L4-Based Real Virtual Machines
– An API Proposal –

Sebastian Biemueller and Uwe Dannowski
System Architecture Group
Universität Karlsruhe (TH), Germany

Abstract— **Virtual machines (VMs) recently regained attention as a solution to problems not only in high-performance computing, servers, and desktops, but in embedded systems as well. For example, network-enabled embedded systems use virtual machines to provide hardened subsystems for banking, encryption, and digital rights management.**

Virtual machine systems and microkernels share a common set of goals such as reliability, security, isolation and, flexibility, so that integrating VMs and microkernels is a promising approach. In fact, modern microkernels already provide the abstractions and mechanisms necessary to cater for virtual machines.

In this paper we show how virtual machine concepts map to the concepts of a microkernel, the L4 microkernel. We identify shortcomings of the current kernel API with respect to virtual machine support and propose a minimalistic set of extensions.

I. Introduction

The microkernel approach is an ideal construction principle for the design of embedded operating systems. It reduces the amount of code that is executing in privileged mode and generally improves security, reliability, and verifiability of the system — aspects crucial for embedded systems [1]. Furthermore, the modular system structure on top of the small kernel contributes flexibility and diversity to the operating system design space and allows, for instance, an untrusted open-source web browsing component to run safely besides a closed-source banking module, or an entertainment console system to run simultaneously with a system component controlling mission-critical special-purpose hardware.

Application of the microkernel-based system construction principle to existing software requires porting. However, software reuse is an important aspect especially in embedded systems where the whole system is often designed for decades whereas rapid development of hardware often obsoletes the underlying platform after only a few years.

Full virtualization bridges this gap. By creating the illusion of a physical machine, a virtual machine monitor can provide a stable interface to software inside the virtual machine despite any changes in the underlying hardware. This is also its weak point: With isolation at the granularity of complete machines, flexible, modular, and efficient systems are hard to build. The virtual machine approach lacks design principles for the construction of the virtual machine monitor, the software inside the virtual machine, and the virtual machine system as a whole.

In this paper we propose the construction of a virtual machine system based on the principles of microkernels. We separate the virtual machine monitor into a necessarily privileged part, the hypervisor, and an unprivileged part, the user-level monitor. The hypervisor and a microkernel are similar enough to justify integration of both. The resulting system comprises a microkernel that also provides for virtual machines and a microkernel-based system on top that includes components for maintaining the virtual machines, as illustrated in Figure 1.

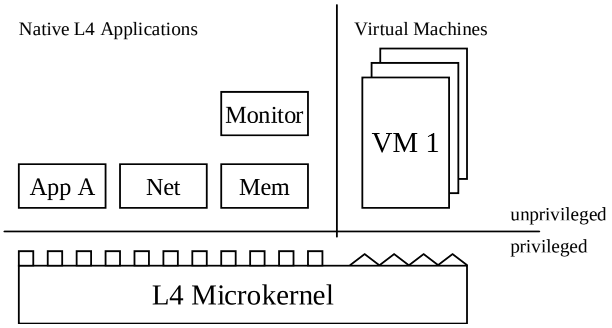

Fig. 1. Native microkernel applications including the unprivileged part of the virtual machine monitor run side-by-side with virtual machines.

Such a system combines the best of both worlds: Virtual machines provide the stable, compatible interface, the platform API, at the granularity of complete machines, whereas the microkernel approach enables construction of arbitrary, well-structured systems.

This paper is organized as follows: In Section II, we briefly describe the operation of a virtual machine, followed by a short introduction to the L4 microkernel. In Section III, we discuss how virtual machine concepts map to L4 concepts and identify shortcomings of the current kernel API. Section IV formulates the necessary changes to the L4 API to support virtual machines. We discuss related work in Section V and conclude the paper in Section VI.

II. Background

A. Virtual Machines

A Virtual Machine (VM) as defined by Goldberg [2] 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. At this level, the 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 devices. These resources can be provided either as pass-through access to a physical resource or by software emulation.

Pass-through access of a physical resource provides the guest with direct access at the full performance of native operation. If the resource can not be securely multiplexed between virtual machines, it must be assigned exclusively to one VM.

