# Device driver reuse via virtual machines


A dissertation submitted to the School of Computer Science and Engineering of
The University of New South Wales in partial fulfillment of the requirements for
the degree of Doctor of Philosophy.

Joshua Thomas LeVasseur

May 2009

# Abstract

Device drivers constitute a significant portion of an operating system's source code. The effort to develop a new driver set is a sobering hurdle to the pursuit of novel operating system ventures. A practical solution is to reuse drivers, but this can contradict design goals in a new operating system. We offer a new approach to device-driver reuse, with a focus on promoting novel operating-system construction, which insulates the new operating system from the invariants of the reused drivers, while also addressing development effort. Our solution runs the drivers along with their original operating systems inside virtual machines, with some minor reuse infrastructure added to the driver's operating system to interface with the rest of the system. This approach turns the drivers into de-privileged applications of the new operating system, which separates their architectures and reduces cross-influences, and improves system dependability.

Virtual machines help reuse drivers, but they also penalize performance. The known solution for improving virtual machine performance, para-virtualization, modifies the operating system to run on a hypervisor, which has an enormous cost: substantial development effort, and abandonment of many of virtualization's benefits such as modularity. These costs contradict our goals for driver reuse: to reduce development effort, and to easily reuse from a variety of operating systems. Thus we introduce a new approach to constructing virtual machines: *pre-virtualization*. Our solution combines the performance of para-virtualization with the modularity of traditional virtual machines. We still modify the operating system, but according to a set of principles called *soft layering* that preserves modularity, and via automation which reduces implementation costs. With pre-virtualization we can easily reuse device drivers.

We describe our driver-reuse approach applied to a real system: we run virtual machines on the L4Ka::Pistachio microkernel, with reused Linux drivers. We include an evaluation and demonstrate that we achieve throughput comparable to

the native Linux drivers, but with moderately higher CPU and memory utilization. Additionally, we describe how to apply pre-virtualization to multiple hypervisor environments. We include an evaluation of pre-virtualization, and demonstrate that it achieves comparable performance to para-virtualization for both the L4Ka::Pistachio and Xen hypervisors, with modularity.

# Originality statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Joshua LeVasseur

# Acknowledgments

My experience working on this project was rewarding and enriching, and a turning point in my life, as all periods of self growth must be. I thank Prof. Gernot Heiser, my supervisor, for relentlessly supporting me throughout the project, and for believing in my ability to succeed. He worked with me on the research and helped guide me through paper writing, and spent considerable time reviewing my thesis and providing feedback. He worked to bring me from the American educational system into the German, which have different requirements for entering a PhD program. He also continued to supervise me from remote, after he transferred to UNSW, while I remained in Germany.

I thank Prof. Gerhard Goos, of the University of Karlsruhe, for his wonderful advice, guidance, and challenges. He graciously accepted me as a student after Prof. Heiser moved to UNSW. His demanding requests guided me into the person I wanted to become. I enjoyed participating in his PhD seminars, which were important training for paper writing, public speaking, and finally the thesis writing.

I thank Prof. Frank Bellosa for accepting me after he become the head of the System Architecture Group in Karlsruhe. He wholeheartedly supported me and went out of his way to ensure that I would complete the thesis, even if this meant ultimately earning the degree via UNSW under Gernot Heiser. He provided me with a position in the group, with many opportunities for guiding Master's students, for lecturing, and continuing my research. This environment contributed to my self growth, and ultimately the completion of the thesis.

I have many thanks for Dr. Volkmar Uhlig, who was also working on his thesis at the time. Through our daily interaction he became the greatest influence on my mental processes, particularly regarding operating systems. He taught and mentored me on operating systems, sharing a lot of the knowledge gained through his interactions with the L4 community. He provided guidance on the PhD process, and we discussed the advice and challenges given to us by our advisers. He also

provided an atmosphere in the group of constant challenge and growth in research, along with critical and *creative* thinking, where you willingly share ideas to make room for more. He became my friend, and helped me to enjoy life in Karlsruhe, to learn a new culture and to see that additional dimensions to life exist.

I am thankful for the people in the Systems Architecture Group in Karlsruhe. Uwe Dannowski, Espen Skoglund, and Jan Stoess helped build L4 and support it. Gerd Liefländer gave wonderful advice for public speaking and writing, James McCuller provided excellent technical support. Andreas Haeberlen demonstrated to me the results of stepping away from the computer to achieve better thinking. And the students gave me opportunities to advise on thesis writing, and they supported the research.

I thank my wife Susanne and my daughter Sophie for supporting me and helping me to achieve my goal. They also inspired many life transformations outside the PhD.

I also thank Rich Uhlig and Sebastian Schönberg's group at Intel, for providing funding for our L4 research projects.

I thank Netronome Systems for employing me while I finished the thesis, and for sponsoring a conference trip.

I thank the anonymous reviewers of this thesis for their insightful and sincere feedback.

I appreciate the interaction with the L4 community that I had over the years, particularly with the other students at Dresden and UNSW. I am excited to see the L4 ideas move out into the world, as they accompany us to new challenges.

# Contents

# List of figures

# List of tables

# Chapter 1

# Introduction

This thesis presents a method of reusing device drivers in new operating system designs. In support of this goal, it also introduces new techniques for creating virtual machines.

The field of operating system design is rich with interesting methodologies and algorithms, but which often remain obscure due to the daunting task of turning them into complete systems. In particular, a complete system requires device support, and device drivers can contribute substantially to the implementation burden — for example, in the general-purpose Linux operating system, version 2.4.1, 70% of the code implements device drivers [CYC+01]. Even in research systems that require only a small set of device drivers, their implementation weight is substantial: the L4Ka::Pistachio microkernel is roughly 13k lines of code, while its driver for the Intel 82540 network controller has 4.5k lines of code (compared to 8.8k lines of code for a production driver in Linux 2.6.9). Besides the implementation burden, other hurdles thwart development of a new driver base, including unavailable devices (particularly for discontinued or updated devices), unprocurable reference source code for the drivers, incomplete or inaccurate documentation for the devices, and test and verification under all operating conditions. Thus we have a variety of incentives to *reuse* existing drivers, which permits immediate access to a wealth of devices, leverages driver maturity, reduces the implementation burden of the novel operating system, and permits developers to focus on the novel aspects of the system.

The reasons for driver reuse are compelling, but many obstacles exist. A fundamental obstruction is that drivers have invariants, and these invariants may conflict with the architecture of the novel operating system, and thus conflict with the pri-

mary reason for driver reuse. Thus driver reuse must isolate the new operating system from conflicts with the invariants. Another obstacle with driver reuse is that drivers require resource provisioning from their operating system, and thus the new operating system needs to provide equivalent resources, which can be a substantial implementation burden.

Several past projects have accomplished driver reuse, but with only partial success. They have usually required access to the source code, required the new operating system to conform to the invariants of the drivers (in some cases overlooking invariants, resulting in malfunction), and implemented substantial compatibility layers for driver resource provisioning.

This thesis shows how to reuse legacy drivers within new operating systems while solving the primary obstacles, thus protecting the novel design of the OS from the drivers' invariants, and reducing implementation burdens. Performance is also a goal, but of lower priority. The basis of the reuse approach is to run the device drivers in virtual machines along with their original operating systems, which supports reuse of binary drivers, offers isolation, and reuses the drivers' operating systems for resource provisioning to the drivers.

Virtual machines are known to have runtime performance costs, and implementation burden. A popular technique for improving virtualization performance is para-virtualization, but it contradicts our goal of rapid development and deployment. We thus introduce a new technique for creating virtual machines — *pre-virtualization* — which combines many of the best features of traditional virtualization *and* para-virtualization, to give good performance and rapid deployment.

## 1.1 Device driver reuse

In this thesis we present the design of a pragmatic driver-reuse system that runs the drivers inside virtual machines, along with their original operating systems. The virtual machines help us solve several problems: they protect the invariants of the novel operating system from the invariants of the reused drivers, they permit easy resource provisioning for the reused drivers (by running the drivers in their original operating systems), they permit substantial code reuse, and they enhance dependability.

Device-driver based operating systems apply software layering to reduce the complexity of supporting many devices. The drivers implement all of the device-specific logic, and present the capabilities of the device through an abstracted in-

terface to the rest of the operating system. The operating system can treat devices interchangeably by substituting different drivers. This layering makes drivers good candidates for software reuse through several methods, all of which have had limited success. We instead treat the driver and its original operating system as the unit of reuse, but since the operating system is not designed for this purpose, we introduce interfacing logic into the original operating system. This interfacing logic permits our reuse environment to take advantage of the drivers' abstracted interfaces, and to thus submit device requests and to handle completed operations.

In this thesis, we discuss the requirements imposed on one's new operating system to support this style of driver reuse. We discuss side effects (such as excessive resource consumption) and how to overcome them. We discuss how to construct adapters that connect one's new operating system to the reused driver infrastructure. Besides the general design strategies, we present a reference implementation, where we reuse device drivers from Linux 2.4 and 2.6 on an L4 microkernel system. We evaluate the performance of the reference implementation, and study some of its behavior, such as resource consumption, and compare this behavior to native device drivers. We also compare and contrast to prior work.

## 1.2 Pre-virtualization

Although we reduce engineering burden for driver reuse by running the drivers within virtual machines, we add engineering burden for constructing virtual machines. Virtual machines have an engineering effort versus performance trade off: the better their performance, the higher their implementation burden tends to become (to work around the traditional method of trapping on privileged instructions, which on the popular x86 architecture is very expensive).

The existing virtual-machine construction techniques contradict our theme to make driver reuse easy to achieve. We thus propose a new approach to constructing virtual machines. Our new approach is based on a concept of *soft layering*, where we dynamically adapt a layered software interface to overcome bottlenecks of the software layering. We call our application of soft layering to virtualization *pre-virtualization*. It retains the modularity of traditional virtualization by conforming to the platform interface, but enables many of the performance benefits of para-virtualization such as running emulation logic within the operating system itself. As part of the effort to reduce engineering burden, it uses automated tools to process the operating system's source code in preparation for soft layering. The

implementation burden then simplifies to a per-hypervisor run-time support module, which binds to the operating system at runtime.

Our approach to constructing virtual machines presented in this thesis advances the state of the art. We describe the concepts behind its design. We discuss the problems of other approaches, and describe how pre-virtualization solves those problems. We describe how to reduce implementation burden via automation. We present a reference implementation that supports several hypervisors: multiple versions of L4, and multiple versions of Xen; and which supports several versions of Linux 2.6 and Linux 2.4. We analyze its performance, and compare to para-virtualization.

# Chapter 2

# Background and related work

Since this thesis describes a method to reuse device drivers by running the device drivers in virtual machines, it discusses both the issues of driver reuse and the issues of virtual machine construction. In support of both topics, this chapter provides background material, describes prior work, and identifies shortcomings in the prior work that this thesis addresses.

## 2.1 Driver reuse

Our fundamental goal is to promote the construction of novel operating systems by reducing implementation burden. Device drivers tend to contribute substantially to the implementation burden of the OS kernel. Thus we seek to reuse device drivers within new OS ventures to increase the chance of success, or at least to provide enough framework to evaluate the novel aspects of the system.

In support of our theme, a driver reuse system should provide the following capabilities:

**Isolation:** The isolation property is to protect the new operating system from the reused drivers. The drivers already conform to an architecture, with invariants that could possibly contradict the design goals of the novel OS. Our primary goal is to support novel system construction, and so the driver-reuse framework must protect the novel system from the invariants of the reused drivers.

**Dependable reuse:** We will use the drivers in a manner that exceeds their design specifications, particularly in that we introduce new infrastructure for interacting with the drivers. The new infrastructure could adversely interact with

the drivers, and thus careful construction techniques are paramount, as is careful understanding of the interfaces used by the drivers. Thus we want to design for error, and so reduce the consequences of error that could arise from misusing the drivers and their interfaces.

**Unmodified driver source code:** For scalable engineering effort, the driver code should be reusable unmodified, or with modifications applied by automated tools.

**Binary driver reuse:** Many drivers are available in binary form only, and thus the reuse approach must be capable of reusing binary drivers. In some OS ecosystems, independent hardware vendors provide proprietary drivers without source code.

We now justify these criteria via an analysis of prior approaches to driver reuse.

### 2.1.1 Driver transplantation

The most common approach for driver reuse is to transplant the drivers from a donor OS to the new OS, supported by a compatibility layer. The compatibility layer provides a life support to the drivers (such as data structures, work scheduling, synchronization primitives, etc.), and provides a driver operational interface that permits the new OS to access the devices. A disadvantage to transplantation is that the compatibility layer must conform precisely to a variety of interfaces and invariants defined by the donor OS. Other disadvantages are that one must write a compatibility layer for each donor OS, and upgrade it to track versions of the donor OS (which is particularly relevant for Linux, because it offers no formal interface for its drivers, and thus a compatibility layer may have severe differences between versions).

The transplantation approach supports both binary reuse and source code reuse. Binary reuse depends upon availability of loadable kernel modules in both the donor and new OS, while source code reuse permits linking the drivers directly to the new OS. An example of binary driver transplantation is the NdisWrapper project[1], which reuses Microsoft Windows networking drivers within Linux. Many projects have transplanted drivers in their source code form [FGB91, GD96, BDR97, Mar99, AAD+02, VMw03].

[1] `http://ndiswrapper.sourceforge.net/`

A transplantation merges two independently developed code bases, glued together with the compatibility layer. Ideally the two subsystems enjoy independence, such that the design of one does not interfere with the design of the other. Past work demonstrates that, despite great effort, conflicts are unavoidable and lead to compromises in the structure of the new OS. Transplantation has several categories of reuse issues, which we further describe.

**Semantic resource conflicts**

The transplanted drivers obtain resources (memory, locks, CPU, etc.) from their new OS, accompanied by new invariants that may differ from the invariants of the donor OS, thus creating a new and risky relationship between the two components. In the reused driver's raw state, its manner of resource use could violate the new invariants. The misuse can cause accidental denial of service (e.g., the reused driver's nonpreemptible interrupt handler consumes enough CPU to reduce the response latency for other subsystems), can cause corruption of a manager's state machine (e.g., invoking a non-reentrant memory allocator at interrupt time [GD96]), or can dead-lock in a multiprocessor system.

These semantic conflicts are due to the nature of OS design. A traditional OS divides bulk platform resources such as memory, processor time, and interrupts between an assortment of subsystems. The OS refines the bulk resources into linked lists, timers, hash tables, top-halves and bottom-halves, and other units acceptable for sharing between the subsystems. The resource refinements impose rules on the use of the resources, and depend on cooperation in maintaining the integrity of the state machines. Modules of independent origin rely on the compatibility layer for approximating this cooperation — for example, when a Linux driver waits for I/O, it removes the current Linux thread from the run queue; to permit reused Linux drivers to dequeue Linux threads, the compatibility layer will allocate a Linux thread control block when entering a reused Linux component [Mar99, AAD+02]; in systems that use asynchronous I/O, the glue layer will instead convert the thread operations into I/O continuation objects [GD96].

**Sharing conflicts**

A transplanted driver typically shares the address space with the new OS kernel. Additionally, the reused drivers typically run in privileged mode with the new OS kernel. The drivers can claim regions of the address space that conflict with the

new OS kernel, and they can make unsupported use of the privileged instructions.

A solution for sharing the address space may be unachievable, due to device driver invariants and non-modular code. For example, the older Linux device drivers, dedicated to the x86 platform, assumed virtual memory was idempotently mapped to physical memory. Reuse of these drivers requires modifications to the drivers or loss in flexibility of the address space layout: the authors in ref. [Mar99] decided not to support such device drivers, because the costs conflicted with their goals; the authors of ref. [GD96] opted to support the drivers by remapping their new OS.

Privileged operations generally have global side effects. When a device driver executes a privileged operation for the purposes of its local module, it likely affects the entire system. A device driver that disables processor interrupts disables them for all devices. Cooperatively designed components plan for the problem; driver reuse spoils cooperative design.

The global nature of privileged operations also permits the consequences of faults to exceed the boundaries of the device driver module. Even when the subsystems are mutually designed to share resources, faults can negate the careful construction, exposing the other system components to a larger fault potential. Via privileged domain and address space sharing, the device driver's faults can easily access the internals of other modules and subvert their state machines.

**Engineering effort**

Device driver reuse reduces engineering effort in OS construction by avoiding reimplementation of the device drivers. Preserving confidence in the correctness of the original drivers is also important. When given device drivers that are already considered to be reliable and correct (error counts tend to reduce over time [CYC+01]), it is hoped that their reuse will carry along the same properties. Confidence in the new system follows from thorough knowledge of the principles behind the system's construction, accompanied by testing.

Reusing device drivers through traditional transplantation reduces the overall engineering effort for constructing a new OS, but it still involves substantial work. Ford et al. report 12% of the OS-Kit code as glue code [FBB+97]. Engineering effort is necessary to extract the reused device drivers from their source operating systems, and to compile and link with the new operating system. The transplant requires glue layers to handle semantic differences and interface translation.

For implementation of a glue layer that gives us confidence in its reliability,

intimate knowledge is required about the functionality, interfaces, and semantics of the reused device drivers. The authors in [GD96, Mar99, AAD+02] all demonstrate intimate knowledge of their source operating systems.

The problems of semantic and resource conflicts multiply as device drivers from several source operating systems are transplanted into the new OS. Intimate knowledge of the internals of each source operating system is indispensable. Driver update tracking can necessitate adaptation effort as well.

### 2.1.2 OS cohosting

VMware has demonstrated an alternative approach for reusing binary drivers in their VMWare Workstation product [SVL01] — a cohosting approach. They run the drivers within their entire original OS, and share the machine with the new OS (in this case, VMware's hypervisor), which is similar to the approach of this thesis. Their approach uses the original OS to provide the life support for the drivers, thus avoiding the need for a large compatibility layer. The driver OS shares the machine with the new OS via time-division multiplexing. The multiplexing uses world switching — the driver OS runs fully privileged when it owns the CPU, but then completely relinquishes the CPU when it switches to the new OS, switching everything including interrupt handlers, page tables, exception handlers, etc. When VMware's hypervisor detects device activity, it switches back to the driver OS (which restores the interrupt handlers of the driver OS, etc.).

This cohosting method offers no trust guarantees: both operating systems run fully privileged in supervisor mode and can interfere with each other, particularly if the driver OS malfunctions.

## 2.2 Dependability

Reusing drivers introduces a risky relationship between the new OS and the reused drivers, and our approach focuses on reducing this risk compared to other designs, i.e., we attempt to increase the dependability of reused drivers.

Tanenbaum and van Steen specify at least four attributes for describing a dependable system: availability, reliability, safety, and maintainability [TvS02]. Perfectly dependable systems are elusive, but we can discuss how to improve system dependability along the available dimensions; we focus on the availability and reliability aspects of dependability.

Our architecture increases dependability for several reasons:

- We avoid the error-prone project of writing a compatibility layer that must mimic the original driver OS, and instead use the driver OS for providing the compatibility layer.
- We focus on implementing a virtual machine rather than a software compatibility layer. A virtual machine conforms to a well-defined interface that rarely changes.
- Via the virtual machine, we isolate the driver OS from the remainder of the system (to the extent permitted by hardware).
- We run the virtual machine at user level, and thus with controlled privileges (to the extent supported by the hardware).
- We can restart many drivers by restarting their VMs.

These approaches have precedence: VMware's cohosting approach [SVL01] has shown how to eliminate the compatibility layer (although replacing it with the cohosting infrastructure); using virtual machines to enhance reliability is an old technique, with the improvements often attributed to smaller and simpler privileged-mode kernels, and isolation of subsystems within VMs [BCG73, Gol74, MD74, KZB+91]; Nooks [SBL03, SABL04] has shown how to isolate drivers from the remainder of the OS (discussed in the next section) and how to restart the drivers; and many projects have moved their drivers to user-level (discussed in the next sections).

### 2.2.1 Nooks

The recent Nooks project [SBL03, SABL04] shares our goal of retrofitting dependability enhancements in commodity systems. Nooks attempts to prevent the vast majority of driver-caused system failures in commodity systems, via practical fault isolation with driver restart. Their solution isolates drivers within protection domains, yet still executes them within the kernel with complete privileges. Without privilege isolation, complete fault isolation is not achieved.

Nooks collocates with the target kernel, adding 22,000 lines of code to the Linux kernel's large footprint, all privileged. The Nooks approach is similar to second generation microkernels (such as L4, EROS, or K42) in providing address space services and synchronous communication across protection domains, but it

doesn't take the next step to deprivilege the isolation domains (and thus exit to user-level, which is a small overhead compared to the cost of address space switching on x86).

To compensate for Linux's intricate subsystem interactions, Nooks includes interposition services to maintain the integrity of invariants and resources shared between drivers. In our approach, we connect drivers at a high abstraction level—the request—and thus avoid the possibility of corrupting one driver's life support by the actions of another driver. Yet Nook's interposition services permit sophisticated fault detection and driver restart.

The more recent SafeDrive project [ZCA+06] offers similar features to Nooks, but with less overhead, due to using compiler-inserted checks based on code annotations added by developers, as opposed to Nook's separate memory spaces. Like Nooks, it is unable to handle malicious drivers. A contemporary project of SafeDrive, the XFI project [EAV+06], also uses software-based protection of drivers, although with compiler-assisted protection and automated checking, thus handling malicious attacks.

### 2.2.2 User-level device drivers

Our device-driver reuse approach executes the drivers at user-level, and thus deprivileged. User-level device driver frameworks are a known construction technique [LBB+91,FGB91,RN93,GSRI93,KM02,Les02,HLM+03,EG04,HBG+06] for enhancing dependability. They are typically deployed in a microkernel environment, although a recent project has proposed user-level drivers to complement the monolithic Linux kernel [LCFD+05].

Many of the user-level driver projects suggest that they increase system dependability by: isolating drivers, deprivileging drivers, offering a means to catch driver faults so that drivers can be restarted/recovered, and offering programmers a development environment that resembles typical application development and is thus simpler than in-kernel development. These characteristics protect independent subsystems, i.e., so that a driver is unable to interfere with a subsystem that makes no use of the driver. Our approach inherits this principle of independence. We would like to suggest that our approach also offers the fault detection and recovery advantages of user-level drivers, but it is beyond the scope of this thesis to prove these properties. Since we reuse drivers, the advantage of the application development model is inapplicable.

The Mungi user-level drivers [LFDH04] can be written in Python, an inter-

preted language, which can reduce the number of bugs due to its pithiness and automatic garbage collection. While developing the Mungi drivers, I/O-level page protections helped detect errant DMA accesses [LH03]. It is also possible to detect errant DMA accesses with filtering [HHF+05].

So far we are only aware of the Nooks [SBL03,SABL04] and SafeDrive [ZCA+06] projects delivering on the fault detection and recovery promises: they provide extensive fault detection (which relies on code that inspects the kernel's internal state machines and thus surpasses the capabilities of user-level drivers that can only detect a subset of errant memory accesses), and they have demonstrated recoverable drivers with minimal disruption to the clients of the restarted drivers. User-level drivers are capable of fault recovery [HBG+07], but fault detection is difficult and unproven, and thus a corrupt driver may distribute corrupt data throughout the system long before the corruption manifests as a detectable errant memory access. An alternative approach to fault detection is to construct the system with the assumption that the drivers are faulty, and to protect data integrity via encryption [SRC84, HLM+03, HPHS04]. Since the integrity approach assumes that the drivers and hardware are faulty, it is compatible with driver reuse.

Like us, the contemporary project Xen [FHN+04a, FHN+04b] creates user-level device drivers via VMs. Their approach is very similar to ours, and will be discussed in Chapter 3.

### 2.2.3 Faulty hardware

Our approach to enhancing dependability does not address faulty hardware. Worse, working around faulty hardware can undermine approaches to improving driver dependability. For example, the CMD640 PCI/IDE bridge can corrupt data written to disk if an interrupt arrives during data transfer — this requires the driver to disable global interrupts [Sha01], which is an unsatisfactory requirement, particularly for deprivileged drivers.

Yet other solutions can be combined with our driver reuse infrastructure for addressing hardware faults, such as using encryption to detect and tolerate corruption [SRC84, HLM+03, HPHS04].

## 2.3 Virtual machine models

This thesis makes use of three categories of features of virtual machines. We introduce each category with a model to relate the VM to pre-existing concepts, while

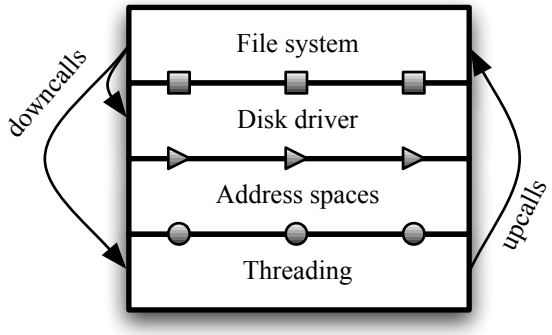



Figure 2.1: Example of layered system construction.

also describing the features required by our generalized driver reuse solution. The three models: (1) from the perspective of building a virtual machine, we use a hierarchical software layering model; (2) VMs use services of their hypervisors, and from this perspective, we model the VM as an application; and (3) guest OSes can collaborate together, and from this perspective, we model the VM as a participant in a component framework.

### 2.3.1 Virtual machine layering

Virtual machines increase the extent of software layering in a system by replacing the hardware with a software module that conforms to the hardware's interface. From the general principles of layering theory we contribute a new approach to constructing VMs, and thus first describe layering.

#### Layering

Hierarchical software layering is a partial ordering between software modules, with the relationship that the higher layer depends on the lower layer [Par72]. The layering promotes module simplification, as Djikstra explained in his description of THE [Dij68], because layering reduces the expression of a module's algorithms to its essential elements, supported by the vocabularies of the lower layers (e.g., a lower layer provides a threading environment for the algorithms of a higher-layer file system). See Figure 2.1 for an example system.

A layer interface makes a statement about the implementation of the layer, forcing some implementation details, while hiding the remaining from the layers above. Thus one can substitute alternative implementations for the layers based

on criteria orthogonal to the interface. This thesis heavily relies on the ability to substitute layers in system construction.

Design by layering involves choosing appropriate abstractions and cut points for the layer interfaces. Although layering's abstractions provide benefits, the abstractions easily introduce inefficiencies due to the lack of transparency into the lower layers' implementation details [PS75]; e.g., the abstractions may prevent the upper layer from using mechanisms that the implementation has, but which are hidden behind the abstractions; or the abstractions may present the idea of infinite resources to the upper layer, when the opposite is the case.

When layers interact with each other, they transform abstractions and transfer control flow. We use the following terms to describe the transformations and control-flow changes:

**downmap:** A higher layer maps its internal abstractions to the abstractions of the lower layers, e.g., the file system converts a file read into an asynchronous disk block read from the disk driver layer, and a thread sleep from the threading layer while waiting for delivery of the disk block.

**downcall:** A control-flow transfer from the higher-level to the lower-level, potentially with message delivery (e.g., the results of the downmap).

**upcall:** Sometimes a lower layer transfers control flow to a higher layer [Cla85] (e.g., for network processing, upwards transfer is the natural direction for packet reception).

Since a lower layer lacks knowledge of upper layers' abstractions, the upcall has no accompanying *upmap*.

Virtual machines reposition software to layers of the hierarchy unanticipated by the original software design. The relocation of modules relative to other modules may invert the ordering of abstractions, so that a lower-level layer runs at higher layers than intended (although, the downward partial ordering is maintained). For example, a virtual machine runs an operating system as an *application* of another operating system, and thus issuing system calls rather than direct accesses to the hardware. When the relocated layer makes a downcall, it attempts to use the wrong abstractions. We introduce an *upmap* operation to accompany the downcall, to handle the inversion of abstractions: it maps the lower-level abstractions to higher-level abstractions. See Figure 2.2.

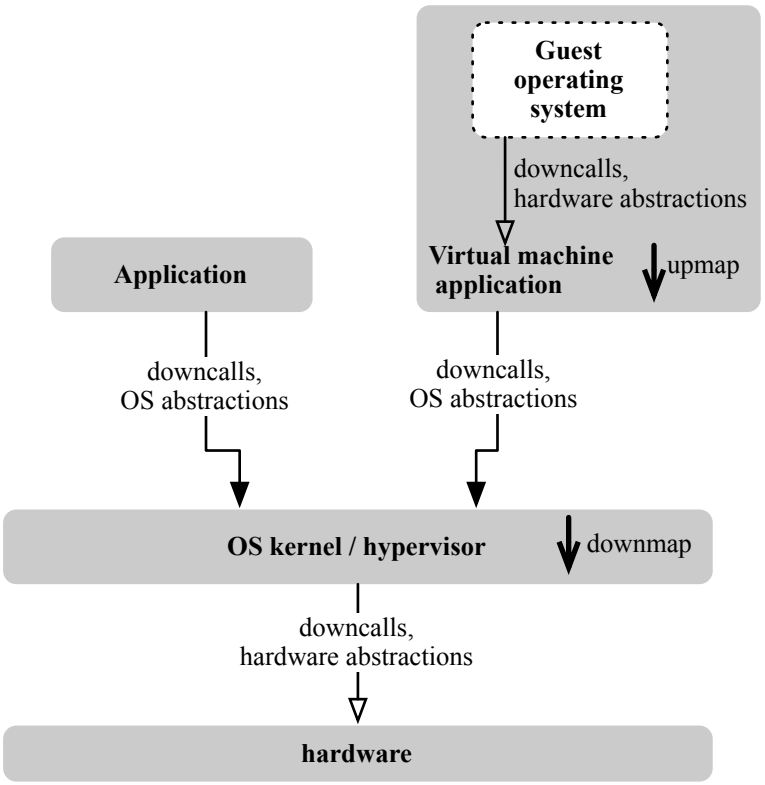



Figure 2.2: The guest operating system communicates with the virtual machine using the hardware abstractions and hardware downcalls. The virtual machine then applies an upmap to transform these low-level abstractions to the higher-level abstractions expected by the hypervisor. The virtual machine executes system calls with the higher-level abstractions as downcalls.

### Virtual machines

The virtual machine environment substitutes software implementations for the machine hardware. The hardware defines the interface between itself and the OS; this interface is well defined, and fairly immutable over time. The software implementation is called the virtual machine monitor (VMM). In this thesis, in contrast to some of the literature, the VMM has no direct access to the privileged hardware; the *hypervisor* has the direct access to the privileged hardware, and the VMM executes as a subsystem of the hypervisor (although potentially within the hypervisor itself). The hypervisor and its applications can be called the *host OS*, and the operating system that runs within the VM the *guest OS*. The hypervisor, like many OS kernels, publishes an abstract interface to multiplex the system resources (e.g., to run multiple VMs), and may even run native applications that use these abstrac-

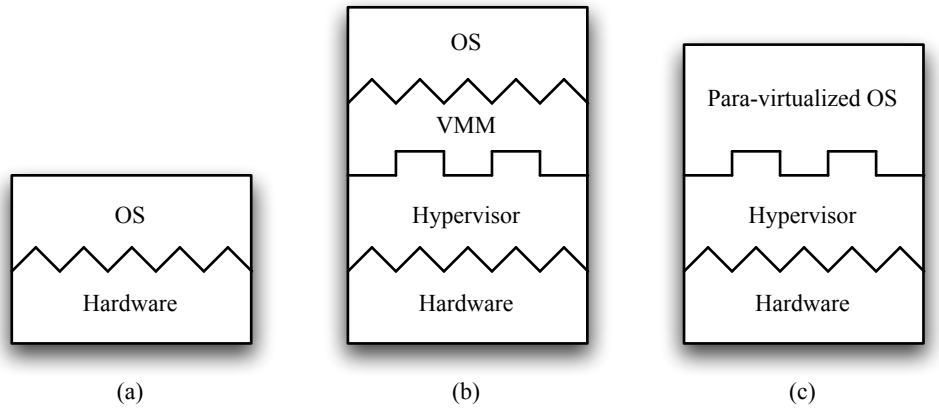



Figure 2.3: In (a) the OS directly accesses the hardware layer. In (b) the VMM communicates with the hypervisor via a high-level interface, while the guest OS communicates with the VMM via the low-level platform interface. In (c) a hand-modified OS integrates the functionality of the VMM, and directly accesses the high-level hypervisor interface.

tions. Thus the VMM and guest OS operate upon the hypervisor's higher-level abstractions, and the VMM maps the guest OS's activity to these higher-level abstractions (see Figures 2.2 and 2.3). The system is recursive: the VMM publishes the machine interface to the guest OS, and the hypervisor uses the machine interface to control the real hardware, and thus it is possible to also insert a VMM below the hypervisor. Some of the literature use VMM and hypervisor interchangeably, while we distinguish between the two since their differences are relevant to this thesis.

Traditional virtualization strictly conforms to the hardware interface, to enable typical operating systems to execute within the virtual machine. For security and resource control, the VMM prevents the guest OS from directly accessing the privileged hardware, and instead maps the guest's system instructions to the interface of the hypervisor. This translation uses an upmap operation, since the hypervisor provides more abstract interfaces suitable for multiplexing resources. Traditional virtualization encounters layering's worst-case transparency [PS75] problems: the OS uses the mechanisms of the privileged hardware, but as a guest in a VM it must use software abstractions (which are implemented in the hypervisor using the very instructions forbidden to the guest OS). The lack of transparency causes performance problems.

Besides performance problems, VM layering disrupts accurate timing [Mac79,

WSG02, BDF+03], entropy generation (for security-related algorithms), and prevents the driver-reuse techniques of this thesis. Timing and driver-reuse with direct hardware access fall outside the scope of virtualization [PG73], and their solutions require increased transparency to the lower layers to either obtain visibility into the resource multiplexing (timing), or to bypass the resource multiplexing (direct hardware access). Entropy helps an OS generate statistically random numbers; a VMM may increase determinism and thus lower the quality of the guest's entropy pools.

A solution for the performance and correctness issues of VMs is to increase the layer transparency between the guest OS and the VM, to give the guest OS knowledge of implementation details in the lower layer beyond the details encoded in the original hardware interface. Many projects have taken this approach: they have modified the OS for more efficient execution within the VM, and have added knowledge to the OS of the abstracted time (see Figure 2.3). IBM called this approach *handshaking* [Mac79], and used it for their early mainframes. Goldberg called it *impure virtualization* [Gol74]. Today this co-design approach is termed *para-virtualization* [WSG02]. These modifications introduce new dependencies between the guest OS and its lower layer, often ruining the modularity of virtualization: the guest OS may no longer execute on raw hardware, within other virtual machine environments, or permit nested VMs.

The solution in this thesis also increases the transparency between layers, but in a manner that preserves the modularity of the original layer interface.

### 2.3.2 Virtual machines as applications

In this thesis, we run a VM with its guest OS and applications in place of a traditional application. We thus first demonstrate that this replacement is in many aspects an equivalent substitution.

An OS kernel and its applications together form a collaborative group of interacting components. This group may communicate over a network with other groups on distant machines; from the perspective across the network, the group is an opaque and single entity, and its implementation as a group is often irrelevant. In a VM, the guest kernel and its applications are also hidden behind an opaque module boundary, represented to the other system components as a single entity, with its internal implementations irrelevant in many contexts. Thus the hypervisor hides the nature of the VM from the other system components, making it difficult to distinguish the side effects of the guest OS from those of any other native

application.

The VM has the properties of a typical application: resource virtualization and delegation to the kernel (e.g., the hypervisor may transparently swap a page to disk that is outside the VM's working set), independence guarantees (application $A$ can give guarantees of its internal correctness independent of application $B$) [Lie95b], and integrity guarantees (application $A$ can communicate with $C$ without interference from $B$) [Lie95b].

The term *virtual machine* may seem to arrogate the concept of virtualized resources and cast VMs as something different from a typical application, when in fact, the *virtual* term applies to the *interface* used to represent the virtualized resources. Virtualized *resources* are common to most operating systems, since the traditional use of an operating system is to multiplex and abstract the hardware resources [Dij68]. Thus the resource management of a hypervisor for its VMs is conceptually similar to the resource management provided by a traditional OS for its applications. An OS's applications consume resources via the kernel's high-level abstractions designed to hide the resource multiplexing inside the kernel, thus synthesizing a virtualized environment: each application has the impression that it owns the machine, with continuous access to the CPU, contiguous access to all of memory, contiguous access to the disk, and streamed access to the network. A virtual machine, as an application of the hypervisor, has a similarly virtualized resource environment, but using different abstractions: the primitives of the hardware.

Despite the similarities between how a hypervisor and a traditional kernel provide virtualized resources to their abstractions, not all kernels can function as hypervisors. Since the VM uses the primitives of the hardware, it may use resources in a manner where no upmap to the high-level abstractions of a traditional OS exists—for example, a VM must create multiple address spaces that it populates with pages from its own supply of pages, but this same action is difficult to achieve under the abstracted POSIX API since it deviates from `fork()` and `exec()` semantics. Although we argue that a VM is a substitute for a normal application, achieving this substitution requires sufficient interface support from the kernel.

