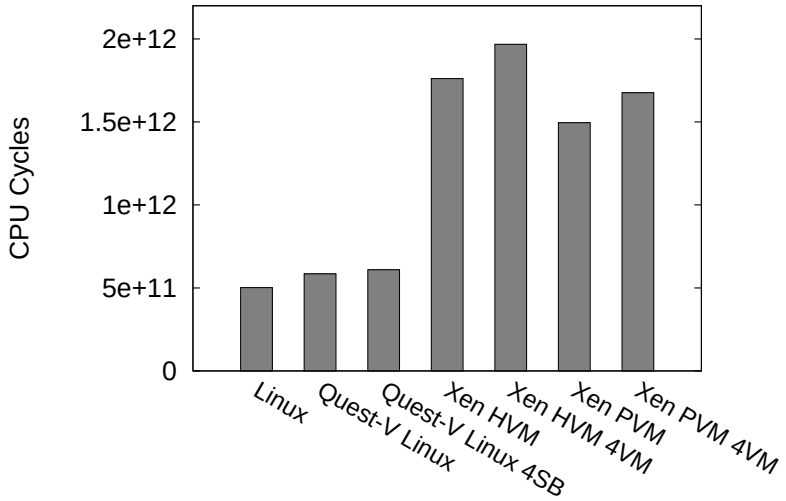

Figure 5.5: Fork-Exec-Wait Micro Benchmark

5.3.3 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 5.3 for three different configurations. In Quest-V, the Linux sandbox was given exclusive control over an integrated HD Graphics 3000 GPU. The results show that Quest-V incurs negligible overhead for HD video decoding and playback in Linux. We also observed (not shown) the same playback frame rate for all three configurations.

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

Table 5.3: mplayer HD Video Benchmark

5.3.4 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 5.6 and 5.7, respectively.

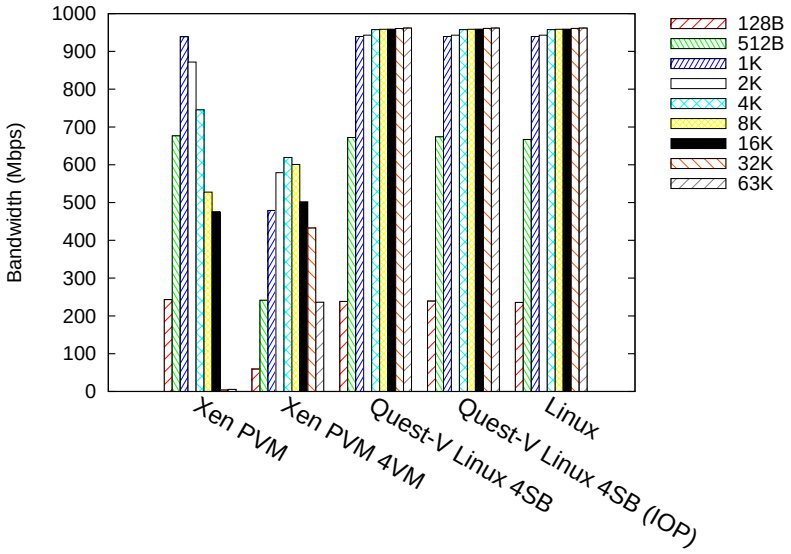

Figure 5.6: netperf UDP Send with Different Packet Sizes

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` [Rus08] is enabled. It can be seen that this helps dramatically improve the UDP bandwidth for small size UDP packets. With 512B packet size, `Xen PVM` outperforms standalone Linux. However, Quest-V Linux exhibits no visible overhead as compared to standalone Linux and outperforms Xen with

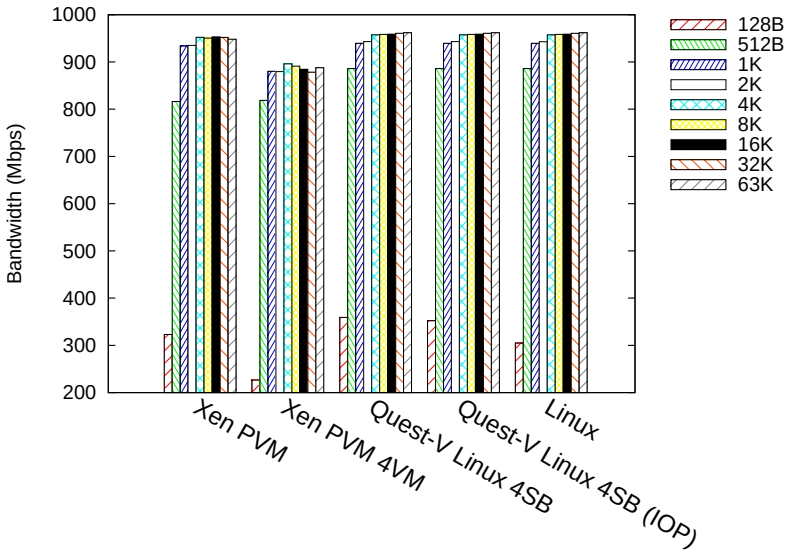

Figure 5.7: netserver UDP Receive

bigger packet sizes and multiple VMs.

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

5.3.5 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 5.8 - 5.13.