Software emulation of a resource is needed for a physical resource which cannot be securely multiplexed, when strong monitoring of the virtual machine's interaction is wanted, or when no physical instance of the resource is available in the host machine. In the virtual machine, each instruction that accesses an emulated resource has to be prevented from execution [3]. Fully virtualizable architectures have the ability to generate traps on such instructions [4]. The VMM catches the trap, and emulates the effects of the instruction on the virtual machine's state and the software model of the resource.

To emulate active components, for instance, a network device, the VMM has to inject interrupts into the virtual machine. The VMM must respect the virtual machine's state which may require to delay the delivery, e.g., the guest has its interrupts disabled.

A virtual machine, while running on the physical processor, needs to be sheltered from events on the physical machine. For example, physical interrupts must not enter directly the virtual machine. Fully virtualizable architectures support this by strongly separating the virtual machine's context from the physical environment, e.g., by performing a world switch before delivering the event.

B. The L4 Microkernel

The L4 microkernel [5] is a second generation microkernel originally developed by Jochen Liedtke at GMD, IBM, and Karlsruhe University. Various versions exist at Karlsruhe University [6], UNSW/NICTA Sydney, and Dresden University of Technology. The kernel offers two abstractions and two major mechanisms.

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.

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.

Mapping 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 identifying a region of the sender's (virtual) address space in an IPC message. Mapping requires mutual agreement of the sender and receiver thread and thus allows save user-level management of address spaces. 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. L4 implements the address space abstraction with whatever hardware mechanisms available, such as TLBs, page tables, and permission bitmaps. The minimum granularity of mapping operations on address spaces is subject to the hardware's capabilities.

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.

Hardware generated events such as *exceptions* and *interrupts* are translated into kernel-generated IPC messages. On a 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 its 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: 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.

III. Design

This section discusses how L4 concepts can be used to provide a virtual machine environment. We first present the general architecture and then discuss selected aspects of VM support in slightly more detail. In fact, L4 already provides the right abstractions and mechanisms to cater for virtual machines; where necessary, we propose minimal extensions or generalizations.

A. Architecture

The architecture is microkernel-based and consists of three major components: the virtual machine, its monitor, and the microkernel.

The *virtual machine* consists of an L4 address space, containing all directly accessible physical resources and one or more L4 threads representing the virtual machine's processors (VCPUs).

The *monitor* defines and maintains the virtual machine's environment. It guarantees isolation by securely controlling the VM's access to physical resources and implements all complex aspects of virtualization, such as emulating instructions accessing resources not directly available to the VM. As a normal L4 thread, the monitor can benefit from the services of other user-level components to build the environment of the virtual machine, for example disk storage or network connectivity.

The *microkernel* provides the execution environment for the VM, the monitor, and the rest of the system. It ensures controlled execution of the guest in the VM, using hardware-supported virtualization techniques where necessary.

B. Resources

To securely isolate a virtual machine, the VM must not have uncontrolled direct access to the physical hardware resources of the host machine. L4's resource mapping mechanism already provides a way to control the permissions of an address space. The monitor simply uses mappings to selectively grant a VM access to a physical resource. In the following, we illustrate this in the context of memory. Some architectures have additional resource spaces, e.g. x86's IO port space. In L4, they are part of the address space abstraction and thus subject to the same mapping mechanism.

Physical Memory: A virtual machine can not have direct access to physical memory as it would circumvent protection. Instead, virtual memory is used [7]. An L4 address space represents the virtual machine's physical memory address space. The monitor populates this space by mapping parts of its own address space into it.

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 complete physical 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 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. Removing these objects would create special cases for VM address spaces resulting in larger changes to the API, and it would preclude later optimizations.

Virtual Memory: We use L4's virtual memory management to provide the VM's physical address space. However, the guest operating system in the VM may want to use virtual memory itself. To maintain the illusion of direct access to memory, the virtual machine system must resolve a guest-virtual address into a host-physical address [7]. 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. Current hardware lacks support for such a cascaded memory translation, called nested paging. Thus, both stages have to be merged into a single translation, the shadow page table, which directly translates a guest-virtual address into a host-physical address. The shadow page table can also be seen as a virtual TLB (vTLB) managed in software and located between the guest's page table and the hardware TLB.