Applications have system calls to obtain services from their kernels. VMs have a parallel, called hypercalls, which provide a control-flow transfer (and message transfer) to the hypervisor (see Figure 2.4). Whether the VM uses traditional virtualization or para-virtualization, the hypercalls are available to guest code that is aware of the VM environment. The hypercalls are conduits that puncture the VM

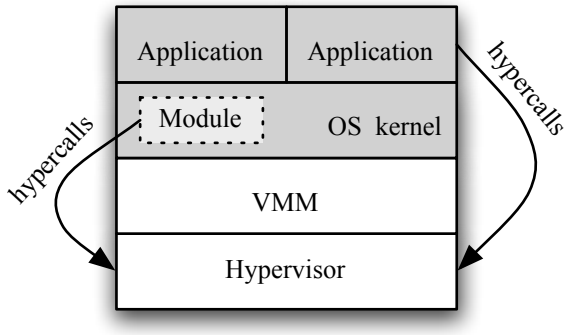



Figure 2.4: The hypervisor accepts hypercalls from applications and kernel modules that are aware of the VM environment.

isolation. Examples of VM-aware services: the user interface tools and file-sharing tools installed by most commercial VM environments, and memory ballooning (a technique to achieve cooperative memory management between the guest kernel and the hypervisor) [Wal02].

Many OSes isolate their applications to provide independence and integrity guarantees. A VM has similar isolation, but spread across several hardware address spaces. The VM's isolation from other VMs is vulnerable to the imperfections that accompany normal application isolation, despite the folklore that VMs have security analogous to physical machine isolation in a concrete room. The VM's secure containment relies on correct implementation of the hypervisor, and correct implementation and access control for all communication that crosses the VM boundary (especially hypercalls and hardware emulation, since a guest OS could use them as attack vectors for unauthorized privilege escalation), and thus security is no automatic feature of the system (see Saltzer's discussion *Weaknesses of the Multics Protection Mechanisms* [Sal74]). A VM system has the advantage that it can run a small trusted computing base [Gol74, MD74], with lower complexity, and thus with few bugs (when bugs are estimated in proportion to code size). Formal verification for hypervisors is a step towards trusted, secure isolation of VMs [TKH05], and due to the complexity of formal verification, relies upon a small source-code base.

The hypervisor provides services to the VM just as a kernel provides services to its applications, and irrespective of either's internals. A VM enhances a legacy operating system with features unanticipated by the original architects, building a new system from the old. It is large-scale legacy reuse, and capa-

ble of satisfying one of computing science's dilemmas: a VM promotes innovation while maintaining compatibility with prior systems, interfaces, and protocols. Example services: pausing a VM by removing the VM from the run queue; roll back of the VM state to discard erroneous or experimental changes; checkpoint and migration across the network [CN01, SCP+02]; intrusion detection [JKDC05]; debugging [KDC05]; co-existence of realtime and non-realtime applications [MHSH01]; and secure computing platforms with strictly controlled information flow [MS00, CN01, GPC+03, SVJ+05].

This thesis treats device drivers as applications, even though the drivers run within virtual machines. The hypervisor thus has leeway to manage and control the resource consumption of the device drivers.

### 2.3.3 Component frameworks

The hypervisor provides the VM the option to cooperate with other applications on the machine, in a network of interacting components [BFHW75, Mac79], e.g., interaction as simple as sharing the same graphical interface, or as complicated as file sharing via page sharing. The collaborating components may be applications native to the hypervisor, or other VMs, and all may be unaware of the internal implementations of the others. This creates component frameworks from virtual machines.

Typically a kernel provides inter-process communication (IPC) mechanisms for component systems, which its applications intimately integrate into their algorithms. A hypervisor also provides IPC, but designed independently of the guest kernel, and thus requires translation into the mechanisms of the guest kernel. The translation may take place externally to the guest OS, such as by mapping the IPC to the legacy network protocols that the guest understands and receives via its virtual network adapter (e.g., file sharing via a network protocol such as NFS or CIFS). Alternatively, the translation may take place within the guest OS, via installable kernel modules or applications that are aware of the hypervisor's IPC, and which map the hypervisor IPC to the guest OS's internal interfaces.

We present three possible component frameworks, which vary on the extent of interaction, and whether they focus on enhancing legacy systems, or building new systems.

In the *classic virtual machine model*, the legacy OS is the principal, and substantially flavors the system. The VM's enhancements to the legacy OS play a minor role, and the legacy OS operates unaware of the enhancements. For example,

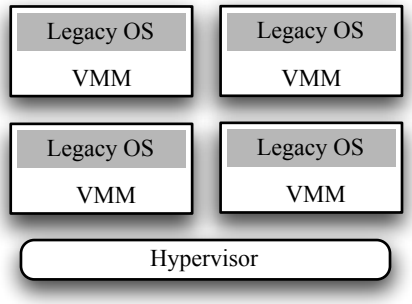



Figure 2.5: Classic model example: server consolidation. The legacy OS instances do not directly invoke features of the hypervisor.

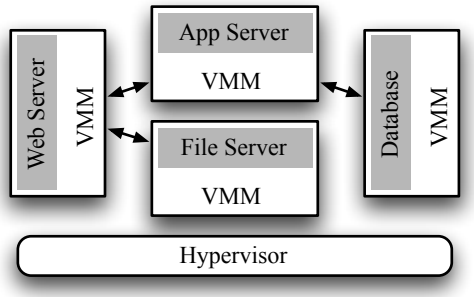



Figure 2.6: Federated model example: tiered Internet server. Multiple legacy OS instances collaborate to form a federated system, inherently tied to the features of the VM environment.

in server consolidation, the VM introduces confinement [BCG73] and performance isolation to fairly share hardware between unrelated servers, to promote extended use of the legacy OS (see Figure 2.5).

In the *federated system model*, the legacy OS serves as a building block for the construction of a system from several legacy OS instances [BFHW75]. Each building block OS benefits from the enhancements of the VM environment, such as subsystem reboots to independently recover from failure, and confinement to prevent the propagation of security breaches. For example, a tiered Internet server divided across multiple virtual machines impedes security compromise from the front-end to the sensitive data in the back-end (see Figure 2.6). The federated system integrates the features of the VM environment into its architecture, and is nearly unable to execute without the VM environment; the alternatives are multiple physical machines, or the merger of all servers onto the same physical machine within a single

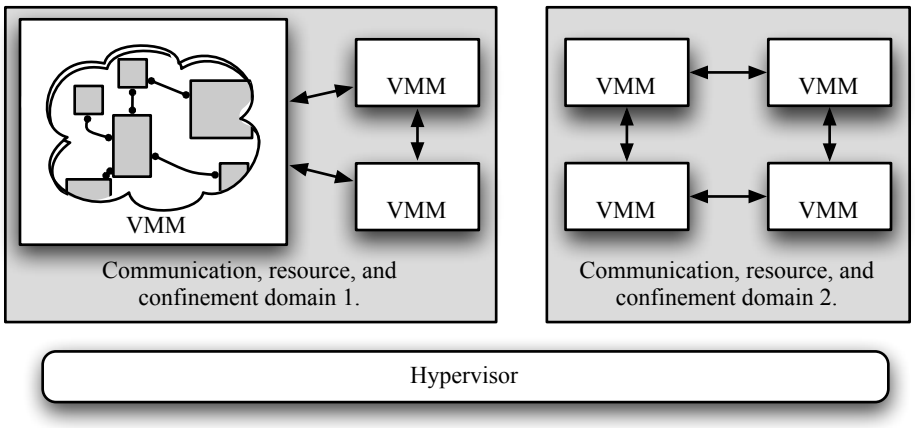



Figure 2.7: Recursive construction technique. Several VMs collaborate in an isolated domain, just as several processes collaborate in each of the VMs (internal processes are visible in the left-most VMM).

legacy OS. Several federated systems can coexist on the same machine, and in this sense, the properties associated with virtual machines are recursive: just as the VM provides an isolation, confinement, and resource domain for the processes within the legacy OS, the hypervisor provides a dedicated isolation, confinement, and resource domain to each federated system, to share amongst the building-block OSs (see Figure 2.7). This recursion may demand recursive VMs as well, to accommodate the preferred hypervisors of each federated system, along with the preferred hypervisor installed by the system administrator on the machine. Contemporary virtual machine technologies do not optimize for recursive VMs.

The *domain model* focuses on the specialties of the hypervisor and its methods of system construction. This model develops system algorithms optimized for the domain, in contrast to the general algorithms of commodity systems. The legacy OS assumes a supporting role and provides reused code, such as device drivers or legacy network protocols. For example, to build an efficient database, uncontested by an OS for the control of memory paging, but supported by reused device drivers, the domain model executes the database as a native hypervisor application using legacy device drivers running in virtual machines (see Figure 2.8).

This thesis relies on component construction with virtual machines: the driver VMs collaborate with and support other system components.

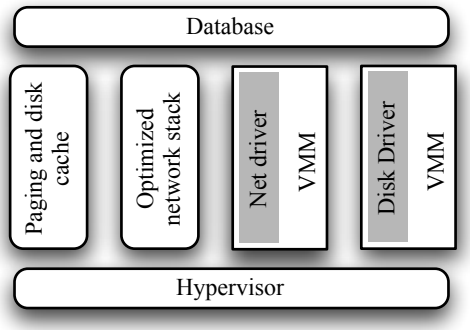



Figure 2.8: Domain model example: database appliance. The database directly accesses the hypervisor's interface, and avoids the inefficiencies typically associated with commodity OS interfaces. Multiple legacy OS instances provide device driver reuse.

## 2.4 Virtual machine construction

The performance and ease of implementation of our driver reuse solution is intimately tied to the virtual machine environment. We describe VM implementation artifacts that impact performance and developer productivity.

The performance of virtual machines relies on bare-metal execution for most guest OS instructions. The virtual machine emulates only those instructions that would interfere with the hypervisor, or would leak hypervisor state where the guest OS expects its own state. We call these instructions the *virtualization-sensitive* instructions [PG73]. The remaining instructions are *innocuous*, and execute on the raw hardware.

A downcall of typical software layering follows function calling conventions: a linked call and return with parameters passed in the register file or stack. The downcall of a virtual machine is more complicated, since the guest OS uses virtualization-sensitive instructions, as opposed to function or system calls. The approaches to virtualization are thus characterized by how they locate and transform the virtualization-sensitive instructions into downcalls.

### 2.4.1 Traditional virtualization

When the virtualization-sensitive instructions are a subset of the privileged instructions, they cause hardware traps when executed by an unprivileged virtual machine. This permits a VMM to locate the sensitive instructions with runtime trapping, to emulate the instructions at the occurrence of the runtime trap, and to

then return control to the guest OS at the instruction following the emulated instruction [PG73].

Not all architectures provide the set of virtualization-sensitive instructions as a subset of the privileged instructions, in particular, the popular x86 architecture [RI00]. This requires more complicated algorithms to locate the sensitive instructions, such as runtime binary scanning. VMware successfully deploys virtual machines on x86 by using algorithms to decode and scan the guest OS for sensitive instructions, to replace them with downcalls [AA06].

Trapping on privileged instructions is an expensive technique on today's superpipelined processors (e.g., in the Pentium line, a trap ranges from 100 to 1000 times more costly than a common integer instruction). Likewise, more efficient downcalls to the VMM that use special instructions to cross the protection domain of the guest OS (hypercalls) are often expensive. To avoid these overheads, VMware also rewrites some of the performance-critical parts of the binary to perform emulation within the protection domain of the guest OS, thus reducing the frequency of transitions to the VMM [AA06].

Hardware acceleration is a known technique to reduce the runtime costs. Most major server processor vendors announced virtualization extensions to their processor lines: Intel's Virtualization Technology (VT) for x86 and Itanium [Int05b, Int05a], AMD's Secure Virtual Machine (SVM) [AMD05], IBM's LPAR for PowerPC [IBM05], and Sun's Hypervisor API specification [Sun06]. These extensions help accelerate the performance of virtual machines. For example, Linux toggles interrupts frequently, with the potential for frequent emulation traps; the new x86 virtualization mode implements interrupt virtualization in hardware, with the option to suppress traps until an interrupt is actually pending for delivery to the guest kernel. The hardware extensions are not comprehensive and typically ignore memory-mapped device virtualization, despite their high virtualization overhead. IBM added a VM assist mode to the System/370 decades ago, which automated shadow page table management, provided a virtual timer, and avoided expensive VM exits by emulating instructions via the CPU rather than the hypervisor [Mac79].

### 2.4.2 Para-virtualization

Para-virtualization is characterized by source code modifications to the guest OS, to convert the sensitive instructions into normal downcalls. It can provide high performance [Mac79,HHL+97,WSG02,BDF+03], and it easily handles x86 and other

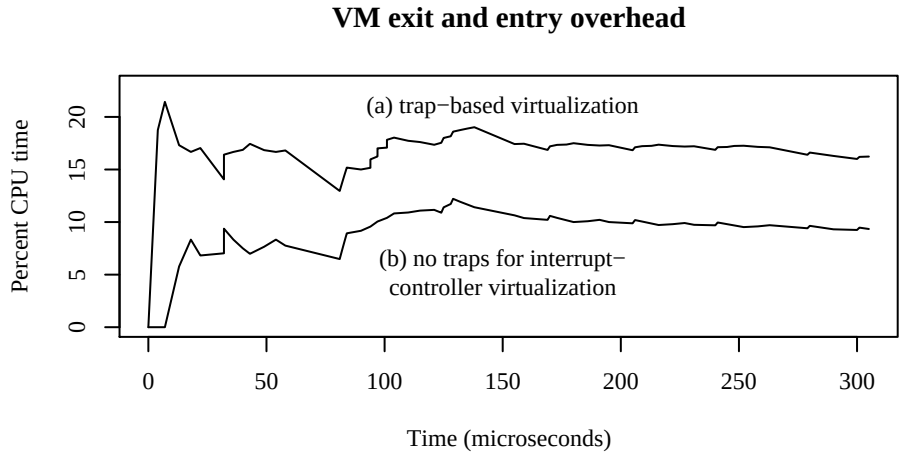



Figure 2.9: The graph studies the CPU overhead due to entering and exiting a VM under a heavy network workload. The VM uses Intel's VT, runs Linux 2.6 as a guest, emulates an interrupt controller, and emulates a gigabit network interface. The overhead represents only the cost of entering and exiting the VM (the trap cost), and excludes emulation costs. The data was collected from a trace of a running VM, with heavy network throughput, and thus frequent interrupts. Each data point shows the aggregate overhead to that point in time from the start of the trace. Line *a* is for traditional, trap-based virtualization. Line *b* estimates the reduced overhead one can achieve by emulating the interrupt controller within the VM's protection domain; it subtracts the overhead due to the exits for the interrupt controller.

architectures that have non-privileged sensitive instructions. Developers manually convert the OS's sensitive instructions into emulation logic, thus supporting emulation within the protection domain of the guest kernel (many projects typically relocate traditional VMM functionality into the guest kernel). The result is that para-virtualization transforms the guest kernel into a first-class application of the hypervisor. This approach deviates from virtualization, since the OS directly uses the abstractions of the hypervisor, rather than using a machine interface that is virtualized to use the abstractions of the hypervisor. See Figure 2.9 for an example of the performance savings that para-virtualization can achieve compared to traditional virtualization.

Para-virtualization involves a fair amount of manual effort to modify the guest kernel code, often by people outside the kernel community, i.e., usually performed by the hypervisor developers themselves. Reducing the manual labor is a goal

of optimized para-virtualization [MC04]; it uses manual modifications to satisfy performance demands, and trap-and-emulate otherwise. An alternative approach automates many of the changes via static rewriting of the assembler [ES03] — at compile time, a translator converts the virtualization-sensitive instructions into downcalls. This approach is incomplete, for it ignores memory instructions (such as page table updates and device register updates).

To obtain performance, para-virtualization compromises developer productivity: it increases effort and offers less value. The following metrics illustrate this:

**VMM modularity:** Para-virtualization integrates the VMM's emulation code into the guest kernel, reducing the ability to share the VMM logic between different OSes (and OS versions). The pursuit of performance often leads to intimate changes spread across the guest kernel, rather than contained in a module. In contrast, the VMM of traditional virtualization is (binary) reusable across different OSes and OS versions.

**Trustworthiness:** Para-virtualization introduces modifications to the guest kernel, often applied by the hypervisor developers rather than by the original kernel developers, potentially introducing new bugs [CYC+01]. Each version of a guest OS requires testing and verification, and inspection of kernel source code to see whether the kernel developers changed structural features since the last version. In traditional virtualization, the VMM implementation is mostly independent of the guest kernel's internals; since it is reusable, reliability testing (and feedback) from a particular guest OS benefits all other guest OSes (but this is no substitute for comprehensive testing).

**Hypervisor neutrality:** The modifications tie the guest kernel to a particular hypervisor. To upgrade the hypervisor either requires backwards binary compatibility in the hypervisor, or updates to the guest OS's source code to suit the latest hypervisor API (and then recompilation).

**OS neutrality:** The manual modifications of para-virtualization pose a high barrier to entry for unsupported OSes. This hampers participation for less popular operating systems within the hypervisor ecosystem, and for novel operating systems.

**Longevity:** VMs offer a means to check-point and restore, whether immediately (such as for runtime migration across the network), or over a long time interval (such as after several intervening iterations of product releases for the

guest OSes and hypervisors). Traditional virtualization supports reactivation of the guest OS on a new edition of the hypervisor; this can provide uninterrupted service across hypervisor upgrades when combined with network migration, can permit retroactive application of bug fixes in the hypervisor to older VMs, and can support the use scenario where the user duplicates a generic VM image for specialization [PGR06]. For para-virtualization, upgrades to hypervisor interfaces interfere with longevity.

Many projects have used the para-virtualization approach, with modifications ranging from minor to major reorganization of the guest kernel. IBM enhanced OS/VS1 and DOS/VS to detect when running within a VM, for better performance [Mac79]. Mach ran Unix as a user-level server [GDFR90] (although the Unix server little resembled its original form) and the Linux kernel as an application [dPSR96]. Härtig, et al., were the first to demonstrate para-virtualization at high performance on a microkernel, running Linux on L4 [HHL+97]; they heavily modified the Linux source in the x86-architectural area to use L4 abstractions. Denali introduced the term *para-virtualization* [WSG02]. XenoLinux uses para-virtualization, and published benchmarks results suggesting that para-virtualization outperforms traditional virtualization [BDF+03], although in the comparison XenoLinux used memory partitioning rather than virtualization. Linux can run a modified Linux as a guest OS [Dik00, HBS02]. For Disco, the authors modified Irix to overcome the virtualization limitations of their CPU [BDR97].

### L4Linux

L4Linux exemplifies the modularity problems of para-virtualization: three universities offer competing (and incompatible) L4-derived microkernels, with more under construction; and the universities have modified four generations of the Linux kernel to run on the microkernels (see Table 2.1). The first L4Linux [HHL+97] used Jochen Liedtke's x86 L4 microkernel [Lie93, Lie95b] (written in assembler). Hazelnut and Pistachio are the L4 microkernels from Universität Karlsruhe, written in C++ yet rivaling Liedtke's performance (Pistachio is the latest, and used in the evaluation of this thesis). The University of New South Wales (UNSW) ported Pistachio to several processor architectures, including embedded systems; and they ported Linux to run at the high-level L4 API (project name is Wombat), permitting it to run on a variety of CPU families. Fiasco is the L4 microkernel from TU Dresden, also written in C++, and supports realtime.

| L4Linux | Liedtke's x86 kernel | Fiasco | Hazelnut | Pistachio |
|---|---|---|---|---|
| 2.0 | ✓ | ✓ | | |
| 2.2 | ✓ | ✓ | | |
| 2.4 | | ✓ | ✓ | ✓ |
| 2.6 (Dresden) | | ✓ | | |
| 2.6 (Karlsruhe) | | | | ✓ |
| 2.6 (UNSW) | | | | ✓ |

Table 2.1: L4Linux demonstrated that high performance para-virtualization is possible on a microkernel, but it involved many man-hours of effort for four generations of Linux and several microkernels. A ✓shows compatibility between a version of L4Linux and a particular microkernel (16 May 2006). Para-virtualization requires proactive effort to enable compatibility between a microkernel and a guest OS, and the data show that for several of the combinations this effort exceeded the perceived usefulness.

L4Linux lacks VMM modularity: porting effort was repeated for each generation of Linux, due to changes in Linux internals, which required L4 developers to understand the Linux kernel internals. Also each group independently duplicated the porting work of the other groups. We have often experienced a lack of trustworthiness with L4Linux. And L4Linux is missing: hypervisor neutrality at the binary and source interfaces, OS neutrality, and longevity. Although L4Linux usually supports research contexts where hypervisor neutrality and longevity are superfluous, it strains developer productivity (which also makes it a good project to train students).

**Xen**

Xen [BDF+03] also exemplifies the modularity problems of para-virtualization. See Table 2.2 for version compatibilities of the Xen hypervisor, Linux, and other OSes. Notice that Xen v3, at its release, supported only a single guest kernel, while the prior version of Xen supported at least six guest kernels.

Xen lacks VMM modularity: porting effort was repeated for each guest kernel, with little code sharing. The ported guest kernels lack hypervisor neutrality: they can only execute on a particular version of the Xen hypervisor, and no alternative hypervisors (such as L4). OS neutrality is missing, as can be seen in Table 2.2 for the upgrade from Xen v2 to Xen v3. Xen is also missing longevity: A XenoLinux 2.6.9 binary is only able to run on the Xen v2 series; the same binary will not execute on Xen v3.

| | Xen v1 | Xen v2 | Xen v3 |
|---|---|---|---|
| XenoLinux 2.4 | ✓ | ✓ | |
| XenoLinux 2.6 | | ✓ | ✓ |
| NetBSD 2.0 | ✓ | ✓ | |
| NetBSD 3.0 | | ✓ | |
| Plan 9 | | ✓ | |
| FreeBSD 5 | | ✓ | |

Table 2.2: Xen has evolved through three hypervisor generations, and several operating system generations. A ✓shows compatibility between a guest OS and a version of the Xen hypervisor (16 May 2006). This data was collected shortly after the release of Xen v3, and shows that most of the para-virtualization effort for Xen v2 was lost, due to changes between Xen's v2 and v3 interfaces; several of those guest OSes now support Xen v3, after manual porting effort.

### 2.4.3 Modular para-virtualization

Several para-virtualization projects have attempted to partially preserve the modularity of traditional virtualization.

Young described para-virtualization [You73] as behavioral changes of the guest OS, enabled at runtime upon detection of running within a VM.

Eiraku and Shinjo [ES03] add trap-and-emulate capability to x86, by using an assembler parser to automatically prefix every sensitive x86 instruction with a trapping instruction. This provides some modularity, including VMM modularity, hypervisor neutrality (except for raw hardware, unless they were to rewrite the trapping instructions with no-op instructions at load time), and longevity. Additionally, since they automate the changes to the guest OS, they have trustworthiness and OS neutrality. The approach sacrifices performance.

vBlades [MC04] and Denali [WSG02] substitute alternative, trappable instructions for the sensitive instructions.

PowerPC Linux uses a function vector that abstracts the machine interface. This indirection supports running the same kernel binary on bare hardware and on IBM's commercial hypervisor. While working on this thesis, we have found that function calls add noticeable runtime overhead for frequently executed instructions on x86.

User-Mode Linux (UMLinux) uses para-virtualization, but packages the virtualization code into a ROM, and modifies the Linux kernel to invoke entry points within the ROM in place of the sensitive instructions [HBS02]. Thus UMLinux can substitute different ROM implementations for the same Linux binary. Addi-

tionally, UMLinux changes several of Linux's device drivers to invoke ROM entry points, rather than to write to device registers; thus each device register access has the cost of a function call, rather than the cost of a virtualization trap.

VMware's recent proposal [VMI06, AAH+06] for a virtual machine interface (VMI) has evolved to include several of the techniques proposed in this thesis. In VMI version 1, VMware modified the Linux kernel to invoke ROM entry points (as in UMLinux [HBS02]), where the ROM is installed at load time and contains emulation code specific to the hypervisor (or even code that permits direct execution on hardware). After the first VMI version became public, our research group started contributing to its design, and VMware has evolved it into an implementation more in line with this thesis. VMI deviates from the base instruction set more than our approach. It provides additional semantic information with some of the instructions. It lacks a device solution. VMI is aimed at manual application to the kernel source code.

Today, the Linux community has finally reacted to the modularity problems of para-virtualization in the face of competing hypervisors, particularly VMware's and Xen's. The Linux community (which can include anyone on the Internet), the Xen team, and VMware are collaborating to add a modular para-virtualization interface to the Linux kernel. The interface is highly derivative of VMI version 2 (and thus this thesis), but forfeits binary modularity for sustaining the community's open source goals — one must thus compile the kernel with foreknowledge of all possible hypervisors that the kernel may execute upon, which is a severe deficiency. This Linux solution offers function overload points, and some instruction rewriting for performance. The Linux community expects hypervisor teams to add the source code of their mapping logic into the Linux source tree. If VMI mapping logic is added, then binary runtime modularity will be achieved.

### 2.4.4 Device virtualization

Guest OSes access information outside the VM via device interfaces, such as disk and network. To convert device interaction into downcalls, traditional VMMs detect and emulate device register accesses via trapping. Device drivers are notorious for having to configure many device registers for a single device transaction [SVL01], thus using trapping for downcalls is intolerably slow. Most VM environments will thus introduce specialized device drivers into the guest kernel, where the drivers are aware of the hypervisor and its downcall mechanisms, and thus avoid trapping. As in para-virtualization, these specialized drivers abandon

modularity: they are specific to both the hypervisor and the guest kernel. They thus require engineering investment for designing and building loadable kernel modules for all the possible guest OSes and hypervisors.

Virtualized devices suffer from another source of performance degradation. Typical high speed devices achieve their performance by operating concurrently to the OS — they use a coprocessor that performs logic in parallel to, and collaboratively with, the host processor. The expectation of this concurrency pervades the structuring of typical OSes. In contrast, virtual devices of the VM environment *share* the host processor with the OS, and thus have different sequencing characteristics, which guest OS structuring may not efficiently support. The prime focus of high speed device virtualization is achieving optimal sequencing that matches the assumptions of the guest OS.

### 2.4.5 Resource virtualization

The layering of virtual machine construction has a weakness: the layers perform redundant and uncoordinated resource management. Coordinated resource management is preferable, but difficult to achieve without sharing high-level information between the guest kernel and the hypervisor — thus para-virtualization has an advantage in that it opens the possibility for collaborative resource management between the hypervisor and guest kernel.

For example, both the hypervisor and guest kernel may perform working set analysis to find pages suitable for swapping to disk. The costs for duplicate analysis can be high, since working set analysis can raise many cache and TLB misses while scanning the page tables, and since the guest kernel's inspection of page tables requires costly synchronization between the virtual page tables and the access bits of the hardware page tables. Additionally, the hypervisor and guest kernel may form opposite conclusions about a page, leading to redundant disk swapping. Several solutions have been used. The VAX VMM team avoided double paging by requiring the working set to be always resident [KZB+91] (i.e., no resource management at the hypervisor level). IBM, via handshaking (para-virtualization), permitted a guest OS to use the hypervisor's paging mechanisms, and to thus avoid double paging [Mac79]. VMware introduces a driver into the guest kernel that transfers ownership of pages between the hypervisor and the guest kernel on demand [Wal02].

A second example: multiprocessor systems are vulnerable to uncoordinated resource management for CPU scheduling: the hypervisor may preempt a virtual

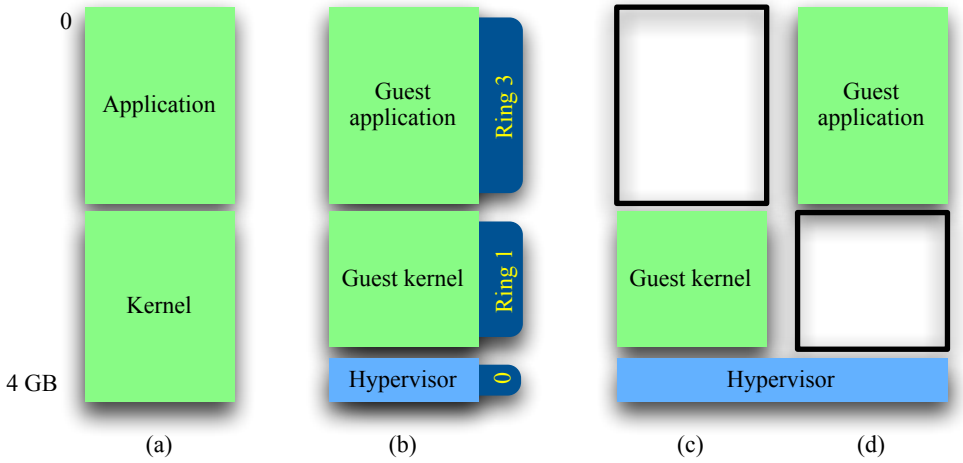



Figure 2.10: In (a), the OS partitions an address space into two privileges: a privileged kernel and an unprivileged application. In (b), the guest kernel uses an intermediate privilege level provided by the x86 hardware through a combination of segmentation and paging (termed *rings*). Finally, if only two privilege levels are available, then the guest kernel runs in a separate address space (c) from the guest application (d), since the hypervisor requires the most privileged level.

CPU that holds a lock, while another virtual CPU spins on that lock, amplifying the typical lock waiting time from microseconds to milliseconds. A possible solution is to modify the guest kernel to inform the hypervisor about lock acquisitions, so that the hypervisor delays preemption [ULSD04].

Since the solutions benefit from collaboration between the hypervisor and guest kernel, their implementations tend to be specific to a hypervisor and guest kernel pair, and thus lack modularity.

### 2.4.6 Address spaces and privileges

The VMM has the important task to efficiently create a three-privileged system, to execute (1) a guest application and (2) a guest kernel on (3) a hypervisor. The construction technique depends on the hardware and hypervisor capabilities, with efficient solutions sometimes demanding para-virtualization.

Kernels and applications run within separate privilege domains to protect the kernel's integrity from the applications. They each have separate logical address spaces. Transitions between applications and the kernel involve hardware privilege changes, typically within the same hardware address space to make the privilege-change hardware efficient (see Figure 2.10 (a)).

A VM environment requires a third privilege domain for the hypervisor. The hypervisor must control the most privileged hardware, and thus executes at the most privileged level, consuming one of the hardware privilege levels. The guest kernel still requires protection from the guest applications, and thus we must use a technique to share the remaining privilege(s) between the guest kernel and its applications. We describe four approaches.

**Hardware protection:** If hardware provides secondary protection independent of the paging, then it is possible to partition the hardware address space into multiple logical spaces — for example, IA64 provides protection keys, and x86 provides segmentation. This approach works only if the hypervisor can use the additional hardware without interfering with the guest's use. The protection keys of IA64 can be virtualized in case of conflict, while the segmentation of x86 will cause problems, especially for recursive VM construction based on segmentation.

The x86 hardware provides four privilege levels, which it calls rings, implemented via segmentation and paging, and the hardware can automatically transition between the rings without hypervisor intervention. The paging prohibits user-level from accessing the memory of any higher privilege, and the segmentation prevents intermediate privileges from accessing higher privileges. Reverse access is permitted, so that each privilege level can access the lower-privileged memory. We can install the guest kernel at an intermediate privilege [KDC03] (see Figure 2.10 (b)). The rings are in limited supply and may collide with recursive VMs and guest kernels that use segmentation. Additionally, the CPU optimizes for the dual privilege modes of paging (e.g., the fast `sysenter` and `sysexit` instructions for system calls), whereas the intermediate privileges must use the slower legacy `int` system call instruction. Alternatively, the hypervisor can ignore the intermediate privilege levels, and execute both the guest kernel and application at user-level, with the guest kernel protected from its applications via a segment limit [KDC03]. This approach requires an indirection via the hypervisor to reconfigure the limit when transitioning between guest user and kernel, although when combined with `sysenter` and `sysexit`, this approach should have performance comparable to the hardware transitions of intermediate privileges.

**Dedicated VM hardware:** Processors with hardware support for hypervisors [Int05b, Int05a, AMD05, IBM05] provide more generalized solutions to the privilege problem (thus potentially supporting recursive VMs), and with performance accelera-

tion. Intel's VT [Int05b], for example, creates three privilege domains via two address spaces: one address space for the hypervisor, and another address space for the guest kernel and guest application to partition using the standard x86 hardware features. The result enables high performance interaction between the guest kernel and guest application, but slower transitions with the hypervisor due to complete address space switches. Although VT mitigates the number of switches to the hypervisor for processor emulation, it lacks a solution to mitigate switches caused by device emulation (high-speed device drivers access memory-mapped device registers; when using VT these raise page faults that the VMM must intercept, and then decode to determine how the faulting instruction is accessing the device register).

**Privilege emulation:** Not all hardware provides additional hardware protection mechanisms (e.g., AMD64 provides segment offsets without segment limits, which prevents its use for creating logical spaces). Additionally, not all hypervisors permit access to the additional hardware (e.g., when using traditional Linux as a hypervisor). We can instead dedicate an address space to each guest privilege level (see Figure 2.10 (c) and (d)). With hypervisor support, it is possible to accomplish this by changing only the page directory entries that implement the guest kernel space (Xen on AMD64 takes this approach). Otherwise, we map each guest privilege domain to a dedicated host address space and perform an address space switch when transitioning between guest kernel and guest application. This is our approach when using the L4Ka::Pistachio microkernel, Windows XP, and Linux as hypervisors. The use of separate address spaces requires us to emulate all memory accesses that cross privileges, i.e., when the guest kernel accesses memory in the guest application. Para-virtualization has been our traditional approach to handle cross-privileged memory accesses when running Linux on the L4Ka::Pistachio microkernel.

**Small address spaces:** The x86 segmentation enables an alternative privilege partitioning approach, called small address spaces [Lie95a, UDS+02], which were originally designed to emulate tagged TLBs on the x86. Rather than partition an address space into multiple privileges, small spaces partitions the user level into multiple logical address spaces each starting at address 0. This permits multiple guest OSes to coexist in the TLB (see Figure 2.11), and thus improves the performance of component systems built from VMs on x86, and are good candidates for running user-level device drivers. Small spaces have a restrictive limitation: each

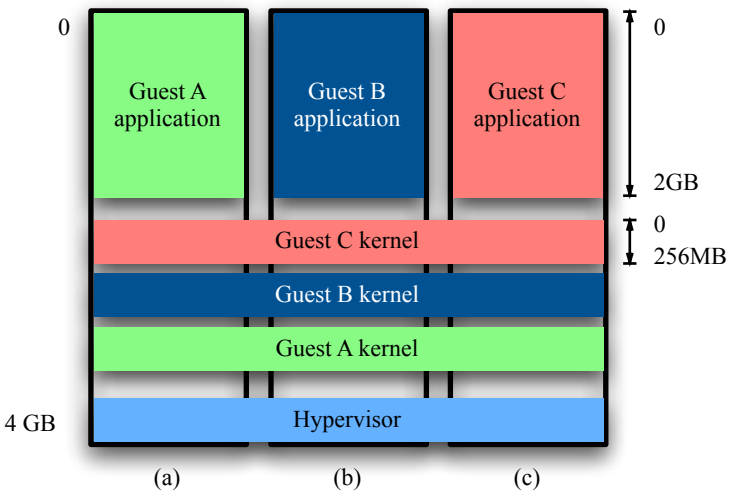



Figure 2.11: (*a*), (*b*), and (*c*) are three separate hardware address spaces, each with several logical address spaces, including three small spaces hosting guest kernels. Sharing multiple logical spaces in a single hardware space lowers TLB costs for component interaction with the guest kernels (e.g., device driver reuse); also scheduling a VM in a small address space is very fast compared to a full hardware space switch.

logical space must start at 0, and must fit within its small limit. In ref. [UDS+02], the authors run Linux applications in a large logical space, and the Linux kernel in a small logical space, which required relocating the Linux kernel to a low address. Linux expects to run at a high address (one of its assumptions is that page faults at low addresses belong to applications and not the kernel), and thus this approach requires adjusting the guest kernel source code.