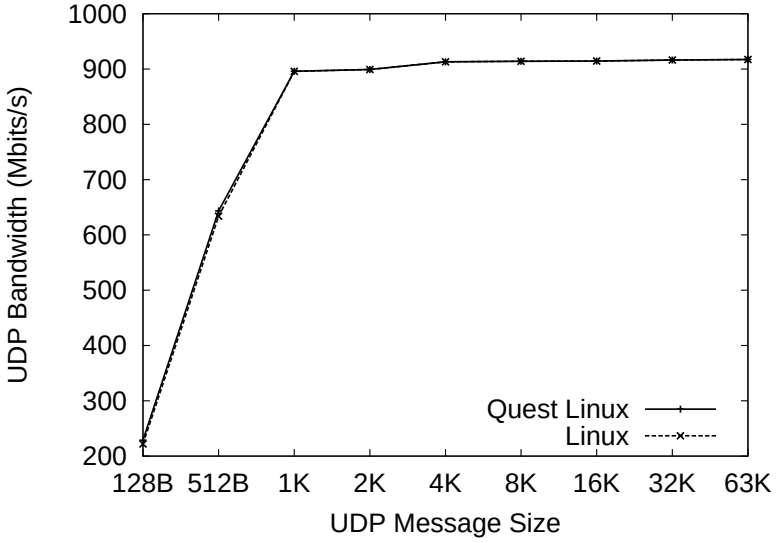

Figure 5.8: UDP Bandwidth

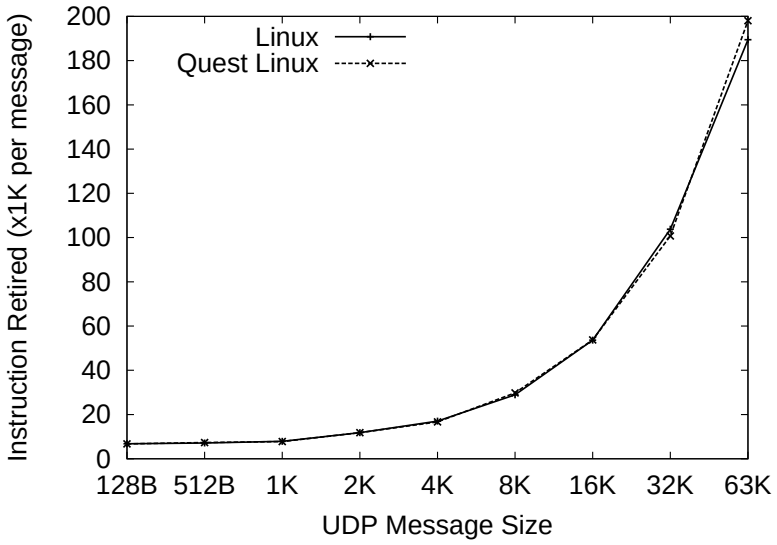

Figure 5.9: Instructions Retired

Figure 5.8 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 5.9, we show the number of instructions retired over each UDP send operation. The results confirm that Quest-V does not interfere with the op-

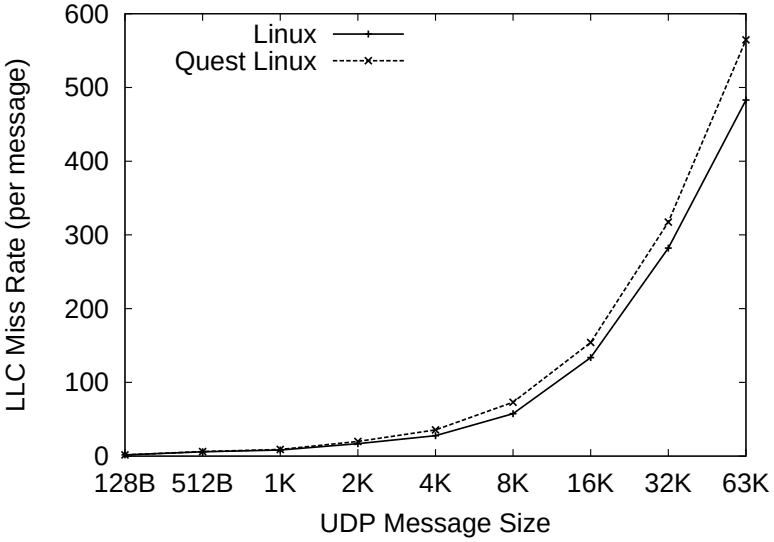

Figure 5.10: Last Level Cache Misses

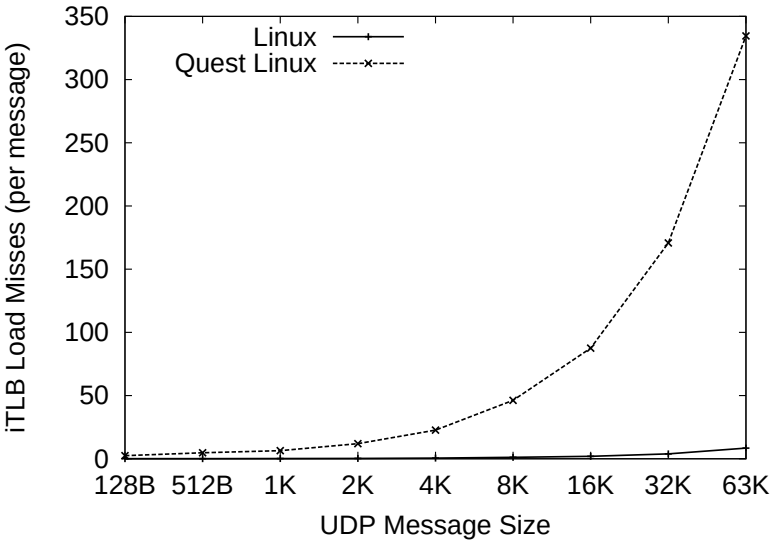

Figure 5.11: iTLB Load Misses

eration of a sandbox since no extra monitor instructions are executed. Figures 5.10 to 5.13 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 Linux. The TLB load and store misses are miss events

