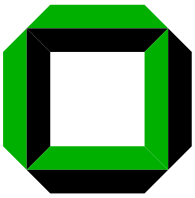


Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Hardware-Supported Virtualization for the L4 Microkernel

Sebastian Biemüller

Diploma Thesis

Verantwortlicher Betreuer: Prof. Dr. Frank Belosa
Betreuender Mitarbeiter: Dipl. Inf. Jan Stöß

29. September 2006

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 29. September 2006

_____

Sebastian Biemüller

Abstract

Despite the immense popularity virtual machines regained in the last years current virtualization environments force significant compromises to the host system architecture. Hypervisor-based virtual machine environments are missing system construction principles; they only provide the coarse grained abstraction of a virtual machine, preventing small and efficient systems. Hosted virtual machine environments reuse the host's operating system services and thereby massively increase the trusted code base, preventing small, reliable and verifiable systems.

We propose a novel approach for the construction of virtual machine environments: We divide the virtual machine environment into a necessarily privileged part, and a user-level monitor. The privileged part and microkernels share a common set of goals such as reliability, security, and isolation so that integrating virtual machines and microkernels is a promising approach. In fact, modern microkernels, such as the L4 microkernel, already provide the abstractions and mechanisms necessary to cater for virtual machines. We identify shortcomings of the current kernel with respect to virtual machine support and add a minimalistic set of extensions. As a normal microkernel application, the user-level monitor can interact with services provided by other components to maintain the virtual machine.

To demonstrate our approach we implemented a prototype virtualization environment on top of the modified microkernel. We successfully run a guest operating system, utilizing all types of services side-by-side with other, native microkernel applications.

Our design preserves the performance of the microkernel, especially the mechanism for inter-process communication needed no adaption. Measurements indicate only minimal side-effects caused by the increased hardware-overhead of the world-switch needed for the virtual machine environments.

Contents

Chapter 1

Introduction

For the last 40 years the complexity of computer hardware increased exponentially driven by Moore's famous law [70] constantly opening fields where computer systems become attractive. Unfortunately, software systems do not keep up with this rapid evolvement resulting in increased pressure on software systems [29] to adapt to these new hardware features and efficiently utilize them. This holds especially for operating systems; they execute directly on the bare hardware to manage and abstract hardware specifics behind a convenient, portable interface which user applications rely on. This thesis combines two approaches which address the stated problem from two different directions.

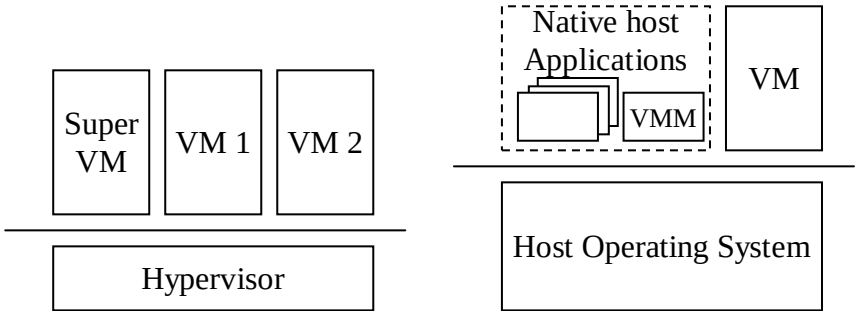
Virtual Machines first introduced by IBM in the 1960s were invented to provide multiple users a seemingly separate, computing system to overcome the limits of former single-user operating systems. In the last 40 years, virtual machines have matured in providing a faithful illusion of a complete machine platform; they now allow legacy reuse of a complete software stack; presenting an attractive approach to solve a variety of problems in dissimilar contexts such as exploiting new hardware features [15], device driver reuse [57], workload consolidation [25] and migration [20], secure computing [69].

Component-based operating systems increase flexibility of system software construction by separating the operating system into isolated components. The base of the system is a microkernel. It only implements the minimal necessary concepts such as computation, isolation, and communication to establish independence and integrity between different subsystems. The goal is to allow construction arbitrary systems on top. Microkernel-based systems allow stepwise innovation in OS technology to cope with the ever-increasing complexity of operating systems.

1.1 The Problem

Systems which are based on only one of the both approaches, either the microkernel system, or the virtual machine system are overly restrictive and do not achieve the required flexibility.

Systems solely based on the virtual machine abstraction, such as the hypervisor-based approach (Figure 1.1(a)) affect the whole system. A small, privileged hypervisor provides a reliable system base. It implements only a single interface, the virtual machine. The virtual machines are isolated from each other and securely executed using



(a) Hypervisor Architecture (b) Hosted Architecture

Figure 1.1: Virtualization Architectures. Missing: The different interfaces provided by the different approaches.

the host machine's resources. The virtual machine as a faithful duplicate of a real machine is often too coarse grained to represent small software components efficiently. This even holds for the virtual machine management logic, which is run in a privileged component, the *super VM*. The super VM often runs a commodity operating system to implement services such as the emulation of the virtual machine environment of the other VMs. Faults in its implementation [19] can thereby affect other virtual machines as well, harming the strong isolation criteria required for the virtual machine environment.

To overcome the construction problems of the virtual machine management logic the *host-based* approach (Figure 1.1(b)) integrates the privileged hypervisor into a host operating system, which directly executes on the physical machine. The host operating system services can be used to implement the virtual machine monitor as an in-kernel module or a user-level application; this avoids the second costly world switch into the super-VM. However, this model has serious drawbacks, too. The host operating system greatly increases the privileged part, and thus the trusted computing base of the system [26, 78]. The reliability and secure isolation of virtual machines is only as hard as the general security of the host operating system. Additionally, running the virtual machine monitor on top of the host operating system increases overhead drastically [26].

The virtual machines strongest point becomes also its weakest: focusing on reuse and isolation they do not offer a core abstraction for communication with the system surrounding the virtual machine. The communication is often emulated via special devices which are part of the platform provided to the guest. It significantly increases the necessary trusted computing base.

On the other hand we presented microkernels. Microkernels are designed to form a flexible system base. Like all other operating systems they define a convenient interface to their user-level applications. This interface is close, but not equal to the hardware platform, preventing full software reuse – requiring porting.

1.2 Approach

In this thesis, we unify the virtual machine- and the microkernel environment in one system. We present a new scheme to construct virtual machine systems: Based on the

concepts of microkernels we construct whole-system virtual machine systems [81] to overcome the deficits of a solely hosted or hypervisor-based approach.

Our approach separates the virtual machine monitor into a necessarily privileged part, called the hypervisor, and an unprivileged part, the user-level monitor. Our approach is based on the thesis that: *hypervisors and microkernels are similar enough to justify integration of both*. The resulting system comprises a microkernel that also provides support for virtual machines and a microkernel-based system on top which includes components for maintaining the virtual machines. Such a system combines the best of both worlds: Virtual machines provide strongly isolated containers with a stable, compatible interface, the hardware platform API for legacy reuse, whereas the microkernel approach enables construction of arbitrary, well-structured systems.

Our design is based on a particular microkernel, the L4 microkernel. The resulting microkernel has only marginally increased complexity caused by the mechanism to support user-level management of the virtual machine, the protocol to communicate virtualization critical events, and the enhanced execution context handling required by the world switches from/to virtual machines. The virtual machine representation in L4 is highly integrated, introducing only minimal changes to the microkernel API. Like all other application workload the virtual machine is described by an L4 address space, which represents the isolation domain, holding all accessible physical resources. L4 threads embody the processor of the virtual machine directly executing the virtual machine's instruction on the physical processor. The communication of virtualization critical events is completely via synchronous IPC; it models the fault like behavior.

The user-level monitor application uses the IPC based virtualization-fault protocol to control the virtual machine. Being a native L4 application, the monitor can use L4's usual mechanisms to control the virtual machine environment. Guest physical memory is provided by L4's mapping mechanism; execution of the virtual machine underlies the usual thread scheduling provided by L4.

We demonstrate the feasibility of our approach by integrating support for full virtualization of the x86 architecture using Intel's virtualization technology. The required changes to the L4 microkernel did not negatively affect the performance of native L4 applications; thus independence is sustained. Especially the inter-process communication mechanism need no modifications. Only minimal overhead is introduced by the unavoidable hardware world switch for entering and exiting the virtual machine.

Outline

This thesis is structured as follows:

Chapter 2:

In the background chapter we give an overview of virtualization and microkernel-based systems. We analyse the requirements of virtualization, describing different virtualization systems and discuss their approaches and resulting system architectures. Further, we introduce the idea of microkernel-based systems describing its goals and concepts and the resulting system architecture. We focus on a particular microkernel, the L4 Microkernel.

Chapter 3:

In the design chapter we present our approach of integrating virtualization features into the L4 microkernel. We present our L4-based virtualization system

architecture focusing on the kernel representation of a virtual machine using microkernel concepts.

Chapter 4:

In the implementation chapter we offer an overview of our prototype implementation of the microkernel extensions and a user-level virtual machine monitor application.

Chapter 5:

In this chapter we present the evaluation of our prototype implementation.

Chapter 6:

Finally includes a summary and gives suggestions for future work.

Chapter 2

Background & Related Work

This thesis brings together two research areas: virtual machines and microkernel-based systems. In Section 2.1 we discuss the general problem of virtualization by reviewing previous work: on the theory of virtualization, the requirements of virtualization, and different virtualization approaches including their resulting architectures. Section 2.2 describes the state of the art in microkernel-based systems laying the ground for many design decisions. We briefly introduce the L4 microkernel paying special importance to virtualization critical mechanisms, such as the representation of hardware mechanisms.

2.1 Virtual Machine Environments

Virtual machine (VM) systems are a major development in computer system designs. By providing a duplicate of one or more computer systems, virtual machines extended computer systems from multi-access, multi-programming, multi-processing to multi-environment architectures. Concurrency, previously only available at a higher level becomes available at the lowest level — the system-level. Virtual machines adopt the common theme of adding an layer of indirection to increase the flexibility.

The term *Virtual Machine* was introduced by IBM in the 1960s; IBM used virtual machines to provide multi-user systems by securely multiplexing several instances of a single-user operating system; each OS running in a virtualized duplicate of the real machine [21]. In the last 40 years the virtual machine technique has matured in providing the illusion of an isolated and exclusive hardware platform to its guest's software stack. Virtual machines represent an attractive approach to solve a variety of problems in quite different contexts:

- Exploiting new hardware features without requiring changes to the software stack [15, 16]
- Improved testing of system software [54]
- Providing a high degree of reliability and security [13, 17, 30, 31, 44, 45, 69, 79]
- Running multiple, different software stacks concurrently on one physical machine e .g . to achieve consolidation or operating system diversity [25, 96]
- Running virtual configurations which are different from the physical system to offer services like migration and ubiquitous computing [17, 20]

2.1.1 Theory of Virtualization

A Virtual Machine as defined by Goldberg [33] is an "efficient hardware-software duplicate of a real machine". The idea of virtual machines is to create the illusion of a physical machine at the lowest level, the platform API. Thus, communication between the guest (operating) system executing inside the VM and the virtual machine environment is completely defined by the behavior of the hardware interface.

The virtual machine monitor (VMM) provides the illusion of the VM's environment using the resources of the host system. It has full control of the virtual machine and can establish full isolation or controlled sharing of resources between the VM, itself, and the rest of the system. The environment of the VM includes all resources of a physical machine: the processor, memory, interrupt lines, IO ports and thus devices. The virtual machine monitor has three characteristics [74]:

- **Equivalence.** The guest running in the virtual machine expects to execute on a physical machine. In order to execute properly the environment provided by the VMM, the virtual machine, needs to represent all aspects of a physical machine. Only the exact timing behavior is impossible to achieve — multiple virtual machines execute concurrently, and the VMM itself needs some time to create and maintain the virtual machine's environment.
- **Resource Control.** The VMM must be in control of the physical system. (i) No virtual machine is allowed to directly access a physical resource not explicitly allocated to it. (ii) The virtual machine monitor needs to be able to transparently regain control of a previously allocated resource.
- **Efficiency.** The virtual machine provides the same hardware platform as the host machine. Efficiency can be achieved by securely using physical resources of the host whenever possible. This is especially important for the virtual machine processor. A "statistically dominant subset of the virtual processor's instructions needs to be executed directly on the real processor" [74].

Virtualization of System Resources

To create the illusion of an exclusively available physical machine all machine components need to be virtualized. The virtualized components must follow the stated characteristics of equivalence, controllability, and efficiency. Depending on their type resources can be virtualized in several ways [34]; Figure 2.1 depicts an overview:

A virtualized resource is represented by a *renamed* physical resource. Renaming enables different objects of the same resource class to represent each other. For example, renaming of physical memory can be provided by virtual memory. Different physical memory frames can be used to represent the same page frames in different virtual machines.

If a physical resource is completely preemptible one instance of a physical resource can be *multiplexed* between different virtual machines; for example, multiplexing of a processor register between different threads. A special case of multiplexing is when the resource is exclusively assigned to one user, here the state need not necessarily preemptible.

If a resource can not be renamed or securely multiplexed the resource needs to be *emulated*. Emulation of resources is usually done by trapping instructions which access these resources. The effect on the virtual machine environment is then emulated in software.

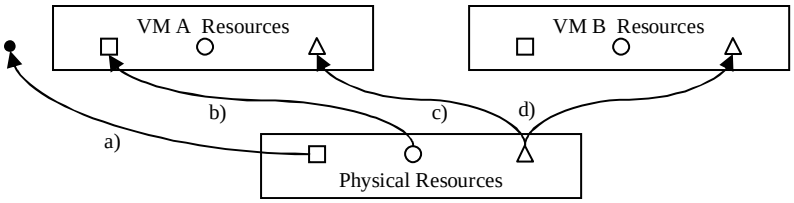


Figure 2.1: Resources of a virtual machine can be provided in several ways. Depending on the characteristics of a given resource it can be: (a) emulated, (b) remapped, (c) mapped, and (d) multiplexed.

The Virtual Processor

The physical processor is the key to the virtualizability of the system. It must provide trap semantics that let a VMM safely, transparently, and directly use the real processor to execute the virtual machine. With these traps on virtualization critical conditions the VMM can use direct execution to create the illusion of a normal physical machine for the software running inside the virtual machine. These traps include external events, related to the physical machine such as interrupts, which must be sheltered from the virtual machine and be delivered to the VMM, and internal events such as the execution of *sensitive* instructions. When the virtual machine executes directly on the physical processor, some instructions – instructions which leak the state of the physical machine uncontrolled into the virtual machine – can not be issued; these instructions are called *sensitive*. Sensitive instructions must be prevented from being executed directly. If all critical instructions are trap-able, the processor is called *fully-virtualizable*.

2.1.2 The Virtual Machine Monitor

The virtual machine monitor functionality can be divided into several modules [74]. The *dispatcher* is the first module. It starts the VMM and initializes its environment. The dispatcher represents the entry-points for the different traps of the physical processor. The dispatcher therefore implements secure multiplexing of the virtual machines. The *allocator* manages the physical resources. It keeps track of the resource usage to guarantee controlled isolation. The *interpreters* are a set of routines which emulate the behavior of the trapped, sensitive instructions.

Architecture

In the *hosted* architecture the virtual machine monitor runs on top of a host operating system. It can use the host operating system's services to implement the virtual machines. The monitor can reuse the host operating system's abstraction and services such as scheduling and resource allocation to implement the virtual machines. On the other hand the monitor may loose full control of the host's hardware to the host operating system; flaws in the host operating system undermine the security of the complete virtual machine system [78]. To minimize the side effects of software not directly related to the virtual machines, the *hypervisor* based architecture minimizes the privileged code running directly on the physical machine. The privileged hypervisor implements only basic scheduling and isolation mechanisms of the dispatcher module. The other components are located in a *super*-VM located on top of the hypervisor. This privileged

virtual machine often contains a classic monolithic OS which includes the device driver emulation and VM management interface.

2.1.3 Selected Virtualization Environments

Tons of virtual machine systems have been developed each for its own special purpose. We classify these approaches into para- and full-virtualization systems and discuss the various approaches regarding their special goals.

Para-Virtualization: Environments

Para-virtualization systems [95] require source code access to the guest system to change it. Para-virtualization does not have the goal to provide an exact duplicate of the platform API. It follows the opposite approach: each guest is ported towards the virtualization system by replacing virtualization critical sensitive operations with high-level hypercalls to the VMM. These changes are often guest specific; para-virtualization requires intimate knowledge of the guest system resulting in high engineering effort [56]. Depending on the required characteristics research found various approaches. Having knowledge of the guest operating system's behavior hypercalls can even be delayed and batched until an unavoidable call to the VMM is necessary. This is used, for example, in the Xen hypervisor [25]. This reduces the number of domain switches and thus increases performance. Stepwise introduction of special hypercalls, the approach presented in [67], removes performance bottlenecks. A host based para-virtualization environment is User Mode Linux, it runs a modified Linux on top of the Linux operating system [24].

VMI [91] provides a generic para-virtualization interface which still requires porting of the guest. It focuses on the unification of the interface between the VMM and the guest for x86 based systems. The OS implementors simply change their operating system to call specific VMI routines stored in a ROM to make their system virtualization friendly. The routine transparently handles virtualization. It allows to switch between different VMMs implementing the VMI interface and native operation.

Full-Virtualization: Environments

Full virtualization environments do not require source code access to the guest; they focus on providing a faithful illusion of the platform API. Modification of the guest's behavior is only possible by using special device drivers for a specific guests, e .g . to provide custom high performance devices such as network cards [52], or to reclaim memory using memory balloons [94]. Prominent full-virtualization environments are the host-based VMware Workstation and the hypervisor-based VMware ESX server [92]. Virtual machines core functionality is to provide an isolated environment.