### 2.4.7 Sensitive memory objects

In some architectures, particularly x86, the OS kernel controls parts of the hardware via memory objects. The memory objects occupy physical memory, in some cases at byte-aligned locations, and provide a place for the CPU to store bulk state off chip. A similar concept is memory-mapped I/O, but which has a subtle difference from memory objects: memory-mapped devices associate page-aligned *virtual addresses* with device registers, while sensitive memory objects associate device behavior with regions of *physical memory* (which may require no virtual memory addresses and no page alignment). The memory objects are an extension to the CPU's interface, and conceptually, the guest kernel issues downcalls to this

aspect of the CPU interface just as with other aspects. On x86, memory objects include the page tables, the task-state segment (TSS), the interrupt descriptor table (IDT), segment descriptor tables, and I/O permission bitmaps.

In a VM the memory objects become an interface to the VMM as opposed to the true hardware, and thus the VM must redirect the guest kernel's downcalls to the VMM. Converting the original downcalls into VMM downcalls is difficult. A constraint is that, just as when running on raw hardware, the guest kernel allocates the memory for these objects from its physical memory pool, which leaves the VM the responsibility of detecting accesses to the memory objects. The VM is unable to permit passthrough access to the hardware for handling these objects, since the objects are virtualization sensitive.

The downcalls may be expected to cause *immediate* side effects. For page tables, modifications will change mappings. For the IDT, changes will install new interrupt handlers. For the segment descriptors, changes will install new segments, and thus changes to the logical address space. The guest kernel may malfunction without the immediate side effects. A possible algorithm to fulfill the immediate side effects is to fabricate downcalls by detecting changes on the objects via write-protection, which would force page faults every time the guest tries to make changes, at which point the VM could re-enable write permissions, perform a single step (via the debug hardware), and then synchronize the changes with the hardware. This is a potential disaster for performance, because if the objects are byte aligned, then they may share page frames with other kernel data, particularly frequently written kernel data, causing false page faults. Para-virtualization can easily solve this problem by modifying the guest kernel to perform high-performance downcalls, such as function calls or hypercalls, or even by allocating dedicated pages to the objects to avoid the false sharing.

The approach to converting the guest's page table accesses into VMM downcalls is performance critical for many workloads (e.g., compiling the Linux kernel itself creates several new processes for each of the hundreds of kernel source files). Some of the approaches for generating the downcalls are:

**Brute force virtualization:** Use conventional page-based protection to protect page tables, forcing traps when the guest accesses them. This involves large overhead just for tracking the page tables, because the hardware may use physical addresses to identify them, while the guest accesses them via an arbitrary number of virtual mappings. Thus the VMM must detect and track the virtual mappings created by the guest for the page tables.

Emulation logic that executes within the guest's address space needs virtual addresses for updating the guest's page tables, otherwise it has to temporarily unprotect the guest's mappings so that the guest's virtual addresses may be used.

**Virtual TLB:** The VMM can treat the host page tables as a virtual TLB for the guest's page tables. This scheme fills the virtual TLB (host page tables) in response to guest page faults, and flushes the virtual TLB in response to guest TLB invalidations. Flush emulation is particularly costly for x86, which lacks an instruction to flush large regions of the virtual address space from the TLB and so instead falls back to flushing the entire address space. Additionally, x86 flushes the TLB upon every address space change, which forces the virtual TLB to be flushed as well. Thus every new activation of an address space involves a series of page faults to build mappings for the working set. The virtual TLB also has to emulate referenced and dirty bits for the page tables on x86 — it must expose these access bits in the guest's page table entries. The access bits represent reads and writes on virtual memory, and thus the VMM can use page protection via the host page tables to raise page faults on the guest's first access to data pages for synchronous updates of the access bits, at which point the VMM promotes the host page table entries to permit continuation of the guest accesses. Thus access-bit maintenance can cause more page faults than would be encountered on native hardware.

**Para-virtualization:** Via para-virtualization we can modify the guest kernel to issue efficient downcalls to the VMM. Yet downcalls still have overhead. Thus para-virtualization also has the capacity to alter the guest kernel's core algorithms to reduce the rate of downcalls. In para-virtualization one probably will treat the host page tables as a virtual TLB, but with the possibility to modify the guest kernel to add a tagged virtual TLB for architectures that lack tagging already, since using a tag reduces the rate of TLB invalidations. For example, since Linux already supports tagged TLBs on several RISC architectures, we can easily add hooks to its x86 port that tell the hypervisor to selectively flush addresses from inactive guest address spaces, rather than invalidating each outbound address space.

**Xen passthrough:** The Xen hypervisor uses a unique approach to managing the page tables, which exposes the host page tables to the guest kernel [BDF+03].

Since the host page tables contain virtualization sensitive state, Xen requires modifications to the guest kernel for explicitly translating host state to guest state when the guest kernel reads the page tables. An advantage for Xen's approach is that the guest kernel has efficient access to the hardware-maintained referenced and dirty bits. Another advantage is that it makes use of the guest's memory for backing the page tables, rather than hypervisor memory, which reduces the burden on the hypervisor's internal memory management. A disadvantage is the effort of modifying Linux to perform the host-to-guest state translation, since the guest kernel may read from the page tables without using an easily indirected abstraction.

**Automation with profile feedback:** An approach that we have proposed in past work is to automatically transform the guest's page table accesses into efficient downcalls. We proposed to locate the downcalls by profiling the guest under several workloads, and then to feed back the results to a second compilation stage. Yet since the profiling may not uncover all potential downcall sites, the VMM runtime must supplement it with more expensive approaches.

**Heuristics** Where the guest kernel follows known patterns for manipulating the page tables the VMM can prepopulate the host page tables with translations, thus reducing page faults and downcalls. Known behavior includes Unix's `fork()` and `exec()`.

**Dedicated VM hardware** Processors with hardware support for hypervisors can help virtualize sensitive memory objects, particularly for x86 [Int05b,Int05a, AMD05]. They can simplify the virtualization burden, and also improve performance.

# Chapter 3

# Device driver reuse

We reuse drivers by executing them within their original operating system inside of a virtual machine, and by submitting operational requests via proxies that we add to the driver's operating system. The approach offers substantial developer productivity (compared to implementing new drivers, and compared to transplanting old drivers) while satisfying the demands of a reuse environment: it insulates the recipient OS from the donor OS by restricting fault propagation, and by controlling resource scheduling; and it separates the architectures of the recipient and donor OS, promoting the prime reason for driver reuse — to write a structurally novel OS.

Drivers interact with their kernels via two interfaces: a resource-provisioning interface that provides building blocks for the drivers (e.g., linked lists, hash tables, interrupt handlers, locks, etc.); and a device-control interface that abstracts a class of devices behind a common interface. Our approach uses the driver's original OS to handle the resource-provisioning interface, thus permitting the driver to exist in its native environment. For the remaining interface — device control — we must implement translation infrastructure to mate the drivers with the recipient OS.

This chapter explains the system architecture for running drivers inside VMs, the problems introduced by the VM environment, and solutions. It then presents a reference implementation for reusing Linux device drivers in an environment based on the L4 microkernel.

## 3.1 Principles

Traditional device driver construction favors intimate relationships between the drivers and their kernel environments, which interferes with easy reuse of drivers, because it demands an engineering investment to emulate these intimate relationships. To achieve easy reuse of device drivers from a wide selection of operating systems, we propose the following architectural principles:

**Resource delegation:** The recipient OS provides only bulk resources to the driver, such as memory at page granularity, thus delegating to the driver the responsibility to further refine the bulk resources. The device driver converts its memory into linked lists and hash tables, it manages its stack layout to support reentrant interrupts, and divides its CPU time between its threads. In many cases the recipient OS may preempt the resources, particularly those outside the driver's working set.

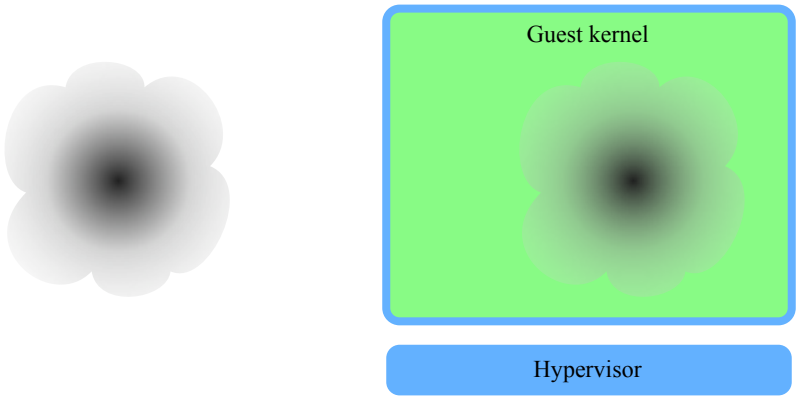
**Separation of address spaces:** The device driver executes within its own address space. This requirement avoids naming conflicts between driver instances, and helps contain memory faults.

**Separation of privilege:** Like applications, the device driver executes in unprivileged mode. It is unable to interfere with other OS components via privileged instructions.

**Secure isolation:** The device driver lacks access to the memory of non-trusting components. Likewise, the device driver is unable to affect the flow of execution in non-trusting components. These same properties also protect the device driver from the other system components. When non-trusting components share memory with the drivers, they are expected to protect their internal integrity; sensitive information is not stored on shared pages, or when it is, shadow copies are maintained in protected areas of the clients [GJP+00].

**Common API:** The driver allocates resources and controls its devices with an API common to all device drivers. The API is well documented, well understood, powerfully expressive, and relatively static.

Most legacy device drivers in their native state violate these proposed design principles. The legacy drivers use internal interfaces of their native operating systems, expect refined resources, execute privileged instructions, and share a global




(a) (b)

Figure 3.1: In *a* we see an isolated driver, with its large and amorphous resource-provisioning interfaces that a driver-reuse environment needs to supply. In *b* we have wrapped the driver with its original OS, thus reducing the burden on the hypervisor to only provide an interface for the well-defined bulk resources of the platform.

address space. On the other hand, their native operating systems partially satisfy our requirements: operating systems provide resource delegation and refinement, and use a common API–the system's instruction set and platform architecture. We can satisfy the principles by running the driver with its native OS inside a virtual machine. See Figure 3.1 for a comparison of lone-driver reuse versus driver reuse within a virtual machine.

## 3.2 Architecture

To reuse and isolate a device driver, we execute it and its native OS within a virtual machine. The VM virtualizes much of the machine, but not the devices controlled by the reused drivers: the VM selectively disables virtualization to permit the device driver OS (DD/OS) to directly access the devices that it controls, thus permitting the DD/OS to access the devices' registers, and to receive hardware interrupts. The VM and hypervisor, however, inhibit the DD/OS from seeing and accessing devices which belong to, or are shared by, other VMs.

The driver is reused in a component framework by clients, which are any processes in the system external to the VM, at a privileged or user level. The VM helps convert the DD/OS into a component for participation in a component framework, as described in Section 2.3.3. To connect the reused drivers to the component

framework, we install translation modules into the DD/OS that proxy between the clients and the DD/OS. The proxy modules serve the client requests, mapping them into sequences of DD/OS primitives for accessing the device; they also convert completed requests into responses to the clients, and propagate asynchronous data events to the clients. See Figure 3.2 for a diagram of the architecture.

The proxy modules execute within the DD/OS, and thus must be written as native components of the DD/OS. We write the proxy modules — they are a primary source of engineering effort in our reuse approach. If we choose the appropriate abstraction level in the DD/OS for writing the proxy modules, we can achieve a high ratio of driver reuse for the lines of code written for the proxy, hopefully without serious performance penalties. For example, a proxy module that interfaces with disk-devices should be able to reuse IDE disks, SCSI disks, floppy disks, optical media, etc., as opposed to reusing a particular device driver. The DD/OS could provide several viable environments for proxy modules with different levels of abstraction, such as running within the kernel and using the primitives of the drivers (e.g., directly sending network packets when the device is ready), or running within the kernel with abstracted interfaces (e.g., queuing network packets for when the device is ready), running at user-level (e.g., using a network socket), or running at user-level with raw device access (e.g., a network raw socket). The interfaces have trade-offs for the amount of code to implement the proxy module, the performance of the proxy module, and the maintenance effort to adapt the proxy modules to upgrades in the DD/OS.

To isolate device drivers from each other, we execute the drivers in separate and co-existing virtual machines. This also enables simultaneous reuse of drivers from incompatible operating systems. When an isolated driver relies on another (e.g., a device needs bus services), then the two DD/OSes are assembled into a client-server relationship. See Figure 3.3 for a diagram of the architecture.

We can build our VMs with full virtualization or para-virtualization, yet the result should be the same: no modifications required for the reused drivers. A driver is the module that controls a specific device (e.g., the Intel Pro/1000 network driver), which excludes all of the common services provided by its OS, such as IRQ management (the Pro/1000 driver uses kernel abstractions for handling interrupts, rather than driving the platform's interrupt controller directly). Although para-virtualization modifies the DD/OS to run as an application of the hypervisor, the modifications tend to focus on basic platform interfaces that the DD/OS abstracts for its device drivers, and so the drivers usually just need recompilation to operate

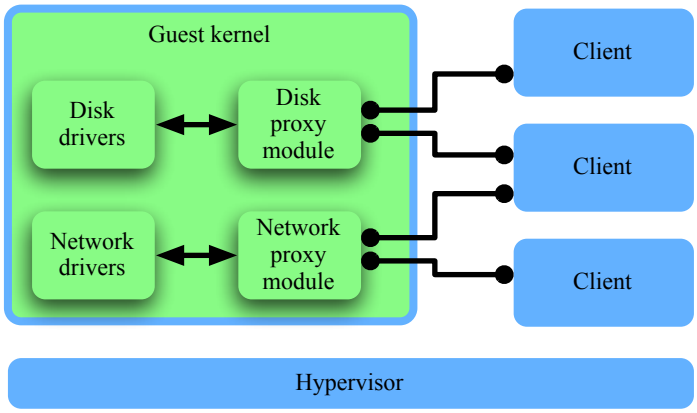



Figure 3.2: Device driver reuse by running the drivers, with their original OS, inside a VM. Native hypervisor applications, the *clients*, use the network and disk drivers. The proxy modules connect the reused drivers to the hypervisor applications.

in the para-virtualized environment.

### 3.2.1 Virtual machine environment

The virtualization architecture has five entities:

- The hypervisor is the privileged kernel, which securely multiplexes the processor between the virtual machines. It runs in privileged mode and enforces protection for memory and IO ports.
- The virtual machine monitor (VMM) allocates and manages resources and implements the virtualization layer, such as translating access faults into device emulations. The VMM can be either collocated with the hypervisor in privileged mode or unprivileged and interacting with the hypervisor through a specialized interface.
- Device driver OSes host unmodified legacy device drivers and have pass-through access to the devices. They control the devices via port IO or memory mapped IO and can initiate DMA. However, the VMM restricts access to only those devices that are managed by each particular DD/OS. The hypervisor treats the driver OSes as applications, as described in Section 2.3.2.
- Clients use device services exported by the DD/OSes, in a traditional client-server scenario. Recursive use of driver OSes is possible; i.e. a client can

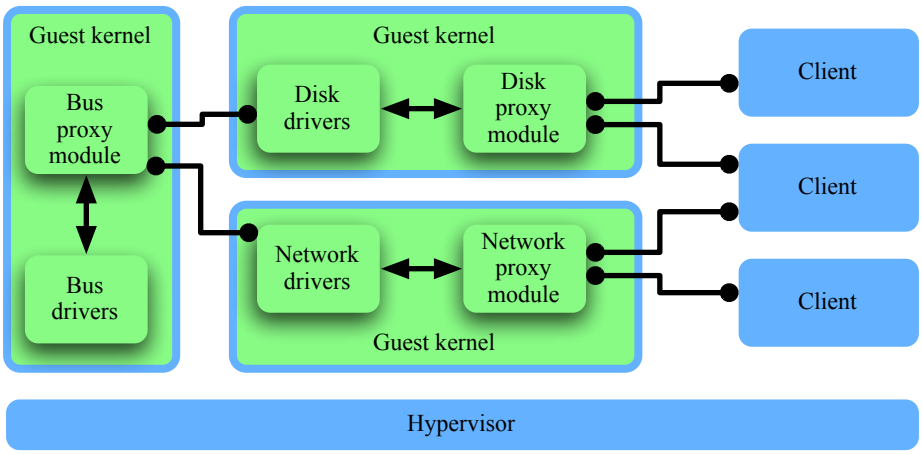



Figure 3.3: Device driver reuse *and* isolation, by running each driver in a separate VM. The block and network DD/OSes recursively use the bus DD/OS, to share the PCI bus. Native hypervisor applications, the *clients*, use the network and disk drivers.

act as a DD/OS server for another client. The client could be the hypervisor itself.

- Translation modules are added to DD/OSes to provide device services to the clients. They provide the interface for the client-to-DD/OS communication, and map client requests into DD/OS primitives. We will focus on translation modules that we can load into the DD/OS kernel, in the manner of a loadable driver or module.

The hypervisor must feature a low-overhead communication mechanism for inter-virtual-machine communication. For message notification, each VM can raise a communication interrupt in another VM and thereby signal a pending request. Similarly, on request completion the DD/OS can raise a completion interrupt in the client OS.

The hypervisor also is expected to provide a mechanism to share memory between multiple virtual machines. The hypervisor and VMM can register memory areas of one VM in another VM's physical memory space, similarly to memory-mapped device drivers.

### 3.2.2 Client requests

The proxy modules publish device control interfaces for use by the clients. The interface for client-to-DD/OS device communication is not defined by the hypervisor nor the VMM but rather left to the specific proxy module implementation. This allows for device-appropriate optimizations such as virtual interrupt coalescing, scatter-gather copying, shared buffers, and producer-consumer rings (see Figure 3.4).

The translation module makes one or more memory pages accessible to the clients and uses interrupts for signaling in a manner suitable for its interface and request requirements. This is very similar to interaction with real hardware devices. When the client signals the DD/OS, the VMM injects a virtual interrupt to cause invocation of the translation module. When the translation module signals the client in response, it invokes a downcall of the VMM.

Each hypervisor environment has unique inter-process communication (IPC) mechanisms, and the translation modules have unique interfaces exposed to the clients. These mechanisms and interfaces inherently relate to sharing data across the VM's protection domain, and thus are completely outside the perception of the DD/OS. To avoid polluting the DD/OS with knowledge of IPC, we implement the IPC logic within the hypervisor-specific translation modules.

## 3.3 Enhancing dependability

Commodity operating systems continue to employ system construction techniques that favor performance over dependability [PBB+02]. If their authors intend to improve system dependability, they face the challenge of enhancing the large existing device driver base, potentially without source code access to all drivers.

Our architecture improves system availability and reliability, while avoiding modifications to the device drivers, via driver isolation within virtual machines. The VM provides a hardware protection domain, deprivileges the driver, and inhibits its access to the remainder of the system (while also protecting the driver from the rest of the system). The use of the virtual machine supports today's systems and is practical in that it avoids a large engineering effort.

The device driver isolation helps to improve *reliability* by preventing fault propagation between independent components. It improves driver *availability* by supporting fine-grained driver restart (virtual machine reboot). Improved driver

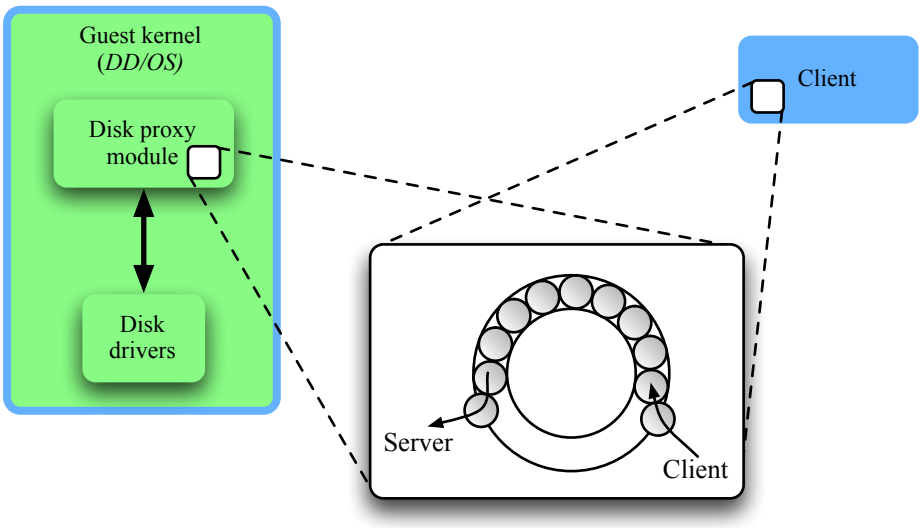



Figure 3.4: The DD/OS and client share memory that contains a producer-consumer ring for batching device requests. The ring permits concurrent collaboration without explicit synchronization. This thesis's reference implementation takes this approach.

availability leads to increased system reliability when clients of the drivers promote fault containment. Proactive restart of drivers, to reset latent errors or to upgrade drivers, reduces dependence on recursive fault containment, thus helping to improve overall system reliability [CF01].

The DD/OS solution supports a continuum of isolation scenarios, from individual driver isolation within dedicated VMs to grouping of all drivers within a single VM. Grouping drivers within the same DD/OS reduces the availability of the DD/OS to that of the least stable driver (if not further). Even with driver grouping, the entire system enjoys the benefits of fault isolation and driver restart.

Driver restart is a response to one of two event types: asynchronous (e.g., in response to fault detection [SBL03], or in response to a malicious driver), or synchronous (e.g., live upgrades [HAW+01] or proactive restart [CF01, CKF+04]). The reboot response to driver failure returns the driver to a known good state: its initial state. The synchronous variant has the advantage of being able to quiesce the DD/OS prior to rebooting, and to negotiate with clients to complete sensitive tasks. Our solution permits restart of any driver via a VM reboot. However, drivers that rely on a hardware reset to reinitialize their devices may not be able to recover their devices.

The interface between the DD/OS and its clients provides a natural layer of indirection to handle the discontinuity in service due to restarts. The indirection *captures* accesses to a restarting driver. The access is either delayed until the connection is transparently restarted [HAW+01] (requiring the DD/OS or the VMM to preserve canonical cached client state across the restart [SABL04]), or reflected back to the client as a fault.

## 3.4 Problems due to virtualization

Driver reuse via virtual machines introduces several issues: the DD/OS consumes resources beyond the needs of a bare device driver, the driver's DMA operations require address translation, the VM may not meet the timing needs of physical hardware, legacy operating systems are not designed to collaborate with other operating systems to control the devices within the system, and the architecture introduces user-level drivers that can interfere with high-speed device sequencing. This section presents solutions to these issues.

### 3.4.1 DMA address translation

DMA uses bus addresses, i.e., the addresses that the devices put onto the bus for routing to the memory banks. On a traditional x86 platform, the processor and devices use identical bus addresses, which are thus the physical addresses stored in the page tables. We call these addresses the *machine* or *bus* addresses, as opposed to the common term *physical*, since virtual machines change the meaning of physical addresses. A VM adds another layer of translation between the guest's physical addresses and the machine addresses. This layer of translation provides the illusion of a virtual machine with contiguous memory. In the normal case of virtualization (with no device pass through), the VMM maps the guest's physical addresses to machine address by intercepting all uses of physical addresses. The VMM has two cases. For the first, virtually paged memory, the VMM intercepts physical memory access by installing override page tables (or TLB entries) into the hardware instead of the guest's page tables (or TLB entries). For the second, virtual device accesses, the VMM carries out the translation upon device access (the VMM virtualizes a small set of real devices, making this a tractable approach). Since all physical accesses, including DMA requests, are intercepted, the VMM confines the guest to its compartment.

The DD/OS introduces a different scenario: the drivers have direct device access, thus bypassing the VMM and its interception of all uses of guest-physical addresses, which inadvertently exposes incorrect addresses to the devices' DMA engines, causing havoc. We need to either introduce the VMM's address interception and translation, or render it unnecessary. The particular approach depends on available hardware features and the virtualization method (full virtualization vs. para-virtualization).

In a para-virtual environment the DD/OS can incorporate the VMM's address translation into its building blocks for device drivers — for example, Linux offers to drivers several functions to convert physical addresses to bus addresses (which are normally idempotent translations on the x86 platform, but other platforms require the translation functions); these functions could be modified to apply the VMM's translation. Additionally, if a device translates a range of physical addresses, then the DMA operation expects the addresses to map to contiguous machine memory; the newer Linux kernels (such as 2.6.16) use large contiguous regions of memory in device operations. In a para-virtual environment, the DD/OS could be modified to consult the VMM when creating contiguous regions of memory for DMA — in Linux this is difficult, since Linux often assumes that the kernel's memory is already contiguous; we solve this by providing a single block of contiguous memory to the entire Linux VM that hosts the DD/OS; the alternative is to use memory trampolines of contiguous memory, to where the data for the DMA operations are temporarily copied (this is the solution used by Xen, since Xen changes the mappings between guest-physical and machine addresses on client device operations, precluding the sustainment of a contiguous VM). The hypervisor also has to support an interface for querying and pinning the VM's memory translations for the duration of the DMA operation (pinning is necessary since the hypervisor could transparently preempt pages from the VM).

Several hardware platforms permit bus address interception and translation for DMA operations via an IO-MMU, while also enforcing access permissions. The IO-MMU permits the VMM and hypervisor to control the translation from the DD/OS's physical addresses to the machine addresses (see Figure 3.5). Thus they enable complete hardware-isolation of the DD/OS, removing device drivers from the trusted computing base [LH03], and provide an alternative to the para-virtual approach. If we want to isolate device drivers from each other, each driver needs its own set of address translations. IO-MMUs tend to lack support for multiple address contexts per device, although the recent Directed I/O from Intel supports a

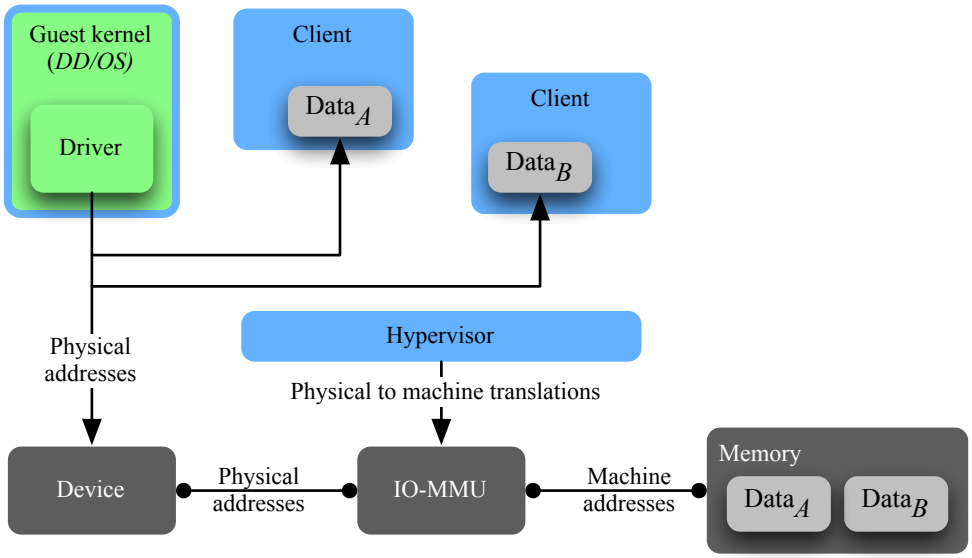



Figure 3.5: The DD/OS communicates directly with the device, and provides it physical addresses that reflect the locations of the data within the client address spaces. The hypervisor configures the IO-MMU to translate the physical addresses into true machine addresses.

per-device context [Int07]. With support for multiple contexts, the hypervisor can permit simultaneous DMA operations from different devices; without support for multiple contexts, the hypervisor must context-switch the IO-MMU between each device driver VM, thus serializing the device access. We have demonstrated this context switching in prior work [LUSG04].

For full virtualization where we must use the guest's built-in DMA address preparation, without an IO-MMU to intercept DMA accesses, we must eliminate the need for interception so that the proper addresses reach the devices. In this case, we provide identity mappings to machine memory for the DD/OS's physical memory. This need not restrict the system to a single DD/OS instance, since the idempotent mappings are strictly needed only for memory that might be exposed to DMA, which excludes quite a bit of a kernel's memory. In many cases device drivers only issue DMA operations on dynamically allocated memory, such as the heap or page pool. Hence, only those pages require the idempotent mapping restriction. Thus when running multiple DD/OS instances, we need a scheme to setup identity mappings as we boot the DD/OS instances one after the other. We can achieve this by using a memory balloon driver [Wal02], which can reclaim pages from one DD/OS for use in other DD/OSes, effectively distributing DMA-

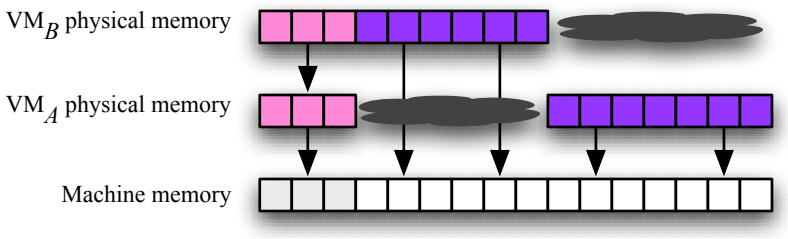



Figure 3.6: DMA memory allocation for two VMs in full virtualization, where guest physical memory must map idempotently to machine memory. The read-only sections of both VM$_A$ and VM$_B$ share the same machine pages. The writable areas of the two VMs use non-overlapping machine memory, protected via collaborative balloon drivers.

capable pages between all DD/OSes (see Figure 3.6). Note that once the memory balloon driver has claimed ownership of a page in a DD/OS, that DD/OS will never attempt to use that page. DMA from static data pages, such as microcode for SCSI controllers, further requires idempotent mapping of data pages. However, dynamic driver instantiation usually places drivers into memory allocated from the page pool anyway. Alternatively, one DD/OS can run completely unrelocated; multiple instances of the same OS can potentially share the read-only parts.

It is important to note that, in the absence of an IO/MMU, all solutions assume well-behaving DD/OSes. Without special hardware support, DD/OSes can still bypass memory protection by performing DMA to physical memory outside their compartments.

### 3.4.2 DMA and trust

Code with unrestricted access to DMA-capable hardware devices can circumvent standard memory protection mechanisms. A malicious driver can potentially elevate its privileges by using DMA to replace hypervisor code or data. In any system without explicit hardware support to restrict DMA accesses, we have to consider DMA-capable device drivers as part of the trusted computing base.

Isolating device drivers in separate virtual machines can still be beneficial. Nooks [SBL03] isolates drivers in separate address spaces, despite being susceptible to DMA attacks, but still reports a successful recovery rate of 99% for synthetically injected driver bugs — the fundamental assumption is that device drivers have a variety of fault sources.

We differentiate between three trust scenarios. In the first scenario only the

client of the DD/OS is untrusted. In the second case both the client as well as the DD/OS are untrusted by the hypervisor. In the third scenario the client and DD/OS also distrust each other. Note that the latter two cases can only be enforced with DMA restrictions as described in the next section.

During a DMA operation, page translations targeted by DMA have to stay constant. If the DD/OS's memory is not statically allocated it has to explicitly pin the memory (in regards to the hypervisor). When the DD/OS initiates DMA in or out of the client's memory to eliminate copying overhead, it must pin that memory as well. In the case that the DD/OS is untrusted, the hypervisor has to enable DMA permissions to the memory and to ensure that the DD/OS cannot run denial-of-service attacks by pinning excessive amounts of physical memory.

When the DD/OS and client distrust each other, further provisions are required. If the DD/OS gets charged for pinning memory, a malicious client could run a denial-of-service attack against the driver. A similar attack by the DD/OS against the client is possible when the DD/OS performs the pinning on behalf of the client. The solution is a cooperative approach with both untrusted parties involved. The client performs the pin operation on its own memory, which eliminates a potential denial-of-service attack by the DD/OS. Then, the DD/OS validates with the hypervisor that the pages are sufficiently pinned. By using time-bound pinning [LUE+99] guaranteed by the hypervisor, the DD/OS can safely perform the DMA operation.

Page translations also have to stay pinned during a VM restart, since a faulting DD/OS may leave a device actively using DMA. All potentially targeted memory thus cannot be reclaimed until the VMM is sure that outstanding DMA operations have either completed or aborted. Likewise, client OSes must not use memory handed out to the faulted DD/OS until its restart has completed.

### 3.4.3 Resource consumption

Each DD/OS consumes resources that extend beyond the inherent needs of the driver itself. The DD/OS needs memory for code and data of the entire OS. Furthermore, each DD/OS has a certain dynamic processing overhead for periodic timers and housekeeping, such as page aging and cleaning. Periodic tasks in DD/OSes lead to cache and TLB footprints, imposing overhead on the clients even when not using any device drivers.

Page sharing, as described in ref. [Wal02], significantly reduces the memory and cache footprint induced by individual DD/OSes. The sharing level can be very

high when the same DD/OS kernel image is used multiple times and customized with loadable device drivers. In particular, the steady-state cache footprint of concurrent DD/OSes is reduced since the same housekeeping code is executed. It is important to note that memory sharing not only reduces overall memory consumption but also the cache footprint for physically tagged caches.

The VMM can further reduce the memory consumption of a VM by swapping unused pages to disk. However, this approach is infeasible for the DD/OS running the swap device itself (and its dependency chain). Hence, standard page swapping is permitted to all but the swap DD/OS. When treating the DD/OS as a black box, we cannot swap unused parts of the swap DD/OS via working set analysis. All parts of the OS must always be in main memory to guarantee full functionality even for rare corner cases.

Besides memory sharing and swapping, we can use three methods to further reduce the memory footprint. Firstly, memory ballooning actively allocates memory in the DD/OS, leading to self-paging [Wal02, Han99]. The freed memory is handed back to the VMM. Secondly, we treat zero pages specially since they can be trivially restored. Finally, we compress [AL91, CCB99] the remaining pages that do not belong to the active working set and that are not safe to swap, and uncompress them on access.

Page swapping and compression are limited to machines with DMA hardware that can fault on accesses to unmapped pages and then restart the DMA operation. Otherwise, a DMA operation could access invalid data (it must be assumed that all pages of a DD/OS are pinned and available for DMA when treating the DD/OS as a black box).

Periodic tasks like timers can create a non-negligible steady-state runtime overhead. In some cases the requirements on the runtime environment for a DD/OS whose sole purpose is to encapsulate a device driver can be weakened in favor of less resource consumption. For example, a certain clock drift is acceptable for an idle VM as long as it does not lead to malfunction of the driver itself, allowing us to schedule OSes less frequently or to simply drop their timer ticks.

### 3.4.4 Timing

Time multiplexing of multiple VMs can violate timing assumptions made in the operating system code. OSes assume linear time and non-interrupted execution. Introducing a virtual time base and slowing down the VM only works if there is no dependence on real time. Hardware devices, however, are not subject to

this virtual time base. Violating the timing assumptions of device drivers, such as short delays using busy waiting or bounded response times, can potentially lead to malfunctioning of the device.[1]

We use a scheduling heuristic to avoid preemption within time critical sections, very similar to our approach to lock-holder preemption avoidance described in [ULSD04]. When consecutive operations are time-bound, operating systems usually disable preemption—for example, by disabling hardware interrupts. When the VMM scheduler would preempt a virtual processor but interrupts are disabled, we postpone the preemption until interrupts are re-enabled, thereby preserving the timing assumptions of the OS. This requires the VMM to trap the re-enable operation. Hard preemption after a maximum period avoids potential denial-of-service attacks by malicious VMs.

### 3.4.5 Shared hardware and recursion

Device drivers assume exclusive access to the hardware device. In many cases exclusiveness can be guaranteed by partitioning the system and only giving device access to a single DD/OS. Inherently shared resources, such as the PCI bus and PCI configuration space, are incompatible with partitioning and require shared and synchronized access for multiple DD/OSes. Following our reuse approach, we give one DD/OS full access to the shared device; all other DD/OSes use driver stubs to access the shared device. The server part in the controlling DD/OS can then apply a fine-grained partitioning policy. For example, our PCI DD/OS partitions devices based on a configuration file, but makes PCI bridges read-only accessible to all client DD/OSes. To simplify VM device discovery, additional virtual devices can be registered.

In a fully virtualized environment, some device drivers cannot be replaced dynamically with our driver stubs for accessing shared devices. Linux 2.4, for example, does not allow substituting the PCI bus driver. In those cases, the clients rely on full hardware emulation for the shared device (just as in a normal VM), rather than driver stubs. For example, a DD/OS that drives a network card needs PCI bus services, and when it tries to access the PCI bus, will be virtualized by the VMM which forwards the operations to the PCI-bus DD/OS. The number of such devices is quite limited. In the case of Linux 2.4 the limitations include PCI, the interrupt

[1] Busy waiting, which relies on correct calibration at boot time, is particularly problematic when the calibration period exceeds a VM scheduling time slice and thus reports a slower processor. A device driver using busy waiting will then undershoot a device's minimal timing requirements.

controller, keyboard, mouse, and real-time clock.