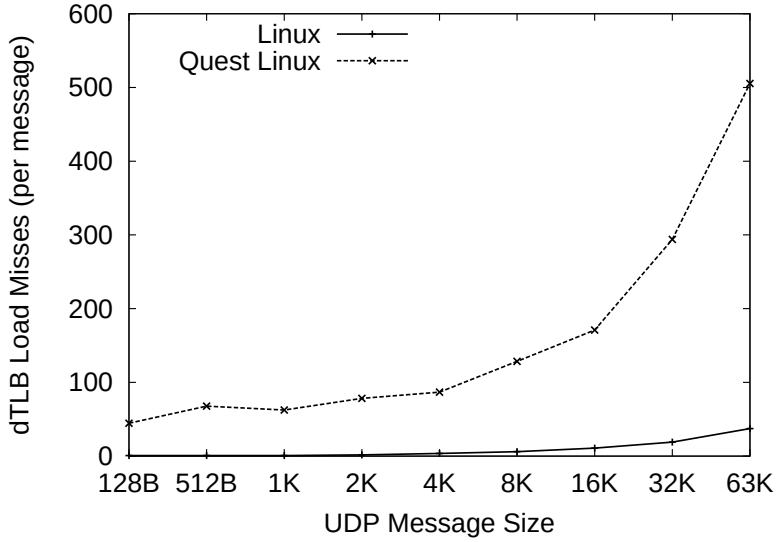

Figure 5.12: dTLB Load Misses

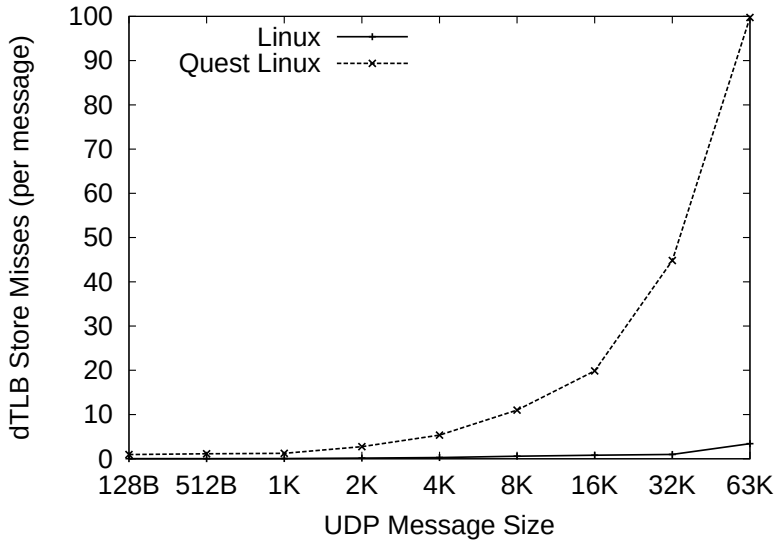

Figure 5.13: dTLB Store Misses

recorded during memory load and store instructions, respectively. As can be seen from these results, Quest-V does incur visible cache or TLB performance overheads. The overheads that do occur are mainly due to the extra levels of address translation caused by EPTs. Since the EPTs and traditional page tables share the same TLBs, TLB contention

is increased when EPTs are activated. The extra last level cache misses are potentially the consequence of extra page table walks caused by EPT TLB misses. However, since the UDP benchmark is I/O-bound, micro-architectural 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 Linux. This was to investigate communication performance without I/O overheads such as DMA and device interrupts. The bandwidth results shown in Figure 5.14 confirm that Quest-V Linux does incur a performance penalty compared to running as a standalone system. Figures 5.15 to 5.19 show similar results for last level cache and TLB misses. In addition to the standalone and Quest Linux configurations used in the previous experiments, we added a new configuration labeled "Quest Linux LP" which uses 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 misses. The UDP bandwidth also improved under this configuration.

5.3.6 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 4160 bytes offsets to avoid cache aliasing effects. The average time for the thread to complete access to a working set of pages was measured over 10 million iterations.

Figures 5.20 and 5.21 compare the performance of a native Quest kernel running in