Emulation, Simulation

On processors not fully-virtualizable sensitive instructions must be prevented to appear in the VM's instruction stream. The most relevant technique is *para-virtualization* [25, 56, 96]; it needs source-code availability of the guest software to patch the sensitive instructions with fix-ups. *Binary rewriting techniques* [7, 23, 80] instead use complex trapping of memory references to transparently change the guest's instructions. *Whole-system simulation* such as Simics [90], QEMU [12], and Bochs [55]

interpret the complete virtual instruction set, such approaches, although very powerful impose huge performance hit and thus violate Golberg's efficiency requirement [35]. In this thesis we focus on fully-virtualizable processors.

2.2 The L4 Microkernel

The L4 microkernel [63] is a second generation microkernel originally developed by Jochen Liedtke at GMD, IBM, and Karlsruhe University. Various versions exist at Karlsruhe University, UNSW/NICTA Sydney, and Dresden University of Technology [83].

2.2.1 The Microkernel Argument

In contrast to monolithic systems, the microkernel-based approach implements operating system functionality like scheduling policies, network services or device drivers into dedicated *user-level* servers. To isolate them these servers are located in different address spaces and provide their services by inter-process communication. This construction principle is very flexible; the system services can be replaced, or run side-by-side like normal applications, allowing coexistence of different resource policies on one system [62] specialized for different applications [53, 65]. The system base forms a microkernel. It implements system-wide features like isolation, resource management, communication, and integrity. The goal of a microkernel is to be policy free and to provide system construction principles. The microkernel is tiny and thus can be made secure [42, 86]. Beginning with Mach's user-level pagers [77, 97]; current second generation kernels provide even more complete resource management in user-level components [40, 62]. The most critical kernel mechanism is IPC; it is master to the functionality and flexibility of the microkernel [59, 64] and the component system on top [14, 32].

2.2.2 Abstractions

Threads are the abstraction of an activity. CPU time is multiplexed between threads bound to the same processor. An L4 thread is represented by its register state (processor registers and virtual registers), a unique global identifier, and an associated address space.

Address spaces provide the abstraction for protection and isolation; resource permissions are bound to an address space. L4 address spaces are no first class object; they are indirectly identified via a thread associated to this particular space. All threads in an address space have the same rights and can freely manipulate each other.

2.2.3 Primitives

Inter process communication (IPC) is the mechanism for data transfer and controlled execution transfer between threads. Message transfers are synchronous and involve exactly two threads. Both sender and receiver have to agree on the format of the message. IPC is the master to the microkernel design [59].

Resource Delegation is the mechanism for controlled transfer of resource permissions between address spaces. Access to a resource is granted by transferring a *map* or *grant* item in an IPC message, and thus allows user-level management of address spaces. Mapping requires mutual agreement of the sender and receiver thread. Map duplicates the resource permissions from the sender's into the receiver's address space; grant moves the permission. The receiver's permissions can only be a subset of the sender's permissions. Mapping can be applied recursively. Revocation of resource rights is done asynchronously through the *unmap* primitive and does not require explicit consent from the receiver of the mapping.

Scheduling. L4 has an in-kernel round-robin *scheduler* that allocates time to threads according to their priority and time-slice length. If the time slice of a thread expires, L4 preempts the thread and schedules the next runnable thread.

Protocols. Hardware-generated events such as *exceptions* and *interrupts* are translated into kernel-generated IPC messages. On an hardware interrupt, the kernel synthesizes a message to a thread that is registered as the handler for that interrupt. The sender appears to be a thread with a special per-interrupt thread identifier. Hardware exceptions are transparent to the faulting thread; the kernel preserves the thread's context. The hardware exceptions are mapped onto an IPC based fault protocol. In the name of the faulting thread, the kernel synthesizes a message with information about the cause of the fault and sends it to the faulting thread's exception handler. The faulting thread is automatically set into a blocking IPC receive operation, waiting for a reply from the exception handler to resume execution. On a page fault exception, the fault message is sent to the pager of the thread, expecting a memory mapping in the reply. The special treatment of the page fault exception has historical reasons.

These protocols allow easy virtualization of physical resources, e.g., paged virtual memory [9, 40]: Under memory pressure, the provider of a page unmaps it from the address space it was mapped to. If a thread now accesses the removed page, a page-fault IPC is sent so that the pager can transparently re-establish the mapping and resume the faulting thread.

2.3 Microkernels and Virtual Machines

There already exists related work which integrates the fields of virtual machines and microkernels. In [39] and [43] the authors argue that microkernels and virtual machines, although their definition is quite different, follow similar goals. The work presented in [68] reasons towards a microkernel-based virtual machine systems for high performance computing.

Extended Virtual Machine Monitor. The work of Hohmuth et . al [46] explores the design space of hybrid virtual machine/microkernel systems with the goal to minimize the trusted computing base (TCB) required to implement the virtual machine environment. It is found that a VMM extended by mechanisms for memory sharing and communication can reach a very small TCB.

Para-virtualization. The L4 microkernel already serves as a host for virtual machines. The L4Linux [41] system, is a para-virtualized Linux kernel, ported to the L4's

API interface. It achieves near native performance.

The L4Ka::Virtualization environment [85] also support the L4 microkernel, as a host. The authors separated the VMM into an isolation critical resource monitor and an uncritical in-place component. The in-place component is co-located to the guest, it can use detailed knowledge of the guest to create the illusion of the virtual machine environment. It uses a technique called after-burning to replace sensitive instructions of the guest with function calls to the in-place component's interpreters which avoids hypercalls. Allocation of physical resources that can not completely handled inside the in-place monitor are translated to resource monitor requests. This indirection allows for (i) guest specific optimizations in the in-place monitor and (ii) host system diversity. The environment supports the L4 microkernel [84] and the Xen hypervisor [25] as a host.

The Spine system [37] forms a reliable system base for an intrusion recovery system. It supports detection and recovery from kernel-level and user-level root-kits. The authors choose a microkernel, the L4 microkernel as the base hypervisor component. The authors argue that only a microkernel achieves the correctness, isolation and performance requirements. The microkernel was preferred over a virtual machine approach because the authors wanted a simple architecture that does not sacrifice performance and support for multiple operating systems. As guest operating systems, to prove recovery services, it uses para-virtualized L4Linux instances.

Fluke. The Fluke system developed by Bryan Ford et al . [28] is a software-based virtualizable architecture. It combines the concepts of microkernels and virtual machines to increase operating system extensibility. The operating system functionality is decomposed "vertically" into layers. In these layers, called *nesters*, the environment, provided to the application can be stepwise refined; for example a *demand paging nester* adds anonymous memory, or a *checkpoint nester* allows defined restart of the application after a system failure. The application's execution environment consists of the hierarchy of nesters it runs on. Thus the application environment only includes the required operating system services.

The prototype system implementation is based on the x86 architecture – on which the fluke microkernel runs. Fluke provides a well defined subset of the x86 architecture to the nesters and applications on top to avoid the trapping of traditional virtual machine systems. Additionally, the fluke kernel adds a low-level interface including thread, address space and IPC mechanisms to the execution environment allowing convenient system construction. To overcome the exponential overhead of virtual machine stacking, the services of Fluke's microkernel are designed to support recursive systems: A system component has full access to all its children. For example recursively defined memory management through remapping, which is based on L4's model. All system services are based on capabilities, which allows to short-circuit the hierarchy of nesters when appropriate.

2.4 Summary

In this chapter we presented related work on virtual machine systems. The virtual machine abstraction represents an isolated duplicate of a physical machine. A virtual machine has three properties: equivalence, resource control and efficiency.

We showed that previous approaches to support virtual machines suffer from at least one of the following deficiencies:

- Para-virtualization systems require porting of at least the guest operating system towards the host system environment. Thus depend on source code access preventing full legacy reuse.
- Hypervisor-based VMMs only allow to run workload embedded in virtual machines. The virtual machine abstraction is insufficient as a general abstraction to construct system services running besides the virtual machines. It is often to coarse grained, preventing fine isolation and efficient interaction between system components. This results in compromises which may harm achievable isolation.
- Host-based virtual machine monitors attach their services onto a host operating system which may cause loss of full control of the host services. The host operating system causes a significant performance overhead. It further affects reliability, being dependent on the host system services, the VMM can only be as reliable as the complex and huge host operating system.

To overcome these deficiencies this thesis proposes the usage of a microkernel-based approach to construct a virtual machine system. Microkernels provide a minimal set of flexible abstractions to construct systems on top; they offer abstractions to express isolation and execution, and mechanisms to control usage of physical resources from user-level. Microkernels and virtual machine systems share a common set of goals such as isolation and resource control. Microkernels and virtual machines thus seem candidates for a tight integration to overcome the deficiencies of both, namely lightweight subsystems and reuse.

Chapter 3

Design

In this chapter, we present a novel approach for the construction of virtual machine systems. Our approach addresses the shortcomings in flexibility of current virtualization system architectures which are either based on a hypervisor or a host operating system. Both approaches have significant drawbacks, harming the virtual machine's main design goals: isolation and robustness. We propose to use microkernel principles to construct the virtual machine system. In the related work chapter we already showed that the microkernel-based approach and the virtual machine systems share a common set of goals such as isolation, resource control, and system integrity, to justify a tight integration of both. The microkernel provides a minimal set of flexible abstractions and mechanisms to construct arbitrary systems on top. We show that modern second generation microkernels already offer the necessary concepts to construct virtual machine environments through their minimal interface. We present a design which uses the concepts of the L4 microkernel to construct a virtual machine system. The resulting system comprises of a microkernel that also provides support for virtual machines and a microkernel-based system on top which includes components for maintaining the virtual machines. It combines the best of both worlds: Virtual machines provide strongly isolated containers with a stable environment, based on the platform API; whereas the microkernel approach enables construction of arbitrary, well-structured systems. The microkernel based approach can be seen as a fusion of the benefits of the hypervisor and the hosted virtual machine architecture.

This chapter is organized as follows: In Section 3.1, we first state our thesis goals and then point out a set of requirements and goals that should apply to our solution. These requirements induce our general architecture. In Section 3.2 we roughly describe the basic architecture of our virtual machine system and the microkernel's view of the VM. In subsequent sections we refine our design by describing how the virtual machine's resources are virtualized using L4's concepts.

3.1 Design Goals

The goal of our thesis is to construct a virtual machine system which is based on the principles of microkernels. The base of our system is an extended version of the L4 microkernel. This decision is based on the thesis that a microkernel provides a more flexible set of abstractions than current virtual machine monitors. The microkernel has to provide support for virtual machines as well as for native microkernel applications.

A valid solution must comply to the microkernel principle: aiming for a minimal, flexible system base which allows construction of arbitrary systems on top [60, 62]. Our solution has to follow this approach and use the microkernel construction principle to build the virtual machine system. Only the necessary privileged mechanisms of the virtual machine monitor are allowed to reside in the privileged kernel. The criteria is functionality: A primitive is only allowed in the microkernel if implementation outside the kernel would prevent the system from being usable. The virtual machine system logic is located in components residing on top of the microkernel.

We define following requirements and goals our approach should achieve:

Preserve Native Execution Environment. Virtualization support is an extension of the microkernel functionality. Our solution has to sustain the core functionality of the microkernel; it should minimize the affect on the service quality of the other system components. Especially the performance of the fundamental mechanism of inter-process communication (IPC); IPC is master to achieve modularity, flexibility, security and scalability [59]. The goal of our design is to reduce the impact to a minimum.

Minimal Extensions. The required microkernel extensions have to be reduced to a minimum [61]. To achieve minimal extensions we require our solution to reuse already provided microkernel abstractions and mechanisms whenever possible; reuse increases efficiency and homogeneity of the overall system. Our goal is to describe the virtual machine concepts via microkernel primitives; we do not want the resulting system to be a co-location of a microkernel and virtual machine hypervisor.

Architecture Independence. The representation of the virtual machine and the mechanisms to manipulate and control its state have to be flexible. Especially they must not rely on specifics of a certain architecture. Following the microkernel principle we want to represent the virtual machine, using policy free abstractions and mechanisms [63]. This allows to apply our approach to other architectures than our target architecture for the implementation. For example, the mechanisms should hide the specifics of different hardware virtualization techniques (like Intel VT-x and AMD Pacifica).

3.2 Proposed Scheme

In this section we give an overview of our microkernel-based virtualization system. First we divide the virtual machine system into components. After that we give an overview of the microkernel's representation of a virtual machine. In the following chapters we refine the representation by mapping the resources which form a virtual machine onto the concepts offered by the L4 microkernel.

3.2.1 System Architecture

Our system is based on a microkernel. The microkernel approach gives a minimal system base, like in hypervisor-based architectures. But unlike hypervisors microkernels additionally provide system construction primitives; allowing to build arbitrary systems on top – including virtual machine environments. In our approach we divide the

virtual machine monitor into a necessary privileged part, represented by the microkernel, and an unprivileged part, the user-level monitor application. The resulting virtual machine system consists of three components:

Virtual Machine: The *virtual machine* is represented like all other user-level applications using L4 abstractions. It is a faithful duplicate of the physical machine platform.

Monitor: The *monitor* defines and maintains the virtual machine environment. It implements the allocator and the interpreters of a conventional VMM architecture: It guarantees isolation by securely controlling the VM's access to physical resources and implements all complex aspects of virtualization, such as device emulation. As a normal L4 application, the monitor can interact with services offered by other user-level components to maintain the environment of the virtual machine, for example disk storage or network connectivity [32].

Microkernel: The *microkernel* provides the execution environment for the virtual machine, the monitor, and the rest of the system. It ensures host-wide constraints such as integrity, and controlled execution of all system components. To securely execute the different components the microkernel offers abstractions for execution and isolation in which all system components are described – including the virtual machines. For our virtual machine subsystem the microkernel implements the dispatcher module of the conventional VMM architecture. It uses hardware virtualization techniques where necessary [87] to run the virtual machine. The microkernel catches virtualization critical hardware events and either handles them transparently or dispatches them to the VM's registered monitor application.

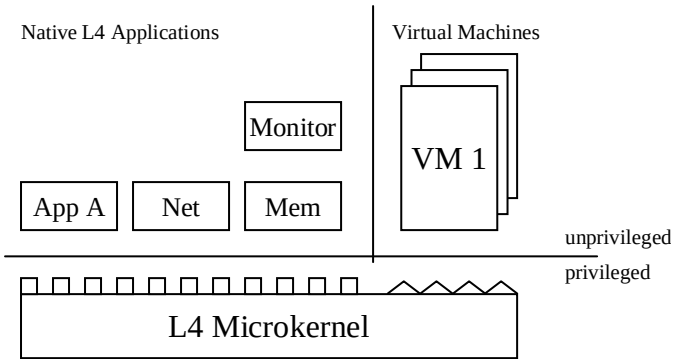


Figure 3.1: Native microkernel applications including the unprivileged part of the virtual machine monitor run side-by-side with virtual machines.

Through the remainder of this chapter we stepwise refine our design and show the virtual machine representation in the L4 microkernel.

3.2.2 Virtual Machine Representation

In this section we give an overview of our virtual machine representation in our microkernel-based virtualization system. The focus of this thesis is the description of a

virtual machine using L4 microkernel concepts including the interface which grants the user-level monitor full control of the VM's resources and execution.

A virtual machine is a hardware-software duplicate of a physical machine system. It includes all platform resources: the processor, physical/virtual memory and devices. In our system the virtual machine representation regards the requirements of three system components:

1. The guest running in the virtual machine requires an efficient, faithful duplicate of the physical machine system.
2. The monitor demands full control of the virtual machine, especially the allocated physical resources and their execution.
3. The microkernel needs a representation of the virtual machine's resources to enforce system-wide constraints such as isolation and integrity.

The L4 microkernel abstracts a virtual machine using an L4 address space. This space holds all resources directly accessible to the virtual machine. It allows L4 and the user-level monitor to enforce strict isolation. The L4 microkernel takes care that the virtual machine can only access the resources with its permissions, thereby establishing isolation and independence – core concepts of a microkernel [60].

L4 normally exports an *extended machine* interface to the user-level applications. This extended machine is similar but not identical to the underlying hardware; for instance L4 provides system calls and installs itself into each virtual address space. Native applications are aware of this changes and developed towards L4's extended machine interface. In a virtual machine environment this is impossible as the guest running in the virtual machine expects an environment which is a faithful duplicate of a real machine. For the virtual machine we introduced a new address space mode, the virtualization mode. In this mode, code executing does not experience L4's extended machine interface but the behavior of the platform API.

Resources. At the time of creation the virtual machine, its address space is completely empty and holds no permission to any physical resources. This establishes strong isolation of the virtual machine; the virtual machine can only operate on resources explicitly allocated to it. The monitor application can then selectively populate the virtual machine with resources, thereby defining the VM's environment.

Depending on the type of resource L4 already provides the required mechanisms of renaming, multiplexing and emulation of resources. In our virtual machine system resources can be divided into several groups:

- **Memory.** Resources with semantics like memory such as physical memory and IO ports can be controlled using L4's resource mapping mechanisms. The user-level monitor uses the already available L4 mechanisms of map/grant and unmap to fully control the access of physical resources in the virtual machine [9, 40].

  The virtualization of memory-based resources is detailed in the Section 3.3.1 for physical memory and Section 3.3.2 for virtual memory.

- **Processor.** The most critical resource is the processor as virtualization is processor-centric [74]. The VM's processor (VCPU) is fundamental to the performance of the virtual machine. In our L4 based virtual machine system, we represent the VCPU with an L4 thread. It allows L4 to securely multiplex the VCPU onto the

physical processors like all other threads in the system. For virtual machines we extend the thread abstraction to hold the complete processor state to handle the world-switch into a virtual machine.

For instance, the monitor application can use L4's thread manipulation primitives such as: create, delete, start and stop to control the execution, which is detailed in Section 3.3.3.

- **Peripheral Devices.** From the view of virtualization peripheral devices are a combination of memory, IO-ports and interrupts. The virtual machine can have either direct physical access as already supported for native L4 applications, or the device can be emulated in software in the basis of instruction faults.