One way to establish the guest-virtual to host-physical translation is to represent the VM's virtual address space as an L4 address space, containing the contents defined by the shadow page table. 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 virtual address space of the virtual machine. However, this approach has several drawbacks:

- L4's mapping mechanism 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 does not expose the distinction between user- and kernel-accessible memory. Extending L4's mapping mechanism accordingly would require to disable this feature 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 vTLB via mapping by the monitor. Since 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.

These problems can be avoided by emulating guest-virtual address spaces transparently inside the kernel. The VM's physical address space is represented by the L4 address space which is maintained by the monitor. Only faults caused by non-present guest-physical memory are propagated to the monitor. Of course, this approach has some disadvantages, too:

- The 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 IPC though.) Furthermore, L4 uses one page table format while the guest operating system may use one of many, increasing complexity of the page table walker for the guest page table.

However, we expect upcoming hardware to natively support nested paging which will remove the need for shadow page tables altogether. 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 to the API.

C. Processor

The processor of a virtual machine is represented by an L4 thread with its associated state (identifier, priority, scheduler, pager, exception handler) extended by the complete architecturally defined processor state. The VCPU behaves like a native L4 thread; especially, the VCPU is scheduled like all other threads in L4.

To emulate the virtual machine's resources such as privileged registers, 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 events: the exception protocol. L4 synthesizes a fault message to the associated user-level handler thread in order to report the reason of the fault, including selected user-visible CPU state. The handler resolves the fault, i.e., by emulating the behavior of the faulting instruction, and sends a reply message back to resume the thread. This message contains the updated CPU register state to be established before resuming the thread.

For virtualization, this static protocol is too inflexible because the VCPU's state is much larger and the relevant state heavily depends on the exact fault reason. Similar to the exception protocol, we propose a *virtualization fault protocol*. For each virtualization fault reason, a pre-defined part of the VCPU state is transferred 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 a *no-resume* item. This item contains a request for additional VCPU state; it causes the VCPU to immediately generate another virtualization fault without resuming the guest. Thus the monitor can iteratively access the complete VCPU register state.

D. Asynchronous Events

To emulate the behavior of active devices, e.g. to inject virtual device interrupts, the monitor has to asynchronously modify the VCPU's state.

L4 already provides a way to asynchronously manipulate another thread through the EXCHANGEREGISTERS system call. EXCHANGEREGISTERS allows manipulation of the thread's instruction, stack pointer and flags register, but only from 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 this model, the monitor can delay asynchronous events and piggyback them on the next 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 as already required by the L4Ka Virtual Machine Technology projects [8], yet with one further extension, a possibility to asynchronously force a virtualization event:

- 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 purposes.
- A *delayed fault* causes the VCPU to raise a virtualization fault on a certain event, for example, the next time the guest is able to receive interrupts. The monitor can then inject pending virtual interrupts.

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

Apart from the VCPU register state, the monitor may need to access the VM's memory. 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.

E. 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 that should not be satisfied with a mapping but trigger an emulation of the accessing instruction.

Therefore, page faults should also use the virtualization fault protocol. Unifying page faults and exception handling is already under discussion in the context of the L4Ka Virtual Machine Technology [8] projects for other reasons such as orthogonality.

IV. Changes to the L4 API

This section describes the changes to the L4 API that enable the design presented in the previous section. The goal is to allow for user-level management of virtual machines with minimal extensions to the L4 API.

The changes fit harmonically into L4's interface; the resulting API is fully backward compatible. No changes to the thread as the abstraction for a multiplexed CPU were necessary. The address space still is the abstraction for isolation and contains the accessible resources. All virtualization-related hardware events are abstracted as kernel-synthesized IPC messages. To handle a virtual machine no further fault handlers had to be added.

A. SpaceControl

The API now differentiates between three modes of an address space. Backwards compatibility is achieved by using two formerly should-be-zero bits to select the mode.