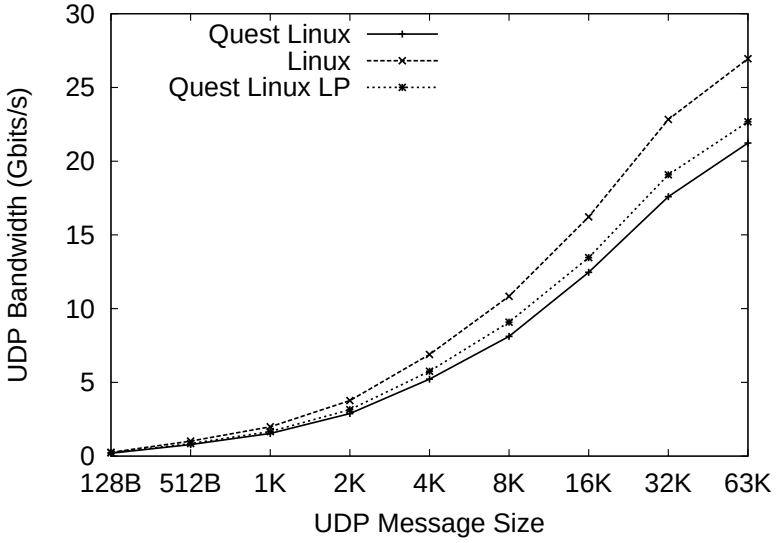

Figure 5.14: Local UDP Bandwidth

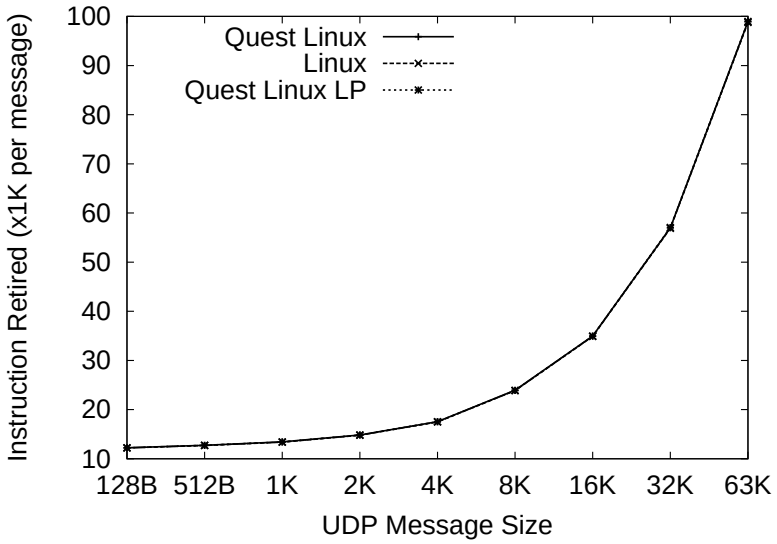

Figure 5.15: Instructions Retired

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

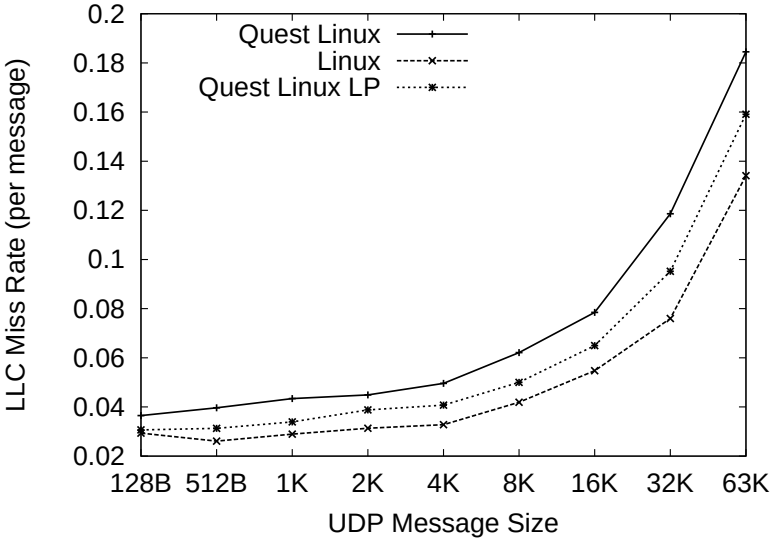

Figure 5.16: Last Level Cache Misses

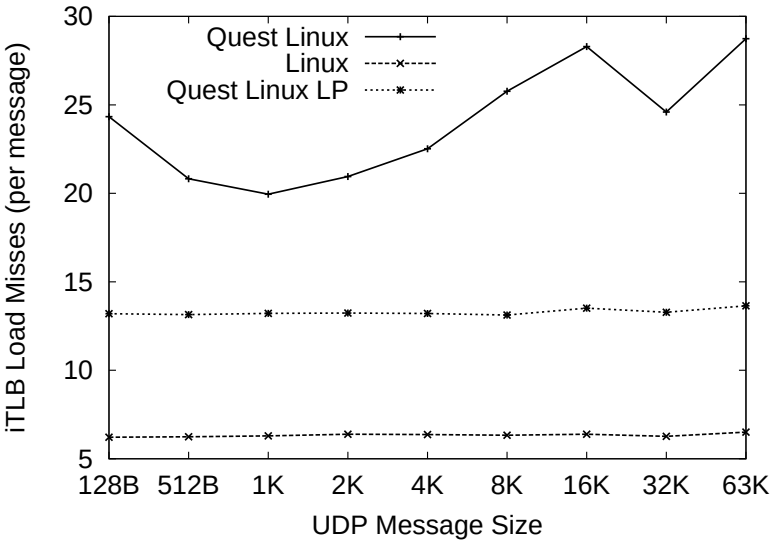

Figure 5.17: iTLB Load Misses

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

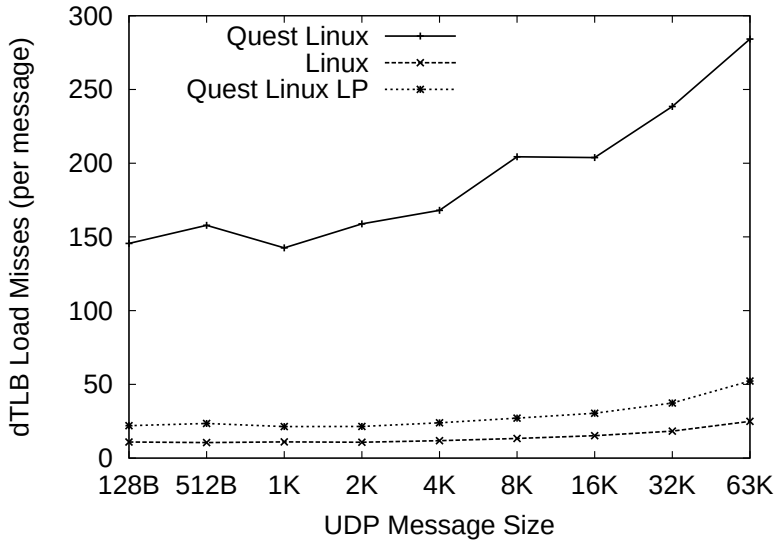

Figure 5.18: dTLB Load Misses

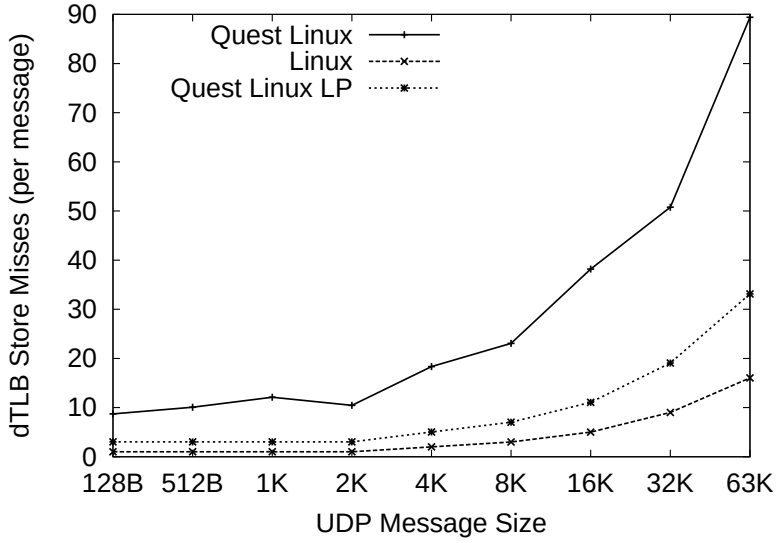

Figure 5.19: dTLB Store Misses

on a base system without virtualization (`Quest Base`) for working sets with less than 512 pages. We believe this is acceptable for safety-critical services found in embedded systems, as they are likely to have relatively small working sets. The cost of a VM-Exit is equivalent to a full TLB flush, but entries will not be flushed in Quest-V sandboxes if they