  In Section 3.3.4 we detail device related specifics.

The user-level monitor can freely define which physical resources are available to the virtual machine and thereby trade the size of a virtual machine with the efficiency; as physical resources avoid time-consuming emulation. In Section 3.3 we refine resource control and discuss the representation of resources for our virtual machine system.

User-Level Control Protocol. Virtualization is processor-centric, the user-level monitor needs access to the VCPU's state to emulate critical instruction and to maintain the virtual machine environment. From the viewpoint of L4, emulation is a matter of trapping. We introduce a new IPC-based fault protocol to grant the monitor full, transparent access to the VCPU to efficiently control the virtual machine's behavior. It dispatches virtualization critical events to the virtual machine's registered monitor. Access to a critical, non-present resource inside the virtual machine generates a trap which causes the virtualizable processor to leave the guest-mode and reenter the privileged-mode into the L4 microkernel. L4 then notifies that event to the VM's monitor application, to handle the fault. This protocol further allows the monitor application to inject virtual events from the device emulation into the virtual processor. It is detailed in Section 3.4.

The virtual machine representation in L4 needs no substantial extensions; L4 already provides the required primitives to manage virtual machines.

3.3 Virtual Machine Resources

In the following sections we present the virtualization of the virtual machine's resources in more detail.

3.3.1 Physical Memory

In this section we discuss the representation of physical memory in the L4 based virtual machine environment. A virtual machine can not have direct access to host physical memory; it would circumvent protection because on most hardware architectures physical memory access is uncontrolled – even on fully virtualizable processors. To virtualize physical memory virtual memory can be used [22]. We follow the approaches presented in [15, 94] and use virtual memory to establish a layer of indirection which allows to fully control the accessible physical memory of a virtual machine. Virtual memory allows remapping of physical machine memory to give each VM the illusion

of a chunk of physical memory starting at address zero. Virtual memory is supported by hardware and thus very efficient.

In L4 an virtual memory space is represented by an address space. We use an L4 address space to represent the virtual machine's physical memory space. The monitor populates this space by using L4 map messages. It maps or grants parts of its own address space to the virtual machine's space.

Being the L4 pager of the virtual machine the user-level monitor can use L4's user-level paging to completely control the management of the physical memory of a virtual machine. Through the page-fault protocol the monitor is notified on physical memory faults of the virtual machine. By mapping and unmapping the user-level monitor can provide memory resources at arbitrary locations and even with decreased rights, thereby allowing full control of the physical memory [40,60]. L4's mapping an unmapping mechanisms are very flexible. For virtual machines they can be used to establish controlled sharing such as content-based physical memory sharing or copy-on-write between different virtual machines [15, 94].

For efficiency reasons, L4 does not offer the complete architecturally defined virtual address space to user level; the kernel keeps part of it for its own purposes. There is, however, no conceptual limitation in L4's mapping mechanism that would prevent managing the whole address space. A guest may require the physical complete address space of the virtual machine, and hardware support for virtualization makes it easy to provide the full address space.

The L4 API defines two mandatory objects in each address space: the kernel interface page and the user thread control block (UTCB) area. Their location is determined by the creator of the address space. Being part of the L4 virtual address space, they will appear as objects in the VM's physical address space. The monitor can freely define their position and thereby effectively hide them from the guest; for example by placing these objects in a unused region of the VM's physical address space. For the virtual machine environment, these objects can optionally removed from the VM's address spaces. To not preclude later optimizations such as efficient inter virtual machine communication (Section 3.5) both objects can still be mapped into the VM's space.

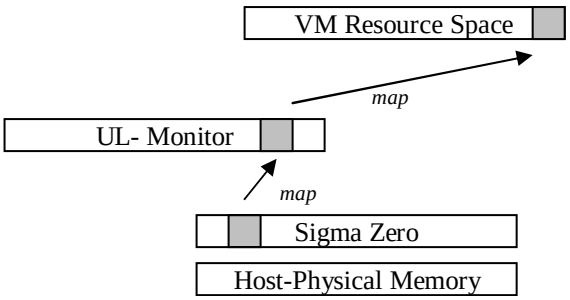


Figure 3.2: Physical memory hierarchy: The user-level monitor uses the virtual memory to provide the virtual machine with "virtualized" physical memory. L4 exports machine memory via the sigma zero address space. The monitor can request memory mappings to its own address space and pass it to the virtual machine.

3.3.2 Virtual Memory

We use L4's virtual memory management concepts to provide the VM's physical address space. However, the guest operating system in the VM usually wants to create virtual memory itself. To maintain the illusion of access to virtual memory, the virtual machine system must resolve a guest-virtual address into a host-physical address [94]. This translation consists of two stages. The first stage translates the guest-virtual address to a guest-physical address via page-tables maintained by the guest operating system located in guest-physical memory. The second stage is determined by the monitor's mapping of guest-physical addresses to host-physical addresses to enforce the resource constraints of the monitor; they overrule the requests of the guest. For example, this indirection can be used to transparently share physical memory of the machine as described in the previous section.

Most hardware lacks support for such a two-tiered memory translation, called nested paging. For efficient execution on the physical processor, both stages have to be merged into a single translation, the shadow page-table, which directly translates a guest-virtual into host-physical addresses. The shadow page-table can also be seen as a virtual TLB (vTLB) as it caches guest-virtual to host-physical translations. Figure 3.3 gives an overview of the shared page-table mechanism.

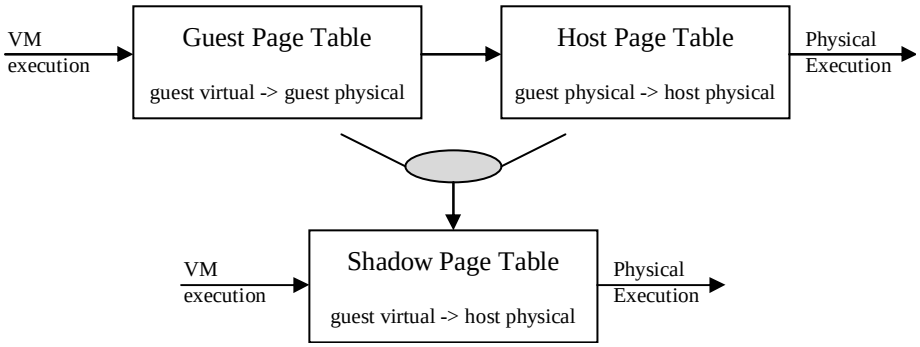


Figure 3.3: The shadow page-table is a fusion of two page–tables. Step one is defined by the guest operating system, located in guest physical memory. Guest physical memory is provided by virtual memory of the host system, which itself is implemented using a page-table. The shadow page-table is a combination of both in a hardware understandable form, to allow physical execution.

In our microkernel-based virtualization system, the generation of the shadow page-table involves all three components of our system architecture. (i) The microkernel needs to enforce isolation and independence between the different L4 address spaces holding the applications and virtual machines. (ii) The monitor application requires full control of the physical resources occupied by the VM. (iii) The guest operating system, needs to create arbitrary virtual address spaces onto its guest-physical memory. We found two different approaches to maintain guest-virtual memory which we discuss next.

Approach 1: User-Level Management

One way to establish the guest-virtual to host-physical translation is to represent the VM's virtual address spaces as an L4 address space, described by the contents of the

shadow page-table. The monitor directly constructs the shadow page-table using L4's map and unmap operations. VM-internal translation faults are propagated to the monitor which then walks the guest operating system's page-table to find the guest-physical address. It then maps pages from its own address space directly into the L4 address space representing the virtual machine, or injects the page-faults into the virtual machine as depicted in Figure 3.4. Page-fault injections accord page-faults which would occur when the guest executes on a physical machine.

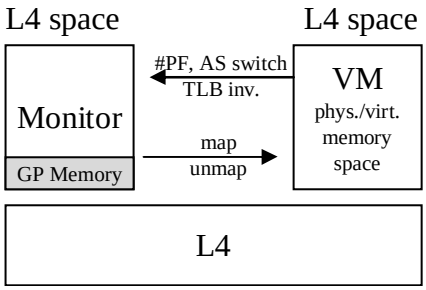


Figure 3.4: User-Level vTLB Management: The user-level monitor L4 application uses L4's mapping mechanism to create the guest virtual address space. Virtual Memory related events in the VM are notified to the monitor application.

This approach allows the monitor application to fully control the guest virtual address spaces. The monitor can use a-priori knowledge of the guest operating system to optimize the virtual address space management. For example by avoiding address space flushes or establishing virtualization-aware drivers into the guest to reduce virtualization overhead.

However, this approach has several serious drawbacks:

- L4's mapping mechanisms abstracts from the underlying hardware page-table. To allow user-level management of address spaces it includes access rights such as read, write and execute, but it does not expose additional processor architecture specific parameters such as the distinction between user- and kernel-accessible memory. Their introduction would require architecture specific extensions of L4's mapping mechanism and these features would have to be disabled for all but VM-address spaces.

- L4 threads are associated with exactly one L4 address space. As a result, only the currently active guest virtual address space can be described by the L4 address space. An address space switch in the VM requires a complete flush and repopulation of the L4 address space of the virtual machine via memory mappings by the monitor. Introducing threadless address spaces, which are needed to represent currently inactive guest virtual address spaces, into the L4 microkernel would incur major API changes; for example the introduction of address space identifiers.

- Updates to the virtual TLB are very frequent operations, efficiency of the vTLB is paramount to the virtual machine's overall performance. We consider the cost of two address space switches and a map operation for every vTLB update too expensive.

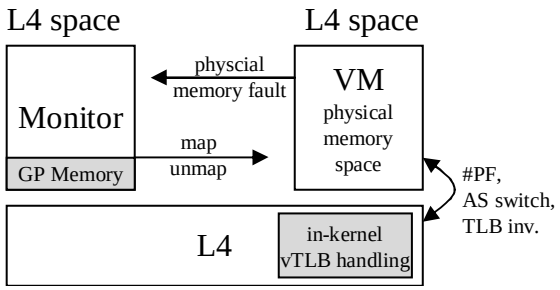


Figure 3.5: In-Kernel vTLB Management: The L4 microkernel handles virtual memory related virtualization events transparently. Only events which are caused by non-available or insufficient rights to physical resources are signaled to the user-level monitor application.

Approach 2: In-kernel Management

The problems of the first approach can be avoided by emulating guest-virtual address spaces transparently inside the L4 microkernel. The VM's physical address space is represented by an L4 address space which is maintained by the monitor. It holds all physical resources of the virtual machine, including the physical memory. The approach is shown in figure 3.5.

The L4 microkernel catches page-faults on guest virtual memory. It walks the guest page-table to resolve the fault. The user-level monitor is only notified if the L4 microkernel detects that the fault is caused by non-present, or insufficient rights on the guest physical memory.

Of course, this approach has some disadvantages, too:

- In-kernel shadow page-table management can perform only very limited optimizations. Without introducing awkward configuration protocols, optimizations based on the knowledge of a specific guest's behavior are not possible.
- The complexity of shadow page-table management is rather high: The vTLB algorithm needs to walk the VM's guest physical memory, which may cause in-kernel page-faults (that can be handled like faults during an string IPC handling). Furthermore, L4 uses one page-table format, while the guest operating system may use one of many, increasing complexity of the L4 page-table walker for the guest page-table.

Taken Approach

The authors of [7, 72] already identified the shadow page-table management of major importance for the overall performance of virtual machines. We expect *upcoming hardware* to natively support nested paging which will remove the need for shadow page-tables altogether [8]. Therefore, we prefer in-kernel vTLB management as a temporary, clean solution at the API level: If hardware support is present, L4 simply uses it without any further changes at the API level.

3.3.3 Processor

The virtual machine's processor (VCPU) is critical for the efficiency of the virtual machine [74]. Similar to other systems we use a thread to represent a virtual machine's processor [24, 26, 56]. The L4 thread abstraction represents a timeslice of the multiplexed physical processors. The execution of the VCPU can be controlled like all other threads in the L4 system. Like all other threads in L4, the VCPU is scheduled based on its timeslice and priority.

Currently L4's extended machine only exports the user accessible general purpose registers to the user-level applications. For the virtual machine we extended the thread abstraction to include the complete physical register set. To securely multiplex its threads, L4 maintains the processor context inside the microkernel. While executing the VCPU directly on the physical processor the fully virtualizable processor architecture helps to retain isolation and control as described in Section 2.1.1. The user-level monitor needs access to the VCPU context in two cases which we examine next.

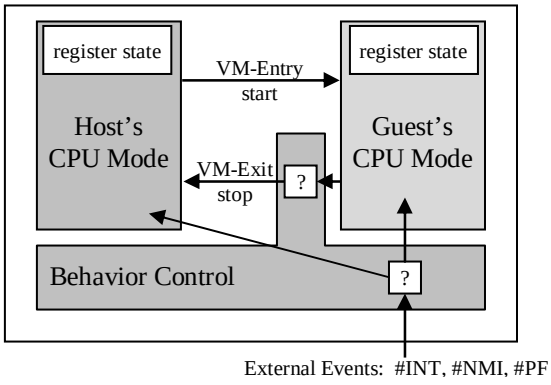


Figure 3.6: A virtualizable processor consists of two execution contexts. The Host mode is active while the VMM runs. It configures the behavior control to securely execute the VM in the guest's mode.

Handling of Critical Instructions

To emulate the virtual machine resources such as privileged registers or devices, the monitor needs to emulate the instructions accessing these resources. Fully virtualizable hardware allows to generate traps on these instructions, which cause an exit out of the virtual machine into the privileged part of the virtual machine monitor (here the L4 microkernel). L4 already provides an abstraction for these hardware exceptions: the IPC-based exception protocol. In the name of the faulting thread, L4 synthesizes a fault message to the associated user-level handler. The message contains the reason of the fault and a static subset of the user-visible CPU state. The handler resolves the fault by emulating the behavior of the faulting instruction, and sends a reply message back to resume the thread. This message contains the new CPU register state, to be established before resuming the thread.

For virtualization, exporting a static set of registers is too inflexible: the overall VCPU state is much lager, but the relevant state is small and depends on the exact fault reason. The authors of [18, 52] showed that emulation of critical instructions is a frequent operation and has strong influence on the virtual machine performance. Similar

to the exception protocol, we propose a *virtualization fault protocol*. For each virtualization fault reason, an (architecturally) predefined part of the VCPU state is transfered in the fault message. In the rare case that the monitor requires more or different state than was sent in the fault message, the virtualization fault protocol provides the *get item*. This item contains a request for additional VCPU state; it causes the VCPU to immediately generate another virtualization fault without resuming execution of the VCPU. Thus the monitor can iteratively access the complete VCPU register state. The exact description of the virtualization fault protocol is described later in Section 3.4.

Asynchronous Execution Control

When emulating the behavior of active devices the monitor must be able to asynchronously modify a VCPU's state, for instance, to inject virtual device interrupts. L4 already provides a way to asynchronously manipulate another thread through the EXCHANGEREGISTERS system call. EXCHANGEREGISTERS allows for native threads certain thread state to be read or written; but only within the same address space. Access to the complete register state has to be emulated by user-level protocols, for example, by inserting a helper thread into the destination address space, reachable via IPC, to do EXCHANGEREGISTERS locally. Inserting an L4 thread transparently into the virtual machine's address space is a major intrusion. The thread needs stub code for its protocol logic mapped into the VM's virtual address space, it needs the ability to invoke IPC, and its presence must not induce any side-effects in the guest.

To avoid introduction of an additional thread, the monitor can delay asynchronous events and piggyback them on the next synchronous virtualization fault reply. However, this is no general solution, because it may delay asynchronous events for too long – yet, it is an efficient optimization for high workload situations. As a minimally invasive method to asynchronously access the VCPU state, we favor the extension of EXCHANGEREGISTERS across address space boundaries, a possibility to asynchronously force a virtualization event. We decided to map the asynchronous access onto the synchronous virtualization fault protocol. The extensions allow the monitor the force the VCPU to sent a virtualization fault. This avoids the introduction of two access formats, and the management of in-kernel message buffers. In-kernel buffers would be needed to store the event until the VCPU is ready to receive it, at a later point in time. The extensions of EXCHANGEREGISTERS include:

- An *immediate fault* causes the VCPU to immediately raise a virtualization fault. The monitor can use this to unconditionally inject events such as non-maskable interrupts or exceptions, or to inspect the VM's state, e.g., for debugging or introspection purposes.

- The *delayed fault* provides an efficient way to find the next point in time the VCPU is ready to receive certain events. This is necessary because the VCPU can not always inject hardware events immediately into the guest, for instance, interrupts when the guest operating system has interrupts disabled. The delayed faults cause the VCPU to raise a virtualization fault when the VCPU is ready to deliver such events – the next time the guest is able to receive interrupts. The monitor can use the later virtualization fault to inject pending virtual interrupts [50].

Allowing a thread's pager and exception handler to invoke EXCHANGEREGISTERS does not introduce any security issues, as a pager is already a strongly trusted compo-

nent for the virtual machine. Apart from the VCPU register state, the monitor may need to access the VM's memory, e.g. to inspect the guest's page-tables. Accessing guest physical memory is not problematic for the monitor since it provided the memory from its own address space or knows the providing component.

3.3.4 Peripheral Devices

Peripheral devices are represented as a combination of physical memory, IO memory, and associated interrupt lines on the level of the processor interface.

Memory Mapped Devices

Memory mapped-devices are located in the guest physical address space. They are accessed by normal load/store operations, which cannot be trapped even by fully virtualizable hardware. Instead, access to memory-mapped devices can be tracked by page-faults on memory which represents the devices. It is sufficient because memory-mapped devices are aligned on the size of page-frames [75]. These page-faults should not be satisfied with a mapping but trigger an emulation of the accessing instruction to trap on further device accesses. Therefore, page-faults should also use the virtualization fault protocol. Unifying page-faults and exception handling is already under discussion in the L4 community for other reasons such as orthogonality of concepts.