### 3.4.6 Sequencing

Achieving full utilization of high-speed devices introduces real-time constraints on the system, particularly for the ever more important network interfaces. A typical device has finite bandwidth, allocated across discrete device requests. The system must submit the requests in a timely fashion to prevent the device from going idle — an idle device is a lost resource, or worse, a claim on the device's future resources if requests are pending but delayed elsewhere in the system.

The device demands more than real-time scheduling of discrete device requests: it also requires that the CPU generate the *data* that accompany the device requests. Since we reuse our drivers in VMs, we add the costs of crossing and scheduling the DD/OS' protection domains to overhead (which on x86 involves many TLB flushes at great expense), which subtracts from time that could be used for feeding the device. Monolithic systems have the advantage of low-latency device accesses, and thus can offer more time for generating the data, and can use synchronous techniques for submitting device operations since they have low latency (only function-call overhead). Using synchronous techniques with driver reuse could add too much overhead (protection-domain crossing overhead), violating the real time constraints.

To illustrate, we consider using a network card at full line rate for transmitting a stream of packets. It transmits Ethernet packets at 1500 bytes each. For sustaining a given bandwidth $W$ (bytes per second), we must submit packets at frequency $F = W/1500$ bytes. The period is $T = 1/F$. For a CPU with speed $H$ (Hertz), the period in cycles per packet is $C = HT$. Given $C$, we can estimate the number of context switches per second we can sustain at full device utilization. For example, we choose a gigabit Ethernet card that sustains a bandwidth of $W = 112$ MB/s, with a CPU that runs at $H = 2.8$ GHz. Then $F = 74667$ packets per second, or one packet every 13.4 $\mu$s. This is a packet every 37500 cycles. Assuming a cost of 700 cycles for the address space switch, and then a subsequent TLB refill cost of 15 entries for 3000 cycles, we can tolerate ten address space switches between packet submissions, with no time remaining for system calls and packet processing. These numbers are highly sensitive to packet size and TLB working set size.

The real-time constraints impose a limit on the number of possible address space switches between the DD/OS and the remainder of the system. The solution is to avoid synchronizing between the DD/OS and its clients for every packet.

Instead, they should use well known concurrency techniques such as producer-consumer rings in shared memory [UK98], so that multiple packets arrive with each address space switch. We still need to schedule the DD/OS to eventually transfer the packets from the producer-consumer ring to the device. Thus the DD/OS's minimum scheduling rate is determined by: the bandwidth of the device, the size of the producer-consumer ring between the DD/OS and client, and the size of the producer-consumer ring between the DD/OS and the network device. For example, in one of the later experiments of this thesis, we use 512 descriptor entries in the device's producer-consumer ring for packet transmit. This requires a minimum scheduling period for the DD/OS of 19.2 million cycles (6.86 ms). But for scatter-gather transmissions, with three descriptors per packet, it reduces to a minimum period of 6.4 million cycles (2.28 ms). A convenient approach to ensuring that the device achieves its scheduling period is to rely on the device's interrupt delivery, but mitigated to the appropriate scheduling period, and using an event that signifies that the producer-consumer ring is *almost* empty, as opposed to empty — if the device ring goes empty, then the device becomes underutilized. Since networking is also often sensitive to the latency of acknowledgment packets, this minimum scheduling period may introduce intolerable latency, in which case a more balanced scheduling rate must be found.

## 3.5 Translation modules

Most of the effort behind driver reuse is for the design and construction of efficient proxy modules, particularly when the clients attempt to use the devices in a manner unanticipated by the DD/OS, which requires effort devising heuristics and algorithms that map the intentions of the clients to the capabilities of the DD/OS. Here we discuss some of the general features that a DD/OS can provide to facilitate construction of the proxy modules. We focus on high performance, and so discuss the interfaces internal to the DD/OS's kernel, rather than the DD/OS's user-level services.

### 3.5.1 Direct DMA

For good performance we want to initiate DMA operations directly on the memory of the driver clients. The alternatives, which we prefer to avoid, are to copy data between the driver clients and the DD/OS, and to remap data between the driver clients' and the DD/OS' address spaces. Data copying not only wastes time, but

also pollutes the cache. The remapping approach introduces costs for hypercalls to create the mappings (since the hypervisor must validate the new mappings), and multiprocessor costs for TLB coherency; additionally, it expects page-granular data objects, which requires modifications to the DD/OS for devices that asynchronously generate data (e.g., inbound network packets).

DMA operations on client memory require: (1) infrastructure within the DD/OS to indirectly name the client's memory *without* allocating virtual addresses for the client's memory; (2) the ability to prevent the DD/OS from accidentally accessing the client's memory (unless the DD/OS first creates a temporary virtual mapping); and (3) the ability to express non-contiguous regions of memory. Many kernels already indirectly address physical memory without using virtual mappings, to handle insufficient virtual address space for direct mappings, although their interfaces may be insufficient — for example, network packets often require virtual mappings since the kernel itself generates the packet headers. A fallback for situations unable to use indirect access is to map the client's data into the physical or virtual address space of the DD/OS, which makes the data addressable (and potentially introducing problems where the guest expects contiguous physical pages for DMA, but where the hypervisor has mapped the physical pages to arbitrary machine pages); or to copy the data (which is our preferred approach for distributing inbound network packets).

### 3.5.2 Correlation

The proxy module correlates client requests with DD/OS requests, so that upon the completion of an operation, the proxy module can release its associated resources and notify the client. Several factors complicate the correlation:

- We want the proxy module to submit multiple device requests, of the same device class, in parallel. This batching amortizes the costs of address space switching between the clients and the DD/OS.
- Some of the devices complete their requests out of order, particularly if the DD/OS reorders the requests behind the driver control interface.
- Some devices support scatter-gather requests (thus containing subrequests), with DMA to multiple non-contiguous regions of memory.
- The proxy may introduce request fragmentation if the DD/OS and client use different request units, and thus the proxy associates $n$ client operands with

$m$ DD/OS operations.

For the correlation we prefer a simple lookup operation that maps the completed DD/OS device request back to the client's original request, while supporting several outstanding requests due to batching. A possible implementation is to store the correlation within the DD/OS request, if the request has an unused field member for such purposes, for this avoids additional data structure management. Alternatively, we would have to maintain and operate upon a dedicated correlation data structure within the proxy module (which would minimally introduce insertion, lookup, and deletion costs).

### 3.5.3 Flow control

The architecture has three independently schedulable entities that produce and consume data at potentially different rates. The client produces data that the DD/OS consumes, and the DD/OS produces data that the device consumes. Some devices have a reverse flow too, where the device produces data that the DD/OS consumes, and the DD/OS produces data that the client consumes. We need a flow control system to equalize the data production and consumption rates. The flow control consists of two parts: shared buffers that permit minor deviations in data rates, and signaling that permits major deviation in data rates. The signaling is an explicit feedback mechanism which permits one entity to ask another to stop the data flow, and to then later restart data flow.

High performance devices will already include flow-control systems. The hypervisor must provide the flow control mechanism between the DD/OS and the clients. The DD/OS is responsible for exposing flow-control interfaces to the proxy modules, to relay the devices' flow-control feedback to the proxy modules. This is a typical feature of kernels, as they specialize in I/O sequencing.

### 3.5.4 Upcalls

Both the client and DD/OS invoke the proxy to initiate and complete device operations; from the perspective of the DD/OS, the invocations to the colocated proxies are upcalls [Cla85]. Upcalls pose a problem of environment: they may execute in an interrupt handler context, the current kernel context (which might be the kernel stack of the active guest application), or a dedicated kernel thread. The environments restrict the actions of the proxy logic, particularly in regards to DD/OS reentrance, i.e., the set of function calls the proxy logic is permitted to execute

within its current context; also the amount of time that the proxy logic is permitted to consume; and the resources that the proxy logic can consume and release (i.e., to avoid deadlock). These are all issues that typical developers must address when writing device drivers.

For the proxy to execute code on behalf of its clients, it needs upcalls triggered by the clients. This is a form of IPC across the VM boundary, which the hypervisor can implement as OS bypass: the hypervisor can activate the *address space* of the DD/OS, and then spawn a thread of execution specific to the proxy module (in which case the proxy module must prearrange stack space for this thread of execution). The proxy can hide its OS bypass from the DD/OS as long as it executes only proxy logic, and accesses only proxy data (without raising page faults). The hypervisor can perform OS bypass to applications of the DD/OS as well, subject to avoiding page faults (since a page fault from an unscheduled application will confuse the DD/OS). When the proxy requires activity from the DD/OS, the proxy must preserve DD/OS integrity, and thus activate a valid DD/OS context. The DD/OS already provides a means to activate a valid context on demand, via interrupt delivery, and thus we synthesize a virtual device interrupt; it permits interruption of the current kernel activity to register a pending event, so that the kernel can schedule an event handler to execute at a convenient time, and in an appropriate context.

The proxy invokes hypercall services during the upcalls. It should use only hypercall services that respect the current execution context of the DD/OS (e.g., quick operations if executed in an interrupt context).

### 3.5.5 Low-level primitives

We favor very low-level device control interfaces to maximize exposure to the features of the devices (i.e., high transparency as argued by Parnas [PS75]), such as checksum calculation on network cards. The abstractions of a DD/OS can interfere with access to the raw features of a device — for example, the DD/OS could hide random-access block devices behind contiguous files, or hide packetized network interfaces behind packet streams.

The device control interfaces should be executable by arbitrary kernel subsystems, and not just system calls from applications.

The various hypervisor environments will require some configuration flexibility within the DD/OS — for example, some hypervisors may transfer network packets between components by remapping pages, in which case each packet re-

quires a dedicated page; or the hypervisor may copy or transfer in place via DMA, in which case packets can share page frames.

### 3.5.6 Resource constraint

When the DD/OS solely provides device services, without any other general purpose computing services, then it is very helpful to reduce the DD/OS's resource consumption to the minimum required for device services. For example, the OS should be able to disable periodic house keeping tasks, to reduce memory consumption, and to exclude superfluous subsystems.

Additionally, it is helpful for the DD/OS to provide resource configurability. For example, the DD/OS could support different link addresses, so that running it within an x86 small address space is feasible [UDS+02].

### 3.5.7 Device management

The proxy should be able to describe the devices and their features to the clients, to pass device events to the clients, and to permit the clients to configure the devices.

Some of the operations:

1. The proxy must be able to discover available devices to offer to the clients. It must be able to report the device capabilities to the clients (e.g., checksum computation offload, jumbo packets, and DVD burning).

2. The client can choose one of several devices to use. The proxy must be able to register with the DD/OS as a consumer of the device, potentially claiming exclusive access.

3. The proxy must be informed of changes in device state, such as network link status, hot plug device events, and device sleep requests.

4. The proxy needs methods for querying the device statistics, such as network error counts.

## 3.6 Linux driver reuse

Linux has been our DD/OS, because it is freely available with source code, it has substantial device support, and it has an active community maintaining and adding device drivers. Linux would be a likely candidate to serve as the DD/OS for other

projects too (Xen has chosen Linux). Linux also fulfills many of our guidelines for easy proxy module construction. We describe here how we implemented the Linux proxy modules for network and disk-like device access, using in-kernel interfaces.

Although Linux helps with building proxy modules, it has deficiencies, particularly for performance, and so some of our solutions take advantage of very detailed knowledge of Linux internals to circumvent the deficiencies (despite criticizing prior driver-reuse approaches for requiring intimate knowledge). Our approach can succeed without this detailed knowledge, and we started without it for our initial implementation, but in the pursuit of performance refined our solution.

### 3.6.1 DMA for client memory

The client sends device requests to the Linux kernel to perform operations on client memory. We generate internal Linux device requests, using Linux's internal memory addressing, which is designed to operate on Linux's memory, as opposed to the memory of the clients. We thus must express the client memory in the terms of Linux's internal addressing.

To generate a DMA address for a data object, Linux first generates a physical address for that object (i.e., the address used within the paging infrastructure), and then converts it to a DMA address (i.e., the bus address). Linux has two approaches for generating physical addresses: (1) convert kernel virtual addresses into physical addresses, which permits the direct conversion of a kernel data object into a physical address; and (2) name the memory via an indirection data structure, which permits a data object to exist without a virtual address. The algorithm for converting virtual to physical is a simple equation, $v = p + C$, where $v$ is the virtual address, $p$ is the physical address, and $C$ is the start of the kernel's virtual address space. The indirection data structure uses an array, where the index of the array represents the physical page. It is thus trivial to convert from a virtual address to an indirection address; the reverse is trivially possible for a subset of physical pages; the remaining indirection address space requires temporary virtual mappings created on the fly.

Linux has insufficient virtual address space (because it shares the virtual address space with its application), and insufficient indirect address space (because the indirect space covers the amount of physical memory in the machine), to name all potential client memory. We have three possible solutions to provide Linux with the ability to name client memory: (1) give the Linux kernel a full-sized virtual address space; (2) remap the kernel's physical address space, dynamically, to

handle all active device operations; or (3) give the Linux kernel a full-sized indirect naming space.

**Full-sized kernel space:** If we give the Linux kernel a full-sized address space, then it can create virtual addresses for all machine memory (unless the machine can address more physical memory than virtual memory), which enables Linux to create valid DMA addresses. We would permanently map all client memory into the DD/OS, and then transform client requests into the internal virtual addresses of the DD/OS. This solution requires the DD/OS to manage its virtual address space in a manner that permits mapping all machine (or only client) memory, rather than to allocate its virtual address space for other uses. Linux can be stretched to a full address space by relinking it to a low address, and by changing some of its assumptions about its memory layout (e.g., Linux currently treats page faults on low addresses as user page faults, but in a full-size address space, they could also be kernel page faults). The authors in ref. [UDS+02] successfully relinked Linux to run at low addresses, but for small spaces (not for a large space). The hypervisor can interfere with the creation of a large space, because the hypervisor itself consumes virtual address space. This approach requires para-virtualization. Linux has also offered experimental support for a full kernel address space [Mol03].

**Dynamic remapping:** Since the VMM introduces a layer of translation between Linux's physical addresses and the true machine addresses, the hypervisor can change the mapping of physical to machine on demand; this permits the proxy to choose a fixed set of virtual and physical addresses for rotating among the clients' machine pages; thus Linux can use its limited set of virtual addresses to temporarily generate physical addresses for all of machine memory. Dynamic page remapping has several problems: (1) a performance cost for hypercalls, since the hypervisor must confirm the legitimacy of the mapping; (2) a performance cost for TLB invalidations, which is especially high on multiprocessor machines; (3) the kernel may not have enough virtual address space for all in-flight device operations; and (4) the solution requires infrastructure to allocate and free the dynamic virtual memory ranges, which also requires tracking the free entries. Without the support of an IO/MMU, this approach also requires para-virtualization. The Xen project uses the remapping approach.

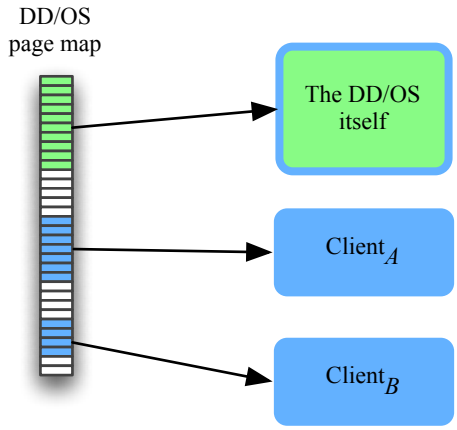



Figure 3.7: Linux represents all of machine memory with the page map data structure. If Linux were to allocate a page map with only enough entries for the memory allocated to its VM, it would be unable to reference client memory; we thus cause Linux to allocate a page map with sufficient entries for client memory too.

**Indirect addressing:** Indirect memory addressing more closely suits the goal of driver reuse, and is the solution that we implement. High-speed devices perform DMA, and thus already use indirect addressing (as opposed to virtual addresses); additionally if they use bus address translation (e.g., an IO/MMU), they can use a different mapping space than the kernel's virtual address space. Furthermore, driver reuse asks the DD/OS to perform device operations, not data inspection or transformation, and thus the DD/OS has no reason to map the client data into its kernel address space. In cases where the drivers must inspect or transform the client data for shared peripheral buses, either the clients can provide the bus-related information separately, or the DD/OS can revert to remapping. Some devices lack DMA engines, and instead require their driver to manually copy data into device registers; for these cases we can also fall back to the dynamic remapping approach, or alternatively copy the data into the DD/OS.

Linux already provides a data structure for memory indirection, the *page map*. To be useful in a DD/OS, this page map must be allocated to match the size of all of machine memory (or all of the client memory areas), rather than the memory allocated to the DD/OS's virtual machine (see Figure 3.7). We currently inform the DD/OS VM of all machine memory, and reserve the client memory in the VM's BIOS, so that Linux detects that the client memory is unusable; if Linux attempts to access the client memory, it will raise a VM fault.

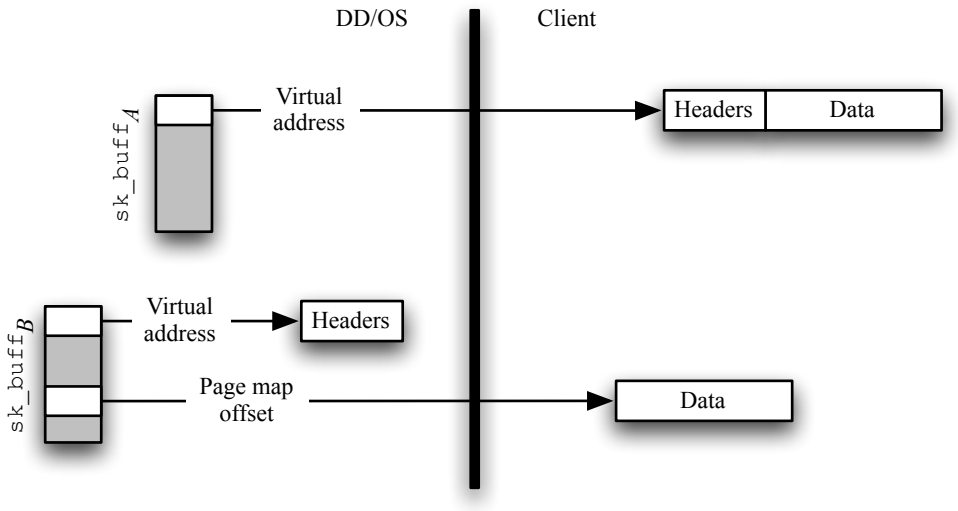



Figure 3.8: $\texttt{sk\_buff}_A$ references the entire packet in the client's memory space using a DD/OS (invalid) virtual address. This is a brittle use of the `sk_buff`, and thus has limitations. If the limitations must be violated, then we can use the alternative configuration in $\texttt{sk\_buff}_B$, which uses a valid DD/OS virtual address for the packet headers, and page map indirection for the remainder of the packet.

### 3.6.2 Network

Our proxy module uses two different internal Linux interfaces for network transmit and receive. We install our proxy as a Linux kernel module, and it uses the internal interfaces of the Linux kernel as close to the device as possible, but still taking advantage of Linux's network packet queuing and flow control.

#### Transmit

Packet transmission requires the DD/OS to send the client's raw packets, which already include packet headers. The DD/OS should avoid inspecting the packets, and must avoid modifying the packets. Linux abstracts packets as an `sk_buff`. The `sk_buff` is a data structure that describes the packet, points at the packet headers, and points at the packet data (even if the data is fragmented across memory). Linux expects to have virtual addresses for the packet's headers, but it can address the data payload indirectly and thus without virtual addresses. As the packets traverse through the Linux network stack, networking subsystems may inspect and duplicate the packets, depending on bridging, routing, and filtering options, in which case Linux may temporarily map the packets.

Our solution uses indirect memory addressing, even where Linux expects a valid virtual address for the beginning of the packet headers. This requires us to fake an indirect address within the virtual address space; we know how Linux converts a virtual address to a DMA address, and we use the inverse operation to fabricate an invalid virtual address, used by the `sk_buff` to point at the primary packet payload. This requires us to ensure that Linux never dereferences our fake virtual address — we configure Linux to transmit outbound packets without inspection. In some reuse scenarios, such as using the DD/OS also as a general purpose OS, the user may configure Linux in a way that requires valid virtual addresses in the `sk_buff`, in which case we can copy some packet headers from the client packet into the `sk_buff`, and use indirect addressing for the remainder of the packet. See Figure 3.8 for a diagram of different `sk_buff` scenarios.

The packet transmission requires the packet to remain pinned in memory for DMA, and thus needs a cleanup phase. Thus we want to inform the proxy module and the client when the packet has been successfully transmitted (or if transmission has been aborted), to start the cleanup phase. The notification should happen in a timely manner, to avoid network protocol timeouts from firing in the client. Thus we require an upcall from the DD/OS kernel to the proxy module, with information about the packets that have completed. Currently we use an `sk_buff` destructor hook in Linux; and we store state in the `sk_buff` payload area to help correlate the `sk_buff` and its fragments with the packet fragments transmitted by the client.

Since the client can send packets faster than the device can transmit them, we need a feedback loop to pause packet transmission (and thus pause the client), and to restart transmission after the device has made forward progress. Linux has an internal flow-control mechanism, behind its `dev_queue_xmit()` function, which enqueues a single `sk_buff` for transmission. Once Linux has control of the packet, it collaborates with the network device to transmit the packet at line speed (or to abort the packet). Furthermore, we use a producer-consumer ring shared between the proxy and each client, which queues packets to transmit. When the queue fills, the client pauses. When the proxy relieves pressure from a paused producer-consumer ring, it awakens the client, to continue with further transmissions.

After the client adds packets to the producer-consumer ring, it asks the hypervisor to signal and schedule the proxy module to transmit the packets. This involves a hypercall and other overhead. We avoid the hypercall if we know that the proxy will soon awaken, either because its device has pending activity that will raise fu-

ture device interrupts, or because other clients have already arranged to awaken the proxy. This permits us to combine all the activity into a single proxy session, rather than to have consecutive proxy sessions separated by expensive context changes to other VMs and activities. We thus share a page between the proxy and the clients, where the proxy can provide information about upcoming events.

### Receive

Inbound packets may be destined for the DD/OS itself, its device clients, or some combination depending on broadcast and multicast properties.

The proxy must inspect every inbound packet for potential delivery to a client; this requires an upcall from the DD/OS to the proxy. The upcall must be able to execute in any context that supports packet processing (e.g., interrupt context). The upcall must pass pointers to the inbound packets, so that the proxy can inspect the packets. If the proxy determines that a packet is for a client, then the proxy can take responsibility of the packet (and thus its cleanup), and remove it from the normal packet handling chain of the DD/OS. The latency between packet reception and hand-off to the proxy should be minimized. Currently we capture the Linux bridging hook, where normally its bridging layer would connect, and thus we prevent the DD/OS from simultaneously serving as a normal network bridge; we capture the bridging hook because it provides a low-latency upcall to the proxy module. Alternatively, we could coexist with and use Linux's bridging logic, but this would increase latency, require us to implement a virtual network interface for the bridge, and involve user-level applications to configure the bridge.

Inbound packet processing is often ignited by a device interrupt after several packets have accumulated on the NIC. Thus the kernel can process these several packets together, and if they are transferred to the same client, they can be transferred together. Batching packets together is important for amortizing hypercall overhead. The proxy needs to know when to send a batch of packets to the clients, and thus must learn from the DD/OS when all pending inbound packets have been processed. We use a second upcall from the DD/OS here: we currently schedule a Linux tasklet via `tasklet_hi_schedule()`, which Linux happens to run immediately after packet processing completes, thus achieving our desired queue-to-deliver latency.

Since inbound packets have unpredictable destinations (or multiple destinations for broadcast and multicast), we must inspect the packets within the DD/OS for their routing destinations, and thus must configure the networking hardware to

deliver all packets into the memory of the DD/OS. This complicates zero-copy network receive. To achieve zero-copy we must remap the pages from the DD/OS to the clients (the solution chosen by Xen), but this requires a dedicated page for each packet buffer, and thus modification to Linux to arrange for dedicated pages. Alternatively, copying the packets from the DD/OS to the clients may achieve nearly the same performance as remapping, while out-performing remapping for small packets. We use data copying to the client's address space, carried out by the hypervisor with a CPU-local temporary mapping, so that only a single copy is performed across the VM boundary (no intermediate buffering within the hypervisor) and without requiring the DD/OS to have virtual addresses for all target client memory. On x86, the copy activates the processor's cache-line transfer optimization, which disables memory bus and cache operations made redundant by the copy. Future commodity network hardware may automatically route packets to different receive queues [CIM97], based on LAN addresses, giving us zero-copy.

Once the proxy finishes transferring the packets to the clients, it must free the packets via a downcall to the DD/OS. The downcall should be safe to execute in any context. We use `dev_kfree_skb_any()`, which is safe to call from all contexts.

#### Device management

Linux currently satisfies most of these features — for example, it has a *notify* subsystem to dispatch events throughout the kernel for users of resources to discover changes within the resources.

### 3.6.3 Block

Linux has an in-kernel *block* interface for handling disks and similar devices. Our proxy module is a loadable Linux kernel module that interfaces with the block subsystem, and enables access to a large number of devices. We support only synchronous operations initiated by the client; there is no asynchronous data flow initiated by the DD/OS as networking offers.

Linux will reorder batched block requests, and thus later requests may finish before earlier requests; the reordered requests interfere with a producer-consumer ring, since a single pending request can impede finalization of the reordered operations that are later in the ring.

The Linux block interface has few assumptions about its consumers — for example, it doesn't expect the requests to originate via the `write()` system call.

The interface is flexible and supports many of the requirements of a proxy module.

## 3.7 L4 Microkernel

We use the L4Ka::Pistachio microkernel as the hypervisor, and as the environment for building new operating systems that reuse the Linux device drivers. The L4Ka::Pistachio microkernel is a derivative of Jochen Liedtke's L4 microkernel family [Lie93, Lie95b]. Liedtke's traditional L4 microkernels were written in assembler; Pistachio is mostly C++, yet it provides comparable performance. It offers few abstractions and mechanisms, with fewer than 13,000 lines of code that run privileged. We use its threading, address space management, and IPC to construct the DD/OS's virtual machine, and to provide communication between the translation modules and their clients.

In Chapter 4 we discuss how to build virtual machines with the L4 mechanisms; here we describe how the translation modules use the L4 mechanisms for driver reuse.

### 3.7.1 Client commands

The proxy listens for commands from the device clients, for controlling device sessions, such as opening, closing, and configuring; and for requests to process fresh operations queued in the producer-consumer rings. L4 supports only synchronous IPC [Lie95b, Lie96]; this requires that the proxy module execute helper L4 threads to wait for incoming IPC requests from the clients. The IPC messages arrive via OS bypass: the L4 microkernel directly activates an L4 thread within the proxy module, inside the address space of the Linux kernel, without regard to the current Linux kernel context. The client delivers the command as an L4 IPC, with a message in the register file. In OS bypass, Linux kernel state may be inconsistent, and thus the L4 thread must first activate a valid Linux context if it will interact with Linux; the valid Linux context is built by asking the VMM to deliver a virtual interrupt to Linux. The virtual interrupt delivery causes the Linux kernel to execute an interrupt handler, which then delivers an upcall to the proxy in interrupt context; the proxy then has the ability to execute commands in interrupt context, and if it needs a more capable kernel context, then it can queue a Linux work task.

For high-speed signaling, the device client can directly raise virtual interrupts within the DD/OS via the DD/OS's VMM. This skips the IPC normally used to contact the proxy's L4 thread.

### 3.7.2 Shared state

The proxy modules share memory with their device clients, including producer-consumer rings, and organizational information about the proxy (such as its L4 thread IDs). To establish this shared memory, the proxy module transfers page rights to the client via L4's memory sharing. L4 introduced the concept of recursive address space construction, managed at user-level [Lie95b, Lie96]. It permits two processes to coordinate page sharing, via an IPC rendezvous, with the page's owner giving access rights to the page's client. The page's owner can also rescind access rights, leading to page faults when the client next accesses the page. Our device clients send an IPC to the proxy to open a device session, and the IPC reply contains the shared memory rights.

### 3.7.3 Network receive

Inbound packets received from the network arrive in the DD/OS, and we copy them to the device client from the proxy, via L4 IPC. All other device activity (e.g., outbound network packets, and disk accesses) take place via DMA operations on client memory, and thus bypass the L4 microkernel.

L4 can transfer up to 16 *strings* of data in a single IPC; thus we queue incoming packets to each client, and copy them to the client in a batch either after all packets have arrived, or when we have accumulated 16 packets. We use a one-way IPC, which causes L4 to copy the packets to the client, and to immediately return to the proxy module; thus the proxy module distributes packets to all clients before activating any single client to start processing their packets.

The L4 microkernel handles the string copy via a rendezvous between two L4 threads; thus the client must have a waiting L4 thread, with sufficient buffers to receive the packets. Since the string copy is synchronous, the L4 microkernel provides a fast string copy without buffering the data within the L4 microkernel. It instead creates a temporary mapping within the L4 microkernel's address space to perform a direct copy. It creates the mapping by copying the two 4MB page-directory entries in the target's address space that swaddle the target string buffer into the page directory of the DD/OS's VM. The page directory is processor local, and thus the temporary-copy mapping is processor local; two processors can perform string-IPC copy in parallel to different targets; and expensive TLB shoot-downs are unnecessary.

If the packet arrival rate causes the proxy to exceed 16 packets per IPC, then the

packets would have to wait in the proxy until the client would have been scheduled, processed its previously received packets, and then put its L4 thread into an IPC wait for the next batch of packets. Instead, the client can execute multiple L4 threads to receive packets, with the proxy sending batches of 16 packets to each thread. The number of threads to choose depends on packet arrival rate, amount of processor time available to the client for processing its packets, and latency for scheduling the client. If latency is high, then the proxy might have to queue many packets, and in worst case, drop packets.

## 3.8 Related work

The Xen project reuses drivers in a manner very similar to our solution, and was developed in parallel. Xen reuses Linux drivers by running them as a part of a para-virtualized Linux (XenoLinux), within VMs on the Xen hypervisor [FHN+04b, FHN+04a]. Interestingly, since their small hypervisor lacks a native OS environment, their user-level drivers require the complete support infrastructure of an OS. We view virtual machines as a means for driver *reuse*, not as a construction principle for user-level drivers — for example, if using a microkernel, then use its application-like development environment for native construction of user-level drivers.

Xen's architecture is similar to ours: it loads proxy modules into the DD/OS to translate between client requests and Linux's internal device operations. Xen's typical performance mode uses memory partitioning rather than shadow page tables. We have assumed shadow page tables in our design, to reclaim unused pages from the DD/OS.

We have partially published the results of this thesis in ref. [LUSG04], in particular, our techniques for handling inbound network packets, and direct DMA for packet transmit. Later, another team saw that Xen's data remapping causes unnecessary overhead, and applied techniques similar to ours for reducing the overhead: they copy network packets on the receive path, and use direct DMA on client memory for packet transmit [MCZ06].

Another L4 project briefly studied reusing Linux drivers [HLM+03], by running the reused Linux drivers in a para-virtualized L4Linux. The drivers connected with the remaining system via translation logic running as a Linux application. Their results showed much higher overhead than our own results.

Microsoft Research also studied running drivers within virtual machines, in

the VEXE'DD project [ERW05]. They prototyped using the Virtual PC product, and attempted to isolate drivers in Windows VMs. Their primary concern was to safely support legacy extensions where ad hoc use of undocumented interfaces was rampant. For added protection in running unreliable extensions, they used a safe mediator to monitor the communication between the reused driver and its clients (similarly to Nooks [SBL03], and Härtig et al. [HLM+03]), while we do not view this as intrinsic to driver reuse, but rather an orthogonal and expensive feature.

## 3.9 Summary

We reuse device drivers by executing them with their original operating systems in virtual machines. We call the device driver operating system the device-driver OS (DD/OS). To connect the drivers to the surrounding system, we install translation modules into the DD/OS that communicate across the VM boundary with clients of the devices, and which convert client requests into internal DD/OS operations. This approach provides scalable engineering effort, because the implementation of a single translation module enables reuse of a variety of device drivers; it improves system dependability by driver fault isolation; and it permits driver reuse within new operating systems that are fundamentally incompatible with the DD/OS architecture. As a demonstration of the technique, we described implementation strategies for reusing Linux device drivers on the L4 microkernel.

# Chapter 4

# Pre-virtualization

Our technique to build virtual machines, called *pre-virtualization*, attempts to combine the performance of para-virtualization with the modularity of traditional virtualization, while making virtualization easier for developers to implement. See Figure 4.1.

The popular alternative to traditional virtualization (particularly on x86) is para-virtualization, for it can improve performance, and it converts the style of engineering work from VMM construction to co-design of the guest kernel and hypervisor. The co-design costs modularity: it ties a guest operating system and a hypervisor together, which restricts the system architecture — for example, Linux adapted to the Xen API is unable to run on alternative hypervisors such as VMware, Linux itself, or a security kernel such as EROS. Furthermore, the lock-in obstructs evolution of its own para-virtualization interface — virtual machines provide the vital ability to run obsoleted operating systems alongside new operating systems, but para-virtualization often lacks this feature, requiring all concurrent instances to be the hypervisor's supported version. Even general purpose operating systems have weaker restrictions for their applications. This lock-in forfeits the modularity of virtualization. Modularity is an intrinsic feature of traditional virtualization, helping to add *layered enhancements* to operating systems, especially when enhanced by people outside the operating system's development community (e.g., Linux server consolidation provided by VMware).

Virtualization and its modularity solve many systems problems, and when it is combined with the performance of para-virtualization, it becomes even more compelling. We show how to achieve both *together*. We still modify the guest operating system, but according to a set of design principles that avoids lock-in, which we

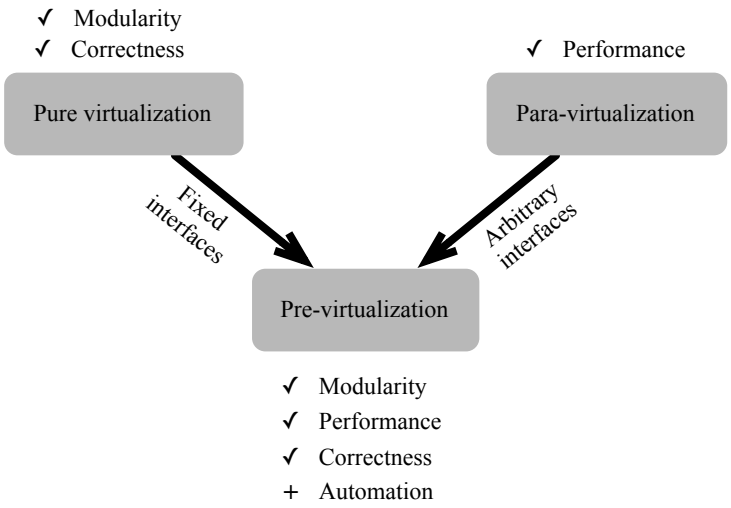



Figure 4.1: Pre-virtualization combines many of the best features of pure virtualization with para-virtualization, while adding automation.

call *soft layering*. Additionally, our approach is highly automated and thus reduces the implementation and maintenance burden of para-virtualization, which supports this thesis's goal of rapid development of a driver reuse environment.

This chapter describes the principles of soft layering, and how we apply them to virtual machines, which we call *pre-virtualization*. We present implementations for two families of x86 hypervisors that have very different APIs, the L4 microkernel and the Xen hypervisor, to demonstrate the versatility of pre-virtualization. For the guest kernel, we used several versions of Linux 2.6 and 2.4, also to demonstrate the versatility of pre-virtualization.

## 4.1 Soft layering

Our goal is a technique to virtualize operating systems for high performance execution on arbitrary hypervisors, while preserving performance when booted on bare hardware (see Figure 4.2). Thus we must focus on backwards compatibility with the native hardware interface, the interface to which the guest operating systems have already been written. The principle of soft layering [Coo83] was developed for such a scenario (originally in network protocol layering). Soft layering requires a strong layer interface, but allows it to be softened in a non-exclusionary manner at runtime. The runtime softening is supported by infrastructure added to the guest kernel source code, but the softening is optional, and variable (partial or

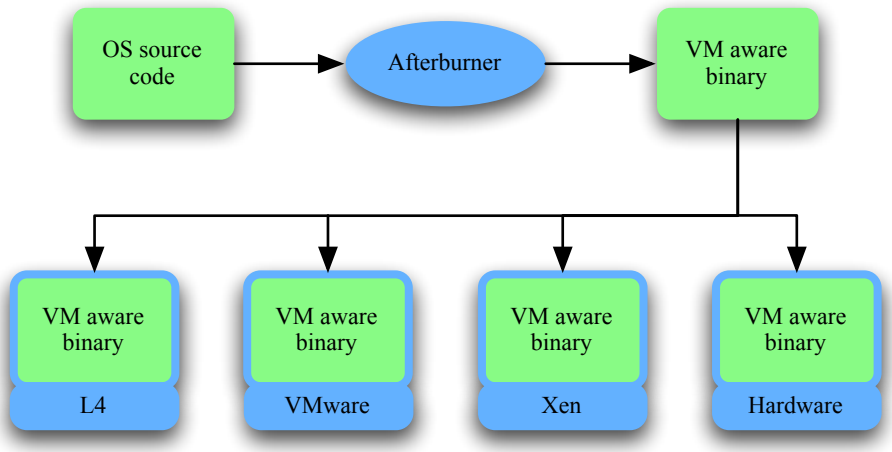



Figure 4.2: Our approach automatically transforms the OS source code to produce a binary that supports soft layering. The binary can boot on a variety of hypervisors, or on bare hardware.

full optimizations are possible, with specialized extensions too).