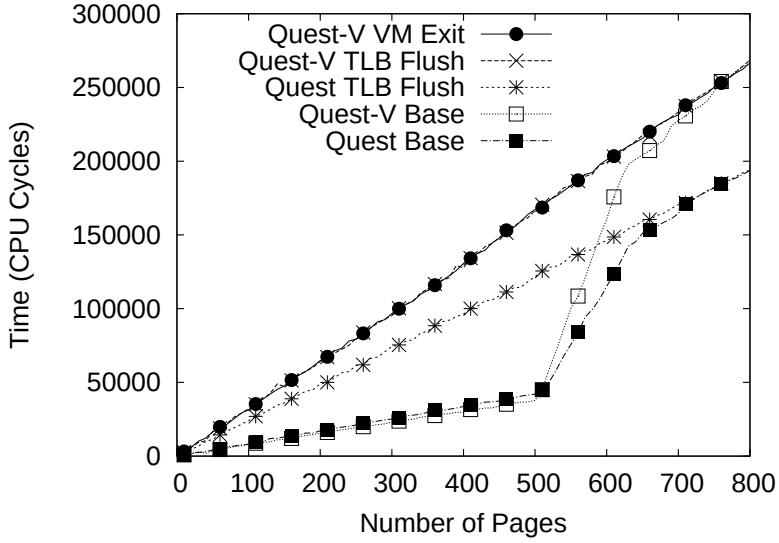

Figure 5.20: Data TLB Performance

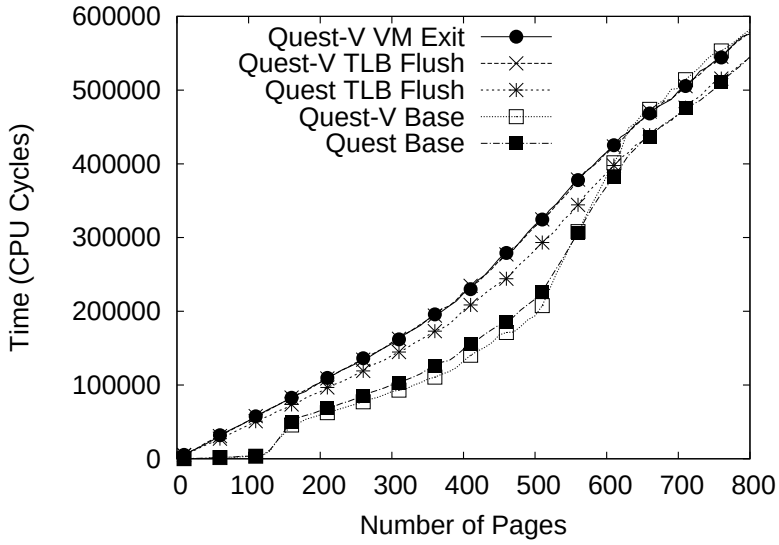

Figure 5.21: Instruction TLB Performance

are within the TLB reach. Note that without the use of TLBs to cache address translations, the EPTs require 5 memory accesses to perform a single guest-physical address (GPA) to host-physical address (HPA) translation. The kernels running the TLB walker use two-level paging for 32-bit virtual addresses, and in the worst-case this leads to 3 memory

accesses for a GVA to GPA translation. However, with virtualization, this causes $3 \times 5 = 15$ memory accesses for a GVA to HPA translation.

5.4 Conclusions

In this chapter, we introduced the third party sandbox kernels supported by Quest-V in addition to Quest sandboxes. Third party sandbox support is necessary in embedded mixed criticality systems for the reuse of existing legacy services, especially those without source code availability. Currently, Quest-V supports both Linux and OSEK/AUTOSAR OS sandboxes. Applications and services developed for these systems can be easily deployed in Quest-V. We paravirtualized a Puppy Linux distribution with 3.8.0 kernel to Quest-V with less than 50 lines of code changed. The Linux sandbox serves as the system front-end when present. All the other sandboxes can be accessed through remote virtual terminals. Processes running in the Linux sandbox can also communicate with processes running in the Quest sandboxes via the shared memory message passing channel APIs we developed for the Linux sandbox. As for OSEK/AUTOSAR OSes, we ported the ERIKA Enterprise OS to Quest-V. The only change we made to the ERIKA OS kernel is to switch the system timer from the PIT to the local APIC timer.

Experiments show that the Quest-V separation kernel does not interfere with normal sandbox operation and incurs no visible performance overhead to various application benchmarks running in the Linux sandbox. Microarchitecture benchmarks show that for applications with relatively large memory footprint, the extra memory translation overhead caused by EPT can become visible. However, we believe this is not a problem for embedded applications with relatively small working sets.

Chapter 6

Conclusions

In this dissertation, we introduced Quest-V, which is an open-source separation kernel built from the ground up for mixed criticality systems such as those seen in the healthcare, automotives, and avionics industries. It uses hardware virtualization to separate system components of different criticalities. Consequently, less important services can be isolated from those of higher criticality on a single multi-/many-core platform, and essential services can be replicated across different sandboxes to ensure availability in the presence of faults.

Quest-V avoids traditional costs associated with hypervisor systems, by statically partitioning machine resources across guest sandboxes, which perform their own scheduling, memory, and I/O management without monitor interference. Sandboxes can communicate via shared memory channels that are mapped to extended page table (EPT) entries. Only trusted monitors are capable of changing entries in these EPTs, preventing guest access to arbitrary memory regions in remote sandboxes. Since Quest-V attempts to avoid VM-exits as much as possible, except to update EPTs for communication channels, bootstrap the sandboxes and handle faults, the TLBs caching EPT mappings are rarely flushed. This benefit comes about due to the fact that multiple guests are not multiplexed onto the same processor core, and in the embedded systems we envision for this work, sandbox working sets will fit within the TLB reach (at least for critical services in Quest sandboxes).

Quest is the default kernel for each Quest-V sandbox. It is a kernel developed for real-time and embedded systems featuring a novel hierarchy of VCPUs implemented as Sporadic Servers, to ensure temporal isolation amongst real-time, safety-critical threads. Quest is designed for mission critical and safety critical tasks in a mixed criticality application. Threads in different Quest sandboxes can communicate with each other through a predictable communication mechanism that provides worst case message passing delay guarantees. Threads and their VCPUs can also migrate from one Quest sandbox to another without violating any VCPU timing guarantees. Experiments show that the predictable communication and migration framework we developed for Quest sandboxes are practical and effective. The temporal and spatial isolation between Quest-V sandboxes also offer software fault containment and allow various fault recovery mechanisms to be implemented. Experiments show that software faults in one sandbox are isolated from other sandboxes. A web server fault recovery example in a Quest sandbox also demonstrates that software faults can be recovered efficiently either locally or remotely without prohibitive monitor overhead. Morever, Quest-V also offers efficient device sharing between Quest sandboxes, though at the cost of lowered safety.