IO Port Space

On some architectures, such as IA-32, there exists an additional address space, the IO port space. In L4 these device ports are part of the L4 address space abstraction. Special port mappings allow the management of permissions of an address space to the physical IO ports [60, 82]. For the virtual machine we use the IO port space to control the VM's permissions to the physical machine's ports. Non-sufficient permissions cause a special virtualization fault message to the user-level monitor, which can use the fault to emulate the associated device in software.

Interrupts

A virtual machine system has to cope with two types of interrupts: (i) physical interrupts coming from a real hardware device and (ii) virtual interrupts, which are created by a software device model representing an emulated device.

The L4 microkernel abstracts processor interrupt lines by special kernel threads. Interrupt events are abstracted by kernel generated IPC messages from the matching kernel thread to one associated handler thread.

In our virtual machine system the user-level monitor can use L4's interrupt IPC mechanism to receive interrupts of physical devices. Using the EXCHANGEREGISTERS system call with delayed faults, as described in Section 3.3.3, the monitor can then inject the physical interrupts and the synthesized virtual interrupts into the virtual machine. Direct delivery of physical interrupts into the VM is not possible as this would require the introduction of an second in-kernel mechanism to buffer interrupt events to deliver them into the VCPU when it is ready to receive the events. Another issue here is that L4 immediately masks interrupt lines when an interrupt appears. The interrupt line is unmasked when the associated handler notifies the kernel that is ready to receive another interrupt via an acknowledge IPC. For the VCPU thread this protocol

can not be used as the required information to emulate this behavior is only available in the interrupt device emulation model of the user-level monitor.

DMA

Using direct memory access in the virtual machine raises isolation issues [57]. To sustain full isolation and control, direct access to DMA-able devices for the virtual machine must be prevented; for example, by emulating the device in the user-level monitor application.

But none the less, it is possible to grant a virtual machine access to a DMA device. Here the user-level monitor who must have access rights to the device, grants access to the virtual machine. It must take care to establish the required degree of isolation by interposing the configuration of the DMA device. The L4 microkernel needs no further mechanisms to support DMA in the virtual machine as a DMA device can be controlled from the user-level monitor like every other device by controlling its physical memory, interrupt lines, and IO ports.

Trapping creates massive overhead which can be reduced by injection of virtualization aware device-drivers into the guest operating system [52]. Upcoming hardware support in the name of IO-MMU [49] or completely virtualizable devices [76] already attack this problem and may need kernel level support.

3.4 User-level Control Protocol

In this section we discuss the design of the communication protocol required to control the virtual machine environment from the user-level monitor.

3.4.1 Analysis of Requirements

The Monitor requires a protocol to (i) access the VCPU's register state, (ii) emulate resources of the VM, (iii) manipulate the VCPU's state and, at last, (iv) to signal certain events of the platform into the VCPU. The microkernel must implement a service which notifies the VCPU's monitor on behalf of the VCPU. The authors of [18, 52] showed that this communication is critical for the virtual machine system performance. Next we describe critical parameters of the control protocol.

Execution Transfer. The required communication mechanism between the VCPU thread and the user-level monitor has RPC semantics. Asynchronous events such as the delivery of interrupts are mapped on the synchronous protocol as described in Section 3.3.3. On a VM-Exit caused by an access to an emulated resource, the physical processor traps, L4 preempts the VCPU thread and notifies the user-level monitor. The VCPU needs to be suspended until the monitor notifies that the effect of the critical instruction has been completely emulated. For optimal efficiency L4 has to transfer execution directly to the user-level monitor – to donate the VCPU's time slice to the monitor.

State Transfer. On a VM-Exit the user-level monitor needs a certain subset of the VCPU register context to handle the fault. The author of [38] shows that most performant message transfer heavily depends on the details of the message and the target

architecture. In the following we discuss the communication schemes in the context of our requirements.

With *shared memory* the user-level monitor and the L4 microkernel have a mapping of the same physical memory allowing to transfer big messages without any copy overhead. In our scenario shared memory has a lot of disadvantages [36]. Shared memory softens the strict separation between the kernel and the user-level monitor. Zero copy transfers requires that the user-level monitor gets full access to the machine representation of the VCPU state. On some architectures this is not possible because the machine representation mixes safety critical state with uncritical state required by the user-level monitor.

In contrast to shared memory, *IPC messages* hide the kernel representation of the VCPU. IPC strictly separates the kernel from the user implementation allowing flexibility. As mentioned before, IPC is L4's core primitive and thus highly optimized [59]; it does not necessarily increase transfer overhead by message copying. L4's IPC is synchronous and allows to establish an RPC protocol. Also virtualization messages are usually very small [7], requiring only minimal copying overhead. Thus our communication scheme is solely based on IPC messages.

Resource Population. IPC messages allow to map resources in L4. IPC messages to VCPU threads allow to the user-level monitor to populate the physical memory into the virtual machine.

Accessing State. Our communication protocol is based on IPC messages. To reduce the copy overhead only the necessarily required state should be transfered between the user-level monitor and the in-kernel VCPU representation. As discussed in Section 3.3.3 the necessary state heavily depends on the architecture an the exact reason of the fault. We identified four cases which must be regarded by our protocol:

1. Some faults have a static behavior and thus require an architecturally predefined subset of the VCPU's state (for example the rdtsc instruction of the IA-32 processor). This state can be contained directly in the fault notification IPC.

2. Some faults have a dynamic behavior. For optimal performance the monitor may need to change the state contained in the fault message during the lifetime of a virtual machine.

3. Faults may have rare corner cases which require much more information to resolve them. To avoid performance overhead for treatment of these rare cases, the protocol needs am mechanism to gather additional state without resuming the VCPU. This mechanism can also be used to fetch additional state, for example, if the complete state does not fit into one IPC message.

4. The resume IPC notification holds VCPU state which is installed before the VCPU thread is resumed.

Addressing State. To identify the involved VCPU registers of a fault message or a VCPU state update the protocol needs to name the addressed VCPU registers. Only the registers which do not have side-effects to the host operation are user-accessible. We give each user-accessible VCPU register an unique index. Only through these indices the user-level monitor can name the registers. All addressable registers are uncritical,

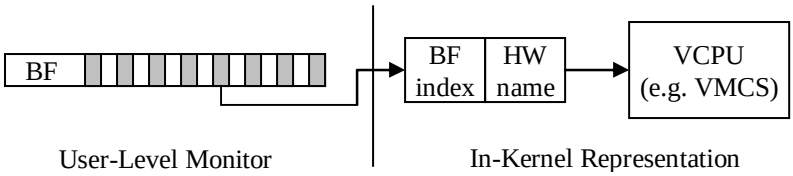


Figure 3.7: The kernel exports the user-level accessible registers via a bit-field. The user-level monitor can use only the bit-field indices to address the VCPU state. The kernel translates the indices into a hardware-usable form by a simple lookup table.

which avoids checks to enforce isolation in the L4 system (Refer to figure 3.7). The registers are ordered in a way which reflects the correlation. VCPU registers which are highly probable to be accessed at the same fault have close index numbers. This allows to find a compressed representation of VCPU register subsets. These sets are usually very small, about ten registers [7]. In figure 3.8 we show a graphical representation of the encoding we present next:

- **Pre-defined Sets.** In cases a fault has a static behavior and requires a constant set of VCPU registers an architecturally predefined set of VCPU registers an efficient encoding.
- **Bit-field.** The bit-field holds one bit for every user-accessible VCPU register. The problem with the bit-field description is its size. Regardless if one or all bits in the field are set, it includes all bits explicitly. To overcome this we found a so called *hierarchical* bit-field. It is divided in partitions of the size of a machine word and a header. The header specifies which words of the bit-field are non-zero and therefore included in the bit-field representation explicitly. A careful ordering of the associated VCPU registers achieves a dense coding.
- **VCPU Mask.** The VCPU mask is derived from the processor mask introduced in [89] and founded on the description of floating point values. The VCPU mask defines a subset of processor registers (the bit-field) in a space efficient way. The mask compresses the bit-field into one machine word. It treats space of the mask description for accuracy.

  To describe an arbitrary set of registers “at least” semantics have to be used. The mask may identify more than the required registers. Here the protocol/user can make trade-offs by using one coarse-grained, or multiple more fine grained masks.

3.4.2 The Virtualization-Fault Protocol

The virtual machine’s CPU is represented by a L4 thread and thus an communication endpoint in L4. The virtualization fault protocol unifies page-fault and exception protocol for VCPU threads. It is based on synchronous IPC. The virtualization fault protocol is defined between the faulting VCPU thread and the thread registered as its pager (the user-level monitor). It allows the pager to get notifications on virtualization-critical events, to access the VCPU register state, and to install resources into the virtual machine using IPC based resource mappings. An example protocol session is sketched in figure 3.9.

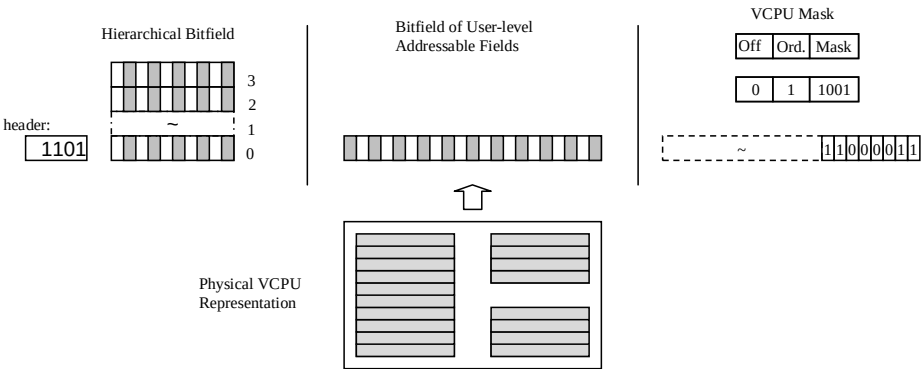


Figure 3.8: This figure shows different formats for the specification of a set of VCPU registers. The representations left and right of the simple bit-field are compressed representations described in the text.

The Fault Message

A *virtualization fault message* is synthesized by the kernel in the name of the VCPU when the VCPU raises a virtualization event. The message contains a word specifying the reason followed by a subset of the VCPU register state. The exact registers depend on the fault reason and the architecture. The kernel implementation uses the CALL IPC to directly transfer the execution, including the remaining timeslice to the pager to reduce latency.

The Reply Message

The pager answers the virtualization fault with a *virtualization reply message*. This message may contain typed items for resource mappings, for example, memory mappings to install physical memory into the virtual machine's environment as described in Section 3.3. The reply message can also be used to modify or to request more VCPU state. Similar to the general IPC protocol, several "typed" items, called *protocol items* are defined. These items are encoded as untyped words in the IPC message of the virtualization reply message to avoid interpretation by the IPC transfer mechanism. This allows to leave L4's IPC path untouched and allows the pager to use L4's fast path IPC. The in-kernel virtualization fault handler on the VCPU side associates a certain meaning to the protocol items (untyped words) which are described next:

- **Set Item.** The *set item* changes exactly one register of the VCPU. The target register is encoded in the item, followed by the value.
- **Set-Group Item.** A *group item* sets a predefined, fixed group of VCPU registers. Predefined groups of registers may be very efficient on some architectures to handle faults which require a static subset of the VCPU's state.
- **Set-Multiple Item.** The *set-multiple item* sets an arbitrary subset of VCPU registers. The target registers are identified by a dense encoding, such as a bit-field or a VCPU mask, followed by the values.
- **Get-Resume Item.** The *get-resume item* requests additional VCPU register state; it uses the encoding of the set-multiple item to identify registers to be

sent. This item, if present, must be the last item in the reply message. It forces the VCPU to immediately send the requested state in another virtualization-fault message without resuming the VCPU.

- **Set-Mask Item.** An architecture may require to make the fault state transferred on a virtualization fault message run-time configurable. The *set-mask item* allows to configure this state for each reason. The state is defined similar to the set-multiple item format.

The refinement and exact encodings for the IA-32 architecture are described in the implementation chapter in Section 4.2.3 and defined in Appendix A.

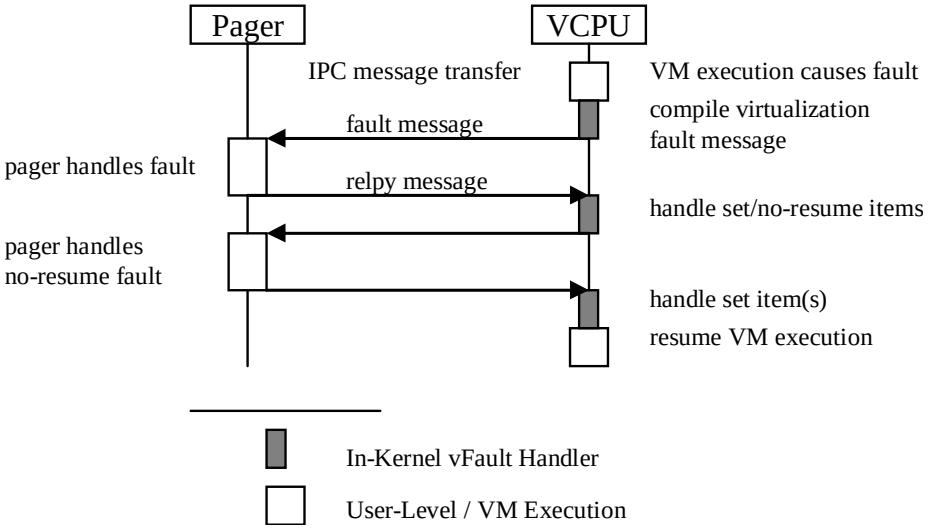


Figure 3.9: The virtualization fault protocol: On a virtualization critical event inside the VM the kernel synthesizes a virtualization fault message and sends it to the VCPU's pager. The pager's first reply holds a get item requesting further state. The second reply only modifies VCPU state and resumes execution of the VCPU.

3.5 Virtual Machine Communication

Virtual machine communication is known since the beginning of virtual machines [10, 66]. It provides an efficient way for communication avoiding complex emulation of communication devices such as network cards. Our virtual machine environment reuses the facilities already available in L4.

As an extended L4 thread we conditionally allow the VCPU to issue selected L4 system-calls. To use the L4 interface the VCPU thread must be able to access its user thread control block (UTCB) and kernel interface page (KIP). Being part of the virtual machine's platform we decided to base our virtual machine communication model on the guest physical address space. The guest physical address space already has a strong representation in L4: it is contained in the address space abstraction which represents the virtual machine. Guest physical memory can be altered via map and unmap operations by the monitor. The guest virtual address space is only represented in a hardware

understandable form by the vTLB. Thus the UTCB area of the virtual machine is part of the guest physical memory space already described in Section 3.3.1. To access the UTCB a virtualization aware driver inside the guest operating system may remap the UTCB object into the guest virtual address space via the guest page-tables.

The virtual machine is allowed to issue following L4 system-calls:

- THREADSWITCH. The VCPU thread releases the physical CPU without blocking on a resource. This is transparent to the monitor, like usual thread scheduling.
- IPC. The VCPU thread can utilize all features of L4's IPC mechanism. Addresses of mapping and strings are treated as references to guest physical addresses.
- UNMAP . As the dual operation to the map, the unmap operation is to be supported as well.

To issue the L4-system call we leverage the *hypercall* instruction which is available on fully virtualizable processors. Executed in the virtual machine, this instruction unconditionally leaves the virtual machine and switches to host mode – in our case the L4 microkernel. We define two modes associated with the instruction if executed in the virtual machine. The monitor can configure the *hypercall* instruction to be interpreted as L4 system-calls or to cause virtualization fault messages.

1. The VCPU can be configured to cause virtualization fault messages to the monitor if the instruction is executed in the virtual machine. This enables the user-level monitor to emulate the instruction behavior allowing full control of the associated semantics providing even recursive virtualization.
2. The hypercall instruction can be mapped on L4's system call interface. It allows the VCPU thread to call the three L4 system calls directly. To enforce isolation, the monitor can use IPC control mechanisms such as *Redirectors* [51] or *Clans & Chiefs* [58] to intercept the IPCs issued by all VCPUs of a virtual machine.

To invoke the L4 IPC and Unmap operations the VCPU thread must lookup its UTCB. The UTCB location can be communicated by a user-level protocol: First the monitor configures the VCPU to generate virtualization faults on hypercall instructions. The monitor checks if the VCPU requests to use L4 services, loads a VCPU register with the UTCB location, and enables the hypercall instruction to be mapped on L4 operations. From now on the VCPU can efficiently invoke L4's IPC. The invoked system call is specified in a special location in the UTCB, or if the ABI register binding of the architecture has free registers, such a register may be used.

3.6 Summary

In this chapter we presented the design of our microkernel-based virtualization system. It consists of three major components: the microkernel, the user-level monitor application and the virtual machine. Our system is based on an extended L4 microkernel. The microkernel needed only minimal extensions on the API level which are fully backwards compatible, i e . invisible to applications which do not require them.

We introduced a new L4 address space mode which reflects the behavior of the platform API. It is required to give the virtual machine a faithful environment and to

keep track of all physical resources which are available to the VM. The L4 thread abstraction represents the virtual machine's processor. Virtualization events are mapped on an IPC based virtualization protocol which allows access to the VCPU's complete register state; it replaces the exception and page-fault protocol of current API threads. Asynchronous access to the VCPU's state is mapped onto the synchronous protocol. L4's current resource mapping allows population of resources into the virtual machine space at the time of a virtualization fault. Access to non-available resources generates virtualization faults; this is congruent to the behavior of the native thread model.