- **Native Mode** This is the normal address space for native L4 threads. No virtualization is used. All L4 services are available.
- **User-Level Virtualization Mode** This mode supports user-level virtualization such as para-virtualization and pre-virtualization. This type of space is not discussed here.
- **Full Virtualization Mode** In full virtualization mode, the physical processor supports virtualization in hardware, the mode addressed in this paper. The address space holds all physical resources of the virtual machine. Initially it is empty, only KIP and UTCB are mapped as parts of the physical address space.

B. ExchangeRegisters

ExchangeRegisters uses two additional flags to raise an asynchronous virtualization fault: One flag, when set, forces the VCPU to raise the fault. A second flag, when set, delays generation of the fault until the VCPU enters a state where interrupts can be accepted. The ExchangeRegisters system call returns, even though generation of the fault may be delayed.

This extension relies on the experimental feature that allows a thread to invoke ExchangeRegisters on all threads for which it is registered as the pager, even in a different address space.

C. Virtualization Fault Protocol

The virtualization fault protocol unifies the page fault and exception protocol for VCPUs. As such, it is based on IPC. The virtualization fault protocol is defined between the faulting VCPU and the thread registered as its pager. It allows the pager to get notifications on virtualization-critical events and to access the VCPU register state.

A *virtualization fault message* is synthesized by the kernel 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 set of registers depends on the fault reason and the architecture.

The pager answers the virtualization fault with a *virtualization reply message*. This message can contain the already available typed items for resource mapping, for example, memory mappings. The reply message can also be used to modify or request more VCPU state. Similar to the general IPC protocol, several new items are defined:

- The *set item* changes exactly one register of the VCPU. The target register is encoded in the item, followed by the new value.
- A *group item* sets a predefined, fixed group of VCPU registers.
- 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, followed by the values.
- The *no-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.

D. Thread-Startup-Protocol

The startup message of a VCPU thread requires a virtualization reply. This message initializes the VCPU register state and activates the VCPU.

V. Related Work

Related work can be found in three areas: virtual machine monitors, microkernels, and integrations of both.

L4Linux [9], a para-virtualized Linux running on the L4 microkernel, was created to evaluate the microkernel's performance. Since then, L4Linux served various projects targeting efficient, secure subsystems using virtualization [10], [11].

LeVasseur et. al introduced pre-virtualization [12], an automated para-virtualization technique, to reduce the effort of porting new guests to their virtualization environment. The guest operating system is compiled with a pre-virtualization compiler that locates virtualization-sensitive instructions and inserts pad bytes for runtime instruction rewriting. The resulting binary can execute on raw hardware as well as in a pre-virtualization environment.

In contrast, binary translation, as used, for example, by VMware to fully virtualize the x86 architecture, transparently traps and patches virtualization-sensitive guest instructions. It achieves good performance but introduces high complexity into the privileged virtual machine monitor [13]. Recent VMware products can also use hardware virtualization extensions where available.

The Xen [14] system is a para-virtualizing VMM for the x86 architecture. Xen's virtual machines are managed by the Xen hypervisor, but all IO is performed on their behalf by a privileged VM. Currently, only a modified Linux operating system is supported as a guest for such a privileged VM. With Xen 3.0, the hypervisor includes mechanisms for memory sharing and efficient communication between virtual machines to allow running each device driver in its own privileged VM in order to achieve stronger isolation [15]. In recent releases, Xen also added support for full virtualization using the x86 hardware virtualization extensions.

The KVM project [16] is another VMM for full virtualization. It adds a kernel driver to the Linux operating system and exports the hardware virtualization features through a special device file. Each virtual machine is represented as a Linux process. A second, associated process maintains the platform environment for the virtual machine. With the driver being a part of the Linux kernel, the VM's trusted computing base includes the whole Linux kernel.

Fluke [17] is a software-based virtualizable architecture. It combines the concepts of microkernels and virtual machines to increase the operating system's extensibility. Operating system functionality is decomposed vertically into layers, called nesters, that allow the environment provided to the application to be stepwise refined. The application's execution environment consists of the hierarchy of nesters the application runs on and thus only includes the operating system services required.

The close relationship between microkernels and virtual machine monitors has been discussed by Hand et . al [18] and Heiser et . al [19]. The authors argue, that microkernels and virtual machines, although their definitions are quite different, follow similar goals. The work presented in [20] reasons towards microkernel-based virtual machine systems for high performance computing.