> The criteria of soft layering when applied to virtualization are:
>
> 1. it must be possible to degrade to a neutral interface, by ignoring the co-design enhancements (thus permitting execution on native hardware and hypervisors that lack support for the soft layering);
> 2. the interface must flexibly adapt at runtime to the algorithms that competitors may provide (thus supporting arbitrary hypervisor interfaces without pre-arrangement).

The *soft* describes the scope and type of modifications that we apply to the source code of the guest OS: the modifications remain close to the original structure of the guest OS, and it uses the neutral platform interface as the default interface (i.e., the OS will execute directly on raw hardware, and the enhancements require activation in a VM). Soft layering forbids changes to the guest OS that would interfere with correct execution on the neutral platform interface, it discour-

ages changes that substantially favor one hypervisor over others, and it discourages changes that penalize the performance of the neutral platform interface.

The decision to activate a soft layer happens at runtime when the hypervisor and guest kernel are joined together. The hypervisor inspects the descriptor of the soft layer that accompanies the guest OS, and determines their compatibility. If incompatible, the hypervisor can abort loading, or activate a subset of the soft layer (to ignore extensions unequally implemented, or rendered unnecessary when using VM hardware acceleration).

To achieve our performance goals we must increase transparency to the hypervisor's internal abstractions (review Section 2.3.1), but without violating the second criterion of soft layering – that the interface must flexibly adapt to the algorithms provided by competitors. Via the virtualization logic that we inject into the guest kernel's protection domain, we map the operating system's use of the platform interface to the hypervisor's efficient primitives. This achieves the same effect as para-virtualization — the guest kernel operates with increased transparency — but the approach to increasing transparency differs. Some of para-virtualization's structural changes fall outside the scope of the platform interface, thus requiring the soft layer to extend beyond the platform interface too. Yet some of co-design's traditional structural changes, such as high-performance network and disk drivers, are unnecessary in our approach, since they can be handled by mapping the device register accesses of standard device drivers to efficient hypervisor abstractions.

## 4.2 Architecture

Para-virtualization has three categories of interfaces between the hypervisor and guest OS that we've identified, and we offer soft-layer alternatives for each:

*Instruction-level* modifications apply at the interface between the virtual machine and the guest kernel, without extending their reach too far into the guest kernel's code. They can extend the underlying architecture (e.g., Denali [WSG02] introduced an idle with timeout instruction), or replace virtualization-sensitive instructions with optimized downcalls.

*Structural* modifications add efficient mappings from the guest kernel's high-level abstractions to the hypervisor's interfaces, thus bypassing the guest kernel's original code that interfaces with the platform. This is particularly useful for providing high-performance devices, and most virtualization projects offer these mappings for network and disk. These modifications are very intrusive to the guest

kernel, and require specific knowledge of the guest kernel's internal abstractions. The structural modifications make certain relationships explicit, and thus simplify the virtualization — for example, many projects adjust the virtual address space of the guest kernel to permit coexistence of a hypervisor, guest kernel, and guest application within a single address space.

*Behavioral* modifications change the algorithms of the guest OS, or introduce parameters to the algorithms, which improve performance when running in the virtualization environment. These modifications focus on the guest kernel, and do not rely on specialized interfaces in the hypervisor, and thus work on raw hardware too. Examples are: avoiding an address space switch when entering the idle loop [SVL01], reducing the timer frequency (and thus the frequency of house keeping work), and isolating virtualization-sensitive memory objects (e.g., x86's descriptor table) on dedicated pages that the hypervisor can write-protect without the performance penalty of false sharing.

Our soft layering approach addresses para-virtualization's instruction-level and structural enhancements with different solutions. Para-virtualization's behavioral modifications are a natural form of soft layering: they avoid interference with OS and hypervisor neutrality, and may achieve self activation — for example, the kernel can detect that certain operations require far more cycles to execute, and thus it changes behavior to match the more expensive operations [ZBG+05]. Behavioral modifications may benefit from visible interfaces to the guest kernel, which we classify as structural enhancements.

### 4.2.1 Instruction level

The performance of virtual machines relies on using bare-metal execution for the frequently executed innocuous instructions, while introducing expensive emulation only for the infrequently executed virtualization-sensitive instructions [PG73]. The emulation traditionally is activated upon traps on the virtualization-sensitive instructions, which is an expensive approach on today's super-pipelined processors. Para-virtualization boosts performance by eliminating the traps (and potentially only for the most frequently executed privileged instructions [MC04]). Yet if the modifications are restricted to hypercall substitution for each instruction, the savings may be small, since hypercalls can have noticeable costs. Thus several para-virtualization projects reduce the number of hypercalls by mapping the low-level instruction sequences into higher-level abstractions via source code modifications. For our approach to have performance comparable to para-virtualization, we must

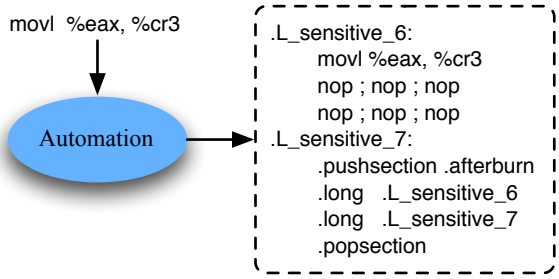



Figure 4.3: Example of assembler preparation for a sensitive x86 instruction — it adds scratch space via `nop` instructions, and adds assembler directives to record the location of the instruction.

also map low-level instructions into higher-level abstractions, but while obeying the criteria of soft layering.

To satisfy the criterion of soft layering that the guest OS should execute directly on raw hardware, we leave the virtualization-sensitive instructions in their original locations. Instead, we enable a hypervisor to quickly locate and rewrite the virtualization-sensitive instructions at runtime. The hypervisor is thus able to replace each individual instruction with optimized emulation logic.

To provide the hypervisor with scratch space for writing the optimized emulation logic, we pad each virtualization-sensitive instruction with a sequence of no-op instructions[1]. The scratch space becomes a permanent addition to the binary, and is even executed when the binary runs on native hardware, which is why we fill the scratch space with no-op instructions. We allocate enough scratch space to minimally hold an instruction for calling out to more complicated emulation logic. See Figure 4.3.

To help the hypervisor quickly locate the sensitive instructions at runtime, we annotate their locations. The runtime instruction rewriter decodes the original instructions to determine intent and the locations of parameters, and writes higher-performance alternatives over the scratch space provided by the no-op padding.

### Complex emulation

In several cases the size of the emulation logic exceeds the size of the scratch space, particularly if the logic for mapping low-level instructions to the higher-level abstractions of the hypervisor is complicated. For these cases, we inject a mapping

[1]Instructions with an architecturally defined relationship to their succeeding instruction must be preceded by their no-op padding, e.g., x86's `sti` instruction.

module into the address space of the guest kernel, in a region reserved from the guest kernel. The code rewriter emits instructions that call this extended mapping module. Thus the extended logic is quickly activated (since it avoids a protection domain crossing), but it is also vulnerable to misbehavior of the guest kernel, and so the emulation logic must execute at the same privilege level as the guest kernel. We term the mapping module the *virtualization-assist module* (VAM).

The virtualization-assist module provides a virtual CPU and device models. The rewritten instructions directly access the virtualization-assist module via function calls or memory references. The virtualization-assist module defers interaction with the hypervisor by batching state changes, thus imitating the behavior of para-virtualization. Since the virtualization-assist module runs at the guest kernel's privilege level, it must execute hypercalls for emulation logic that otherwise could subvert hypervisor security. The hypercall permits the hypervisor to authorize such side effects (for example, memory mappings).

The mapping module is specific to the hypervisor, but neutral to the guest OS since its exported interface is that of the raw hardware platform. Thus a hypervisor need implement only a single mapping module for use by any conformant guest kernel. Additionally, since the binding to the mapping module takes place at runtime, the guest kernel can execute on a variety of incompatible hypervisors, and a running guest kernel can migrate between incompatible hypervisors (which is especially useful across hypervisor upgrades). See Figure 4.4 for a comparison of soft-layering to pure virtualization and para-virtualization. See Figure 4.5 for an illustrated timeline of the guest OS startup.

#### Automation

For the easy use of soft layering, we apply the instruction-level changes automatically at the assembler stage [ES03]. Thus we avoid many manual changes to the guest OS's source code, which reduces the man-power cost of para-virtualization's high performance. This automatic step permits us to package most of the soft layer as a conceptual module independent of the guest kernel — they are combined at compile time (see Figure 4.2). This source code modularity is particularly useful for the fast-paced development of open source kernels, where developers edit the latest edition of a kernel, and the users (via the distributions) grab any edition for production use. By automating, one can re-apply the latest version of the soft layer interface (e.g., to an obsoleted source base).

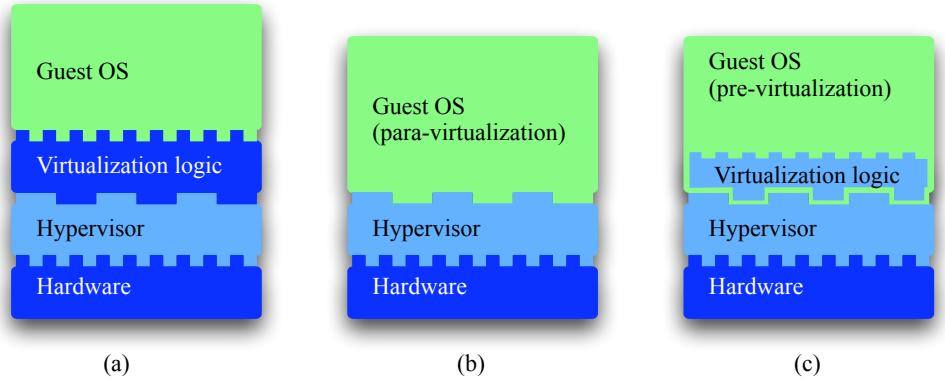



Figure 4.4: A comparison of different virtualization approaches, and their layered interfaces. Traditional emulation is used in *a*. Para-virtualization is used in *b*. And soft layering is demonstrated in *c*.

### Memory instructions

A variety of memory objects are also virtualization-sensitive, such as memory-mapped device registers and x86's page tables. When instructions access these objects, we reclassify them from innocuous to virtualization-sensitive instructions. We apply the soft-layering technique to these instructions, but only in contexts where they access these memory objects — innocuous uses of the instructions remain untouched. We try to automatically classify memory instructions as innocuous or virtualization sensitive at the compiler stage, by parsing the guest kernel's C code. To distinguish between memory types, our automated tool uses data-type analysis, and overloads only those memory-access instructions for sensitive data types. In some cases the guest code may abstract a particular memory access with an access function, and we can use our parser to redefine the memory-access instructions of such functions. A developer must identify the data types and access functions of senstive memory, and report them to the automated tool; thus we have only partial automation. If the guest kernel makes insufficient use of data types and access functions, and thus we are unable to automatically disambiguate the innocuous from the virtualization-sensitive with high accuracy, then we manually apply the instruction no-op padding and annotations via C language macros or compiler hints [BS04]. It would be possible to disambiguate at run time (as does traditional virtualization), but that adds too much overhead.

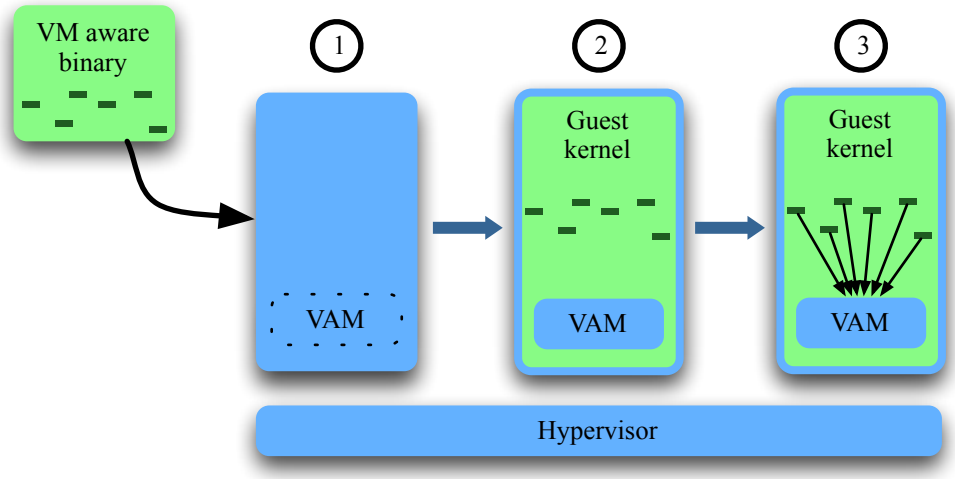



Figure 4.5: (1) The virtualization-assist module starts with a virgin VM, (2) unpacks the guest kernel into the VM, (3) rewrites the sensitive instructions to invoke the virtualization-assist module, and then boots the guest OS.

### 4.2.2 Structural

While the instruction-level interface conforms to a standard that the processor manufacturer defines, the structural changes of para-virtualization are open-ended and have no standard. This lack of restraint is one of para-virtualization's attractions, for it enables custom solutions that reflect the expertise of the developers for their problem domains.

Structural changes can comply with the criteria of soft layering — soft layering only demands that the structural changes obey an interface convention for identifying and enabling the changes. When the structural changes conform to soft layering, they can support particular hypervisors or guest kernels while all others can ignore the structural changes (or incrementally add support). A possible interface scheme is to use function-call overloads, each identified by a universal unique identifier [RPC97].

Some structural changes permanently alter the design of the kernel, such as changes in the guest kernel's address space layout, which operate even when executing on raw hardware. Other changes replace subsystems of the guest kernel, and are suitable for function-call overloading in the soft layer (i.e., rewriting the guest kernel to invoke an optimized function within the mapping module). Another class of changes adds functionality to the guest kernel where it did not exist before, as runtime installable kernel modules, such as memory ballooning [Wal02] (which

enables cooperative resource multiplexing with the hypervisor).

The structural modifications require conventions and standards for the important overload points, and documentation of the side effects, to provide a commons that independent hypervisor and kernel developers may use. The danger is divergence since there is no well-defined interface for guidance, in contrast to the instruction-level soft layering.

Multiple approaches to function-call overloading are possible, such as rewriting all invocation points (e.g., dynamic linking [DD67]), using an indirection table (e.g., COM [Bro95, Rog97]), or installing kernel-specific loadable modules. We currently use a combination of an indirection table and loadable modules, where the loadable modules have the ability to access code and data in the virtualization-assist module.

### 4.2.3 The virtualization-assist module

The virtualization-assist module includes comprehensive virtualization code for mapping the activity of the guest kernel into the high-level abstractions of the hypervisor. This is especially useful for repurposing general-purpose kernels as hypervisors (such as Linux on Linux, or Linux on the L4 microkernel). The virtualization-assist module not only performs the downcalls to the hypervisor, it also intercepts upcalls to the guest kernel from the hypervisor (e.g., for fault and interrupt emulation). Since these operations are part of the platform interface, they are agnostic to the guest kernel.

Some of the downcalls have no direct mappings to the hypervisor's interface, e.g., updates to the x86 TSS or PCI BAR configurations. Once the VAM intercepts these downcalls, it uses traditional virtualization techniques to map the operations to the interfaces of the hypervisor. The goal is to let the OS use the raw hardware interface (just as in full virtualization).

#### Indivisible instructions

The virtualization-assist module installs itself into the guest OS at load time by rewriting the guest's virtualization-sensitive instructions and by hooking its function overloads (see Figure 4.5). Where we replace the original, indivisible instructions of the guest kernel with emulation sequences of many instructions, we must respect the indivisibility of the original instructions in regards to faults and interrupts. The guest kernel could malfunction if exposed to virtualization-assist

module state in an interrupt frame, and so we also treat function overloads as indivisible operations. In either case the emulation could be long running, or with unpredictable latency, which is the nature of virtualization.[2] To avoid reentrance, we structure the virtualization-assist module as an event processor: the guest kernel requests a service, and the virtualization-assist module returns to the guest kernel only after completing the service, or it may roll back to handle a mid-flight interruption. The guest kernel is unaware of the emulation code's activity, just as in normal thread switching a thread is unaware of its preemption. If the guest kernel has dependencies on real time, then we assume that the kernel authors already over provisioned to handle the nondeterminism of real hardware and application workloads.

#### Code expansion

The guest kernel may execute several of the virtualization-sensitive instructions frequently, e.g., Linux often toggles interrupt delivery. The performance of the virtualization-assist module for emulating these frequently-executed instructions depends on its algorithms and how it chooses between instruction expansion and hypercalls; for the frequent instructions we want to avoid both. For example, using a hypercall to toggle interrupt delivery (so that the hypervisor avoids delivering virtual interrupts) would add too much overhead. Instead we rely on a general algorithm: we use a virtual CPU within the virtualization-assist module that models the interrupt status, and we replace the interrupt toggling instructions with one or two memory instructions that update the virtual CPU's status flags; the emulation overhead is thus eliminated. The hypervisor will deliver virtual interrupts despite the status of the virtual CPU's interrupt status flag, but since the virtualization-assist module intercepts the hypervisor's upcalls, it enforces the virtual interrupt status flag, and can queue the pending interrupt for later delivery when the guest kernel reactivates its interrupts via the virtual CPU.

### 4.2.4 Device emulation

Soft layering helps virtualize standard devices with high performance; all other virtualization approaches depend on special device drivers for performance, which

[2]The guest kernel is a sequential process concerned about forward progress but not the rate of forward progress [Dij68].

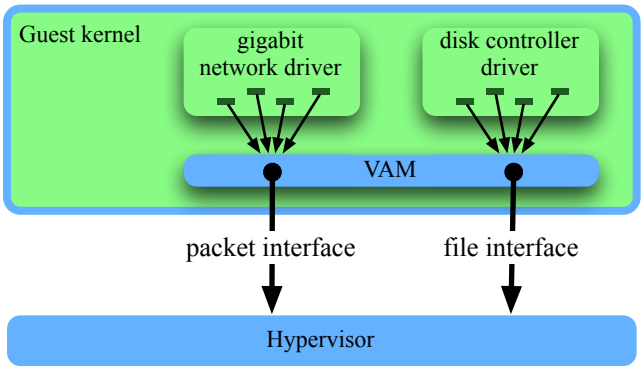



Figure 4.6: High-performance device access via pre-virtualization: the virtualization-assist module upmaps the low-level device-register accesses of the guest OS to the device interfaces of the hypervisor.

inherently tie the guest OS to a particular hypervisor. We follow the neutral platform API to provide the modularity of strict layering.

Device drivers issue frequent device register accesses, notorious for performance bottlenecks when emulated via traps [SVL01]. We convert these device register accesses into efficient downcalls — we use instruction-level soft layering to rewrite the code to invoke the virtualization-assist module. The virtualization-assist module models the device, and batches state changes to minimize interaction with the hypervisor. See Figure 4.6 for a diagram.

Networking throughput is particularly sensitive to batching: if the batching adds too much latency to transmitted packets, then throughput deteriorates; if the virtualization-assist module transmits the packets prematurely, then throughput deteriorates due to hypercall overhead. The batching problem plagues the specialized device drivers of other approaches too, since many kernels hide the high-level batching information from their device drivers. The speeds of gigabit networking require more comprehensive batching than for 100Mbit networking [SVL01]. In past work with Linux, we used a callback executed immediately after the networking subsystem, which provided good performance [LUSG04]; for driver emulation we infer this information from the low-level activity of the guest kernel, and initiate packet transmission when the guest kernel returns from interrupt, or returns to user, which are both points in time when the kernel is switching between subsystems and has thus completed packet processing.

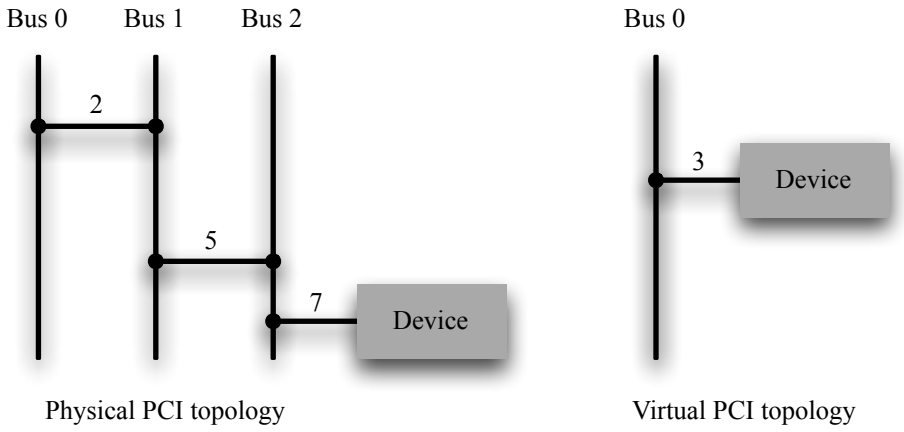



Figure 4.7: When configured for selective device pass through, the virtualization-assist module presents a virtual PCI bus to the guest OS, which forwards PCI configuration requests to the actual device.

### 4.2.5 Device pass through

To support device driver reuse, the virtualization-assist module permits the guest kernel's device drivers to register for real device-interrupt notifications, to create memory mappings for device registers, and to access the I/O port space. Yet it never grants to the guest kernel full control of the platform, to prevent interference between the hypervisor and DD/OS, and thus must virtualize some aspects of the platform. In some cases, these platform devices will be fully virtualized, and in others, the devices will have partial pass through.

When several DD/OSes work together to control the devices, they will often have to share a bus. In this case, the virtualization-assist module will also partially virtualize the bus. Figure 4.7 shows an example of how a DD/OS controls a device connected to a virtualized PCI bus — the DD/OS permits the driver to access and configure the PCI control registers of the device, but the virtualization-assist module hides the true bus topology by presenting a virtual bus topology, and thus the DD/OS is unable to see other devices.

## 4.3 Guest preparation

Soft layering involves modifications to the guest kernel, although we apply most in an automated manner at the compilation stage. The modifications fall under three categories: sensitive instructions, sensitive memory instructions, and structural.

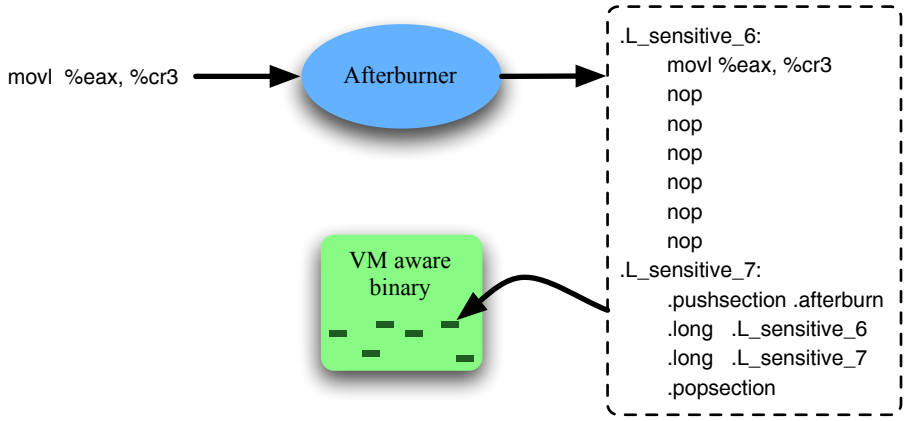



Figure 4.8: The Afterburner automatically transforms sensitive assembler instructions, inserts no-op padding, and records the location of the instruction (in the `.afterburn` ELF section in this example).

### 4.3.1 Sensitive instructions

To add soft layering for virtualization-sensitive instructions to a kernel, we parse and transform the assembler code (whether compiler generated or hand written). We wrote an assembler parser and transformer using ANTLR [PQ95]; it builds an abstract syntax tree (AST) from the assembler, walks and transforms the tree, and then emits new assembler code.

The most basic transformation adds no-op padding around the virtualization-sensitive instructions, while recording within an executable section the start and end addresses of the instruction and its no-op window (see Figure 4.8). The no-op instructions stretch basic blocks, but since at this stage basic block boundaries are symbolic, the stretching is transparent. x86 has a special case where the kernel sometimes restarts an in-kernel system call by decrementing the return address by two bytes; this can be handled by careful emulation in the VAM.

More sophisticated annotations are possible, such as recording register data flow based on the basic block information integrated into the AST.

### 4.3.2 Sensitive memory instructions

An automated solution for pre-virtualizing the memory instructions must disambiguate the sensitive from the innocuous. We implemented a data-type analysis engine that processes the guest kernel's high-level source to determine the sensitive memory operations based on data type. For example, Linux accesses a page table

entry (PTE) via a `pte_t *` data type. Our implementation uses a gcc-compatible C parser written in ANTLR, and redefines the assignment operator based on data type (similar to C++ operator overloading). Our modifications (1) force the operation to use an easily decodable memory instruction, and (2) add the no-op padding around the instruction. The automated parsing tool involves three phases:

- The developer identifies the sensitive memory types, such as those representing page-table entries, in the kernel code. The developer also identifies access functions for sensitive memory types that should be overloaded. The developer provides this information to the tool. This requires the developer to competently read the code and to identify the sensitive memory operations.

- The developer configures the kernel's build infrastructure to invoke the tool on each source file after running the C preprocessor, but before compiling the code. The tool processes each source file, and identifies where the code accesses sensitive memory. It distinguishes between function local data (e.g., items in the register file or the stack), and system memory. The tool assumes that sensitive memory only occupies the system memory.

- The tool transforms the C code. If overloading a memory-access *instruction*, then it adds no-op padding and annotations, and forces use of an easily decodable memory instruction. If overloading a *function*, then it converts the function call into a indirect function call that the virtualization-assist module can overload, and adds annotations. The tool emits the transformed code to the compiler, which outputs assembler, which we send as input to the pre-virtualization assembler stage.

Refer to ref. [Yan07] for more details regarding the C transformation tool.

We integrate Linux page table and page directory accesses into the soft layer. Although manually hooking page table writes in Linux is fairly simple due to Linux's abstractions, page-table reads are far more difficult to do by hand, but are easily handled by the automated C parser which detects the page-table reads via data-type analysis.

To virtualize device drivers that access memory-mapped device registers, we apply instruction-level soft layering to their access instructions. Each driver requires a different set of annotations to permit the virtualization-assist module to distinguish one driver from another at runtime. Since Linux already abstracts accesses to device registers with read and write functions, we can easily redefine

them, either using the automated C parser, or by manually adding a prolog to each driver that overloads the read and write functions via the C preprocessor.

### 4.3.3 Structural

Our primary structural modification allocates a hole within the virtual address space of Linux for the virtualization-assist module and hypervisor. The hole's size is currently a compile-time constant of the Linux kernel. If the hole is very large, e.g., for running Linux on Linux, then we relink the Linux kernel to a lower address to provide sufficient room for the hole.

To support pass-through device access, for the driver reuse of this thesis, we added a function overload to perform DMA address translation. We did not pursue automatic application of these overloads, and instead manually added them to the source code, because Linux lacks consistent and comprehensive abstractions that both handle forward and reverse translation, and which enforce invariants regarding contiguous regions of memory that DMA operations may expect. Note that normal virtualization has no need for DMA overloading — driver reuse requires the DMA overloading.

To support the L4 microkernel with decent performance, we added other function-call overloads. In the normal case these overloads use the default Linux implementation; when running on the L4 microkernel, we overload the function calls to invoke replacement functions within the virtualization-assist module. These overloads permit us to control how Linux accesses user memory from the kernel's address space, and permit us to efficiently map Linux threads to L4 threads. The overloads implement techniques described in prior work for running Linux on L4 [HHL+97].

## 4.4 Runtime environment

We divide the virtualization-assist module into two parts: a front-end that emulates the platform interface, and a back-end that interfaces with the hypervisor. The rewritten sensitive instructions of the guest kernel interact with the front-end, and their side effects propagate to the back-end, and eventually to the hypervisor. Up-calls from the hypervisor (e.g., interrupt notifications) interact with the back-end, and propagate to the front-end. See Figure 4.9 for a diagram.

Although we try to match the performance of para-virtualization, the virtualization-assist module is more restricted by the constraints of soft layering. Thus the

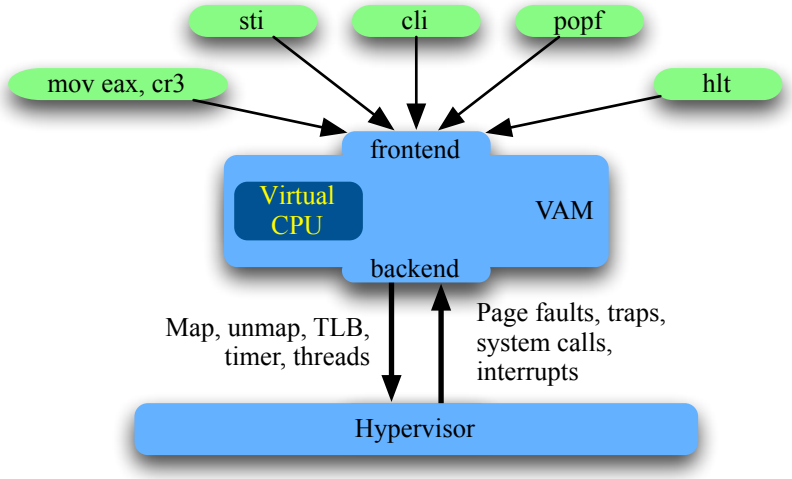



Figure 4.9: The virtualization-assist module frontend interfaces with the guest kernel, and the backend interfaces with the hypervisor.

virtualization-assist module implements heuristics to attempt to match the performance of traditional para-virtualization modifications. This may not always be possible, but so far we have been successful.

### 4.4.1 Indivisible instructions

We must be careful to avoid corrupting or confusing the guest kernel, since we execute a substantial portion of the emulation logic within the address space of the guest kernel, and thus open the opportunity for careless emulation. The guest kernel is unaware of our emulation logic, and is expecting the rewritten instructions to follow the semantics of the original instructions. This includes side effects to the register file.

Immediately before entering the emulation logic, the register file contains guest kernel state. We call this the *boundary* state. After entering the emulation logic, the virtualization-assist module becomes owner of the register file, and installs its own state. The virtualization-assist module must thus preserve the boundary state, so that it can restore the state when exiting the emulation logic.

The virtualization-assist module may exit the emulation logic via two paths:

1. The normal case is a return to the instruction following the currently emulated instruction. The return is similar to a function call return, although more conservative. Normal functions automatically preserve and restore part

of the register file, and since we use the compiler to generate much of the emulation logic, we rely on the compiler to preserve and restore part of the register file. For the remaining registers, we manually preserve and restore them at the virtualization-assist module boundary.

2. An interrupt arrives, which requires the virtualization-assist module to branch to an interrupt handler. Eventually, the guest kernel's interrupt handler will return to the interrupted code. In many cases the guest kernel ignores the identity of the interrupted code, and will reactivate it without question. Yet a variety of kernel algorithms require the guest kernel to inspect and identify the interrupted code (such as for instruction sampling, or exception handling), in which case, if the interrupt frame points at emulation logic, the guest kernel could malfunction or become confused. Thus it is important that the register file and interrupt frame contain the proper boundary state upon entering the interrupt handler. This requires that we carefully handle asynchronous events that arrive during execution of emulation logic.

The virtualization-assist module handles three interrupt conditions that require attention to the boundary CPU state: (1) when the guest kernel executes an instruction that enables interrupts, and interrupts are already pending; (2) interrupts arrive during instruction emulation; and (3) interrupts arrive during un-interruptible hypercalls and must be detected after hypercall completion.

For case 1, before entering the front-end we allocate a redirection frame on the stack in preparation for jumping to an interrupt handler when exiting the virtualization-assist module. If no interrupt is pending (the common case), the virtualization-assist module discards the redirect frame, and then returns to the guest kernel with full boundary state restored. If an interrupt was pending, then the redirect frame causes control transfer to the interrupt handler, leaves behind a proper interrupt frame, and preserves the boundary CPU state. This is particularly useful for x86's `iret` instruction (return from interrupt), because it cleanly emulates the hardware behavior when transitioning from kernel to user: interrupts are delivered as if they interrupted the user context, not the kernel context.

Case 2 is important for `iret` and idle emulation, since these both involve race conditions in checking for already pending interrupts. For interrupt arrival during `iret`, we roll-back and restart the front-end, so that it follows the route for case 1. For interrupt arrival during idle, we roll forward to abort the idle hypercall, and then deliver the interrupt using the redirection frame.

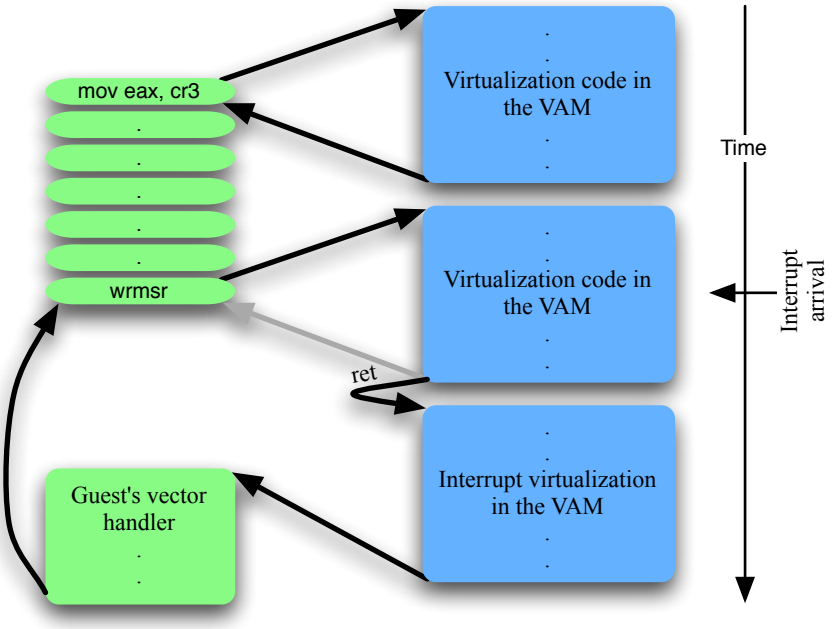



Figure 4.10: When virtualized, an emulation sequence replaces a single instruction. To respect the semantics of interrupt delivery on the original hardware, we avoid delivering interrupts in the midst of an emulation sequence. Here, the virtualization-assist module reschedules itself to execute interrupt emulation after the `wrmsr` sequence, thus entering the guest's vector handler with the proper boundary CPU state.

Case 3 requires manual inspection of pending interrupts, since the hypervisor avoids delivering an upcall. If an interrupt is pending, we alter the virtualization-assist module's boundary return address to enter an interrupt dispatcher, then unwind the function call stack to restore the guest kernel's boundary state, and then enter the interrupt dispatch emulation. See Figure 4.10.

### 4.4.2 Instruction rewriting

When loading the guest kernel, the virtualization-assist module locates the soft-layer annotations within the guest kernel's executable. Via the annotations, the virtualization-assist module locates the sensitive instructions, decodes the instructions to determine their intentions and register use, and then generates optimized replacement code. The replacement code either invokes the virtualization-assist module via a function call, or updates the virtual CPU via a memory operation.

Minimizing the instruction expansion is crucial for the frequently executed instructions. For x86, the critical instructions are those that manipulate segment registers and toggle interrupt delivery. The segment register operations are simple to inline within the guest kernel's instruction stream, by emitting code that directly accesses the virtual CPU of the virtualization-assist module[3] for flat segments). For toggling interrupts, we use the same strategy, which causes us to deviate from the hardware interrupt delivery behavior; the hardware automatically delivers pending interrupts, but its emulation would cause unjustifiable code expansion (we have found that the common case has no pending interrupts because the guest executes these instructions at a higher rate than the interrupt arrival rate). Instead we use a heuristic to deliver pending interrupts at a later time:[4] when the kernel enters the idle loop, transitions to user mode, returns from interrupt, or completes a long-running hypercall. Our heuristic may increase interrupt latency, but running within a VM environment already increases the latency due to arbitrary preemption of the VM.