In fact, the privileged part of the virtual machine monitor and the microkernel have many similarities. The most important are isolation and resource control. The microkernel's user-level resource control already requires methods which are reused and slightly extended for the virtual machine. The biggest extension comes with the thread abstraction. Currently, the microkernel only provides unprivileged access to the processor and a reduced register set. The much bigger privileged set requires a more sophisticated protocol, but the basic functionality, synchronous and asynchronous access is already available, which caused only minimal extensions on the API level.

The last point we have the check if the environment provided by L4 fulfills the virtual machine criteria and if the design meets our goals stated at the beginning of this chapter.

Based on our description in Section 2.1 a virtual machine environment has to sustain:

- **Equivalence.** The virtual machine is given a complete address space which can hold any resources available in the physical machine. All resource accesses in a virtual machine can be configured to cause resource faults. These faults allow the user-level monitor to faithfully emulate the behavior of non-available physical resources.

  Using physical resources may effect the isolation of the virtual machine and thus harms the equivalence criteria. Here the user-level monitor application is responsible for only providing the resources which sustain the required isolation for a particular virtual machine instance.

- **Efficiency.** To sustain efficiency the virtual machine environment can use physical resources whenever possible. The representation of the virtual machines resources can even change during runtime of the virtual machine to achieve system-wide performance trade-offs.

- **Resource Control.** Through L4's resource mapping and thread management mechanisms the user-level monitor has full control of the resources acquired by the virtual machine. The monitor can asynchronously, without consent of the virtual machine, preempt any physical resource held by the virtual machine. Thus the resource control requirement is achieved.

  Using L4's resource control mechanism the user-level monitor can define a configuration which fits the requirements of a particular virtual machine; it can define the trade-off between equivalence and efficiency, for the virtual machine itself, as well as on the system level, e .g ., participating on a system wide resource management scheme.

Chapter 4

Implementation

In this chapter we present a prototype implementation of our extended microkernel and a user-level monitor application. The implementation is based on the fully virtualizable IA-32 processor architecture using the Intel virtualization technology VT-x.

In Section 4.1 we discuss the architectural features of the IA-32 processor extensions. In Section 4.2 we describe the implementation of the virtualization extensions to the L4 microkernel. In Section 4.3 we briefly sketch our prototype implementation of the user-level monitor application.

4.1 The IA-32 Processor Architecture

Our implementation of the microkernel-based virtualization environment uses the Intel IA-32 architecture with its virtualization extensions (VT-x). It is expected that our approach also applies to comparable, fully-virtualizable architectures as well, such as [8, 47].

Virtualization Extensions (VT-x)

The basic IA32 architecture is not fully virtualizable. In [78] the authors identified several critical instructions which prohibit the IA-32 processors from being fully virtualizable. To prevent execution of these critical instructions virtualization systems need to make high trade-offs: either requiring highly complex approaches, such as para-virtualization [25] or binary translation, and instruction tracking [23, 80].

To overcome these CPU virtualization issues Intel recently introduced virtualization extensions to the IA-32 processor. With these extensions the IA-32 processor is not only virtualizable, but provides convenient mechanisms to create a virtualization environment. Rich Uhlig et al . give an overview of the virtualization extensions in [72, 87] — more detailed information can be found in [50]. We will sketch the major points which help to understand the implications on our implementation.

The virtualization extensions introduce a new processor mode, the *root mode*. It is orthogonal to the processor's legacy protection ring model; allowing the VMM and the guest to use its intended privilege levels. A VMM runs in VT-x root mode and has full control of the physical processor. The non-root mode provides an alternative IA-32 environment controlled by the VMM and designed to support a VM. VT-x introduces

two new atomic transitions called VM-Exit and VM-Entry. The VM-Entry switches from root mode into non-root mode; the VM-Exit vice versa.

The virtual machine control structure (VMCS) is a new control structure located in a physical memory page-frame. It manages the behavior of VM-Exits and VM-Entries and the behavior of the processor when it operates in non-root mode. The VMCS is divided in two main partitions, the guest-state and the host-state area. The areas contain fields which correspond to different components of the physical processor. A VM-Entry loads the state from the guest region, resuming the execution of a virtual machine. A VM-Exit safes the actual processor state into the guest area and loads the state from the host area.

In non-root operation the processor behavior can be fully controlled based on execution controls stored in the VMCS by the VMM. These fields allow the VMM to flexibly define which instructions and events cause VM-Exits. On the IA-32 architecture the CR0 and the CR4 registers are critical as they control the processor operation. For efficiency reasons the VMM may wish to retain control only on some of these bits, for example the bits which control virtual memory management including paging, but not others such as the floating point unit. Here the VMCS highly integrates the processor state experienced by the guest and the physical state used for execution. VT-x includes a guest and host mask that a VMM can use to indicate which bits it wants to protect. Guest writes can freely modify the unmasked bits, changing the operation of the physical processor. But an attempt to modify a masked bit causes a VM-Exit. The VMCS also includes a read shadow whose value is returned for masked bits to decuple the VM representation from the physical processor configuration. This fine granular mechanisms greatly reduce the virtualization overhead.

The VMCS offers bitmaps to flexibly control which of the 32 processor exceptions cause VM-Exits. It further includes a pointer to an I/O bitmap to control the access to the physical machine's IO ports, and a newly introduced MSR bitmap which controls direct read / write access to the model specific registers. To efficiently virtualize interrupts the VT-x extensions provide two controls. When the "external interrupt" control is enabled, all physical interrupts – which appear while the processor is in non-root mode – cause VM-Exits. This shelters a VM from physical events which are not related to the active virtual machine. The second control offers delivery of virtual interrupts (synthesized by the VMM) into a guest; the "interrupt-window exiting" control causes a VM-Exit if guest software signals that it is ready to receive interrupts by unmasking its interrupt flag in the EFLAGS register. This avoids polling of the guest's processor state.

The VM-Entry can be configured to inject events such as exceptions and virtual interrupts into the guest. These events are delivered to the guest after the guest state is completely loaded from the VMCS. The events are delivered through the guest's IDT, just as if the injected event had occurred immediately after the VM-Entry. This avoids complex emulation of the delivery mechanism.

All VM-Exits use a common entry point into the VMM, which is defined in the host-state area of the VMCS. The VM-Exit saves detailed information of the exit reason into the VMCS. For many instructions this information is sufficient to handle the fault very efficiently. For example an instruction which accesses a control register: Here the exit information provides (i) the identity of the control register (e .g . cr3), (ii) the mode of the access (read or write), (iii) the second operand (which general purpose register is accessed). On other faults the exit information provides the guest linear address, avoiding complex, time consuming inspection of the guest segmentation configuration.

4.2 Microkernel Extensions

In this section we present the implementation of the virtualization extensions into the L4 microkernel. For the implementation we used the virtualization extensions provided by Intel VT-x. Our current implementation is for the 32-bit protected mode of the L4 microkernel. We currently support only this mode in the guest, too. This is due to the limitation of our rudimentary implementation of the shadow page-table.

The L4 microkernel is the privileged component of the system and therefore runs in the root mode of VT-x. At system start up, L4 detects, enables, and initializes the virtualization extensions of the processor. The kernel announces hardware virtualization support with a special feature string in the kernel interface page, the KIP (see Chapter A).

Our virtualization extensions require small changes on the thread and the address space abstractions. The address space is extended to include a new "virtualization mode" which exports the semantics of the platform API. A Thread inside this space is the VCPU of the virtual machine. In the following we present our extensions in a way which resembles the life cycle of a VCPU thread. First we present the implementation of the extension of L4's thread control block (Section 4.2.1). After the creation of the VCPU thread it starts to execute in the VM which causes hardware traps (VM-Exits); their handling is described in Section 4.2.2. Some VM-Exits are propagated to the user-level monitor which is done using the virtualization fault protocol. The virtualization fault protocol replaces the page-fault and exception protocol of native L4 threads; it is described in Section 4.2.3.

After the handling of the VCPU thread we describe the implementation of memory based resources of the virtual machine: physical memory (Section 4.2.4) and virtual memory (Section 4.2.5). The remaining resources are shown in Section 4.2.6.

4.2.1 VCPU Thread

In the design chapter we decided to represent the processor of a virtual machine by an L4 thread. A thread abstracts a share of the time-multiplexed physical processor and naturally establishes secure and controlled sharing of the physical processor between native applications and virtual machines. To represent the processor of a virtual machine (the VCPU) we extended L4's thread control blocks (TCBs). They now include the VMCS structure containing the processor state. The VMCS does not contain the general purpose registers. We store them at the beginning of the kernel stack which usually holds the hardware exception frame.

VMCS Controls. To control the behavior of the virtual processor VT-x introduces control fields to specify which physical resources cause a VM-Exit. In the L4 microkernel environment we separated the controls into three parts:

- **Kernel-Controlled.** These controls are not accessible by the user-level monitor; i.e. the user-level monitor can not change them, as it would have effects on the host mode – the L4 microkernel. These controls include all fields of the host-area.

  Kernel controlled fields also include the scheduling related events such as hlt and pause which always cause a virtualization fault sent to the user-level monitor.

- **Implicit.** These controls are automatically managed by the L4 microkernel. The user-level monitor indirectly manages these controls using L4's map and unmap mechanisms. Implicitly controlled virtualization events are for example the faulting on IO and MSR accesses. If the VCPU accesses a non-available resource L4 synthesizes a virtualization fault.
- **User-Controlled.** Controls of this group are exclusively controlled by the user-level monitor, for instance the interrupt window exiting, or the time-stamp counter access control. These VMCS fields are exported as VCPU registers and accessible using the virtualization fault protocol.

VMCS Access. The VMCS is located in physical memory, but it cannot be accessed by normal load/store operations. VT-x introduces new read and write instructions to access the various fields contained in the VMCS. The addressed field is an operand encoded in a general purpose register. Thus, to access the fields assembler code needs to be written. To increase the type safety of VMCS accesses in the L4 microkernel, we used C++ templates to bind the different VMCS indices to their types. Listing 4.1 gives an example.

As described in Section 3.3.3, the user-level monitor is not allowed to directly access the VMCS. It only can access the uncritical fields which are exported to user-level and defined in a bit-field. Thus, validation of user requests is reduced to bounds checking with the bit-field indices. The translation of bit-field indices into VMCS fields is done using a simple array. The user-level monitor can access the VCPU state, which includes selected VMCS fields, using the virtualization fault protocol.

VCPU Creation. The initialization of the VCPU consists of two stages. In the first stage, at creation of the VCPU thread, all resources (including the VMCS) are allocated and preinitialized. This especially includes the host-state area. The VCPU thread is not put in an IPC receive for a startup message to receive its IP and SP, like native threads. Instead, the VCPU thread expects a virtualization fault reply message. This allows the user-level to completely configure and initialize the register state of the VCPU before even the first instruction inside the virtual machine is issued.

After the user-level monitor sends the virtualization fault reply message, the VCPU thread finalizes the initialization. The return_to_user() function is changed to not return to user-level but to enter the virtual machine for the first time using the vmlaunch instruction.

VCPU Dispatching. The VCPU thread is scheduled like all other native threads of a L4 system. The thread switch is extended to maintain the physical processor's VMCS pointer register. Mapping the VCPU onto a L4 thread does not require any additional thread states in L4. Activity states such as wait on "inter-processor-interrupt", which are provided by VT-x, are not relevant to the L4 microkernel but fully accessible to the user-level monitor, to emulate SMP systems.

The hlt and pause instruction cause a physical processor to enter sleep mode until the processor receives an hardware interrupt. For the VCPU these instruction cause VM-Exits which is mapped onto virtualization faults. They can not be handled transparently by L4 because the VCPU thread is not allowed to resume its execution until a virtual interrupt is delivered to the processor. The virtualization fault message automatically blocks the VCPU thread thus avoids that it is being scheduled, until the monitor decides to resume the VCPU, for instance, by injecting an virtual interrupt.

```
template<u32_t index, typename T = word_t> class vmcs_field
{
public:
    // Assign T to vmcs_field ; T needs a member raw
    vmcs_field& operator= (T val)
        {
            ia32_vmwrite(index, val.raw);
            return *this;
        }

    // Assign vmcs_field to T ; T needs a member raw
    operator T()
        {
            T val;
            val.raw = ia32_vmread(index);
            return val;
        }
};

// Specialize template for word_t ; word_t does not have a .raw member
template<u32_t index> class vmcs_field<index, word_t> {
public:
    // Assign word_t to vmcs_field
    vmcs_field& operator= (word_t val)
        {
            ia32_vmwrite(index, val);
            return *this;
        }

    // Assign vmcs_field to word_t
    operator word_t()
            return ia32_vmread(index);
};

#define     VMCS_IDX_PB     0x4000

class pb_t
{
public:
    union {
        u32_t raw;
        struct {
            s32_t extint_exit    :  1;
            s32_t res1           :  2;
            s32_t nmi_exit       :  1;
            s32_t res2           : 28;
        } __attribute__((packed));
    };
};

typedef vmcs_field<VMCS_PB_IDX, pb_t>  f_pb_t;
```

Listing 4.1: The above listing shows the type safe encapsulation of the vmread and vmwrite instructions. Using templates we are able to bind the required object type on the used VMCS index. The output generated by the GCC compiler does not incur any overhead.

4.2.2 VM-Exit Handler

L4's VM-Exit handler is the single entry point of the VCPU thread on a VM-Exit. A VM-Exit occurs on virtualization critical events. The processor stores the current processor context (the VCPU context) into the guest state area of the current VMCS in a well defined precise way and does a world switch, loading the host state, thus resuming execution in root-mode at the beginning of the VM-Exit handler. The handler stores the remaining processor context (mostly the general purpose registers) onto the kernel stack. Restoring L4's execution context additionally requires to reinitialize the segment registers, the EFLAGS, and to the size of the TSS segment. Afterwards, the fault reason is examined by inspecting the VMCS fields. The faults can be grouped into three types:

1. Internal event: Events such as instructions that affect the vTLB management are treated in the L4 microkernel and thus handled transparently.

2. Monitor event: Resource faults (such as physical memory, IO ports, and MSRs) and the emulation of most critical instructions belongs to this group. In the name of the faulting VCPU L4 sends a virtualization fault message to the registered pager (i . e . user-level monitor) to handle the fault.

3. External events: Events which are not related to the VCPU execution cause VM-Exits, too, resuming execution in the VM-Exit handler. For physical interrupts we use the int instruction to issue an software interrupt to emulate the delivery of the hardware interrupt though L4's IDT.

After handling the fault the VCPU context is reloaded and the VCPU regains control of the physical processor using the vmresume instruction which synchronously does a VM-Entry into the non-root mode.

4.2.3 Virtualization Fault Protocol

The virtualization fault protocol grants the user-level monitor full access to the VCPU thread. It is required to emulate critical instructions, to control the behavior of the VCPU, to access the guest's processor state, and to populate memory into the virtual machine. The virtualization fault protocol replaces the memory fault protocols and the exception messages used for native L4 threads. As described in the design chapter, the protocol consists of two messages: The kernel generated fault message and the reply message which comes from the VCPU's pager.

Fault Message. The fault message has a static message layout. In the first message register it holds the reason of the fault, followed by a fault specific subset of the VCPU's register state. The state can be defined by the user-level monitor using a *set-mask* protocol item in the reply message. This item includes a VCPU bit-field. For each VM-Exit reason that results in a virtualization fault a bit-field is stored in the VCPU thread's control block and used to generate the fault message. After the message is stored, L4 issues a blocking in-kernel IPC on behalf of the VCPU thread to the registered pager. This IPC blocks, the VCPU thread is not scheduled until the pager sends a reply message.

An alternative solution would be to use a fixed fault message content. As we aim for full virtualization the goal is to provide a faithful duplicate of the real machine, so

the required emulation does not depend on the specifics of a guest. Thus the required work is fixed by the specifics of the hardware architecture and a fixed protocol would be sufficient. Unfortunately, our current implementation of the user-level monitor is not complete enough to run more complex operating systems which utilize all hardware features to validate this thesis.

Reply Message. The reply message allows the user-level monitor to modify the VCPU state, to read additional VCPU state, and to resume execution. To distinguish these different operations the reply message can hold different protocol items.

The reply handler interprets the untyped words contained in the UTCB as protocol items of the virtualization fault protocol. These words are transfered by the in-kernel IPC transfer mechanism. We support several protocol items which are shown in Appendix A.3. To interpret the items we use the same mechanisms as used in L4's extended IPC message transfer. Handling of read requests, which are encoded as *get items*, require to store these items temporary; they cannot be handled directly as this would overwrite the current message. We store these *get items* in a region local to the physical processor. After the message is completely examined, all *get items* are processed, and the requested VCPU register values are stored in the UTCB for the second fault message. If the message contained no *get items* the execution of the VCPU thread returns into the VM to resume execution.

Asynchronous VCPU Access

The user-level monitor requires asynchronous access to the VCPU and the VM to implement the behavior of the emulated virtual machine environment. As discussed in 3.3.3 the user-level monitor must use the EXCHANGEREGISTERS system call to force the VCPU thread into a virtualization fault to access the VCPU's state.

Our changes to the L4 API include the required extensions to the EXCHANGEREGISTERS system call as detailed in A.2. The implementation is straightforward. It distinguishes between local and cross processor operation. The operation is always run on the processor the VCPU is associated to. This guarantees that the VCPU is currently not running and thus can be freely manipulated.

- **Delayed Fault.** We currently implemented the "interrupt-window exit" control to force a virtualization fault message when the guest signals to be able to receive interrupts. The implementation simply sets the "interrupt-window exit" control in the VCPU thread's VMCS.
- **Immediate Fault.** The immediate fault requires that the VCPU thread enters the virtualization fault protocol immediately. The implementation uses L4's in kernel notification mechanisms. It offers a method to force execution of in kernel functions. We force the VCPU thread to call the in-kernel virtualization fault handler.