In addition to Quest sandboxes, Quest-V also support third party sandbox kernels such as Linux and OSEK/AUTOSAR OSes. All third party sandboxes supported by Quest-V can only be bootstrapped by a Quest kernel. Currently, we ported a Puppy Linux distribution with Linux 3.8.0 kernel to serve as the front-end to Quest-V, providing a window manager and graphical interface. All the other sandboxes can be accessed from the Linux front-end through virtual terminals we developed on top of shared memory channels. Low criticality applications running in the Linux front-end can communicate with the critical tasks running in the Quest sandboxes via shared memory message passing APIs available in both Linux and Quest sandboxes. More recently, we ported ERIKA Enterprise OS,

an OSEK single processor operating system designed for automotive ECUs, to Quest-V specifically for automotive applications. Experiments show that the Quest-V separation kernel does not interfere with normal sandbox operations and incurs no visible performance overhead to the sandbox kernels. Various application benchmarks in the Linux front-end exibit bare-metal Linux performance. These results show that our Quest-V separation kernel implementation is consistent with our initial design philosophies.

Quest-V distributed monitors occupy a small memory footprint compared to traditional hypervisors. They are used only to partition resources, assist in fault recovery and establish inter-sandbox communication channels. By comparison, traditional hypervisors need extra functionality to multiplex guest virtual machines on a shared set of hardware resources. In Quest-V, each sandbox manages its own partitioned set of resources so hypervisor-based virtualization of those resources is eliminated. Similarly, Quest-V monitors are not required for most service requests, which heightens the security of the system.

Chapter 7

Future Work

This chapter describes future directions for Quest-V development.

7.1 VCPU Migration Policy

In chapter 4.3, we introduced the predictable service migration mechanism used by Quest kernel to migrate threads and their VCPUs between Quest sandboxes. We mentioned that VCPU migration is useful for several reasons amongst which are the need to balance loads across sandboxes of the same criticality levels and to guarantee the schedulability of VCPUs and threads. However, currently there is no policy available in the Quest kernel that makes migration decisions according to these objectives.

A potential policy could be designed around the optimization and balance of three aspects of the system: microarchitectural resource contention, CPU utilization, and power consumption. To mitigate microarchitectural resource contention, the Quest kernel needs to understand the microarchitectural resource consumption of each thread and attempt to migrate threads and their VCPUs for optimal co-runner selection that minimize cache and memory bus contention [WZWZ13]. If cache partitioning techniques such as page coloring are utilized, the migration policy should attempt to assign threads and VCPUs to different Quest sandboxes to maximize cache utilization. CPU utilization of a Quest sandbox is primarily limited by the VCPU scheduling policy introduced in chapter 4.1.

The predictable migration mechanism in Quest kernel already guarantees that migration requests that violate the VCPU utilization bound will be rejected. However, the migration policy could attempt to migrate VCPUs in under utilized sandbox to other sandboxes and change the C-state of the processor core to conserve power.

In summary, in addition to the predictable migration mechanism introduced in this dissertation, a migration policy in the Quest kernel that considers all the above criteria and respects other restrictions such as I/O device sandbox affinity is needed in order to make appropriate migration decisions that optimize system behavior under different application requirements.

7.2 Dynamic Resource Partitioning

On future many-core platforms with large number of cores and I/O peripherals, the static resource partitioning scheme of Quest-V separation kernel introduced in chapter 3.2 will become increasingly inflexible. To increase system wide resource allocation flexibility and reduce power consumption, dynamic CPU, memory, and I/O device repartitioning schemes should be considered in future Quest-V development.

Quest-V sandboxes should be able to dynamically add and remove processor cores. This allows the processor cores to be powered on and off individually in response to workload changes. Multi-core sandbox also allows the sandbox kernel and its applications to exploit the physical parallelism if necessary. An approach similar to the Barrelfish/DC [ZGKR14] *boot driver* should be considered in Quest-V separation kernel which allows monitors to decouple sandboxes from the processor cores. Similarly, dynamic I/O device assignment should also be supported to satisfy flexible I/O demands and reduce power consumption. Once the device blacklists in the monitors are made dynamic, I/O device repartitioning can be supported with proper device de-initialization and re-

initialization in the corresponding sandboxes. Finally, sandboxes should also be allowed to dynamically adjust their memory demands. Although preferably this should be carried out at a larger granularity than a single memory page in order to reduce potential monitor traps.

In addition to resource management flexibility and power efficiency, the decoupling of sandbox states from processor core and peripheral states can also help simplify the sandbox migration and fault recovery procedure. However, we have to guarantee that the dynamic resource partitioning process is predictable. The real-time requirements of the applications running in the safety critical and mission critical sandboxes can never be violated.

7.3 Fault Recovery

As mentioned in chapter 4.4, even though the sandbox isolation provided by the Quest-V separation kernel offers fault isolation and serves as a platform to construct various fault recovery mechanisms, current Quest-V implementation lacks the policy and mechanism support for general fault recovery routines. For instance, in remote recovery, the local monitor has to choose a target remote sandbox before recovery begins. There are many possible policies for choosing a target sandbox that will resume an affected service request. One simple approach is to pick any available sandbox in random order, or according to a round-robin policy. This is essentially the approach in current Quest-V implementation. In more complex decision-making situations, a sandbox may be chosen according to its current load and criticality level.

As another example, after a fault is detected, the faulting software components must be identified by the monitor. In addition to examining the system context at the time the fault happened, Quest-V should allow applications or drivers to register their dependencies if

automatic discovery is not possible. In 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.

Finally, as the first step of any fault recovery effort, fault detection is, itself, a complicated topic. In addition to the obvious fault events such as EPT violation, more elaborate schemes for identifying faults are needed. If a fault does not automatically trigger a trap into the monitor, it can be forced by a fault handler issuing an appropriate instruction. To guard against compromised sandboxes that lose the capability to pass control to their monitor as part of fault recovery, certain procedures should be adopted. One such approach would be to periodically force traps to the monitor using a preemption timeout [Inta]. 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. However, this approach forces us to give up certain level of the efficiency and safety achieved through avoiding monitor intervention in normal sandbox operations in Quest-V.

7.4 AUTOSAR Extensions Support

In chapter 5.2 we introduced how a single processor ERIKA Enterprise OSEK operating system can be bootstrapped in a Quest-V sandbox via the *kexec* command. However, as we mentioned, OSEK/AUTOSAR OS sandboxes in Quest-V currently can not easily communicate with each other and other sandboxes. In the future, we will investigate the

support of AUTOSAR OS extensions such as AUTOSAR Communication Subsystem and multi-core AUTOSAR OS in the Quest-V separation kernel.