### 4.4.3 Xen/x86 hypervisor back-end

The x86 Xen API resembles the hardware API, even using the hardware `iret` instruction to transition from guest kernel to guest user without indirecting through the hypervisor. Still, the virtualization-assist module intercepts all privileged instructions and upcalls to enforce the integrity of the virtualization. We configure Xen to deliver interrupts, events, and x86 traps to the virtualization-assist module, which updates the virtual CPU state machine and then transitions to the guest kernel's handlers. The virtualization-assist module intercepts transitions to user-mode (the `iret` instruction), updates the virtual CPU, and then completes the transition. We optimistically assume a system call for each kernel entry, and thus avoid virtualization overhead on the system call path, permitting direct activation of the guest kernel's system call handler — on `iret`, we preconfigure the vCPU state for a system call, and permit Xen's system-call upcall to bypass the virtualization-assist module; all other upcalls activate the virtualization-assist module, which cancels the system-call setup, and reconfigures the vCPU for the actual upcall (note that the configuration is minor, but by permitting bypass of the virtualization-assist module for system calls, we avoid TLB, cache, and branch misses in the

[3] (

[4] We detect special cases, such as `sti;nop;cli` (which enables interrupts for a single cycle), and rewrite them for synchronous delivery.

virtualization-assist module, which are noticeable savings under in benchmarks — our pre-virtualized system calls are faster than Xen's para-virtualized system-call handling in microbenchmarks).

Xen's API for constructing page mappings uses the guest OS's page tables as the actual x86 hardware page tables. The virtualization-assist module virtualizes these hardware page tables for the guest OS, and thus intercepts accesses to the page tables. This is the most complicated aspect of the API, because Xen prohibits writable mappings to the page tables; the virtualization-assist module tracks the guest's page usage, and transparently write-protects mappings to page tables. Xen 3 changed this part of the API from Xen 2, yet our virtualization-assist module permits our Linux binaries to execute on both Xen 2 and Xen 3.

The virtualization-assist module handles Xen's OS bootstrap (what Xen calls *start of day*), registering its own handlers where a para-virtualized OS would normally register handlers, and configuring the virtual memory system to resemble the environment expected by an OS on raw hardware (although bypassing x86's 16-bit mode and starting directly in 32-bit mode). When the guest OS starts initializing its virtual devices, the virtualization-assist module updates the original start-of-day parameters to match the guest OS's configuration (e.g., the timer frequency).

### 4.4.4 L4 microkernel back-end

The L4 API is a set of portable microkernel abstractions, and is thus high-level. The API is very different from Xen's x86-specific API, yet soft layering supports both by mapping the neutral platform API to the hypervisor API, and we use the same x86 front-end for both.

For performance reasons, we associate one L4 address space with each guest address space, providing shadow page tables [PPTH72] (which cache the guest's mappings within the L4 kernel with TLB semantics). Thus L4 provides its own page tables for the hardware page tables, and the virtualization-assist module synchronizes the translations of the guest's page tables with those of the L4 page tables. The virtualization-assist module can update the shadow page tables optimistically or lazily.

We virtualize privileges by mapping each privilege to a separate L4 address space. Thus although a single guest page table describes both the Linux kernel and application address spaces, we treat them as two separate logical spaces, and map them to different L4 address spaces. This could put a noticeable load on system resources by adding two sets of shadow page tables for each guest application, but

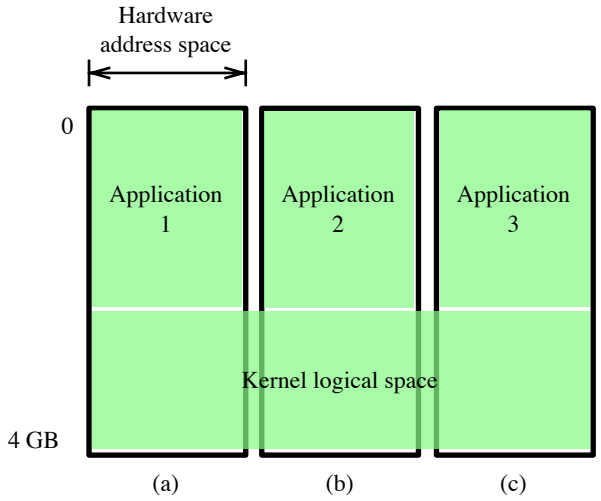



Figure 4.11: Linux creates several *logical* address spaces (the applications 1, 2, 3, and the kernel), implemented by *hardware* address spaces *a*, *b*, and *c* (using the page tables). To sustain a logical kernel space that spans multiple hardware spaces, Linux must maintain and synchronize consistent mappings across the hardware spaces. This also provides Linux an opportunity to lock these translations in the TLB, which is a critical x86 performance optimization to reduce TLB misses.

we can make an optimization: Linux maintains a fairly consistent logical address space for the kernel across all guest page tables, and thus we can represent the kernel space with a single shadow page table (Härtig et al. described this technique in ref. [HHL+97]; they also described problems with alternative approaches in their section *The Dual-Space Mistake*). Occasionally Linux defines translations for a particular application within the kernel space, and we flush these from the kernel's shadow page table when changing page tables. Thus we can avoid switching shadow page tables when Linux changes the current page table since when Linux changes it, it is at privileged level, and the logical space is the same before and after the switch (see Figure 4.11); we only need to change the shadow page table upon privilege changes (such as at `iret`, see Figure 4.12).

We map many of the x86 architectural events to L4 IPC: transitions from kernel to user, system calls executed by user, exceptions executed by user, and virtual interrupts. See Figure 4.13.

L4 lacks asynchronous event delivery — it requires a rendezvous of two threads via IPC. Since we map hardware interrupts and timer events onto IPC, we instantiate an additional L4 thread within the virtualization-assist module that receives asynchronous event notifications and either directly manipulates the state of the

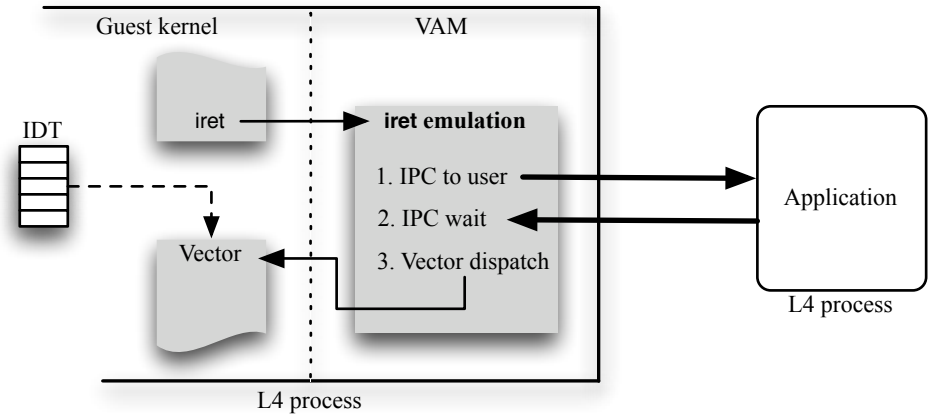



Figure 4.12: The x86 `iret` instruction can be used to transfer from kernel to an application. Its emulation performs an L4 address space change, via L4 IPC. If the L4 process does not yet exist, the emulation allocates and initializes a new L4 process, to map to the guest application. The L4 processes store no guest state, which makes the L4 processes easily reassignable (in case we migrate over the network to another machine, or if we hit a resource limit and are unable to allocate a new L4 process but must reuse an old).

L4 VM thread or updates the virtual CPU model (e.g., register a pending interrupt when interrupts are disabled). See Figure 4.14 for a relationship of events to the L4 threads within the virtualization-assist module.

As described in Section 4.3.3, we added several structural hooks to the guest OS, to accommodate virtualization inefficiencies in the L4 API. One hook is a consequence of separating guest kernel and applications into separate shadow spaces: the guest kernel needs to access the user data using kernel addresses [HHL+97, Mol03].

The virtualization-assist module bootstraps the guest OS. It prepares the L4 application environment to suit virtualization of a guest OS (primarily setting up the virtual memory). The virtualization-assist module starts executing the guest at its 32-bit entry point, thus skipping 16-bit boot logic.

### 4.4.5 Network device emulation

Supporting our goal for high-speed device virtualization that conforms to soft-layering principles, we implemented a device model for the DP83820 gigabit network card. The guest need only provide a DP83820 network device driver, and via

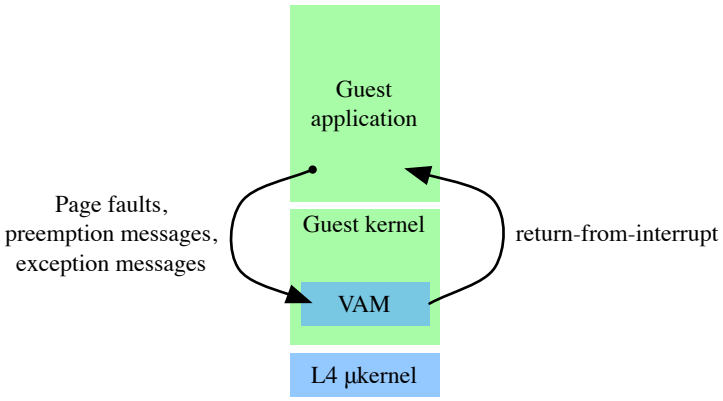



Figure 4.13: L4 IPC abstracts x86 architectural events, particularly when they cross the protection boundary between the guest's applications and kernel. The virtualization-assist module orchestrates the IPC.

its virtualization, is able to use any physical device supported by the hypervisor.

The DP83820 device interface supports packet batching in producer-consumer rings, and packets are guaranteed to be pinned in memory for the DMA operation, supporting zero-copy sending in a VM environment (see Figure 4.15). It additionally permits hardware acceleration with scatter-gather packet assembly, and checksum offload (if the hypervisor's interface supports these operations, which our driver-reuse environment does support). A drawback of this device is lack of support in older operating systems such as Linux 2.2.

We split the DP83820 model into a front-end and a back-end. The front-end models the device registers, applies heuristics to determine when to transmit packets, and manages the DP83820 producer-consumer rings. The back-end sends and receives packets via the networking API of the hypervisor.

We have only implemented a back-end for the L4 environment, which is the driver reuse network translation module of this thesis (Section 3.6.2). See Figure 4.16 for a diagram of the DP83820 model, rewritten at runtime to use the driver reuse framework.

## 4.5 Hypervisor-neutral binary

We generate a guest-kernel binary that can execute on different hypervisors, as well as on raw hardware. We designed the solution to avoid creating dependencies on known hypervisors at compile and link time. Instead it is runtime adaptable to different hypervisors, especially future versions of existing hypervisors. The

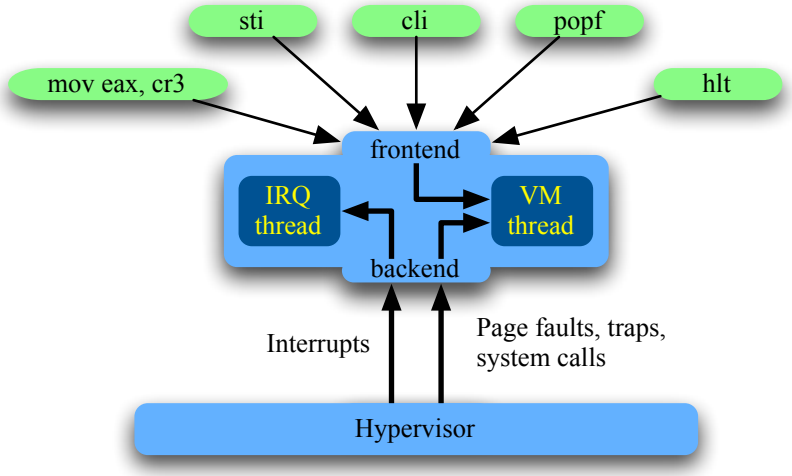



Figure 4.14: The L4 virtualization-assist module has two threads for handling requests, corresponding to synchronous and asynchronous requests.

exception is if the target hypervisor deviates so far from the platform interface that it requires structural changes in the guest kernel.

The major difference between our binaries and the original binaries, in terms of code generated by the compiler, is that ours contain padding around virtualization-sensitive instructions. The padding provides space for the virtualization-assist module to rewrite with optimized code. To quickly find these instructions at runtime, we also attach an annotation table to the binary. In the case of Linux, the table is an additional section in its ELF binary. Each row in the table contains the start and stop address of a virtualization-sensitive instruction and its surrounding padding. The annotations contain no more information, since the virtualization-assist module decodes the instructions at runtime to identify them.

Additionally, the binary contains a table to identify the function overloads. Each row in the table contains a globally unique identifier to identify the type of overload, and the address of the function pointer to overload.

L4 requires some structural changes to the guest kernel. L4 requires the guest OS to run in a compressed, 3 GB virtual address space, and so we relocate the link address from the typical 3 GB virtual address to 2 GB. We use a static link address (as opposed to a variable and runtime configurable address, which is possible). Thus when using the same guest binary for Xen, L4, and raw hardware, it runs on all at the 2 GB link address.

The guest kernel binary has no other support for the pre-virtualization environ-

ment. It carries none of the emulation code installed by the virtualization-assist module. The virtualization-assist module supplies the necessary replacement emulation code. The virtualization-assist module and guest kernel are separate binaries; they can be distributed independently of each other.

## 4.6 Discussion

General-purpose OSes have a history of providing standardized and abstracted interfaces to their applications. Para-virtualization attempts the same — but this approach is deficient principally because abstractions lock-in architectural decisions [PS75, EK95], while in contrast, the neutral platform interface is expressive and powerful, permitting a variety of hypervisor architectures. Soft layering provides the architectural freedom we desire for hypervisor construction. Soft layering also supports special hypervisor interfaces via two mechanisms: (1) function call overloading, and (2) passing high-level semantic information to the platform instructions (in otherwise unused registers) for use by the virtualization-assist module (e.g., a PTE's virtual address), and ignored by the hardware; both would require standardization, but satisfy the soft-layer criterion that the hypervisor can ignore the extra information.

The soft layer follows the neutral platform interface to maximize the chances that independent parties can successfully use the soft layer. Yet the soft layer still requires agreement on an interface: how to locate and identify the instruction-level changes, the size of the no-op windows that pad virtualization-sensitive instructions, the format of the annotations, and the semantics of the function overloading for structural changes. A soft layer forces the standard to focus more on information, and less on system architecture, thus facilitating standardization since it is unnecessary to favor one hypervisor of another. Additionally, the soft layer can degrade in case of interface mismatch; in worst case, a hypervisor can rely on privileged-instruction trapping to locate the sensitive instructions, and to then rewrite the instructions using the integrated no-op padding, while ignoring other elements of the soft layer.

## 4.7 Summary

We presented the soft-layering approach to virtualization, which provides the modularity of traditional virtualization, while supporting many of the advantages of

para-virtualization. Soft layering offers a set of design principles to guide the modifications to an OS, with a goal to support efficient execution on a variety of hypervisors. The principles: (1) permit fallback to the neutral platform interface, and (2) adapt at runtime to the interfaces that competitors may provide. Our application of soft layering, called pre-virtualization, also reduces the effort of para-virtualization by introducing automation. We demonstrated the feasibility of pre-virtualization by designing solutions for several dissimilar hypervisors with the same approach and infrastructure.

We believe that pre-virtualization enables other exciting approaches we would like to explore in the future. This includes migration of live guests between incompatible hypervisors, after serializing the CPU and device state in a canonical format; the target hypervisor would rewrite the annotated instructions, and then restore the CPU and device state, using its own virtualization-assist module. Also soft layering can optimize recursive VM design, by bypassing redundant resource management in the stack of VMs, e.g., avoiding redundant working-set analysis.

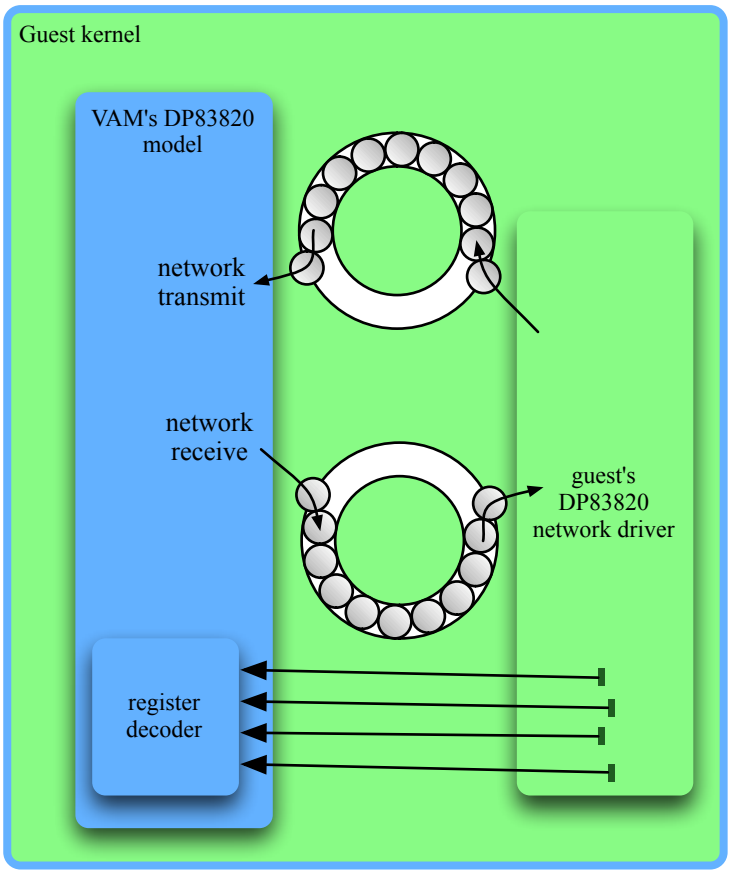



Figure 4.15: The guest's DP83820 driver interacts solely with the VAM's DP83820 device model, using producer-consumer rings that batch packet transfer, and with function calls into the VAM (we rewrite the device-register access instructions to invoke the VAM). The DP83820 has explicit ownership of the packet buffers in the receive and transmit rings, and thus the VAM can safely transfer packets with DMA, avoiding copying. Additionally, the DP83820 rings support scatter-gather packet assembly, and checksum offload, permitting hardware acceleration for packet transfer.

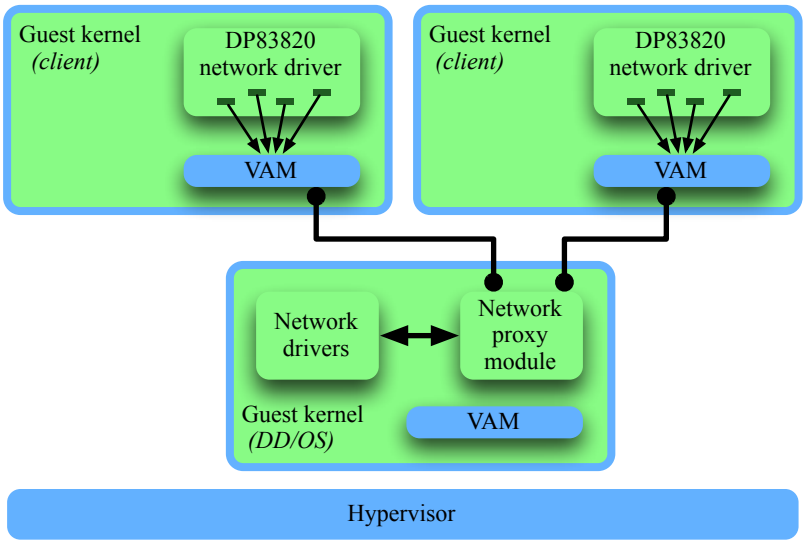



Figure 4.16: To achieve high speed networking, without investing in custom drivers for the VM environment, each guest OS uses its standard DP83820 driver, but pre-virtualized. At runtime the guest's driver is rewritten so that the instructions that normally access the device registers instead invoke methods within the virtualization-assist module's DP83820 device model. The virtualization-assist module upmaps the DP83820 networking operations to the higher-level operations of the driver reuse framework.

# Chapter 5

# Evaluation

We performed comparative performance analyses to study both driver reuse and pre-virtualization. We looked at performance, overheads, and the additional sources of resource consumption introduced by our approaches.

Most performance numbers are reported with an approximate 95% confidence interval, calculated using Student's t distribution with 4 degrees of freedom (i.e., 5 independent benchmark runs), and sometimes 9 degrees of freedom.

## 5.1 Device driver reuse

We implemented a driver reuse system following the architecture described in Chapter 3. We evaluated two approaches to virtualizing the DD/OS: the pre-virtualization technique described in Chapter 4, and para-virtualization; we did not evaluate full virtualization.

Our operating environment is based on the L4Ka::Pistachio microkernel,[1] which offers only a handful of rudimentary native drivers. It is one of the projects described in this thesis's introduction that tries to introduce novel OS mechanisms, but lacks drivers to build a full-fledged OS, and so needs a driver-reuse environment.

We contrast the performance of our reused drivers to two other environments: the original drivers as they exist in their donor system, and native L4 drivers (which run at user-level).

[1] The L4Ka::Pistachio microkernel is available as source code from the System Architecture Group's web site at the University of Karlsruhe (TH), as well as from `http://l4ka.org/projects/pistachio`.

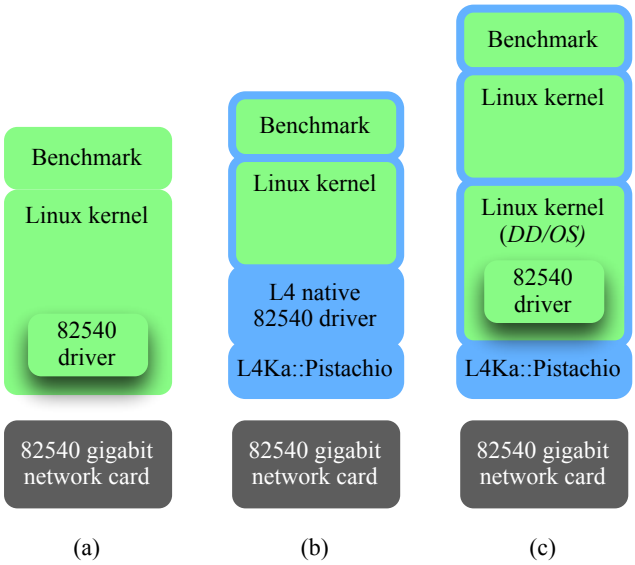



Figure 5.1: In studying the overheads of user-level drivers we used a networking benchmark. The device was an Intel 82540 gigabit card. In (*a*) we have the baseline configuration, with a Linux driver running in kernel mode. In (*b*) we have a native L4 user-level driver controlling the network card. In (*c*) we have a driver reuse environment, where the driver also runs at user level, but within a VM and with its entire Linux kernel.

Our reused drivers originated from Linux, a monolithic kernel that normally runs its drivers in privileged mode. To determine the costs of driver reuse compared to their original environment, we compared our driver-reuse performance to native Linux performance. Since our driver reuse approach runs the drivers at user-level, which can be a more costly approach for driver construction (e.g., on x86, user-level drivers add the cost of flushing the TLB when switching to and from the driver), we further refined the cost burden of driver reuse by comparing to a native L4 user-level driver, to distinguish between the overheads of our approach and the overheads due to user-level drivers.

We configured a normal Linux system as the baseline, a device driver reuse system to closely resemble the baseline, and a user-level driver system that resembled the driver reuse system. See Figure 5.1 for a diagram comparing the architectures of the three systems. The baseline Linux and the driver-reuse system used identical device driver code. They ran the same benchmarks, utilizing the same protocol stacks and the same OS infrastructure. They differed in their architectures: the baseline used its native privileged-mode device driver environment, while our

driver reuse system used one or several DD/OS VMs with the proxy modules, and a benchmark VM as the driver client that ran the benchmark code. The user-level driver environment ran the same benchmarks running within a benchmark VM, but connected to native L4 user-level drivers.

### 5.1.1 Components

The driver-reuse system used the L4Ka::Pistachio microkernel as the hypervisor. The DD/OS and the client OS used Linux 2.6.8.1 and Linux 2.6.9. The VMM was a native L4 application called the Resource Monitor; it coordinates resources such as memory, device mappings, and I/O port mappings for the DD/OS instances and the client OS.

To minimize resource consumption in the DD/OS, we configured the Linux kernel, via its build configuration, to include only the device drivers and functionality essential to handle the devices intended to be used in the benchmarks. The runtime environment of each DD/OS is a ROM image that initializes into a single-user mode with almost no application presence, and thus low burden on the DD/OS's house keeping chores.

All components communicate via L4 mechanisms. These mechanisms include the ability to establish shared pages, perform high-speed synchronous IPC, and to efficiently copy memory between address spaces. The mechanisms are coordinated by object interfaces defined in a high-level IDL, which are converted to optimized assembler inlined in the C code with the IDL4 compiler [HLP+00].

The machine used a Pentium 4 processor at 2.8 GHz with a 1MB L2 cache. It ran Debian 3.1 from the local SATA disk. The network card was an Intel gigabit card based on the 82540 chip. For networking tests, we used a remote client, which had a 1.4 GHz Pentium 4 processor, 256MB of memory, a 64MB Debian RAM disk, an Intel 82540 gigabit Ethernet PCI card, and executed a native Linux 2.6.8.1 kernel. They were connected via a gigabit switch.

### 5.1.2 User-level device drivers

Our first benchmark series studies performance of network benchmarks for native Linux, driver reuse, and our L4 native network driver. We used network benchmarks because they heavily exercise the device driver: networking benchmarks are often I/O bound, and they use small units of data transfer causing frequent driver interactions. Thus they highlight the overheads of driver reuse. Networking is also

an important feature in a world that highly values the Internet.

Our network device is an Intel 82540 gigabit network card [Int06] (also known as the *e1000*, and the *Pro/1000*). We used Linux's native e1000 driver for the baseline and for driver reuse. For our L4 native user-level driver, we constructed a driver for the Intel 82540 based on Intel's portable e1000 driver code (which is the same code used by Linux). This L4 driver also served as the basis of our benchmarks in ref. [ULSD04].

### TCP/IP networking

We ran the Netperf benchmark on native Linux for the baseline, as it is one of the common networking benchmarks in the literature. The goal is to compare throughput and CPU utilization of the different driver environments. We predicted that user-level drivers add a large source of of additional CPU utilization, with driver-reuse adding even more. Additionally, we predicted that as long as we are I/O bound, all driver environments would achieve similar throughputs.

Netperf transferred 1 GB of data, using the following Netperf command line: `-l -1073741824 -- -m 32768 -M 32768 -s 262144 -S 262144`. In our L4 environments, Netperf ran in a benchmark VM running Linux with network services provided by L4.

Figure 5.2 and Figure 5.3 present the results of the *Netperf send* benchmark. Figure 5.4 and Figure 5.5 present the results of the *Netperf receive* benchmark. The $x$-axis is the total benchmark runtime, which represents throughput (shorter bars are better). Additionally, each bar is divided into *active CPU time* and *idle CPU time*, with *active* time shown as a percentage of the total run time. In each case, the throughput is nearly equal, since the benchmark is I/O bound.

We show both versions of Linux because 2.6.9 introduced a substantial performance improvement between the two versions by rearchitecting some of the networking paths. We started with 2.6.8.1, the results of which we reported in prior publications [LU04, LUSG04]. We then later upgraded to 2.6.9. As the graphs show, our driver reuse did not benefit from 2.6.9's performance improvements — it only accentuates the additional costs of user-level drivers. As expected, the user-level drivers consumed more active CPU time than the native kernel-level Linux drivers. The user-level drivers introduce address space switching, which is fairly expensive on x86 due to TLB and instruction trace cache flushes, as well as the time required to execute the address space change instruction. Additionally, the system structuring introduces overhead due to more components, since it runs on

L4Ka::Pistachio in addition to Linux. We also believe that we have throughput loss due to our batching interfering with the reliability and congestion control of TCP, which becomes more evident with 2.6.9 — we explore this more in the following benchmark.

In comparing the native L4 driver to driver reuse, which are both user-level drivers, we see that the driver reuse environment consumed more CPU time. Since both drivers are based on the same portable Intel e1000 driver code, we rule out a less efficient driver. We assume that the additional overhead comes from the DD/OS and its side effects. We explore some of the DD/OS's overhead in the following sections.

The driver-reuse results include two bars, labeled *Driver reuse, custom client* and *Driver reuse, DP83820 client*, which compared two approaches for providing networking services to the client Linux. The custom client is a Linux device driver written specifically for the L4 environment. The DP83820 client is a pre-virtualized network adapter that reuses Linux's integrated DP83820 device driver. See Figure 5.6 for a comparison of the two approaches to providing networking services to the client Linux. Our pre-virtualized network adapter outperformed the custom client in the send case.

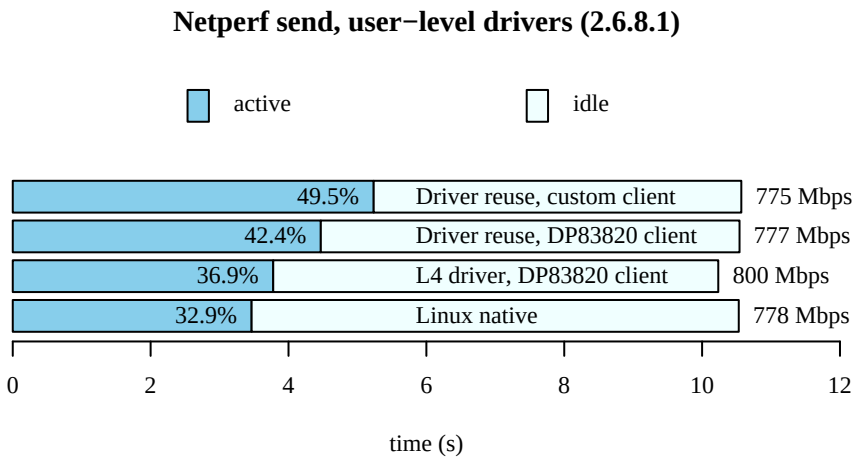



Figure 5.2: The performance of the Netperf send benchmark with Linux 2.6.8.1. The 95% confidence interval is at worst ± 0.1%

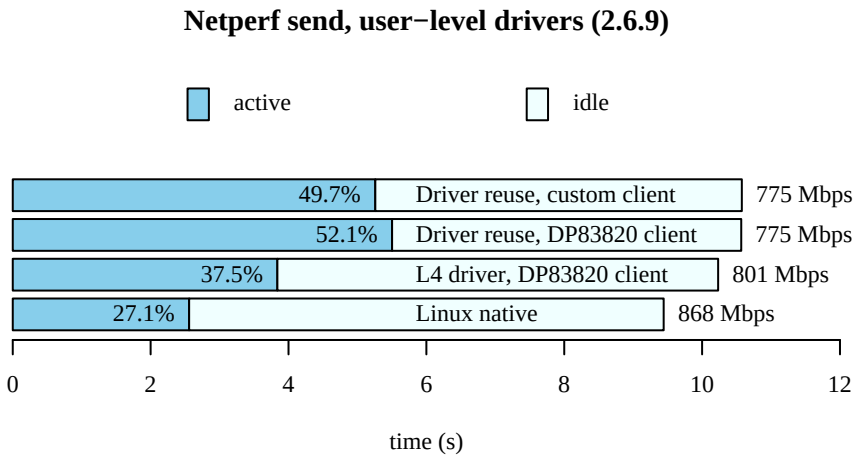



Figure 5.3: The performance of the Netperf send benchmark with Linux 2.6.9. The 95% confidence interval is at worst ± 0.1%

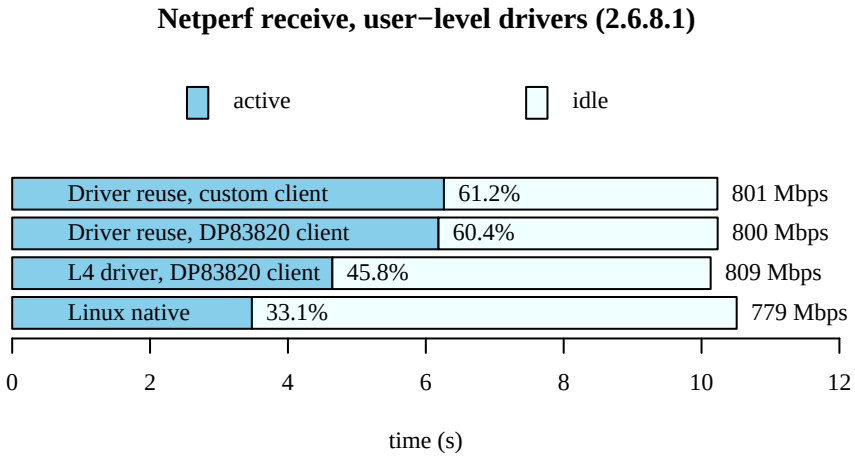



Figure 5.4: The performance of the Netperf receive benchmark with Linux 2.6.8.1. The 95% confidence interval is at worst ± 0.3%

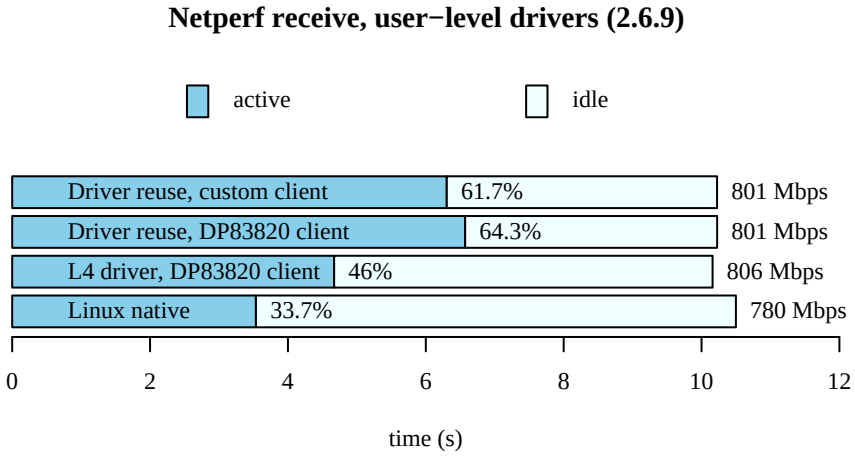



Figure 5.5: The performance of the Netperf receive benchmark with Linux 2.6.9. The 95% confidence interval is at worst ± 0.5%

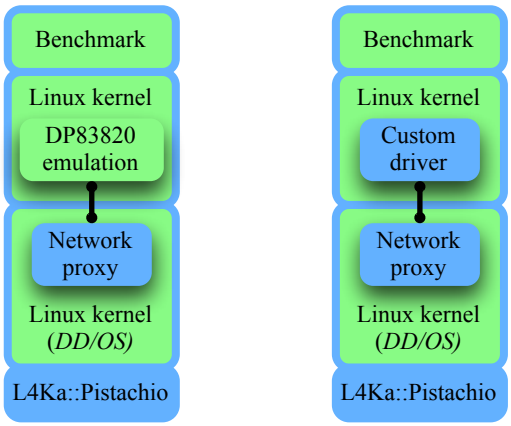



Figure 5.6: Since the benchmark client itself is a VM, we need to virtualize its networking interface to attach it to our driver environment. We developed two approaches. In (*a*) we use pre-virtualization to prepare Linux's DP83820 device driver for use as an interface, and then emulate the DP83820 with our VM runtime. In (*b*) we use a para-virtual custom network driver that we wrote for Linux.

### UDP networking

We believe that the TCP/IP benchmark results reveal two types of overheads: overhead due to additional CPU utilization, and throughput degradation due to additional queuing time. The TCP/IP protocol [TCP81] requires a balance between latency and throughput, which becomes harder to achieve in user-level driver frameworks where one batches packets to reduce the rate of address-space transitions. See Figure 5.7 for a diagram that shows how TCP/IP control packets influence its channel utilization. Figure 5.8 shows how the additional software layers in a driver-reuse environment further influence TCP/IP's network channel utilization. We have three goals for the following benchmark: (1) to show that TCP/IP reliability and congestion control might interact adversely with our setups (although we don't measure the extent and scope of phenomena related to this issue, as that is a matter of balance between tuning for a variety of benchmarks, and is thus beyond the scope of this thesis); (2) to show overheads of our driver-reuse environment at full channel utilization of the gigabit link; and (3) to determine whether we achieve the same throughput as the native driver.