4.2.4 Physical Memory Space

The virtual machine is represented by an L4 address space. All threads within the same address space have the same resource permissions. To faithfully express a virtual machine we added a new mode to the L4 address space. The new virtualization mode can be requested at address space creation via additional control bits on SPACECONTROL

(refer to A.1). In this mode the virtual machine has access to the complete virtual address space size and holds the virtual machine's physical memory space. At the time of creation it is empty, allowing the monitor full control of the virtual machine; the L4 microkernel is not mapped into the memory space; it only contains the necessary KIP and UTCB areas. All threads assigned to it behave like VCPUs as detailed in 4.2.1.

The virtual machine's physical memory address space can be fully controlled by the user-level monitor using L4's memory management mechanisms: map, grant, and unmap. As all other spaces it is fully integrated in L4's mapping hierarchy and thus part of the mapping database.

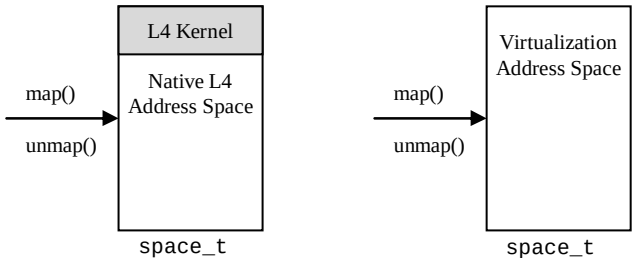


Figure 4.1: A virtual machine is represented by L4's abstraction for isolation, the address space. For virtual machines we introduced a new address space mode, the virtualization space. It changes the environment experienced by the code running in the address space from L4's extended machine to the expected platform API. This requires to remove the kernel from the virtual machine's memory space. The address space can still be manipulated using L4's address space manipulation operations such as mapping and thread creation.

- **Map.** The monitor uses L4's map operation to populate the virtual machine environment. Memory resource mappings are part of the virtualization reply message. These mappings are automatically populated into the guest physical memory space as the VCPU is associated it. We extended the address space implementation (`space_t`) to allow mappings into the virtual memory region normally reserved to the kernel.

- **Unmap** allows the monitor to revoke access to physical memory and query the accessed and dirty information of a physical memory frame. This access information includes the information of the shadow page-table, too. Revocation of guest physical memory has to remove parts of the shadow page-table which point to revoked memory; the vTLB handler has to take care that its reference information is included on an UNMAP operation.

- **Physical Memory Fault.** Access to physical memory which is not mapped in the VM's address space causes resource faults and thereby virtualization messages. Regions which are occupied by the KIP and UTCB are handled transparently by L4.

4.2.5 Shadow Page Table Management

The shadow page table implements the virtual memory management of a virtual machine. The shadow page-table caches the two staged translation (from guest-virtual to

guest-physical to host-physical) in a single translation (guest-virtual to host-physical) for direct use by the processor. We consider this a temporary solution until upcoming hardware natively supports this translation in hardware [7].

The shadow page-table can be seen as an additional, virtual TLB in the memory hierarchy located between the guest's page-tables and the physical processor's TLB. The shadow page-table handler behaves similar to a processor TLB. It caches address translations. This cache can become inconsistent to the description required by the guest. Similar to a processor TLB the shadow page-table management uses page-faults, the invlpg and the load CR3, CR0, CR4 instructions to enforce synchronization. In the virtual machine environment, these instructions cause VM-Exits to emulate their behavior.

There exist two types of inconsistencies:

- **Shadow Page-Table.** In these cases the shadow page-table is inconsistent with the guest which corresponds to an update of a hardware TLB miss/fault.

  The most frequent update is triggered by a page-fault in the virtual machine. The shadow page-table handler examines the guest's page-table and synthesizes an entry in the shadow page table. If the guest's page-table itself does not contain a valid entry the fault is a true page-fault and injected into the virtual machine.

  Another cause for a synchronization is an address space switch, which is triggered by a load CR3 instruction. It requires a flush of the shadow page-table which corresponds to the flush of the hardware TLB. Strategies for optimizing this critical operation are not examined in this thesis as they require complex trade-offs. This thesis focuses on the integration of the mechanisms using concepts of the L4 microkernel.

  The invlpg instruction is used by the guest operating system to enforce consistency of the TLB, thus the shadow page-table has to remove the matching page-table entries, too. (The hardware TLB consistency has already been established by the implicit hardware TLB flush at the world switch).

  Changes on the handling of virtual memory of the VCPU, for instance, enabling and disabling paging cause a flush, too. The shadow page-table manager adapts its behavior.

- **Guest Page-Table.** These inconsistencies are caused by the processor's updates of accessed and dirty bits on the page-table. The shadow page-table mechanism needs to emulate that behavior as the processor only updates the shadow page-table. The access bit is emulated by marking non accessed pages in the guest page-table non present in the shadow page-table. The resulting 'virtual' page-fault is used to propagate the accessed bit update into the guest. The dirty bit is propagated in a similar way by marking non dirty pages in the guest read only in the shadow page-table.

The emulation of super pages (4M pages) of guest physical memory by several 4K pages at host level may require additional care. Flushing such an emulated super page requires the revocation of multiple entries. As the representation of the guest page-table is not necessary consistent with the shadow page-table, the shadow page-table mechanism can not use the guest page-table to decide the size of the area identified by the invlpg instruction. Here, the shadow page-table itself has to carry additional information that it emulates a guest super page, causing a invlpg on such a region to

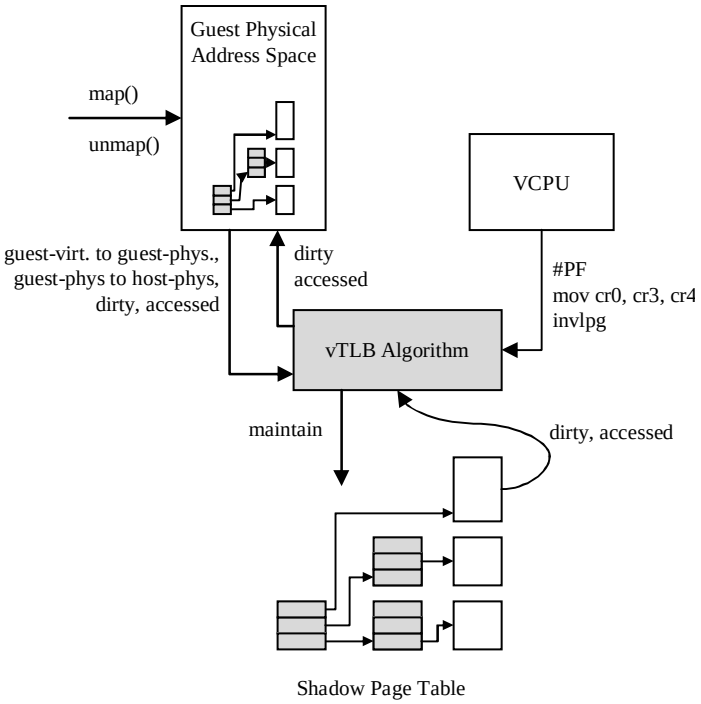


Figure 4.2: The vTLB (shadow page-table) uses the guest guest's page tables, located in guest physical memory, and the virtual memory related instruction issued on the VCPU to maintain the shadow page table. The physical processor updates access information (access and dirty bits) directly into the shadow page-table. These bits must be propagated into the guest's page-tables.

flush multiple 4K pages. The meta information can be stored in the page directory entry's "available bits" of the shadow page-table.

Integration in L4

The L4 microkernel implements page-tables with space_t objects. All page-tables are derived via map operations from the sender's space and thus recursively derived from the sigma zero address space. The dependencies of the mapping hierarchy are expressed in the mapping database (MDB) which is located in a page attached directly behind the hardware page-table. It contains the link to the sender's (parent) page-table entry from which it is derived via the map operation.

Full MDB Integration. Using the in-kernel services to construct the shadow page-table fully integrates the pages into L4's recursive mapping tree. The shadow page tables are constructed from the guest physical address space using the map_fpage() operation. This approach automatically provides support for the UNMAP operation which allows the user-level monitor to revoke access and to query access information of the shadow page-tables by unmapping guest physical memory. Unfortunately, this approach has some drawbacks. The map_fpage() operation needs virtualization specific extensions to allow manipulation of page-table fields which L4 defines statically

in native operation, for instance the protection information (the super-user bit). The shadow-page table format uses the format provided by the space_t object and thus is static, it cannot be adapted to express the guests mode of operation in the most efficient way. The tight integration of the shadow page-table creates overhead in time (for maintaining the mapping hierarchy) and space (to store the back-link). Shadow page-table updates are very frequent operations. The mapping hierarchy information is needed only in the rare cases the user-level monitor changes the physical memory provided to a VM.

MDB Coherence. To avoid the above deficiencies the shadow page-table could be detached from L4's mapping hierarchy. Coherence is established lazily on UNMAP operations on the guest physical memory. The shadow page-table manager now can freely choose the optimal representation to effectively and efficiently represent the guest's address space layout. Detaching the vTLB raises synchronization issues between the recursive mapping tree and the shadow page-table. In the current implementation, a revocation of guest physical memory issued by an UNMAP operation causes a complete flush of the shadow page-table. The UNMAP system call further allows to query the access information (accessed and dirty) of a given memory page. This information is a bitwise or of all page-table entries which transitively derived the memory from the space issuing the operation. The virtual machine's guest physical memory space is automatically included, as it is part of L4's mapping hierarchy. A virtual machine's virtual memory spaces must be included, too. Here we decided to update this information eagerly into the page-tables forming the guest physical address space at the same time the guest's page-table is updated. This avoids time consuming parsing of the complete shadow page-table at the time of the UNMAP operation, or handling of an inverted page-table to track guest physical to guest virtual references.

We implemented both approaches to construct the shadow page-tables. Unfortunately, we were not able to make a final decision as this requires a more advanced shadow page-table algorithm which is out of the scope of this thesis.

Optimization: Guest Page Table Walks

In the presented model we require temporary mappings of portions of the guest physical memory into the kernel address space to access the guest's page-table, for instance, to translate guest virtual into guest physical addresses. The L4 kernel space has only very limited capacity for temporary mappings and multiplexing it is expensive as it requires TLB invalidations. Additionally, accesses to these temporary remapping requires complex, time-consuming offsetting to translate a guest physical into a kernel addressable form which further reduces efficiency.

A direct switch into the guest physical address space is impossible as it does not have the L4 microkernel mapped into it. To overcome these issues we decided to introduce a special address space, the *vTLB-handler space*. Figure 4.3 gives an overview. This space is only for implemtation, it is not visible to the user. It has the L4 microkernel mapped into the upper gigabyte, like all native L4 address spaces. The user-level portion of the address space contains selected chunks of the guest physical address space. This allows to use the hardware TLB to efficiently translate guest-physical into host-physical addresses, thus to walk and access the guest's page-table. In our current implementation, we simply mirror the lower three gigabytes of guest physical mem-

ory. On a VM-Exit, when the execution switches to L4, this vTLB-handler space is activated, as most events are related to memory virtualization.

In our current implementation we use a rather simple population scheme for the vTLB-handler space. Memory mappings into the guest physical memory space are simply mirrored into the handler space if they are located in the user portion of the address space. This is achieved by a special treatment of the virtual machine address space in the mapping implementation of map_fpage(). An UNMAP on a region in the guest physical address space implies the revocation of matching memory in the handler space, too. This happens automatically, as the handler space received all its mappings via map_fpage().

While doing a guest page-table walk the in-kernel vTLB algorithm may cause page-faults in the handler space. These page-faults are caught and transformed into virtualization fault messages to the virtual machine's pager thread. To handle such faults, the state of the VCPU (such as the IP and the fault address) would be enough to reconstruct the exact reason. For efficiency reasons, we extended the VCPU state with two additional fields which contain 1) the guest-physical address which causes the fault and 2) the access mode (such as read or write). The user-level monitor can then check this address if it matches with the location of a memory-mapped device, or if it is a true physical memory resource fault. This behavior conforms to the expected behavior for hardware supported vTLB management as stated in AMD's Pacifica specification [8].

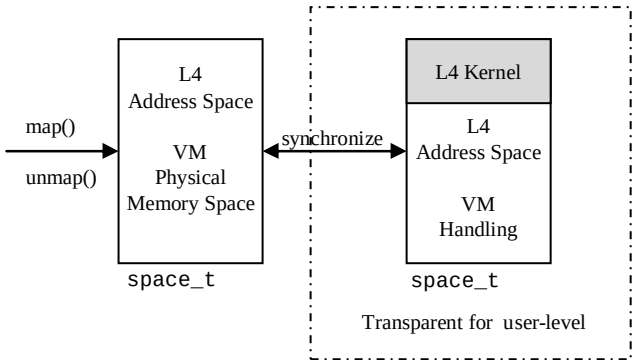


Figure 4.3: Efficient handling of guest page-table walks requires access to huge portions of the guest physical address space. The vTLB-handler space contains the lower 3 GB of the guest physical address space, avoiding temporary mappings in the L4 microkernel.

Optimization: Prefetching Shadow Page-Table Entries

The vTLB operations such as the miss, fill, and status information updates of the guest page-table entries (e.g. the accessed and dirty bits) are very frequent. Under normal conditions they are the most frequent operations in the complete platform virtualization and therefore major to the overall virtualization performance. To optimize the performance of the software implementation of the vTLB, advanced caching similar to software loaded TLB handlers can be used [11, 71, 73, 88, 98].

In the virtualization system the vTLB handlers can use more advanced mechanisms as the VM-Exit causes a high, overhead socked. Optimization of the vTLB requires to

do more than doing the required minimum to resolve the current fault, but to avoid misses in the future, for instance, prefetching or tracking of page-table writes [7].

Prefetching includes the GDT, as well as the current instruction and stack pointer.

4.2.6 Further Resources

IO Port Space. To control pass-through access of a virtual machine to the physical IO ports we reuse L4's IO bitmap implementation. We configure all VCPUs in a virtual machine to use the same bitmap. The monitor can use IO-fpage mappings to manipulate the virtual machine's access. Access to an IO-port which is not available to the virtual machine generates a virtualization fault.

Model Specific Registers. The hardware virtualization provided by VT-x supports three modes for management of model specific registers (MSR): (i) physical access to global MSRs, (ii) virtual machine local values which are saved and restored on a world switch, and (iii) emulation of the MSRs by trapping the instructions which access the MSRs.

Currently, L4 allows only the privileged address spaces to access the model specific registers. In these spaces the execution of the rdmsr and wrmsr instruction causes a general protection fault on the processor which the kernel uses to transparently emulate the behavior of these instructions. We extended L4's current model to unify the handling of virtual machines and the native microkernel environment. We allow each address space controlled access to MSRs similar to the handling of IO ports. Access is controlled by a MSR space which is part of L4's address space abstraction.

For native operation the MSR-fpage establishes mappings which grant/map read and write access to a physical MSR. The accessing instructions still cause general protection faults but depending on the actual rights of an address space, the L4 microkernel emulates the instruction, or synthesizes a MSR fault IPC to the pager thread. The UNMAP system call is extended to support revocation of MSR access permissions.

For the virtual machine environment we support all hardware modes.

- **Physical MSR.** Mapping a MSR to a virtual machine allows the virtual machine to access the machine's MSR. To distinguish between VM local and system-wide access we leverage the x-bit of the MSR-fpage. It defines if a virtual machine has its own instance of the MSR or uses the global value (like the native L4 address space).
- **Emulating MSRs.** Access of a virtual machine to a MSR which insufficient permissions causes a virtualization fault message, which the user-level monitor can use to emulate the effect of the instruction.

Time Stamp Counter. The time stamp conter register (TSC) is a processor register which is automatically increased on every processor cycle. The guest operation system may use it to measure time distances. To maintain a consistent view with other – off processor – timing devices such as the programmable interrupt timer device (PIT) or the real time clock (CMOS RTC), the user-level monitor needs full control of the time stamp counter register [93]. The hardware virtualization extensions provide two modes to virtualize the time stamp counter register: emulation or offsetting. In emulation mode, access to the counter register generates a fault, causing a virtualization fault message to the user-level monitor. Emulation causes high overhead if the guest

frequently accesses the register. The authors in [93] suggest to first enable emulation and then switch to the physical TSC for performance reasons. In offsetting mode the guest can directly access the physical TSC register. But accesses to the register returns the physical TSC modified by an offset. This allows full hardware performance and retains a VCPU local timestamp value.

To completely isolate the VCPU the user-level monitor needs to keep track when the VCPU thread is scheduled to run on the physical processor so that it can adapt the offset. Unfortunately, L4 does not provide a way to get notifications when a thread is being scheduled. In our implementation the L4 microkernel automatically maintains a consistent view of the time stamp counter register (refer to Figure 4.4). When L4 preempts a VCPU thread it saves the current time stamp counter. On reactivation, L4 modifies the VCPU's offset so that the VCPU's time stamp counter resumes at its last value. In result the VCPU thread does not leak information if it has been preempted by L4. The user-level monitor, on the other hand, keeps a consistent view between the time stamp counter value and the other timing devices by synchronizing the offset with the other platform devices.

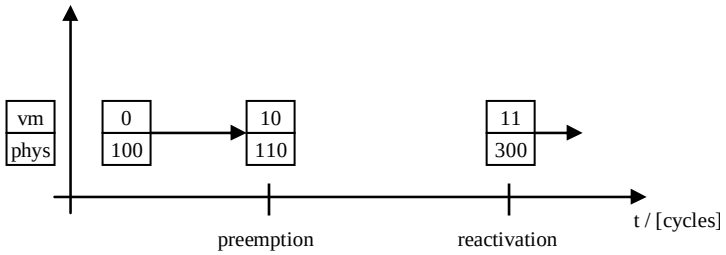


Figure 4.4: The in-kernel TSC handler maintains the VM local time stamp counter offset to create a monotonically increasing counter.

Floating Point Resources. In our approach the L4 microkernel transparently multiplexes the VCPU registers including the FPU state. L4 already supports lazy FPU multiplexing. We extended L4's model to include VT-x's FPU virtualization based on the description proposed in [72].