In an attempt to harness the power of multi-core micro-controllers available in automobile industry and standardize current vehicle system architecture, the AUTOSAR standard introduced support for multi-core operating system since its 4.0 release. AUTOSAR adopted a loosely coupled design philosophy in which tasks are statically partitioned to each per core scheduler. Multi-core OS in AUTOSAR is specified as an OS that shares the same configuration and most of the code, but operates on different data structures for each core. This design simplifies the scheduling policy based on static priority while at the same time exploits the concurrency provided by multi-core processors. An example natural task partitioning might be powertrain angle versus time control tasks.

Besides providing separate schedulers for each core in a multi-core processor, AUTOSAR multi-core OS specification also added several extra system services for cross core communication and synchronization. Part of the existing system services are also extended or adapted to accommodate additional states. Communication between different OS applications (an application is a collection of Tasks) are handled with different frameworks. Communication between applications over core boundaries in multi-core systems is handled by Inter OS Application Communicator (IOC). Communication between separate ECUs is handed to AUTOSAR Communication Subsystem.

It is not hard to see that the AUTOSAR multi-core OS design adopts the distributed system on a chip philosophy with explicit communication framework. This coincides with the Quest-V separation kernel model. By providing the AUTOSAR multi-core specific support in each monitor (extra services and service extensions), we should be able to convert existing single core AUTOSAR OS implementations into multi-core compliant system with minimum overhead and engineering effort. The AUTOSAR multi-core utili-

ties and IOC interface implementations should be provided to existing OSEK/AUTOSAR OSes (e.g. ERIKA Enterprise OS) through libraries. By linking with these libraries, OSEK/AUTOSAR applications running on Quest-V in different sandboxes will be able to take advantage of the AUTOSAR multi-core extensions and IOC communication framework even if each sandbox is still running a single processor OSEK/AUTOSAR OS kernel. The support for AUTOSAR Communication Subsystem should also be extended to Quest and Linux sandboxes for mixed criticality applications in the automobiles industry.

7.5 Distributed Programming Model

Parallel applications developed for the Quest sandbox can be deployed across multiple sandboxes. Different processes can communicate with each other via the predictable communication framework described in chapter 4.2. However, due to the distributed design of the Quest-V separation kernel, the programming model for cross sandbox parallel application development is complicated. Communication channels are managed manually and mappings of processes to sandboxes have to be resolved manually before execution. A new programming model and runtime support are necessary to simplify the cross sandbox parallel application development in Quest-V.

Instead of using C and the POSIX thread (*pthread*) model, a concurrent programming language with explicit communication interface would be more appropriate. Features from existing concurrent languages such as Erlang [ERL] and Elixir [ELI] should be considered. The predictable communication framework utilities should be integrated into the language semantics. Additionally, a system runtime deployed across all Quest sandboxes is also necessary to distribute threads and processes of an application onto multiple Quest sandboxes efficiently and transparently. A policy that considers sandbox workload, resource contention, communication overhead, and device locality should be investigated to opti-

mize the system resource utilization and application performance.

Bibliography

[AA06] Keith Adams and Ole 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, 2006.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[Alb99] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pages 248–259, November 1999.

[Ame81] Stanley R. Ames, Jr. Security Kernels: A Solution or a Problem? In *IEEE Symposium on Security and Privacy*, pages 141–141, April 1981.

[And72] J. P. Anderson. Computer Security Technology Planning Study. *ESD-TR-73-51, Volume I and II, AD 758206, AD 772806*, October 1972.

[ARI08] ARINC 653 - An Avionics Standard for Safe, Partitioned Systems. Wind River Systems / IEEE Seminar, August 2008.

[AUT] AUTOSAR: AUTomotive Open System ARchitecture. `http://www.autosar.org`.

[Aut92] Federal Aviation Authority. Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA/DO-178B, RTCA, Inc, 1992.

[Avi67] Algirdas Avižienis. Design of Fault-tolerant Computers. In *Proceedings of the Fall Joint Computer Conference*, pages 733–743, Anaheim, California, 1967.

[Avi75] Algirdas Avižienis. Fault-tolerance and Fault-intolerance: Complementary Approaches to Reliable Computing. In *Proceedings of the International Conference on Reliable Software*, pages 458–464, New York, NY, USA, 1975.

[Avi85] Algirdas Avižienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, pages 1491–1501, 1985.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A Research Agenda for Mixed-Criticality Systems. 2009.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh 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.

[BBM+12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *the 10th USENIX conference on Operating Systems Design and Implementation*, pages 335–348, 2012.

[BD14] Alan Burns and Robert I. Davis. Mixed Criticality Systems - A Review. Technical report, University of York, 2014.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey 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.

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–156, 1997.

[BHR89] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple Operating Systems on One Processor Complex. *IBM Systems Journal*, 28:104–123, March 1989.

[BL74] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corporation, October 1974.

[BR13] Reto Buerki and Adrian-Ken Rueegsegger. Muen - An x86/64 Separation Kernel for High Assurance. Technical report, University of Applied Sciences Rapperswil (HSR), 2013.

[BWCC+08] 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. Corey: An operating system for many cores. In *the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[CPS+13] Marco Caccamo, Rodolfo Pellizzoni, Lui Sha, Gang Yao, and Heechul Yun. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[CRD+95] 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.

[Cre81] R.J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25:483–490, 1981.

[CRM10] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *the European Dependable Computing Conference*, pages 67–72, 2010.

[CS07] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *International Conference on Supercomputing (ICS '07)*, pages 242–252, June 2007.

[DLS97] 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.

[DLW11] Matthew Danish, Ye Li, and Richard West. Virtual-CPU scheduling in the Quest operating system. In *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*, pages 169–179, 2011.

[DOC] Docker. `https://www.docker.com/`.

[DSN06] Haakon Dybdahl, Per Stenström, and Lasse Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *High Performance Computing*, volume 4297/2006, pages 22–34, 2006.

[ELI] Elixir Programming Language. http://elixir-lang.org/.

[ERI] ERIKA Enterprise. http://erika.tuxfamily.org/drupal/.

[ERL] Erlang Programming Language. http://www.erlang.org/.

[GAH+12] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 411–422, 2012.