The work of Hohmuth et . al [21] 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.

Our work implements the same point in that design space, yet we reach it by including support for hardware-assisted full virtualization into a microkernel. The microkernel design allows for fine-grained, component-based systems with a small trusted computing base whereas support for full virtualization enables reuse of legacy (operating) systems only known from purely virtual machine systems.

VI. Conclusion & Future Work

In this paper we presented a minimalistic set of extensions to the API of the L4 microkernel. They enable user-level management of virtual machines based on L4 abstractions and mechanisms and are fully transparent to threads which do not require these virtualization features. This is achieved by defining minimal, backwards-compatible extensions. This work shows that a microkernel and the privileged part of a virtual machine monitor can be integrated, providing the advantages of both worlds in one system without affecting the microkernel's overall performance. Especially, there are no changes to the carefully crafted IPC path.

Currently, a prototype implementation of the proposed API is under development. The target architecture is IA-32 using Intel's virtualization extensions (VT-x) [22].

Since this paper is an API proposal, we encourage any type of constructive feedback.

References

[1] G. Heiser, "Secure embedded systems need microkernels," *The USENIX Magazine*, vol. 30, no. 6, pp. 9–13, Dec. 2005.

[2] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer Magazine*, vol. 7, no. 6, June 1974.

[3] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2001, pp. 1–14.

[4] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, July 1974.

[5] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, Sept. 1996.

[6] University of Karlsruhe, System Architecture Group, "L4 experimental kernel reference manual," Feb. 2006.

[7] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating System Design and Implementation*. Boston, MA, USA: USENIX Association, Dec. 2002, pp. 181–194.

[8] The L4Ka Team, "The l4ka virtual machine technology. http://www.l4ka.org/projects/virtualization."

[9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of microkernel-based systems," in *Proceedings of the 16th Symposium on Operating System Principles (SOSP), St. Malo, France*. ACM Press, Oct. 5–8 1997. [Online]. Available: http://l4ka.org/publications/

[10] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter, "The Nizza secure-system architecture," in *Proceedings of the The First International Conference on Collaborative Computing: Networking, Applications and Worksharing, December 19–21, 2005, San Jose, CA, USA*. San Jose, CA, USA: IEEE Press, Dec. 2005.

[11] C. Helmuth, A. Warg, and N. Freske, "Micro-sina – hands-on experiences with the Nizza security architecture," in *Proceedings of the D.A.CH Security 2005, Darmstadt, Germany*, Mar. 2005.

[12] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser, "Pre-virtualization: Slashing the cost of virtualization," Fakultät für Informatik, Universität Karlsruhe (TH), Tech. Rep. 2005-30, Nov. 2005.

[13] K. Adams and O. 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*, Oct. 21–25 2006.

[14] 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, New York, USA: ACM Press, Oct. 2003, pp. 164–177.

[15] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallik, "Xen 3.0 and the art of virtualization," in *Proceedings of the Linux Symposium, Ottawa, Ontario, Canada*, July 20–23 2005, pp. 65–77.

[16] Qumranet, "KVM: Kernel-based virtual maschine for linux. http://kvm.sourceforge.net."

[17] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *Proceedins of the 2nd Symposium on Operating System Design and Implementation (OSDI), Seattle, Washington, USA*. USENIX Association, Oct. 1996, pp. 137–151.

[18] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. 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*. USENIX Association, June 2005.

[19] G. Heiser, V. Uhlig, and J. LeVasseur, "Are virtual-machine monitors microkernels done right?" National ITC Australia and University of New South Wales, Tech. Rep. PA0005103, Oct. 2005.

[20] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *ACM Sigops Operating System Review*, vol. 40, no. 2, pp. 8–11, Apr. 2006.

[21] M. Hohmuth, M. Peter, H. Hartig, and J. S. Shapiro, "Reducing tcb size by using untrusted components – small kernels versus virtual-machine monitors," in *Proceedings of the 11th ACM SIGOPS European Workshop*. Leuven, Belgium: ACM Press, Aug. 2004.

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