4.3 The User-Level Monitor

The user-level monitor application creates and maintains the virtual machine environment. In this section we describe our implementation of a small user-level monitor application which utilizes important aspects of virtualization. In its current implementation it is able to run the L4 microkernel as a guest. In the following we describe key points of our implementation.

4.3.1 Architecture

The user-level monitor implements the allocator and the interpreters of a conventional VMM architecture. The monitor is a normal L4 application which can interact with other system servers to implement its services. In its current implementation the user-level monitor runs as the privileged root-task of a L4 system. This is only to reduce

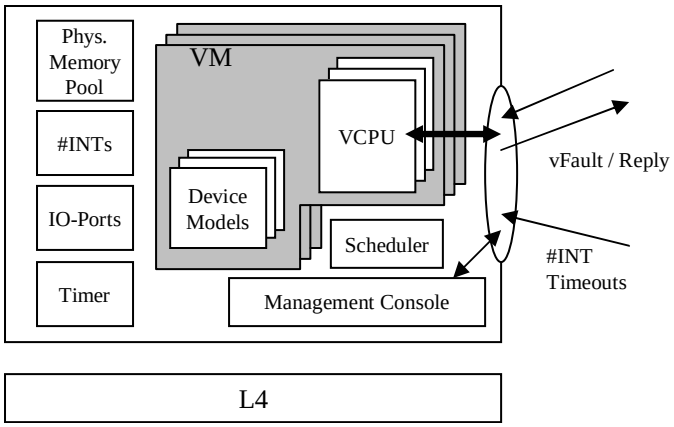


Figure 4.5: User-Level Monitor Architecture: The monitor consists of services which handle the physical resources. The user has full control of the monitor through the management console. The virtual machine representation contains all resources available to it, especially the device models. The monitor provides an IPC interface which dispatches incoming events to the virtual machine.

the implementation complexity because it allows the monitor to have full access on all system resources. All system services are based on IPC an thus can be divided into isolated system servers of a component based system, for instance into a task-server or memory-server.

At system startup the user-level monitor grabs all physical resources including physical memory, IO-ports and interrupts. It partitions the physical resources onto the started virtual machines to securely isolate them. The granted physical memory is registered in a local memory page pool. The page pool supports 4K and 4M pages. This increases the efficiency of the guest mappings.

After all physical resources are registered the user-level monitor installs its global services. The timer service offers time based events in the user-level monitor. The physical interrupt handler manages physical interrupts, for example the keyboard interrupts for the management console. The management console provides the user full access for resource control and debugging of the virtual machines and the monitor itself.

The last service is the module manager. It parses the system configuration and then starts to initialize the requested virtual machines. Virtual machine initialization is presented in Section 4.3.2. After all virtual machines are created, the user level-monitor enters the event loop to service incoming requests. All events are based on L4's IPC. Upon receiving an IPC message it inspects the message which may be of three different types:

- **Virtualization Faults.** The monitor receives virtualization fault messages of the running VCPU threads. These faults are handled using the faulting virtual machine, which is detailed in the next section.
- **Physical Interrupts.** The monitor receives all physical interrupts designated to the monitor itself or the virtual machines. These interrupts may be used to trigger device emulations or may be directly injected into the VCPU.

- **Time-based Events.** The IPC receive-timeouts of the event loop are used to handle time-based events in the user-level monitor.

4.3.2 Virtual Machine Representation

The virtual machine representation is the major module of the user-level monitor application. It contains a full representation of the virtual machine including the allocated physical resources, the virtual processor handling, and the device emulators forming the emulated machine platform.

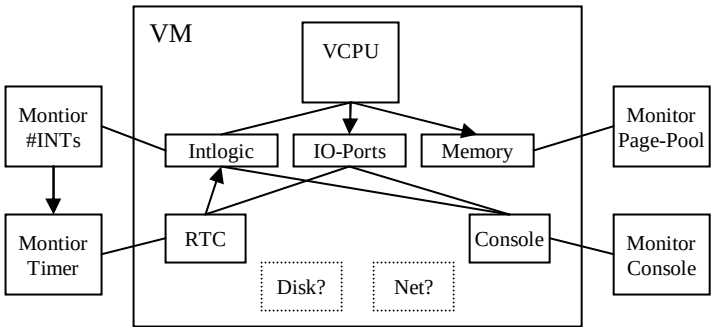


Figure 4.6: Virtual machine representation in the user-level monitor.

Processor

The processor is the major endpoint for the virtualization fault protocol as VM-faults are caused by the VCPU threads. The processor receives all virtualization related events. The processor module handles the L4 virtualization fault messages and the reply. At creation of a VCPU thread it defines the state transfered on a virtualization fault. It holds the VCPU state at the time of the fault. Depending on the fault reason the processor module may trigger device emulation, for instance on a fault caused by in or out on devices. To efficiently translate the fault message into typed VCPU registers we store all VCPU registers in an array arranged on the index in the mask bit-field. This array is overlaid with the exact types of the VCPU register fields. The array is only used for the load/store to handle the fault protocol. To handle faults the exact types can be used, to achieve type safety. Handling of the fault the monitor requires updates on a subset of the registers retrieved with the fault message. The VCPU registers contained in a fault message are simply identified by a bit-field. This field is directly used for the reply mask of the virtualization reply message

Scheduling. The user-level monitor application must be able to control the execution of the virtual machine's processors and needs to keep track of their execution status. Similar to the domain scheduling used in Xen [18] we introduced VCPU thread states to keep track of the different states of a virtual processor. But unlike Xen's implementation we only require two states, as the actual scheduling on the physical processor is done within L4. In any point in time, a VCPU thread can be in one of the following states:

- **Running.** The VCPU thread is currently running the guest and scheduled like any other thread in the L4 system.

- **Blocked.** The VCPU thread does not execute guest instructions. It has send a virtualization fault and is now waiting for the virtualization reply message to resume its execution. Some events such as the execution of the hlt and pause instructions require to suspend the VCPU thread until it is reactivated by an interrupt. In such cases the event loop must not reply to the current VCPU thread but simply wait for other IPC messages.

Physical Memory

Guest physical memory is handled by a page-table like structure in the user-level monitor, the region manager. It maps guest physical addresses of a virtual machine to addresses valid in the user-level monitor. On a physical memory fault the monitor allocates memory from its local page pool, registers it in the VM's region manager and maps the memory into the virtual machine using the virtualization reply message. The region manager holds additional control bit in its page-table to indicate that regions of the guest physical address space are occupied by a guest's device. Access to such a region is not serviced with a mapping but triggers the emulation of the registered memory mapped device.

Startup Environment

The virtual machine monitor currently creates a startup environment which complies to the multiboot standard. This environment is required to run the L4 microkernel as a guest. The processor is set into protected mode with paging disabled using a flat segment model [27]. The monitor copies the guest operating image into the guest physical memory and installs the multiboot meta information structures. After that, the VCPU's register context is initialized as required by the standard using a virtualization reply message to start the VCPU thread. In the case of a L4 guest, kickstart runs and later hands over to L4.

Most operating systems expect the real mode to do the first stages of startup. Currently, we do not support the real-mode execution as it is not directly supported by the Intel processor [50]. The real-mode execution environment must be emulated in software, for example by porting whole system simulators such as Bochs [55] (refer to Section 2.1) to the user-level monitor. The virtual machine environment is first emulated in software; when the guest tries to switch into a processor mode directly supported by the physical processor the user-level monitor can migrate the guest into a virtual machine environment supported by L4 microkernel's virtual machine. The software simulator requires support to externalize the state and load it into the L4 virtual machine.

Device Emulation

In the following sections we roughly describe our implementation of selected device emulation modules.

Interrupt Logic. To manage the interrupts of a virtual machine we implemented an interrupt logic based on the Intel 8269A chip [48]. We implemented two models for interrupt delivery: synchronous piggy-backing of interrupts on virtualization fault

replies, and an asynchronous requests for interrupt delivery using EXCHANGEREGISTERS. A device which raises a virtual interrupt can decide which method should be used to deliver the interrupt into the VM.

Timer Device. Timing in the virtual machines is based on the model proposed in [93]. In our current implementation this model includes the handling of the timestamp counter register and the CMOS real-time clock.

For the virtualization of the real-time clock we need to inject periodic events into the virtual machine. To handle such events we introduced an event queue to the monitor. After the virtual machine has configured its RTC device model by virtualization faults, the configuration is transformed into an event and enqueued into the list. On a virtualization reply, the head of the list is used to determine the timeout of the following IPC receive operation of the event loop. When the monitor thread receives an IPC it is checked if the IPC receive failed because of a timeout violation. This indicates that an event from the event queue needs to be processed. For efficiency reasons we do not handle only the first event of the event queue but all events in a certain time span. In case of the real time clock device, the event raises a timer interrupt at the virtual interrupt controller of the associated virtual machine. The pending virtual interrupt then injected into the VCPU thread to signal a timer tick. The event list distinguishes between one shot and periodic events. Periodic events such as the timer event are not erased but automatically enqueued into the list based on their cycle time.

Console I/O Interface. To control the operating system inside the virtual machine we implemented a simple serial console IO device emulation. It allows basic interaction with the virtual machine. The device emulates an NS16550A chip. For the emulation we configured the VCPU to trap on in and out instructions on the devices port range.

Data output of the virtual machine into the device emulation of the serial console can be configured to be directly displayed on the management console of the user-level monitor. To feed the virtual machine with input we introduced a management interface to the monitor which is based on the L4 kernel debugger interface. It grabs the physical keyboard and allows to deposit keyboard input into console devices of arbitrary virtual machines.

DMA. Direct memory of a virtual machine can be provided in two ways: emulation and direct access to a DMA-able physical device. Emulation can be achieved like all other devices. Direct access of the virtual machine to the physical device needs special care. The monitor has to trap the start of the DMA operation. It must patch the configuration of the start and end addresses of the transfer. The virtual machine uses guest-physical addresses which need to be translated into host physical addresses. The monitor also has to be aware that for the time of the DMA operation the addressed memory block must not be unmapped by itself or its providers, higher-up in the mapping hierarchy. To ensure this, the monitor may priorly do an UNMAP of the memory and exchange this memory location with a mapping of special memory from its pinned memory pool.

Chapter 5

Evaluation

This chapter intentionally left out. For more information please contact contact@l4ka.org.

Chapter 6

Conclusion

This chapter concludes this diploma thesis with a summary followed by a suggestion for further work.

6.1 Contribution of This Work

This thesis addressed microkernel-based virtualization systems. It integrated a generic interface to the L4 microkernel, to maintain virtual machines from user-level. Microkernel-based, hardware-supported full virtualization systems are a novel approach and have not been published in previous work. Our approach has several benefits.

The microkernel approach introduces only a thin layer of trusted software similar to hypervisor based architectures. Further, it allows to co-locate virtual machines and native microkernel applications like host based virtual machine architectures. The microkernel establishes fine grained resource and execution control. Native microkernel applications include services to manage of the virtual machines. The virtual machines are controlled using L4's native mechanisms. A new address space mode was introduced to represent the virtual machine environment. For efficiency reasons, a new fault protocol was introduced to control the VCPU.

The Contributions of this thesis are:

L4 API proposal. (see Appendix A)

This thesis specified an API proposal to integrate the virtualization extensions for the Intel VT-x to the L4 microkernel. The extensions are fully integrated into L4's native interface and use L4's native mechanisms to control the virtual machine. The extensions are fully backwards compatible by defining only system call parameters previously defined as must-be-zero, thus invisible for native applications which do not require these extensions.

A prototype implementation. (Section 4)

We developed a prototype microkernel which implements the virtualization extensions and a user-level monitor. Both components cover all important parts of virtualization as proposed L4 API; it allows to run the L4 microkernel as a guest operating system.

6.2 Suggestions for Future Work

This thesis raises following major areas for future work:

First, the implementation focused on the Intel virtualization extensions VT-x. In the next step the virtualization extensions provided by AMD Pacifica should be included. Second, support for SMP systems may require extensions in the microkernel as well as the user-level monitor application. Third, the microkernel requires a more advanced implementation of the shadow page-table mechanism, to support all virtual memory modes of the IA-32 architecture. Further, performance improvements in the handling of the address spaces switches, which require complex three-way trade-offs among trace costs, hidden page-faults and context-switch costs, are required. Forth, we need a more sophisticated user-level monitor application which provides support for real-mode emulation to run legacy operating systems. The monitor should also provide more device models to fully utilize the kernel extensions. Fifth, the virtualization protocol may be an attractive replacement for the static exception protocol used for native L4 applications. It may provide an convenient way to access the execution context of architecture which have a huge register set, for instance, the IA-64 architecture.

Appendix A

Proposed L4 API Extensions

Besides normal thread execution, L4 provides an virtualization mode: full virtualization mode. Virtualization is tightly integrated with L4's normal execution model. In virtualization mode, threads have access to an extended ISA and have restricted access to L4-specific features.

Full virtualization mode (FVM) uses IA32's virtualization hardware extensions: Intel VT-x or AMD Pacifica. Threads that execute in FVM have access to an extended architecture that includes parts of the privileged instruction set. VT-x and Pacifica partially mirror the processor state in shadow registers or require intervention of an external virtual machine monitor.

Address Space

In full virtualization mode, the L4 execution and resource model is mapped onto a *physical* machine model. A thread that executes in a FVM has access to the privileged part of the platform architecture and runs with an additional memory translation. Depending on the hardware support for double paging, L4 either utilizes the hardware feature or provides a transparent translation of guest-virtual-to-host-physical translations (as opposed to guest-virtual to guest-physical).

Extended Thread State

An thread inside a FVM space represents a virtualized physical processor for the virtualization FVM space. It holds all privileged and unprivileged registers of the physical processor. VM-faults cause virtualization fault messages to efficiently manage critical instructions. Virtualization fault replies allow mapping memory into the FVM space an protocol items allow read/write access to the VCPU' state. EXCHANGEREGISTERS grants asynchronous access by forcing virtualization faults.

The virtualization extensions introduced new kernel feature strings:

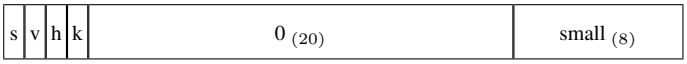
String	Feature
"uvm"	Kernel has user-level virtualization support enabled.
"hvm-vt"	Kernel has full virtualization support using Intel's VT-x.
"hvm-pacifica"	Kernel has full virtualization support using AMD's Pacifica.

A.1 SPACECONTROL

The SPACECONTROL system call has an architecture dependent *control* parameter to specify various address space characteristics. For IA-32, the *control* parameter has the following semantics.

Input Parameters

control



v	The v field denotes the virtualization mode for all threads in the address space. The v field can only be specified for inactive address spaces and is ignored for active address spaces. The availability of the virtualization features is announced as a KIP feature string.
v=0	An address space with no virtualization support.
v=1, h=0	*User-level virtualization mode* provides binary compatibility for user-level applications.
v=1, h=1	*Full virtualization mode* is the hardware assisted virtualization support for IA-32, either Intel's VT-x or AMD's Pacifica. In full virtualization mode, the complete address space is empty and under control of the thread. The thread's state is extended by IA32's processor state including control registers, all segment selectors, debuging registers etc.
v=1, k=1	The k-field indicates that KIP and UTCB should be mapped into the address space. If $k = 0$ the KIP and UTCB areas are not mapped into the address space.

Output Parameters

control

v	Indicates if the requested virtualization mode was effective ($v = 1$). Zero if $v = 0$ in the input parameter.

A.2 EXCHANGEREGISTERS

The EXCHANGEREGISTERS system call has architecture depentent control flags to specify control parameters. The virtualization extensions added three new control bits to force a VCPU to sent a virtualization fault message.

Input Parameters

control

$D = 1$	Force an *direct* virtualization fault with a fault reason of -1.
$I = 1$	Force an *delayed* virtualization fault. The fault is delayed until the VCPU is able to receive *interrupts*. The fault message contains the context of reason 7.
$N = 1$	Force an *delayed* virtualization fault. The fault is delayed until the VCPU is able to receive *non maskable interrupts*. The fault reason is not architecturally defined yet.

Output Parameters

No new output parameters.

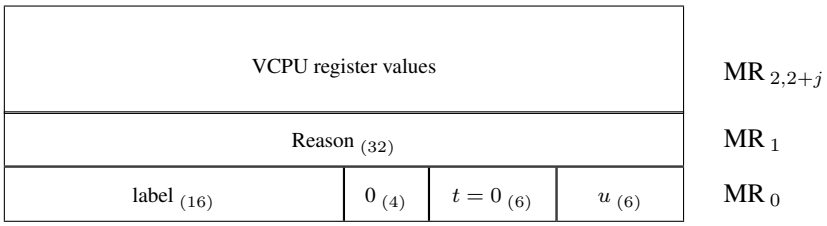
A.3 Virtualization Fault Protocol

The virtualization protocol is defined between a VCPU thread and its registered pager thread. It substitutes the page fault and exception protocol.

A.3.1 Fault Message

The fault message is sent to the VCPU thread's pager at VM-faults which are not handled directly by the L4 microkernel.

To Pager



u	The number of untyped message items. All VCPU registers and the word holding the *reason* are encoded as *untyped* items.
reason	The *reason* which caused the fault.
vcpu registers	The vcpu registers matching the *reason's* VCPU mask; the VCPU mask was specified previously using a *set-fault* item. The message contains j registers.

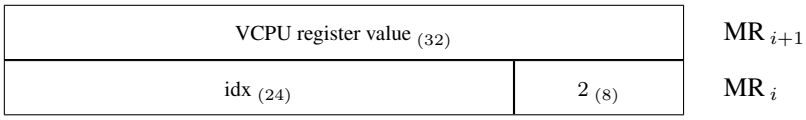
A.3.2 Reply Message

The virtualization reply message resumes execution of the VCPU thread. It may hold typed items. To access the VCPU state several *protocol items* are defined.