[GLSS77] B. D. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. VM/370 Security Retrofit Program. In *Proceedings of the Annual Conference of the ACM*, pages 411–418, Seattle, Washington, 1977.

[Hab08] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008:8, 2008.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The Performance of μ-kernel-based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–77, 1997.

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

[INTb] INTEGRITY-178B RTOS. http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[Iye04] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *the 18th Annual International Conference on Supercomputing*, pages 257–266, 2004.

[JAI] Jailhouse Partitioning Hypervisor. https://github.com/siemens/jailhouse.

[JPN+10] Adhiraj Joshi, Swapnil Pimpale, Mandar Naik, Swapnil Rathi, and Kiran Pawar. Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system. pages 101–108, 2010.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architectures and Compilation Techniques (PACT '04)*, October 2004.

[KEH+09] 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.

seL4: Formal verification of an OS kernel. In *the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[KWMH96] John G. Kassakian, Hans-Christoph Wolf, John M. Miller, and Charles J. Hurton. Automotive Electrical Systems Circa 2005. *IEEE Spectrum*, 33(8):22–27, August 1996.

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LSK04] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *International Symposium on High-Performance Computer Architecture*, pages 176–185, 2004.

[LWCM14] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable Communication and Migration in the Quest-V Separation Kernel. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS)*, Rome, Italy, December 2-5 2014.

[LWI] lwIP: http://savannah.nongnu.org/projects/lwip/.

[LWS+74] Steven B. Lipner, William A. Wulf, Roger R. Schell, Gerald J. Popek, Peter G. Neumann, Clark Weissman, and Theodore A. Linden. Security Kernels. In *Proceedings of the American Federation of Information Processing Societies National Computer Conference*, pages 973–980, May 1974.

[LXC] Linux Containers. `https://linuxcontainers.org/`.

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

[MD79] E. J. McCauley and P. J. Drongowski. KSOS - The Design of a Secure Operating System. In *Proceedings of the American Federation of Information Processing Societies Conference*, volume 48, pages 345–353, 1979.

[MEN] Mentor Embedded Hypervisor. `http://www.mentor.com/embedded-software/hypervisor/`.

[Mil76] Jonathan K. Millen. Security Kernel Validation in Practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[MS70] R.A. Meyer and L.H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9:199–218, 1970.

[MUE] Muen Separation Kernel. `http://muen.codelabs.ch/`.

[MUL] INTEGRITY Multivisor. `http://www.ghs.com/products/rtos/integrity_virtualization.html`.

[NB13] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *the 24th ACM Symposium on Operating Systems Principles*, pages 116–132, 2013.

[NEX] Nexans 2014 Press Release. `http://www.nexans.com/`.

[NH00] Paul Nicastri and Henry Huang. 42V PowerNet: providing the vehicle electrical power for the 21st century. Technical report, SAE International, 2000.

[NSN+11] Yoshinari Nomura, Ryota Senzaki, Daiki Nakahara, Hiroshi Ushio, Tetsuya Kataoka, and Hideo Taniguchi. Mint: Booting Multiple Linux Kernels on a Multicore Processor. In *Proceedings of the International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 555–560, Washington, DC, USA, 2011.

[OSE] OSEK: Open Systems and their Interfaces for the Electronics in Motor Vehicles. `http://www.osek-vdx.org/`.

[PCI] Pci. `http://wiki.osdev.org/PCI`.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17:412–421, 1974.

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

[PKK+79] Gerald J. Popek, Mark Kampe, Charles S. Kline, Allen Stoughton, Michael Urban, and Evelyn J. Walton. UCLA Secure UNIX. In *Proceedings of the American Federation of Information Processing Societies Conference*, volume 48, pages 355–364, 1979.

[POP] Popcorn Linux. `http://popcornlinux.org/`.

[PUP] Puppy Linux. `http://www.puppylinux.org`.

[PvSK90] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of Safety-Critical Software. *Communications of the ACM*, pages 636–648, June 1990.

[RAJ00] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *the 27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[RLT06] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Parallel Architectures and Compilation Techniques (PACT '06)*, pages 2–12, September 2006.

[Rus81] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 12–21, 1981.

[Rus08] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[SB96] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

[SBWH10] Mark Stanovich, Theodore P. Baker, An I Wang, and Michael Gonzalez 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.

[Sim90] H.R. Simpson. Four-slot Fully Asynchronous Communication Mechanism. *IEEE Proceedings Part E: Computers and Digital Techniques*, 137(1):17–30, Jan 1990.

[SKI08] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set pinning: Managing shared caches in CMPs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, March 2008.

[SKLR11] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 401–412, 2011.

[SMI08] T. Shimosawa, H. Matsuba, and Y. Ishikawa. Logical Partitioning without Architectural Supports. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 355–364, July 2008.

[SOL] Solaris Zones. http://docs.oracle.com/cd/E26502_01/html/E29024/toc.html.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 275–287, Lisbon, Portugal, 2007.

[SRD04] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, April 2004.

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

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[VIR] VirtualBox. https://www.virtualbox.org/.

[VMW] VMware Workstation. http://www.vmware.com/products/workstation.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (FOS): The case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43:76–85, 2009.

[Wat06] C. B. Watkins. Integrated Modular Avionics: Managing the allocation of shared intersystem resources. In *Proceedings of the 25th Digital Avionics Systems Conference*, pages 1–12, 2006.

[WHD+09] D. Williams, Wei Hu, J.W. Davidson, J.D. Hiser, J.C. Knight, and A. Nguyen-Tuong. Security through Diversity. *Security & Privacy, IEEE*, 7:26–33, Jan 2009.

[WIN] Wind River Hypervisor. http://www.windriver.com/products/hypervisor/.

[WZW+08] Richard West, Puneet Zaroo, Carl Waldspurger, Xiao Zhang, and Haoqiang Zheng. 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.

[WZWZ10] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4), December 2010. Special VMware Track.

[WZWZ13] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. *Multicore Technology: Architecture, Reconfiguration and Modeling*, chapter 8. CRC Press, ISBN-10: 1439880638, 2013.

[YWCL14] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System using Page Coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Alberta, Canada, August 24-27 2014.

[ZGKR14] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–31, Broomfield, CO, October 2014.

[ZW06] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *the 27th IEEE Real-Time Systems Symposium*, December 2006.

Curriculum Vitae