The benchmark uses the UDP protocol, which lacks the reliability and congestion control of TCP/IP. This also makes it harder to benchmark, because reliable conclusion of the benchmark now requires a network link that reliably transmits all

packets. A UDP benchmark is fairly easy to run, and not finding one that suited our goals, we wrote our own microbenchmark called *UDP Send Blast*. It sends data as fast as possible to a recipient, using UDP as the transport. The parameters: it sent 1GB of data at full MTU, with 256kB in-kernel socket buffers, using `send()` on user-level buffers of 32,768-bytes of data. Since it uses no reliability and congestion control, it can lose packets due to channel congestion and corruption, but we integrated a mechanism to count received packets. Also, due to the lack of reliability, we do not know when the recipient has finished receiving the last of the data; thus we time only the throughput of the sender's NIC for transmitting the data.

We predicted that the benchmark would achieve full throughput in each driver environment, but at higher CPU utilization for user-level drivers, and even more CPU utilization for driver reuse.

Figure 5.9 provides Linux 2.6.8.1 performance, and Figure 5.10 provides Linux 2.6.9 performance. In contrast to the Netperf benchmark, both versions of native Linux achieved the same throughput and similar CPU utilization (thus Linux 2.6.9 has no performance advantage to 2.6.8.1). Additionally, the driver-reuse environment achieved the same throughput as native Linux, as did the L4 native driver except for when using Linux 2.6.8.1 — we assume that it is a tuning issue that could be corrected with further work. The L4 user-level drivers had lower CPU utilization compared to native Linux. The driver-reuse had quite a bit more CPU utilization compared to the other two systems. Thus driver reuse adds additional overhead.

This benchmark was setup to achieve full channel utilization, which it seems to accomplish since driver reuse provided the same throughput as native Linux. Since driver reuse can achieve full channel utilization, but does not in the Netperf TCP/IP benchmark, we believe that TCP/IP and user-level drivers interact in a way that impedes full channel utilization — this could be an interesting topic for the virtualization community to research in the future.

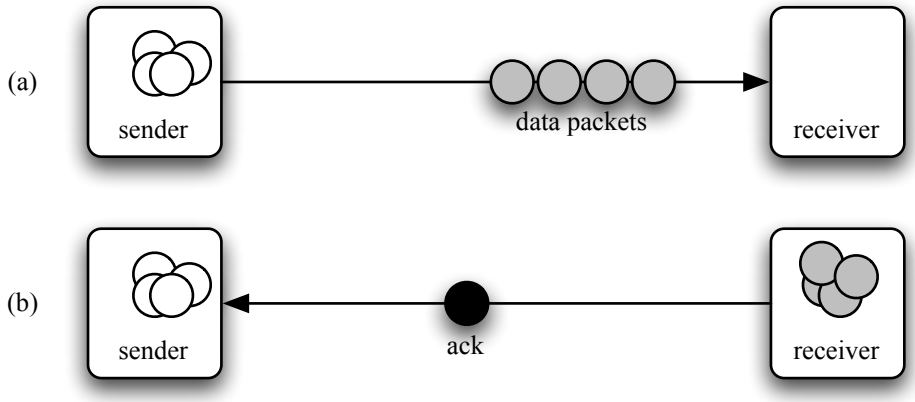



Figure 5.7: TCP's reliability, flow-control, and congestion-control algorithms reduce its network channel utilization. The utilization is the time using the physical network medium divided by the total time used by the network protocol. In *(a)*, the sender puts a subset of the pending data packets onto the medium, but then waits for the *ack* packet (in *(b)*) before proceeding with further data packets.

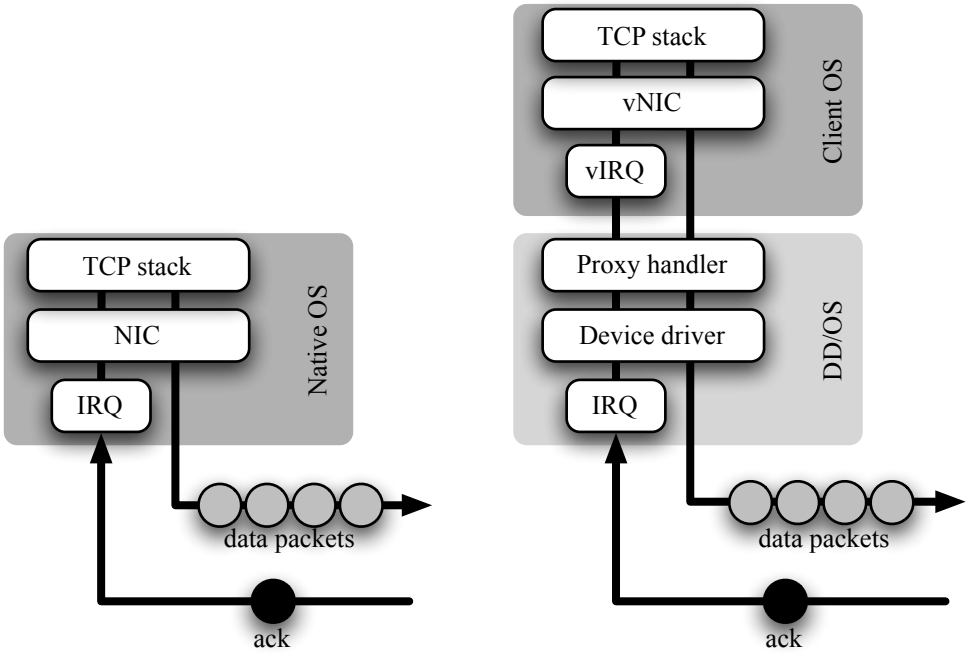



Figure 5.8: Driver reuse increases the latency of the round trip of network packets through the TCP stack. This additional latency decreases network channel utilization. The *ack* packet also defies the packet batching optimization of normal data packets. TCP's windowing can also defy batching, for it may become unsynchronized with the batching, so that the window releases one packet while waiting for its *ack* packet, but our batching algorithm delays the delivery of that packet to the device, thus delaying forward progress of the TCP window.

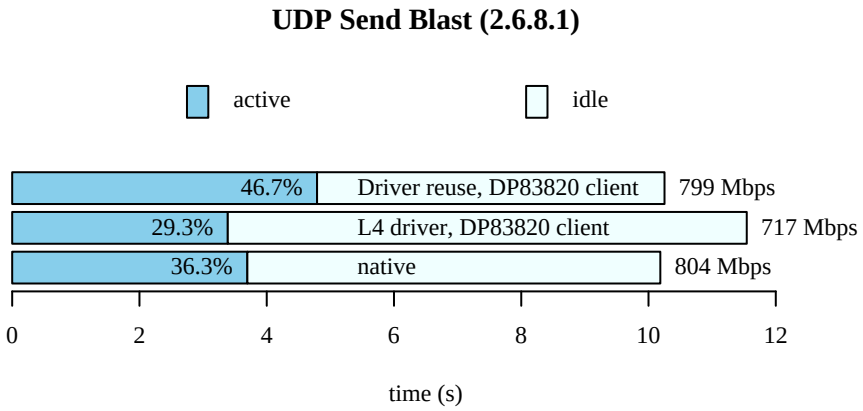



Figure 5.9: The performance of the UDP send blast benchmark with Linux 2.6.8.1. The 95% confidence interval is at worst $\pm$ 0.5%

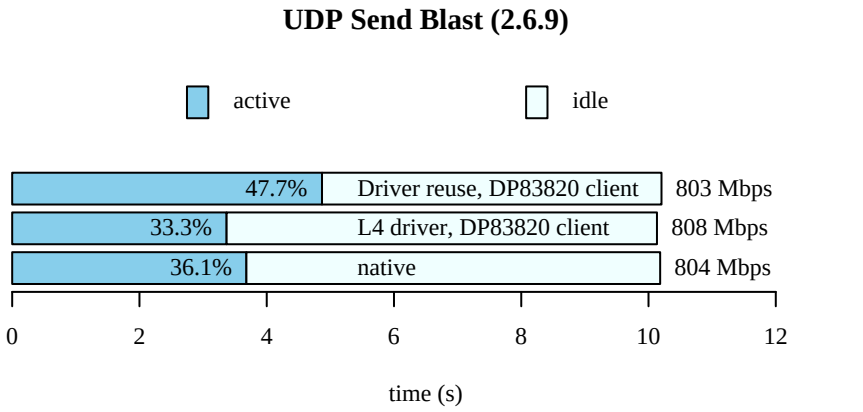



Figure 5.10: The performance of the UDP send blast benchmark with Linux 2.6.9. The 95% confidence interval for the L4 driver is $\pm$ 15%, while the others are at worst $\pm$ 0.5%.

### Working sets versus time

Driver reuse adds overheads, due to internal behavior of the DD/OS, and activity performed in the proxy modules. We can observe that behavior indirectly by monitoring how the DD/OS interacts with the system's bulk resources over time, including its active memory footprint (its working set). This benchmark observes the working set of a DD/OS during a Netperf benchmark. Our goal is to account for potential sources of overheads. We predicted that the DD/OS would have a larger working set, because it likely has more background activity; this activity, along with its corresponding TLB miss rate, would contribute to the higher CPU utilization observed in the networking benchmarks.

We compared the working sets of the Netperf benchmarks for driver reuse and the L4 native driver. See Figure 5.11 and Figure 5.12 for the Netperf working set activity versus time. The working set activity was determined by monitoring the page utilization of the DD/OS and the L4 driver, with page reads and writes accumulated over 90 ms intervals, from the Linux 2.6.8.1 setup. The results show that driver reuse via VMs can consume far more resources than a custom-built driver. In both cases the working set sizes are bounded due to reuse of packet buffers, a finite number of pages that can be accessed within 90 ms, and due to the direct DMA transfer of outbound packets. The working sets drop significantly after the benchmark ends and the drivers go idle (idle working sets are addressed in 5.1.4). The DD/OS working sets have periodic spikes (roughly a 2-second period), even after the benchmark terminates, which suggest housekeeping activity within the DD/OS.

A user-level L4 application measured the working sets, which used costly system calls for scanning the entire driver memory space (the system call can query the page status of only 16 pages per invocation), and thus had noticeable overhead; the overhead severely disturbed the DD/OS during the *Netperf receive* benchmark, but not the L4 native driver, because the DD/OS has a larger memory space to scan (the L4 native driver consumes very little memory). Given the inaccuracy of the sampling, we use these results as a rough guide, with the conclusion that we have larger working set footprints for the DD/OS, and thus more activity per packet.

### Processor utilization versus time

We also measured CPU utilization of the Netperf benchmarks over time, sampling CPU utilization every 90 ms (by sampling the CPU's performance counters). The

benchmark results from Netperf, etc., have so far only revealed the mean CPU utilization for the benchmark, which is a number that hides many details of system behavior. In contrast, these time plots show whether CPU utilization is consistent across the benchmark. We predicted that CPU utilization is consistently higher across the benchmark, due to the additional code paths executed per packet. See Figure 5.13 and Figure 5.14 for the plots (for Linux 2.6.8.1). At our 90 ms sampling resolution with the x86 performance counters, the CPU utilization is consistently more costly for driver reuse in VMs than for a native user-level driver.

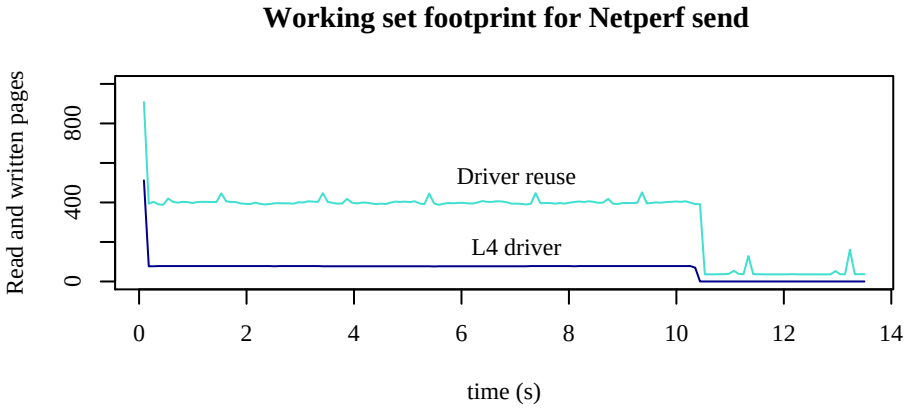



Figure 5.11: The working set footprint of Netperf send, versus time, sampled at 90 ms intervals. For driver reuse, the working set has DD/OS contributions only. For the L4 driver, the working set concerns the driver only.

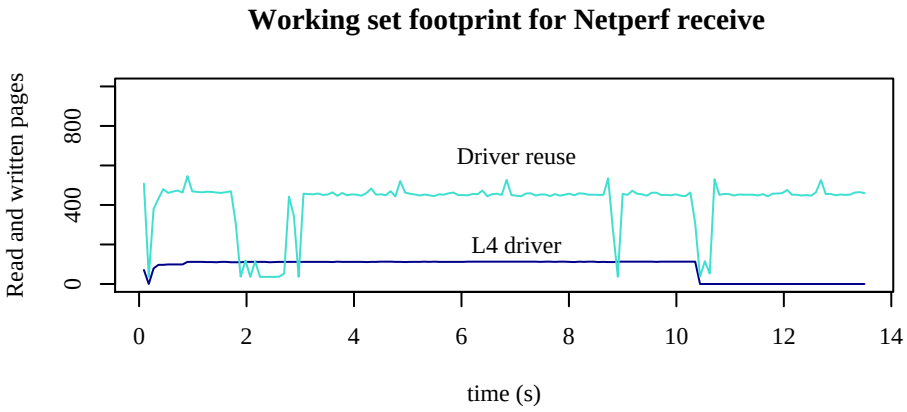



Figure 5.12: The CPU working set footprint of Netperf send, versus time, sampled at 90 ms intervals. The overhead of sampling severely interferes with the benchmark in driver reuse. For driver reuse, the working set has DD/OS contributions only. For the L4 driver, the working set concerns the driver only.

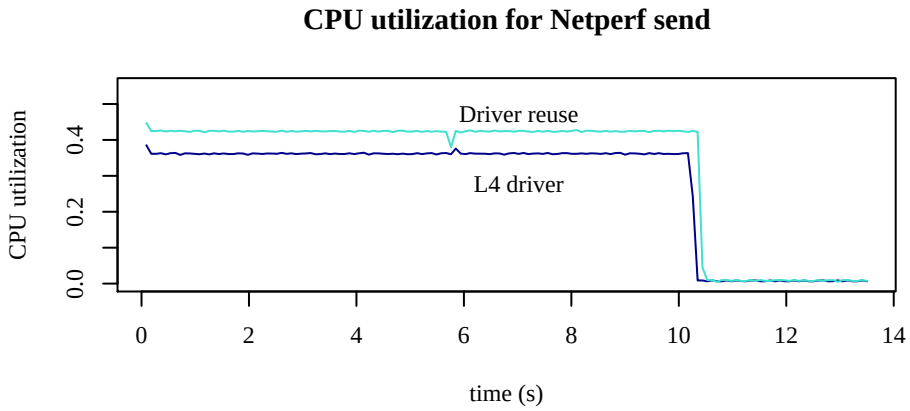



Figure 5.13: The CPU utilization of Netperf-send versus time, sampled at 90 ms intervals.

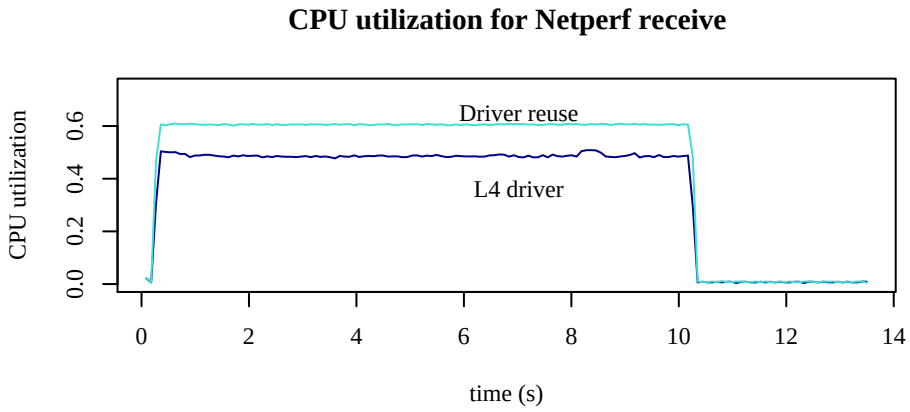



Figure 5.14: The CPU utilization of Netperf-receive versus time, sampled at 90 ms intervals.

### 5.1.3 Isolated versus composite

Our driver-reuse technique can provide the isolation properties of typical user-level drivers. This involves running each driver in a dedicated VM, with the driver's entire DD/OS. Thus each isolated driver adds the overheads of the DD/OS to the system. We built a driver-reuse system with this isolated driver approach, and evaluated it.

The system isolated the network, disk, and bus drivers. Each driver ran in a dedicated VM, with the network and disk VMs reusing the PCI services of the PCI VM. In all other aspects, it shares the setups of prior benchmarks, except where stated.

To observe the overheads of isolated drivers we run a benchmark that exercises both network and disk simultaneously, thus potentially exciting complicated dynamic behavior. Since such a benchmark combines many variables related to our system, we build towards it, by first studying network dynamics, then disk dynamics, and then the combination of the two.

#### Network

The first benchmark focused on networking, using the TTCP benchmark (another common benchmark in the literature), with the goal to uncover performance deltas between running drivers in isolated VMs versus running all drivers within a single VM. We predicted that we would see minor additional overheads for the isolated driver case, since the networking benchmark exercises a single driver.

The disk VM had no load during the benchmark. We compared performance to native Linux, and to a driver-reuse system that ran all drivers in a single, composite VM. We ran the benchmark with para-virtualization and pre-virtualization, using Linux 2.4 and Linux 2.6.8.1. All produced similar results, and we present the data from the para-virtualized Linux 2.6.8.1. See Figure 5.15 for the TTCP send benchmark, with Ethernet's standard 1500-byte MTU. See Figure 5.16 for the TTCP send benchmark, with a 500-byte MTU (causing the kernel to fragment the data into smaller packets). See Figure 5.17 for the TTCP receive benchmark, with 1500-byte MTU. And see Figure 5.18 for the TTCP receive benchmark, with 500-byte MTU. The disk and PCI VMs had no active contributions to the benchmarks, and only performed house-keeping tasks during the benchmarks, which contributed very little noticeable overhead to the benchmarks. Thus when an isolated system uses a single driver, the other (idle) drivers contribute little noticeable

overheads.

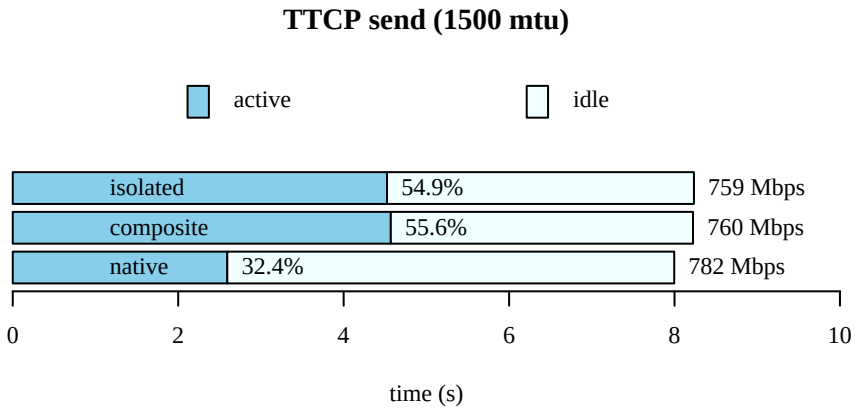



Figure 5.15: The performance of the TTCP-send benchmark with 1500-byte mtu, for native Linux, all reused drivers in a composite VM, and reused drivers in isolated VMs. The 95% confidence interval is at worst ± 0.2%

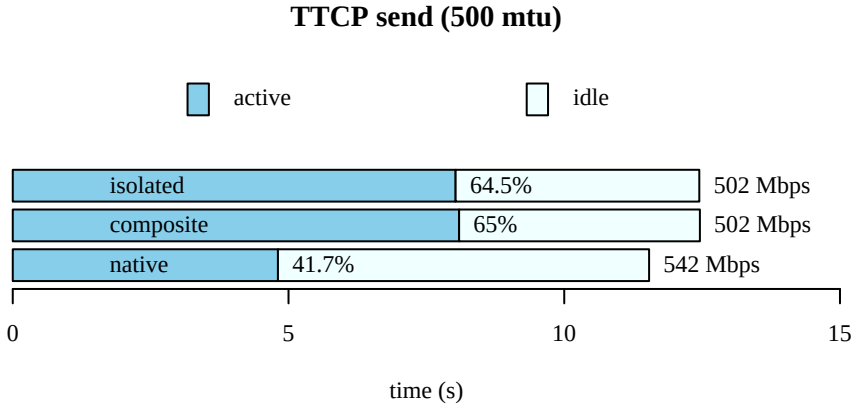



Figure 5.16: The performance of the TTCP-send benchmark with 500-byte MTU, for native Linux, all reused drivers in a composite VM, and reused drivers in isolated VMs. The 95% confidence interval is at worst ± 0.4%

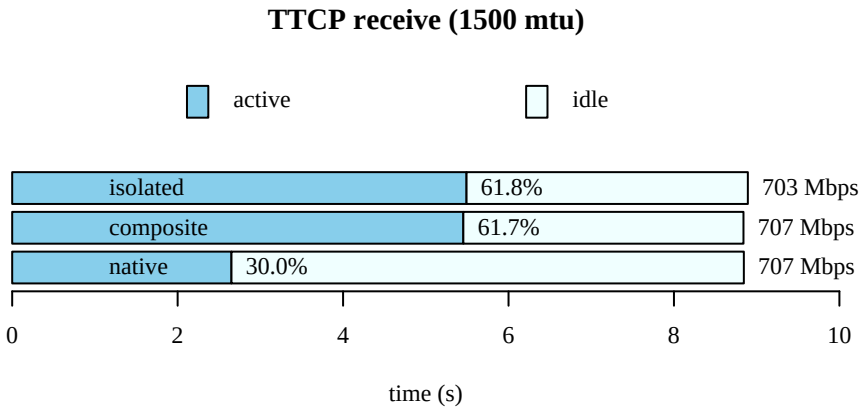



Figure 5.17: The performance of the TTCP-receive benchmark with 1500-byte MTU, for native Linux, all reused drivers in a composite VM, and reused drivers in isolated VMs. The 95% confidence interval is at worst $\pm$ 0.3%

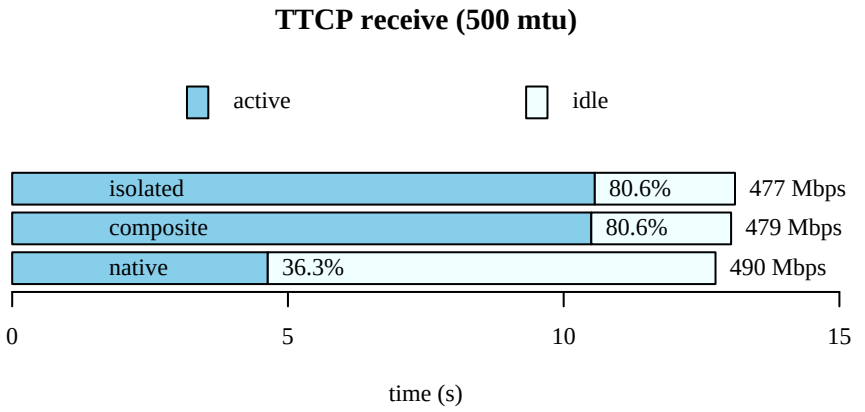



Figure 5.18: The performance of the TTCP-receive benchmark with 500-byte MTU, for native Linux, all reused drivers in a composite VM, and reused drivers in isolated VMs. The 95% confidence interval is at worst $\pm$ 0.3%

### Disk

The second benchmark studied disk performance and overheads. Disks have low performance, and thus we predicted that driver reuse would add little noticeable performance overhead, and so we ran the following experiments to confirm. The network driver had no load. The benchmark ran within the client VM, and reused the disk driver of the DD/OS. We wrote a *streaming* disk benchmark to highlight the costs of driver reuse:

- It streams data to/from consecutive disk blocks, to avoid hiding driver overheads behind random-access disk latency.
- It bypasses the client's buffer cache (using a Linux raw device), and thus we ensure that every disk I/O interacts with the driver and the hardware.
- It bypasses the client's file system (by directly accessing the disk partition), thus excluding file system behavior from the results.

We varied block size from 512 bytes to 64 kilobytes while transferring 2GB of data. We compared native Linux to a composite driver reuse system; and to a driver reuse system that isolated the disk, network, and PCI drivers in separate VMs. In all cases, the read bandwidth averaged 51 MB/s, and the write bandwidth averaged 50 MB/s. The primary difference was the amount of CPU utilization: native Linux had less CPU utilization, particularly for the block sizes smaller than the page size, which had more driver interaction per byte. See Figure 5.19 for the read CPU utilization, and Figure 5.20 for the write CPU utilization.

For large block sizes, we have confirmed our expectation that driver reuse would have slight overheads for the disk; for smaller block sizes, the overheads become more noticeable.

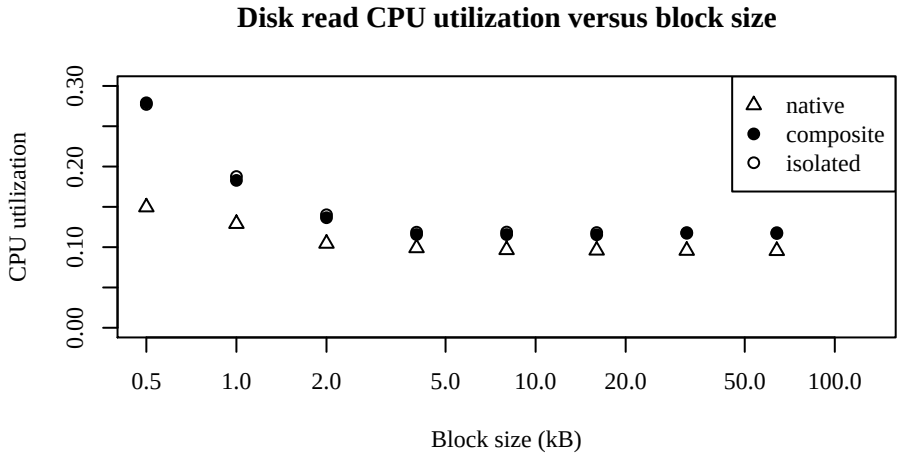



Figure 5.19: The CPU utilization of the streaming disk read microbenchmark, for: *native* Linux; a *composite* DD/OS that ran all drivers; and a DD/OS system with *isolated* drivers for network, disk, and PCI. Bandwidth averaged 51 MB/s.

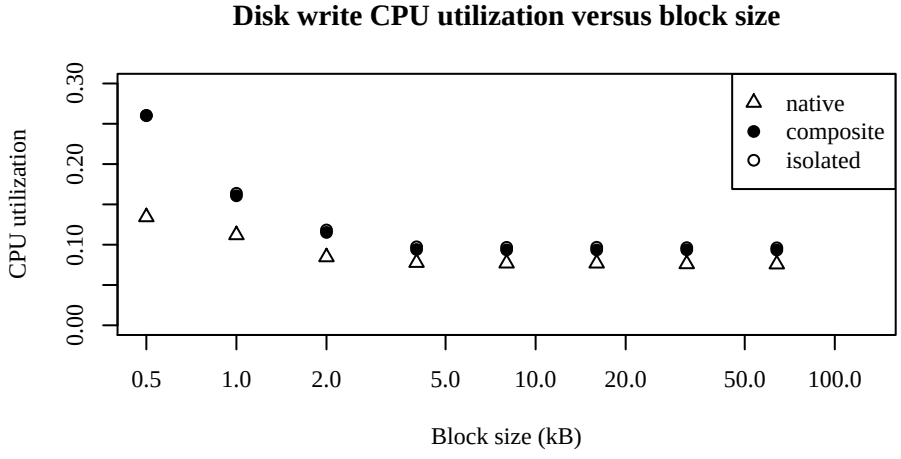



Figure 5.20: The CPU utilization of the streaming disk write microbenchmark, for: *native* Linux; a *composite* DD/OS that ran all drivers; and a DD/OS system with *isolated* drivers for network, disk, and PCI. Bandwidth average 50 MB/s.

### Network and disk

So far the benchmarks for studying isolated versus composite performance have considered either network or disk activity, but not both simultaneously. This third benchmark combines them, so that both drivers contribute actively to the benchmark.

The benchmark ran an NFS server on the test system, with load provided by a remote client running the PostMark benchmark. The PostMark benchmark emulates the file transaction behavior of an Internet e-mail server. The NFS server used our driver reuse framework, and was configured as in the TTCP benchmark.

We expected the PostMark benchmark, running against an NFS server, to generate less I/O load than the prior network and disk micro benchmarks. Thus we did not expect the network and disk drivers to interfere with each other by disturbing sequencing. This led us to predict that the following benchmark would have identical performance whether running on native Linux, running all reused drivers in a single DD/OS, or when running isolated DD/OS VMs; but with higher CPU utilization for reused drivers.

The results confirmed our prediction. The performance of the NFS server was nearly identical for all driver scenarios — for native Linux, for isolated reused drivers, and for reused drivers running in a composite VM — with a mean runtime of 345 seconds. We computed an approximate 95% confidence interval, calculated with Student's t distribution with 4 degrees of freedom (i.e., 5 independent benchmark runs). Table 5.1 lists the results. The time duration of the driver reuse scenarios was slightly longer than for native Linux, but less than the confidence interval for native Linux. Both the isolated and consolidated-driver reuse configurations had higher CPU utilization than native Linux. See Figure 5.21 for CPU utilization traces of the NFS server machine during the benchmark. The benchmark starts with a large CPU utilization spike due to file creation. We configured PostMark for files sizes ranging from 500-bytes to 1MB, a working set of 1000 files, and 10000 file transactions.

The results suggest that reusing multiple drivers at the same time is feasible, whether the drivers are isolated in separate VMs, or running within a single VM.

| | duration (s) | overhead | 95% confidence interval |
|---|---|---|---|
| native Linux | 343 | | ±3% |
| isolated drivers | 344 | 0.23% | ±1% |
| composite drivers | 349 | 1.6% | ±3% |

Table 5.1: PostMark throughput for different driver scenarios; compares native Linux, to running all drivers (PCI, disk, and network) in isolated VMs, to running all drivers in a single VM (composite). The *overhead* column compares the driver-reuse scenarios to native Linux.

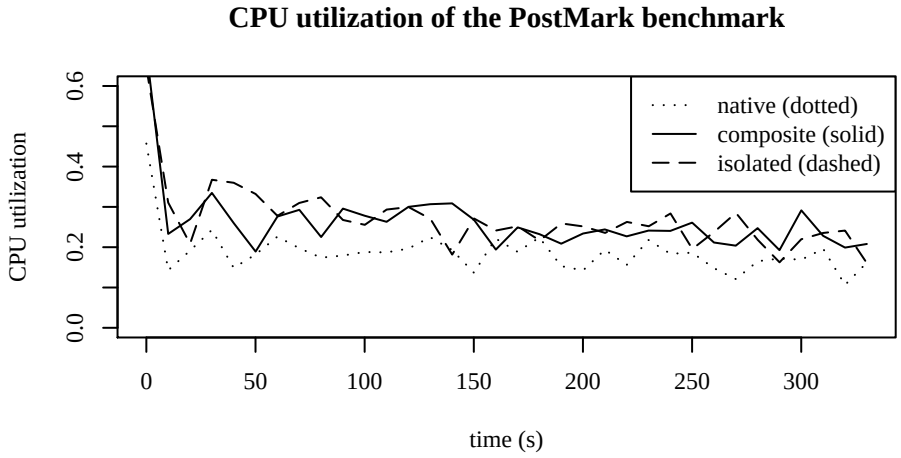



Figure 5.21: The CPU utilization for an NFS server serving a remote PostMark benchmark. Compares native Linux, to running all drivers in a single VM (composite), and to running all drivers (PCI, disk, and network) in isolated VMs.

### 5.1.4 Other overheads

This section considers other overheads of driver reuse, including active and steady-state page working set sizes (and the possibility of page reclamation), steady-state CPU utilization, TLB utilization, and cache utilization. See the preceding sections for throughput and active CPU utilization.

#### Working set

Figures 5.11, 5.12, and 5.22 present the working sets versus time of network and disk activity. Each working set contains the read and written pages of the DD/OS handling the driver, aggregated over a 90 ms sampling period. They exclude the working sets of the benchmark VMs. The working sets are an additional cost

of driver reuse that native Linux lacks; they consume memory that could have been used elsewhere, and worsen the TLB and cache miss rates of driver-reuse (as demonstrated in the following sections).

Figures 5.11 and 5.12 compare between driver-reuse and an L4 user-level driver for the Netperf benchmark. The working set for the driver-reuse scenario is over double that of the L4 user-level driver. The L4 driver is small and architected to perform solely driver activity, while the DD/OS executes house-keeping code, as well as many other code paths (such as quality of service, network filtering, etc.).

The network activity in Figure 5.22 was generated by the TTCP benchmark running in the client VM. In all of the networking scenarios, the send and receive working sets of the DD/OS are bounded. They are bounded by the number of packets that the DD/OS and networking hardware can process in a finite amount of time. They are further bounded by Linux algorithms, which reuse packet buffers from a pool of packet buffers.

The disk activity was generated by unarchiving the source code of the Linux kernel within the client VM. This benchmark read the archive from disk, and wrote the files back to disk, but many of the operations probably interacted solely with the client VM's buffer cache, and thus did not lead to interaction with the DD/OS while we monitored its working set size.

The steady-state working set reflects the DD/OS's working set with no active loading in the client VM. It thus reflects the read and written pages of an idle VM that pursued nothing more than house-keeping activities.

### Memory reclamation

The DD/OS has a larger memory footprint than a simple driver, and it may not be possible to always swap the unused pages of the DD/OS to disk since the DD/OS may drive the swap device. Yet the DD/OS does not use all of its allocated memory for active workloads. Thus it might be possible to reclaim some of the DD/OS's memory. We evaluated this proposition with a model, from which we generated estimates of potential memory reclamation. We did not implement dynamic memory sharing, particularly due to lack of hardware support, and so provide estimates instead.

We created our estimates offline, using a snapshot of an active driver-reuse system with isolated drivers. The tested configuration included three DD/OS instances, one each for PCI, IDE, and network. The PCI VM was configured with 12MB of memory, and the others with 20MB memory each. We ran the PostMark

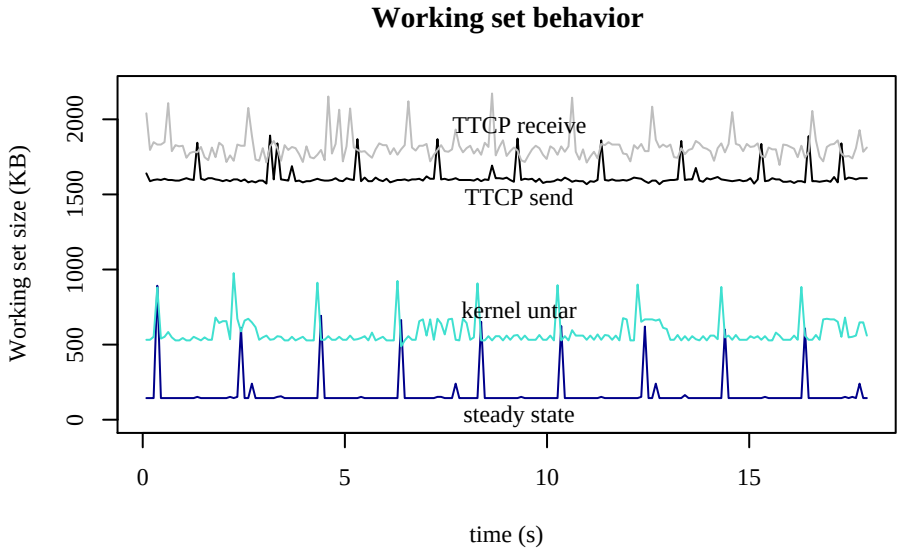



Figure 5.22: The working sets of several disk and network benchmarks, and of an idle system, versus time. Each sample is a 90ms aggregate of the read and written pages of the DD/OS.