Protocol Items

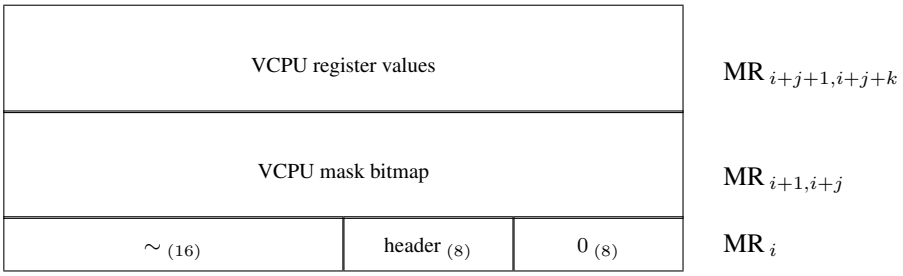
The reply message may consist of several protocol items which are presented next. There exist two types of items: get and set.

Set-Single This item writes exactly one VCPU register.



idx	The addressed VCPU register.
value	The value written to the specified register.

Set-Multiple This item allows to change multiple VCPU registers simultaneously.

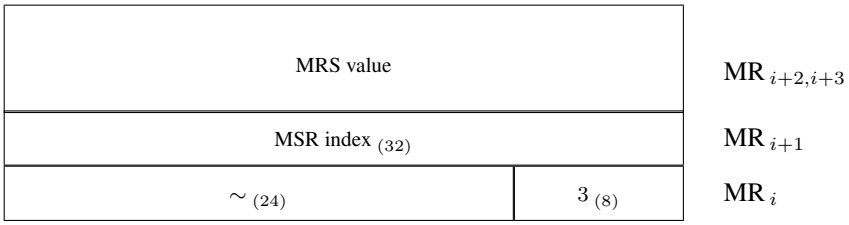


header If bit number n of the *bitmap header* is one, word n of the *mask bitmap* is contained in the item. Words not contained in the item are defined to $zero$. The j words holding the vcpu mask bitmap are ordered with increasing n.

mask Holds the j, possibly compressed, bit-field indicating the addressed VCPU registers. Bits not expressed in the mask field directly are defined to be $zero$.

values The content of the addressed VCPU registers. The VCPU registers have to be attached in the order of increasing bit-field indices.

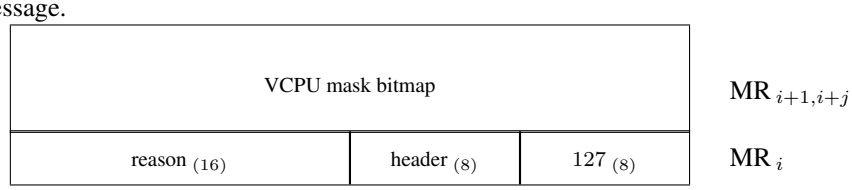
Set-MSR Item This item writes exactly one model specific register (MSR).



index The addressed MSR.

value The value to be written into the specified MSR.

Set-Fault Item This item sets the default content of a specific virtualization fault message.



header If bit number n of the *bitmap header* is one, word n of the *mask bitmap* is contained in the item. Words not contained in the item are defined to $zero$. The j words holding the vcpu mask bitmap are ordered with increasing n.

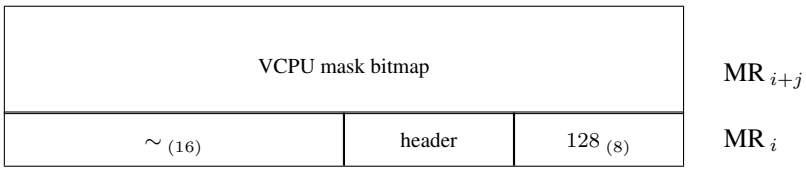
mask Holds the j, possibly compressed, bit-field indicating the addressed VCPU registers. Bits not expressed in the mask field directly are defined to be $zero$.

reason Specifies the virtualization fault reason for which the new mask is to be used.

Get-Single Request to read a single VCPU register.

index The requested VCPU register.

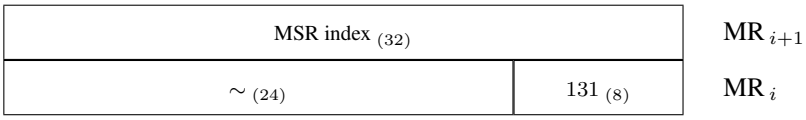
Get-Multiple Request to read multiple VCPU registers.



header If bit number n of the *bitmap header* is one, word n of the *mask bitmap* is contained in the item. Words not contained in the item are defined to $zero$. The j words holding the vcpu mask bitmap are ordered with increasing n.

mask Holds the j, possibly compressed, bit-field indicating the addressed VCPU registers. Bits not expressed in the mask field directly are defined to be $zero$.

Get-MSR Item Request to read a model specific register (MSR).

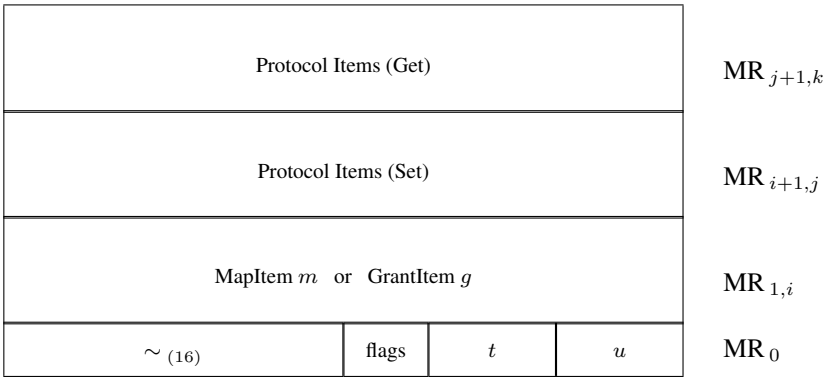


index The requested model specific register (MSR).

Reply Message

The reply message can contain several different types of items: map and grant items to populate resources into the virtual machine, and get/set items to access the VCPU registers. If the reply message contains any get items the VCPU thread does not resume execution but immediately sends the requested VCPU registers. All get items must be specified after the set items at the end of the message.

From Pager.



t Number of typed MRs occupied by MapItem and GrantItem .

u Number of untyped MRs occupied by virtualization fault protocol items.

A.3.3 Thread-Startup Protocol

The thread startup protocol for VCPU threads – threads created inside a virtualization space – requires to send a virtualization fault reply message for the VCPU thread's pager.

A.4 MSR-Fpage

Access to processor's model specific registers is controlled via. The minimal granularity is 1. An MSR-fpage of size $2^{s'}$ has a 2^s-aligned offset address $sndbase + offset$, i.e $offset \bmod 2^s$=0.

control

r	Allow read access to the specified MSRs.
w	Allow write access to the specified MSRs.
g	Ignored for mappings into non-FVM spaces. For mappings into FVM space $v = 0$ grants access to the system MSR. On $v = 0$ the kernel installs a VCPU local MSRs which is transparently multiplexed.
s'	$2^{s'}$ is the size of the region.
offset	Offset specifies the lowest 16 bits of a MSR base address.

Bibliography

[1] ACM Press, December 1995.

[2] IEEE, October 1996.

[3] ACM Press, October 5–8 1997.

[4] ACM Press, October 2003.

[5] USENIX, April 2005.

[6] USENIX, June 2005.

[7] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, USA*, October 21–25 2006.

[8] Advanced Micro Devices, Inc., One AMD Place, P.O. Box 3453, Sunnyvale, California, USA. *AMD64 Virtualization Codenamed 'Pacifica' Technology*, May 2005. Order number 33047.

[9] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC), Bond University, Gold Coast, QLD, Australia*. IEEE Computer Society, January 29–February 2 2001.

[10] J. D. Bagley, E. R. Floto, S. C. Hsieh, and V. Watson. Sharing data and services in a virtual machine system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Austin, TX, USA*, pages 82–88. ACM Press, 1975.

[11] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedins of the 1st Symposium on Operating System Design and Implementation (OSDI), Monterey, California, USA*, pages 243–253. ACM Press, 1994.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, Anaheim, CA, USA* [5], pages 41–46.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th Symposium on Operating System Prinicples (SOSP), Copper Mountain Resort, CO USA* [1], pages 1–11.

[14] John Bruno, Jose Brustoloni, Eran Gabber, Avi Silberschatz, and Christopher Small. Pebble: A component-based operating system for embedded applications. In *Workshop on Embedded Systems (ES), Cambridge, MA, USA*, pages 55–65. USENIX, May 29–31 1999.

[15] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions Computer Systems*, 15(4):412–447, November 1997.

[16] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *USENIX Annual Technical Conference, Anaheim, CA, USA* [5], pages 1–4.

[17] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS VIII), Elmau, Germany*, pages 14–19. USENIX, May 2001.

[18] Ludmila Cherkasova and Rob Gardner. Measuring cpu overhead for i/o processing in the Xen virtual machine monitor. In *USENIX Annual Technical Conference, Anaheim, CA, USA* [5], pages 387–390.

[19] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th Symposium on Operating System Principles (SOSP), Banniff, CANADA*. ACM Press, October 2001.

[20] Christopher Clark, Keir Fraser, Jacob Gorm Hansen Steven Hand, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, Boston, Massachusetts, USA*. USENIX, May 2005.

[21] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), September 1981.

[22] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, September 1970.

[23] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with segemented architecture, October 1998. US Patent: 6397242.

[24] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Linux Showcase Conference, Atlanta, GA, USA*. USENIX, October 2000.

[25] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles, Bolton Landing, NY, USA* [4], pages 164–177.

[26] Samuel T. King George W. Dunlap and Peter M. Chen. Operating system support for virtual machines. In *USENIX Annual Technical Conference San Antinio, Texas, USA*, pages 71–84. USENIX, June 2003.

[27] Brian Ford. The multiboot standard 0.6 . `http://www.uruk.org/orig-grub/boot-proposal.html`.

[28] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedins of the 2nd Symposium on Operating System Design and Implementation (OSDI), Seattle, Washington, USA*, pages 137–151. USENIX, October 1996.

[29] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[30] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles, Bolton Landing, NY, USA* [4], pages 193–206.

[31] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium, San Diego, California, USA*, February 2003.

[32] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, Kolding Denmark, September17–20 2000. ACM Press.

[33] Robert Philip Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Division of Engineering and Applied Physics, Harward Universitry, Cambridge, MA, USA, February 1972.

[34] Robert Philip Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems, Cambridge, Massachusetts, United States*, pages 74–112. ACM Press, 1973.

[35] Robert Philip Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.

[36] Stefan Götz. Asynchronous communication using synchronous ipc primitives. Master's thesis, System Architecture Group, University of Kalrsruhe, May 2003.

[37] Julian B. Grizzard and Henry L. Owen. On a μ-kernel based system architecture enabling recovery from rootkits. In *Proceedings of the 1st International Workshop on Critical Infrastructure Protection (IWCIP), Darmstadt, Germany*. IEEE, November 2005.

[38] Andreas Haeberlen. Using platform-specific optimizations in stub-code generation, July 2002.

[39] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS X), Santa Fe, NM, USA* [6].

[40] Herman Härtig, J. Wolter, and Jochen Liedtke. Flexible sized page objects. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, USA* [2].

[41] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th Symposium on Operating System Principles (SOSP), St. Malo, France* [3].

[42] Gernot Heiser. Secure embedded systems need microkernels. *The USENIX Magazine*, 30(6):9–13, December 2005.

[43] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? Technical Report PA0005103, National ITC Australia and University of New South Wales, October 2005.

[44] Christian Helmuth, Alexander Warg, and Norman Freske. Micro-sina – hands-on experiences with the Nizza security architecture. In *Proceedings of the D.A.CH Security, Darmstadt, Germany*, March 2005.

[45] Christian Helmuth, Andreas Westfeld, and Michael Sobirey. μSINA – eine mirokernbasierte systemarchitektur für sichere systemkomponenten. In *In achter Deutscher IT-Sicherheitskongress des BSI*, pages 439–453. Secumedia-Verlag, May 2003.

[46] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing tcb size by using untrusted components – small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop, Leuven, Belgium*. ACM Press, August 2004.

[47] IBM. *PowerPC Operating Environment Architecture, Book III*, 2005.

[48] Intel Corporation. *8259A Programmable Interrupt Controller*. Santa Clara, CA, USA, December 1988. Order number 231468-003.

[49] Intel Corporation. *Lagrande Technology Architectural Overview*. Santa Clara, CA, USA, September 2003. Order number 252491.

[50] Intel Corporation. *Intel IA-32 Architecture Software Developer's Manual: Volume 3B: System Programming Guide, Part 2*. Santa Clara, CA, USA, January 2006. Order number 253669.

[51] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using ipc redirection. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII), Rio Rico, AZ, USA*, March 29–30 1999.

[52] Ganesh Venkitachalam Jeremy Sugerman and Beng-Hong Lim. Virtualizing i/o devices on VMware workstation's hosted virtual machine monitor. In *USENIX Anual Technical Conference, Boston, Massachussets, USA*. USENIX, June 2001.

[53] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceeno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating System Principles (SOSP), St. Malo, France* [3], pages 52–65.

[54] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference, Anaheim, CA, USA* [5], pages 1–15.

[55] Kevin Lawton. The bochs ia-32 simulator project. `http://bochs.sourceforge.net`.

[56] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[57] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA*. USENIX, December 2004.

[58] Jochen Liedtke. *Architektur von Rechensystemen*. Springer-Verlag, March 1992.

[59] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles (SOSP), Asheville, NC, USA*. ACM Press, December 1993.

[60] Jochen Liedtke. On μ-kernel constuction. In *Proceedings of the 15th Symposium on Operating System Prinicples (SOSP), Copper Mountain Resort, CO USA* [1].

[61] Jochen Liedtke. Microkernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, USA* [2].

[62] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[63] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The L4Ka vision, April 2001.

[64] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS-VI), Cape Cod, MA, USA*. USENIX, May 5–6 1997.

[65] Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the lava hit-server. In *USENIX Anual Technical Conference, New Orleans, Lousiana, USA*. USENIX, June 15–19 1998.

[66] R.A. MacKinnon. The changing virtual machine enviornment: Interfaces to readl hardware, virtual hardware, and other virtual machines. *IBM Systems Journal*, 18(1), 1979.

[67] Daniel J. Magenheimer and Thomas W. Chrisitan. vblades: Optimized paravirtualization for the itanium processor family. In *Prceedings of the Third Virtual Machine Research and Technology Symposium*, pages 73–82. ACM Press, May 6–7 2004.

[68] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *ACM Sigops Operating System Review*, 40(2):8–11, April 2006.

[69] R. Meushaw and D. Simard. NetTop: Commercial technology in heigh assurance applications. *Tech Trend Notes*, 9(4), 2000.

[70] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[71] David Nagle, Richard Uhlig, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-mannaged tlbs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA), San Diego, CA, USA*, pages 27–38, May 1993.

[72] Gil Neiger, Amy Santoni, Felix Lang, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–178, August 2006.

[73] Gilbert Neiger, Stepen Chou, Eric Cota-Robles, Stalinselvaray Jeyasingh, Alain Kagi, Michael Kozuch, Richard Uhlig, and Sebastian Schönberg. Virtual tranlation lookaside buffer, December 2000. US Patent: 6907600.

[74] Gerald J. Popek and Robert Philip Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[75] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallik. Xen 3.0 and the art of virtualization. In *Proceedings of the Linux Symposium, Ottawa, Ontario, Canada*, pages 65–77, July 20–23 2005.

[76] Himanshu Raj, Karsten Schwan, and Jimi Xenidis. Scalable i/o virtualization via self-virtualizing devices. Technical report, February 2006.

[77] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Palo Alto, CA, USA*. ACM Press, October 5–8 1987.

[78] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the Ninth USENIX Security Symposium, Denver, Colorado*. USENIX, August 2000.

[79] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, JL Griffin, and S. Berger. sHype: secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM TJ. Watson Research Center, Yorktown Heights, NY, February 2005.

[80] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[81] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer Magazine*, 28(5):32–38, May 2005.

[82] Jan Stoess. I/o-flexpages on the x86-architecture, May 31 2002.

[83] The L4 Team. L4hq. Home of the L4 community, `http://l4hq.org`.

[84] The L4Ka Team. The L4Ka::Pistachio microkernel. `http://www.l4ka.org`.

[85] The L4Ka Team. The l4ka::virtualization resource monitor. `http://www.l4ka.org/projects/virtualization/resourcemon/`.

[86] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification – now! In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS X), Santa Fe, NM, USA* [6].

[87] R. Uhlig, G. Neiger, D. Rogers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *IEEE Computer*, 38(5):48–56, May 2005.

[88] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed tlbs. *ACM Transactions on Computer Systems (TOCS)*, 12(3):175–205, 1994.

[89] Volkmar Uhlig. *Scalability of Micokernel-Based Systems*. PhD thesis, System Architecture Group, University of Karlsruhe, June 2005.

[90] Virtutech Inc. Selected publications on simics. `http://www.virtutech.com/about/research/selected-pubs.html`.

[91] VMware. Virtual machine interface specification. `http://www.vmware.com/vmi`.

[92] VMware. VMware ESX Server. `http://www.vmware.com/products/server/esx_features.html`.

[93] VMware. Timekeeping in vmware virtual machines, July 2005.

[94] Carl A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation, Boston, Massachusetts, USA*. USENIX, December 2002.

[95] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2001.

[96] A. Whitakter, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI), Boston, MA, USA*. USENIX, December 2002.

[97] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating System Principles (SOSP), Austin, TX, USA*, pages 63–76. ACM Press, November 8–11 1987.

[98] Gerald D. Zuraski and Micael T. Clark. Translation lookaside buffer flush filter, January 2003. US Patent: 6510508.