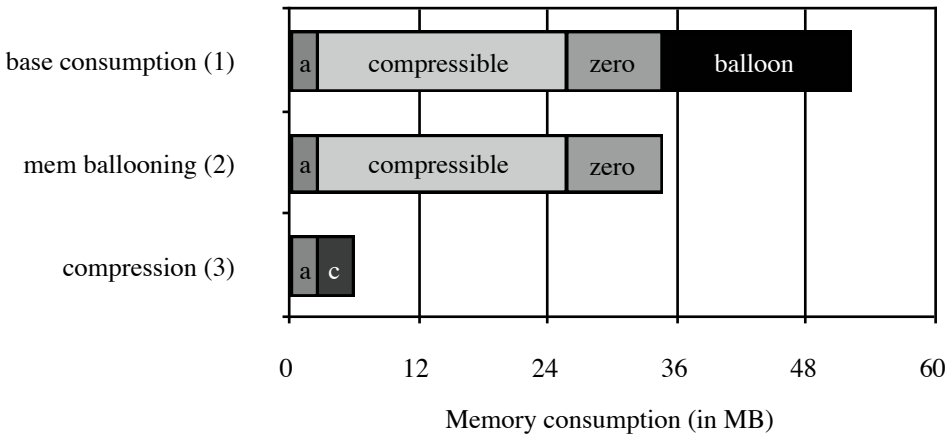



Figure 5.23: The potential for reclaiming memory from DD/OS VMs. The active working set is *a*. In (1) is the combined memory consumption of disk, network, and PCI DD/OS instances. The disk and network VMs each had a 20MB memory allocation, and the PCI VM had a 12MB allocation. Their memory consumption is categorized. In (2) we have the memory consumption after using memory ballooning to reclaim memory from the VMs. This is the most practical approach. In (3) we further attempt to reduce the memory consumption by compressing pages (in conjunction with an IO-MMU for page protection). The data compresses substantially, and is *c*. This includes duplicate pages and zero pages.

benchmark on a remote machine, accessing an NFS file server on the driver-reuse system. The NFS file server served files from a local IDE disk, over the network to the remote system running PostMark. The active memory working set for all DD/OS instances of the NFS server was 2.5MB.

All systems can reduce memory consumption through cooperative memory sharing, e.g., memory ballooning [Wal02]. With a balloon driver in each DD/OS to cooperatively ask the DD/OS to release unused memory, we predict that we can reclaim 33% of the memory. For systems without an IO-MMU for DMA quarantining, memory ballooning is the only option. Even in systems with an IO-MMU that supports recoverable DMA translation faults, the devices and the applications using the devices may not be able to recover from DMA faults, and thus will also be limited to memory ballooning.

For systems with an IO-MMU that inhibits DMA accesses to inaccessible pages, and with devices that can recover from the ensuing DMA translation faults, we can potentially recover more unused memory, uncooperatively. We can make unused pages inaccessible and then compress them [CCB99] (and restore the pages if a DD/OS or devices attempt to access the pages). When the IO-MMU has read-only page access rights, we can also share identical pages between the active and idle working sets via copy-on-write. Our proposed algorithm for using the IO-MMU's capabilities to reclaim unused pages:

1. Determine the active working sets.

2. Search for duplicate pages among the three DD/OS instances, whether the pages are in an active working set or not. Waldspurger described an efficient algorithm for detecting pages duplicated between VMs [Wal02].

3. If a duplicated page is in an active working set, then we consider all other duplicates as active too, and retain them as an active, shared page. All-zero pages are a special case, and we map them to a global zero-page (rather than search for duplicates).

4. For the pages outside the active working sets, compress them and store the compressed copies in memory, thus freeing their original pages. For inactive duplicate pages, store a single compressed version.

For our test case, up to 89% of the allocated memory could be reclaimed with this algorithm, reducing the overall memory footprint of the three DD/OS instances to 6MB. See Figure 5.23 for an analysis of the example session.

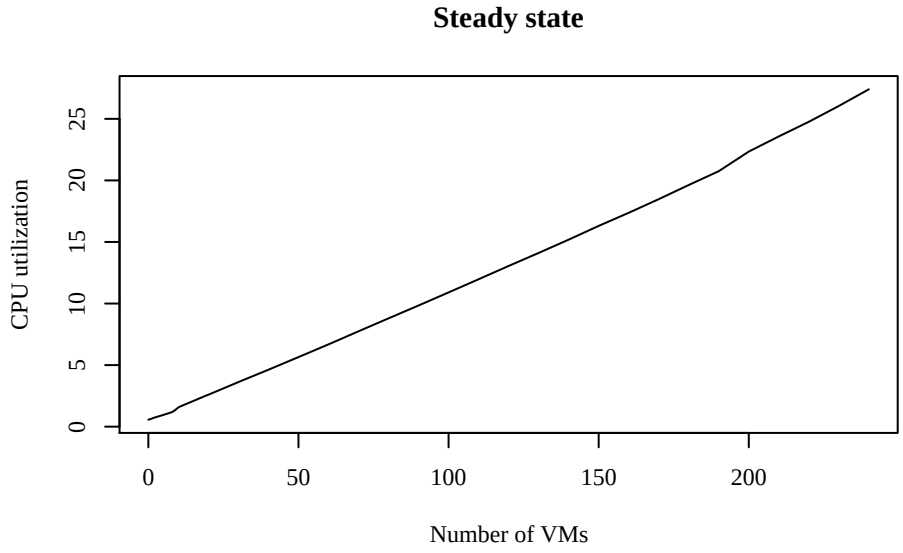



Figure 5.24: The incremental CPU utilization of idle VMs.

Without an IO-MMU, gray-box knowledge also enables DD/OS paging. For example, the memory of Linux's page map is never used for a DMA operation, and is thus reclaimable. Furthermore, the network and block DD/OS each had a contiguous 6.9 MB identical region in their page maps, suitable for sharing. The proxy modules could easily expose this information to the hypervisor's memory management logic.

### Idle processor consumption

When isolating device drivers in separate VMs, each VM will add slight overhead while running their idle loops, thus reducing the available CPU for the active work load. This is a concern for building systems from many DD/OS instances. Figure 5.24 shows the incremental CPU utilization of adding unused VMs to the system. They are all running their idle loops (which periodically wake to run house keeping tasks), but not performing useful work. For a handful of isolated drivers, the idle CPU overhead is affordable. The idle VMs were small, ran from RAM disks, had no devices, and had no memory reclamation.

### TLB and cache consumption

We can detect DD/OS activity indirectly by monitoring platform resource utilization. We observed TLB miss rates and L2-cache miss rates, using the CPU's performance monitors. We predicted that driver reuse would have higher miss rates than native Linux, thus representing more activity per driver operation, as well as more CPU utilization servicing the misses. We also predicted that driver reuse would have higher rates compared to user-level drivers.

We collected TLB miss rates for data and instructions (DTLB and ITLB), and L2-cache miss rates, for most of the benchmarks. The miss rates are for the entire system.

Table 5.2 and Table 5.3 list the cache miss rates for the Netperf send and receive benchmarks, respectively. The tables contain data for Linux 2.6.8.1 and Linux 2.6.9, and compare native Linux to the L4 user-level driver to the driver-reuse system. The miss rates are per millisecond.

We also plotted cache miss rates versus time for the PostMark benchmark. See Figure 5.25 for the DTLB misses, Figure 5.26 for the ITLB misses, and Figure 5.27 for the L2 misses.

The driver reuse system has significantly higher TLB miss rates than native Linux. It also exceeds the miss rates for the L4 user-level driver, but not significantly. Many of the TLB misses are due to the additional address-space switches inherent with user-level drivers, and which on x86 flush the TLB. Thus every time a driver client requires services from a user-level driver, the x86 system flushes the TLB twice (when scheduling the driver, and then when returning to the driver client); in contrast native Linux can lock the drivers' TLB entries thus avoiding TLB misses since the drivers live in a global kernel logical space. The driver-reuse system has a larger working-set footprint than the L4 driver (as described in Section 5.1.4), and thus more TLB misses than the L4 driver.

An operating system competes with its applications for space in the TLB and cache, despite that an OS exists to run applications. Thus a system based on driver reuse (or user-level drivers) will experience higher pressure on the limited TLB and cache, potentially reducing overall performance.

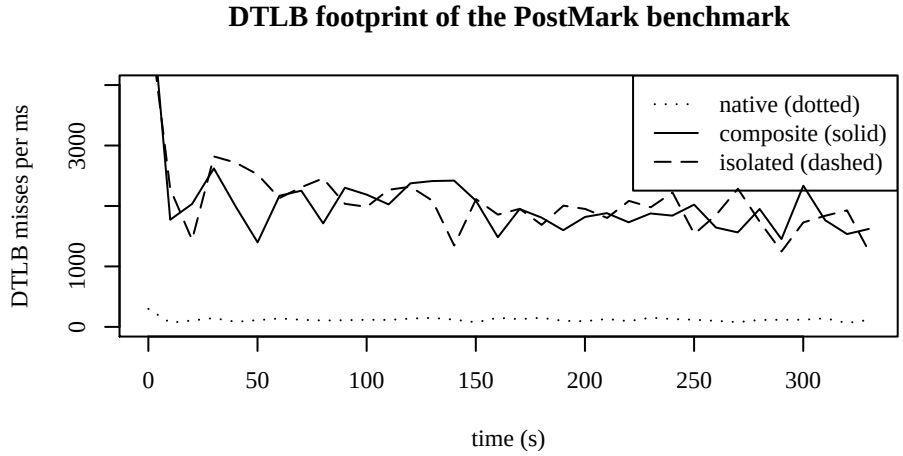



Figure 5.25: DTLB misses per millisecond, for the duration of the PostMark benchmark.

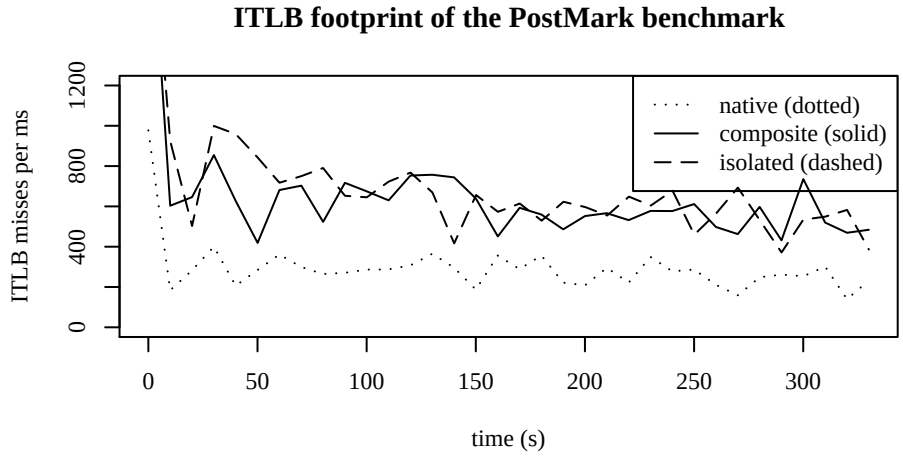



Figure 5.26: ITLB misses per millisecond, for the duration of the PostMark benchmark.

| Linux 2.6.8.1 | DTLB misses/ms | ITLB misses/ms | L2 misses/ms |
|---|---|---|---|
| native | 89.6 | 1.3 | 338 |
| L4 driver | 2190 | 455 | 266 |
| Driver reuse | 2450 | 551 | 312 |
| **Linux 2.6.9** | | | |
| native | 78.6 | 1.3 | 175 |
| L4 driver | 2270 | 477 | 270 |
| Driver reuse | 4300 | 1320 | 356 |

Table 5.2: Cache misses for Netperf send. The 95% confidence interval is at worst $\pm$ 2%

| Linux 2.6.8.1 | DTLB misses/ms | ITLB misses/ms | L2 misses/ms |
|---|---|---|---|
| native | 168 | 1.63 | 469 |
| L4 driver | 3190 | 569 | 1190 |
| Driver reuse | 5280 | 1490 | 682 |
| **Linux 2.6.9** | | | |
| native | 144 | 1.47 | 535 |
| L4 driver | 3180 | 599 | 1190 |
| Driver reuse | 5840 | 1600 | 822 |

Table 5.3: Cache misses for Netperf receive. For native Linux, the 95% confidence interval is $\pm$ 7.5%, while the others are at worst $\pm$ 2%.

### 5.1.5 Engineering effort

We have a single data point for analyzing productivity: the implementation of this thesis, which consisted of both engineering and exploration. We estimate the engineering effort in man hours and in lines of code.

The proxy modules and custom client device drivers for the block and network, along with the user-level VMM, involved two man months of effort, originally for Linux 2.4. The PCI support involved another week of effort.

We easily upgraded the 2.4 network proxy module for use in Linux 2.6, with minor changes. However the 2.4 block proxy module was incompatible with 2.6's internal API (Linux 2.6 introduced a new block subsystem). We thus wrote new block proxy and client device drivers for 2.6. This involved approximately another half man month of effort.

See Table 5.4 for an itemization of the lines of code. The table distinguishes between lines specific to the proxy modules added to the server, lines specific to the custom device drivers added to the client, and additional lines that are common (and are counted once).

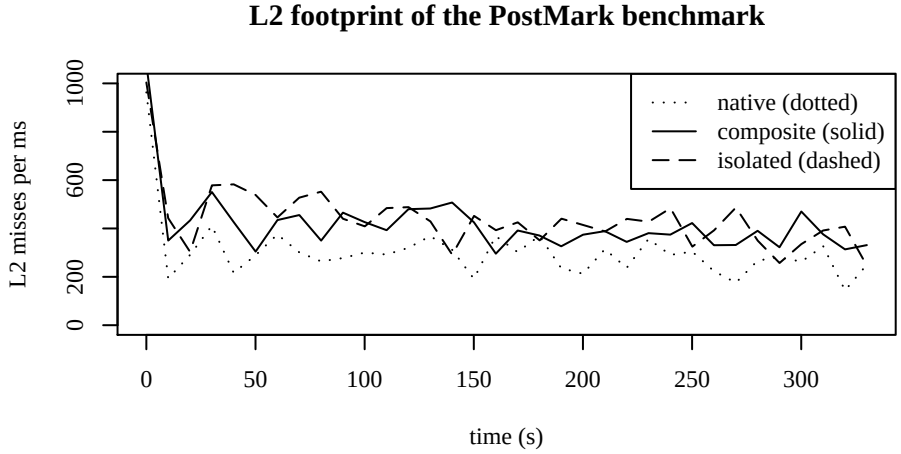



Figure 5.27: L2-cache misses per millisecond, for the duration of the PostMark benchmark.

| | server | client | common | total |
|---|---|---|---|---|
| network | 1152 | 770 | 244 | 2166 |
| block 2.4 | 805 | 659 | 108 | 1572 |
| block 2.6 | 751 | 546 | 0 | 1297 |
| PCI | 596 | 209 | 52 | 857 |
| common | 0 | 0 | 620 | 620 |
| total | 3304 | 2184 | 1024 | |

Table 5.4: Itemization of source lines of code used to implement our evaluation environment. Common lines are counted once.

The engineering effort enabled us to successfully reuse Linux device drivers with all of our lab hardware. The following drivers were tested: Intel gigabit, Intel 100 Mbit, Tulip (with a variety of Tulip compatible hardware), Broadcom gigabit, pcnet32, ATA and SATA IDE, and a variety of uniprocessor and SMP chipsets for Intel Pentium 3/4 and AMD Opteron processors.

## 5.2 Pre-virtualization

In analyzing pre-virtualization, we show whether it satisfies the goals presented in earlier chapters. Some of the goals to demonstrate: that it has runtime modularity, that it compares to para-virtualization for performance, that it offers a good solution for device emulation, and that it has low developer overhead.

The benchmark setups used identical configurations as much as possible, to ensure that any performance differences were the result of the techniques of virtualization. We compiled Linux with minimal feature sets, and configured it to use a 100Hz timer, and the XT-PIC interrupt controller (our APIC emulation is incomplete). Additionally, we used the slow legacy x86 `int` system call, as required by some virtualization environments.

The test machine was a 2.8GHz Pentium 4, constrained to 256MB of RAM, and ran Debian 3.1 from the local SATA disk. Our implementation is available from `http://l4ka.org/projects/virtualization/afterburn`.

### 5.2.1 Modularity

Modularity has two aspects: whether we easily support different hypervisors with runtime adaptation; and whether we can easily apply pre-virtualization to different guest operating systems.

To demonstrate runtime hypervisor modularity, we generated a Linux kernel binary that conforms to soft-layering principles. The binary is annotated, to help locate all of the soft-layering additions. And the binary has integrated scratch space, to provide rewrite space. As described in Section 4.3, the soft-layering includes instruction-level, behavioral, and structural elements. We then chose two hypervisor families, and wrote runtime infrastructure to support the pre-virtualized Linux binaries on both families. The hypervisors include Xen versions 2 and 3, and two versions of the L4Ka::Pistachio microkernel: the traditional API, and an API enhanced for virtualization. This runtime infrastructure is independent to the Linux binary — we combined them at boot time. We then ran the same pre-virtualized Linux kernel binary on all hypervisors, and on raw hardware. We present the corresponding benchmark data in the following sections.

To demonstrate guest OS modularity, we applied pre-virtualization to several versions of Linux: Linux 2.4.28, Linux 2.6.8.1, Linux 2.6.9, and Linux 2.6.16. We present some of the corresponding benchmark data in the following sections.

| Annotation type | Linux 2.6.9 | Linux 2.4.28 |
|---|---|---|
| instructions | 5181 | 3035 |
| PDE writes | 17 | 20 |
| PDE reads | 36 | 26 |
| PTE writes | 33 | 30 |
| PTE reads | 20 | 18 |
| PTE bit ops | 5 | 3 |
| DP83820 | 103 | 111 |

Table 5.5: The categories of annotations in our pre-virtualization implementation, and the number of annotations (including automatic and manual), for x86. *PDE* refers to page directory entries, and *PTE* refers to page table entries.

### 5.2.2 Code expansion

Pre-virtualization potentially introduces code expansion everywhere it emulates a sensitive instruction, which can degrade performance. We informally evaluate this cost for the Netperf receive benchmark. Although the Linux kernel has many sensitive instructions that we emulate (see Table 5.5), the Netperf benchmark executes only several of them frequently, as listed in Table 5.6 (the virtualization-assist module collected these statistics with its emulation code, using a special call to synchronize with the start and stop of the benchmark). We describe these critical instructions.

The `pushf` and `popf` instructions read and write the x86 flags register. Their primary use in the OS is to toggle interrupt delivery, and rarely to manipulate the other flag bits; OS code compiled with gcc invokes these instructions via inlined assembler, which discards the application flag bits, and thus we ignore these bits. It is sufficient to replace these instructions with a single `push` or `pop` instruction that directly accesses the virtual CPU, and to rely on heuristics for delivering pending interrupts.

The `cli` and `sti` instructions disable and enable interrupts. We replace them with a single bit clear or set instruction each, relying on heuristics to deliver pending interrupts.

The instructions for reading and writing segment registers translate into one or two instructions for manipulating the virtual CPU (when running in a flat segment mode).

The remaining instructions, `iret`, `hlt`, and `out`, expand significantly, but are also the least frequently executed.

The `iret` instruction returns from interrupt. Its emulation code checks for

| Instruction | Count | Count per interrupt |
|---|---|---|
| `cli` | 6772992 | 73.9 |
| `pushf` | 6715828 | 73.3 |
| `popf` | 5290060 | 57.7 |
| `sti` | 1572769 | 17.2 |
| write segment | 739040 | 8.0 |
| read segment | 735252 | 8.0 |
| port `out` | 278737 | 3.0 |
| `iret` | 184760 | 2.0 |
| `hlt` | 91528 | 1.0 |
| write ds | 369520 | 4.0 |
| write es | 369520 | 4.0 |
| read ds | 184760 | 2.0 |
| read es | 184760 | 2.0 |
| read fs | 182866 | 2.0 |
| read gs | 182866 | 2.0 |

Table 5.6: Execution profile of the most popular sensitive instructions during the Netperf receive benchmark for Linux 2.6.9. Each column lists the number of invocations, where the *count* column is for the entire benchmark.

pending interrupts, updates virtual CPU state, updates the `iret` stack frame, and checks for pending device activity. If it must deliver a pending interrupt or handle device activity, then it potentially branches to considerable emulation code. In the common case, its expansion factor is fairly small, as seen in the LMbench2 results for the null call (see Section 5.2.4).

The idle loop uses `hlt` to transfer the processor into a low power state. While this operation is not performance-critical outside a VM environment, it can penalize a VM system via wasted cycles which ought to be used to run other VMs. Its emulation code checks for pending interrupts, and puts the VM to sleep via a hypercall if necessary.

The `out` instruction writes to device ports, and thus has code expansion for the device emulation. If the port number is an immediate value, as for the XT-PIC, then the rewritten code directly calls the target device model. Otherwise the emulation code executes a table lookup on the port number. The `out` instruction costs over 1k cycles on a Pentium 4, masking the performance costs of the emulation code in many cases.

Our optimizations minimize code expansion for the critical instructions. In several cases, we substitute faster instructions for the privileged instructions (e.g., replacing `sti` and `cli` with bit-set and bit-clear instructions).

### 5.2.3 Device Emulation

We implemented a device emulation model for the network according to the descriptions in the prior sections. It provides to the guest OS high-performance virtualized access to the network, via a standard device driver—the DP83820.

We compared our performance against a custom device driver that runs in the client Linux VM, as described in the driver-reuse sections of this thesis. The benchmark reused the Linux e1000 network driver to control the actual device, running in a DD/OS. The proxy module running within the DD/OS emulated the DP83820 device model, and translated its device operations into the internal API of the Linux DD/OS.

#### Performance

We ran the Netperf benchmark to compare the performance of the custom client driver to the DP83820 pre-virtualized device emulation. Refer to Figure 5.2 and Figure 5.3 for Netperf-send performance. Refer to Figure 5.4 and Figure 5.5 for Netperf-receive performance. The pre-virtualized DP83820 driver had similar performance to the custom client driver.

#### Analysis

The DP83820 Linux device driver accesses the device registers from 114 different instruction locations (we determined the coverage by manually annotating Linux). The device has 64 memory-mapped registers, several of which are reserved or unused by Linux. During the Netperf benchmarks, only 8 of the 114 instruction locations were executed, and these eight locations accessed five device registers (for handling interrupts, adding transmit descriptors, and for reigniting receive and transmit engines). See Table 5.7 for invocation counts of the five device registers. The invocation counts are high, and easily show the benefit of pre-virtualization, which uses relatively cheap function calls to the virtualization-assist module to update device state, rather than the traps that accompany pure virtualization; and then it batches multiple packets when finally executing an expensive hypercall to process packets.

| register | receive count | packets per | send count | packets per |
|---|---|---|---|---|
| CR | 91845 | 7.8 | 741560 | 0.97 |
| ISR | 91804 | 7.8 | 81049 | 8.8 |
| IMR | 184091 | 3.9 | 243120 | 2.9 |
| IHR | 91804 | 7.8 | 81049 | 8.8 |
| TXDP | 91798 | 7.8 | 81039 | 8.8 |

Table 5.7: Execution profile of the active DP83820 device registers during the Netperf receive and send benchmarks, with total number of accesses for each register, plus the average number of data packets per register access. The benchmark sent a gigabyte of data on a gigabit link.

### 5.2.4 Performance

For our performance analysis, we demonstrate that pre-virtualization performs similarly to its alternatives. Thus we compare pre-virtualization to para-virtualization on several hypervisor environments. Additionally, we demonstrate that a pre-virtualized binary offers no overhead for running on raw hardware, and thus compare a pre-virtualized binary running on raw hardware (no hypervisor) to native Linux.

We study only the costs of pre-virtualization, independent of driver reuse. Thus the benchmarks all had direct access to the devices — there was no device-driver reuse with multiple collaborating virtual machines and proxy modules.

We compared two hypervisors (Xen version 2 and L4Ka::Pistachio) and several versions of Linux (2.4.28, 2.6.8.1, and 2.6.9). We lacked consistent support for a particular version of para-virtualized Linux on each hypervisor, and so do not compare between hypervisors, but only between pre-virtualization versus no pre-virtualization. We used Linux 2.4.28 and Linux 2.6.9 for Xen. We used Linux 2.6.8.1 for L4Ka::Linux.

We ran several benchmarks: a Linux kernel build (a macro benchmark), Netperf (a microbenchmark for networking), and LMbench2 (a microbenchmark for operating system metrics).

#### Linux kernel build

The kernel-build benchmark unarchived the Linux kernel source archive, and built a particular configuration. It heavily used the disk and CPU. It executed many processes, exercising `fork()`, `exec()`, the normal page fault handling code, and thus stressing the memory subsystem; and accessed many files and used pipes,

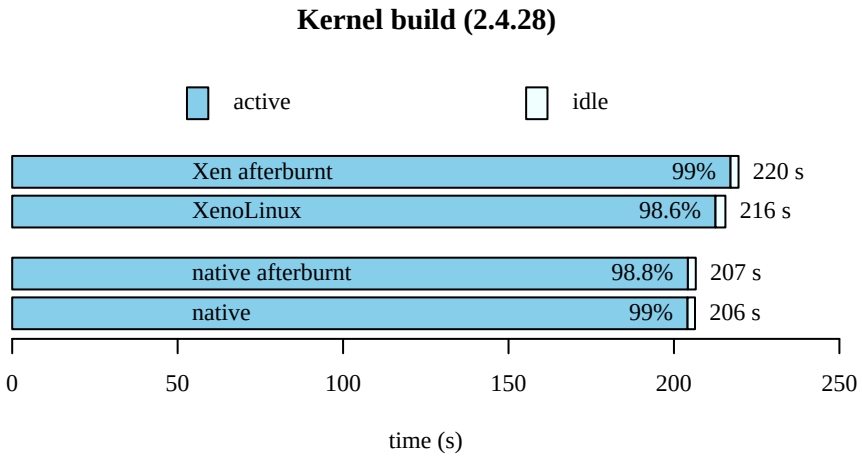



Figure 5.28: Results from the kernel-build benchmark, running on variations of Linux 2.4.28, with direct access to the devices. It compares the performance of para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt*) on the Xen hypervisor. It also compares a native Linux (*native*) to that of a pre-virtualized binary running on raw hardware (*native afterburnt*). The 95% confidence interval is at worst $\pm$ 0.2%

thus stressing the system-call interface. When running on Linux 2.6.9 on x86, the benchmark created around 4050 new processes, generated around 24k address-space switches (of which 19.4k were process switches), 4.56M system calls, 3.3M page faults, and between 3.6k and 5.2k device interrupts.

We measured and compared benchmark duration and CPU utilization. For Linux 2.4.28, see Figure 5.28. For Linux 2.6.8.1, see Figure 5.29. For Linux 2.6.9, see Figure 5.30. In all cases, pre-virtualization had similar performance to para-virtualization. For 2.4.28, pre-virtualization had 1.8% throughput degradation. For 2.6.8.1, pre-virtualization had a 1% throughput degradation. For 2.6.9, pre-virtualization had a 0.8% throughput degradation. And when running on raw hardware, the pre-virtualized binary demonstrated similar performance to native Linux.

The performance degradation for the pre-virtualized binary on Xen is due to more page-table hypercalls. The performance degradation of the pre-virtualized binary on L4 is due to fewer structural modifications compared to L4Ka::Linux.

### Netperf

The Netperf benchmark transferred a gigabyte of information via TCP/IP, at standard Ethernet packet size, with 256kB socket buffers. These are I/O-intensive benchmarks, producing around 82k device interrupts while sending, and 93k device interrupts while receiving (in total) — an order of magnitude more device interrupts than during the Linux kernel build. There were two orders of magnitude fewer system calls than for the kernel build: around 33k for send, and 92k for receive.

We measured throughput (expressed as a duration), and CPU utilization. For Linux 2.4.28, see Figures 5.31 and 5.32. For Linux 2.6.8.1, see Figures 5.33 and 5.34. For Linux 2.6.9, see Figures 5.35 and 5.36. In all cases, throughput was nearly identical, with some differences in CPU utilization. Our L4 system provides event counters which allow us to monitor kernel events such as interrupts, protection domain crossings, and traps caused by guest OSes. We found that the event-counter signature of the para-virtualized Linux on L4 nearly matched the signature of pre-virtualized Linux on L4.

### LMbench2

Table 5.8 summarizes the results from several of the LMbench2 micro benchmarks (updated from the original lmbench [MS96]).

As in the kernel-build benchmark, the higher overheads compared to para-virtualized Xen for `fork()`, `exec()`, and for starting `/bin/sh` on x86 seem to be due to an excessive number of hypercalls for page table maintenance.

Of interest is that pre-virtualized Xen has lower overhead for null system calls than para-virtualized Xen. This is due our architectural decision to handle interrupt-delivery race conditions off the fast path, within the interrupt handler, by rolling forward or backwards critical sections.

Several of the Xen metrics outperform native Linux. The results reported by Barham et al. [BDF+03] confirm this for para-virtualization. Potentially this is due to virtualization of expensive hardware instructions, where the virtualization substitutes lower-cost software emulation, such as interrupt masking and unmasking on the system-call path (see Section 5.2.2).

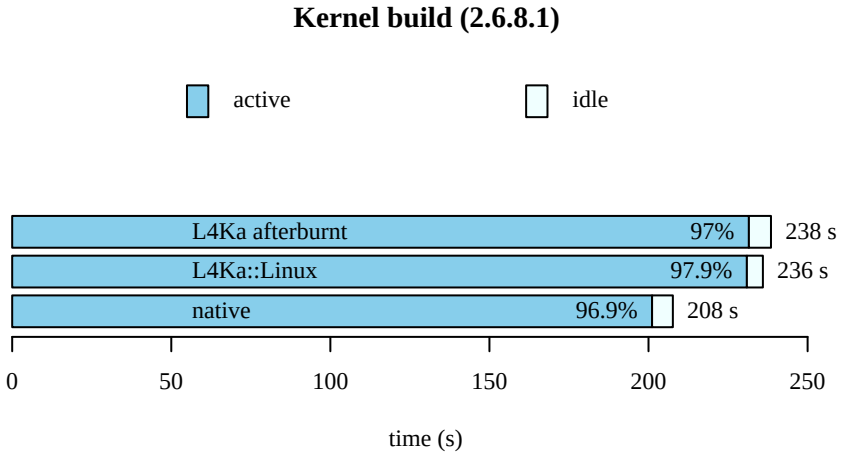



Figure 5.29: The kernel-build benchmark with variations of Linux 2.6.8.1 and direct device access. Compares the performance of para-virtualization (*L4Ka::Linux*) to pre-virtualization (*L4Ka afterburnt*) for the L4 microkernel, and includes the performance of native Linux (*native*). The 95% confidence interval is at worst $\pm$ 1%

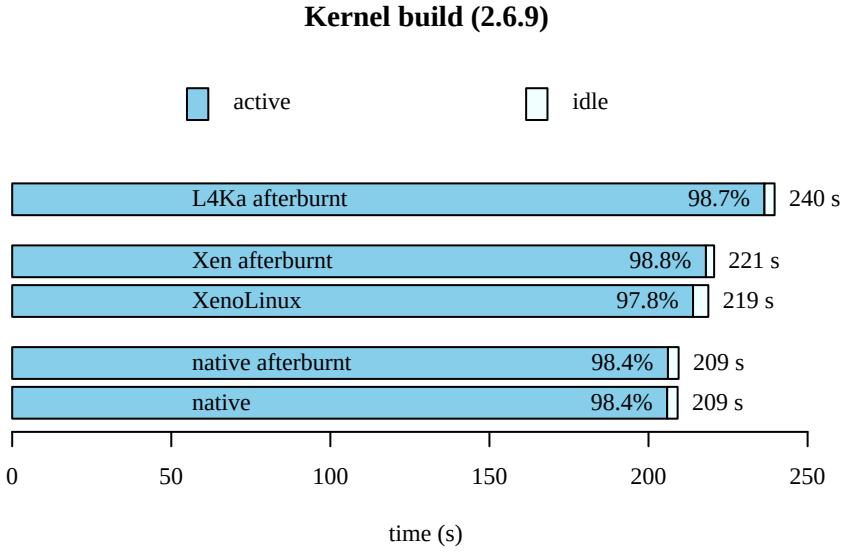



Figure 5.30: The kernel-build benchmark with variations of Linux 2.6.9 and direct device access. Compares the performance of para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt*) for the Xen hypervisor, and the performance of a native Linux (*native*) to a pre-virtualized binary running on raw hardware (*native afterburnt*). Also includes the performance of running pre-virtualized Linux on L4 (*L4Ka afterburnt*). The 95% confidence interval is at worst $\pm$ 0.5%

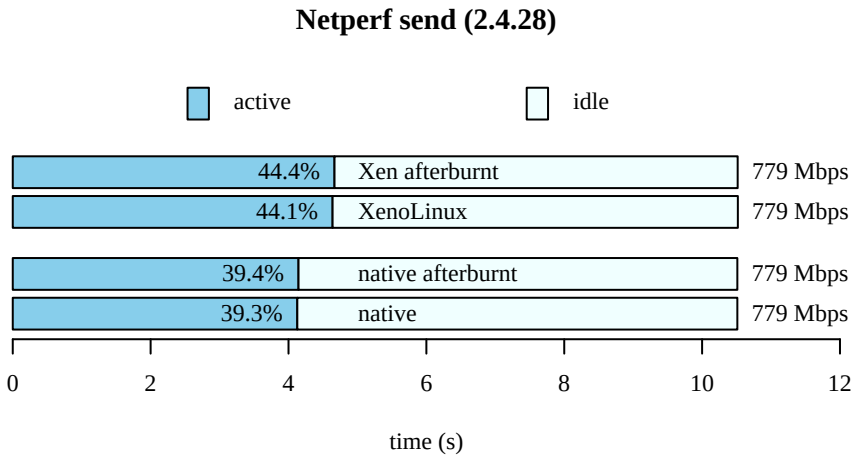



Figure 5.31: The Netperf-send benchmark running variations of Linux 2.4.28 with direct device access. It compares the performance of para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt*), and the performance of native Linux (*native*) to a pre-virtualized binary running on raw hardware (*native afterburnt*). The 95% confidence interval is at worst $\pm$ 0.3%

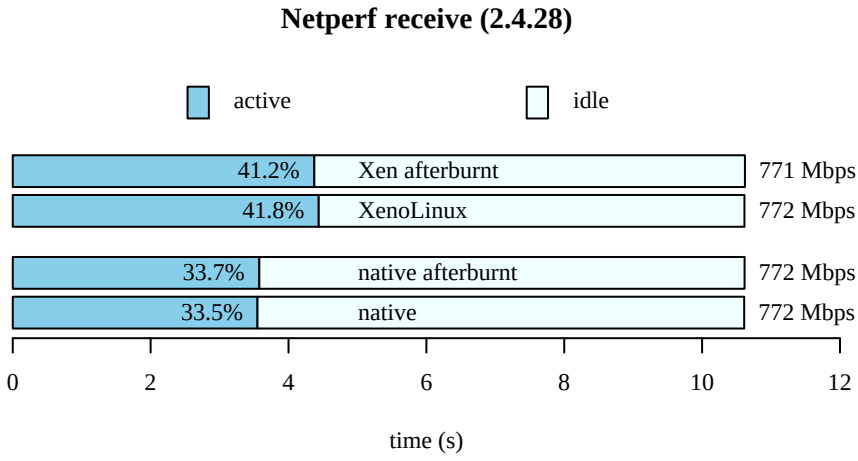



Figure 5.32: Results from the Netperf-receive benchmark running variations of Linux 2.4.28 with direct device access. It compares the performance of para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt*), and the performance of native Linux (*native*) to a pre-virtualized binary running on raw hardware (*native afterburnt*). The 95% confidence interval is at worst $\pm$ 0.2%

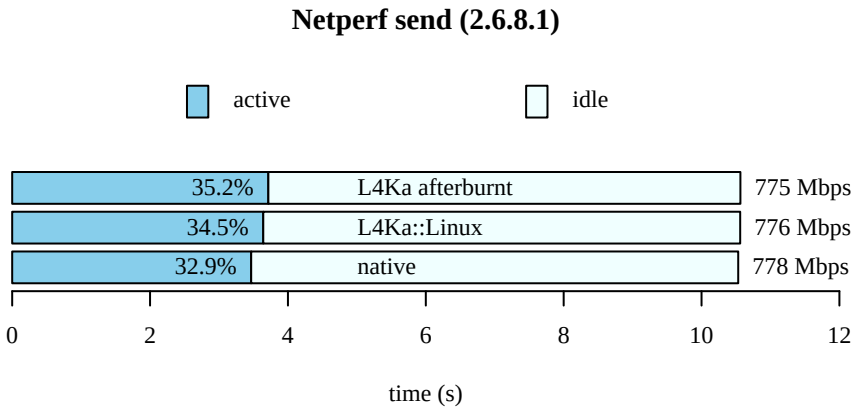



Figure 5.33: The Netperf-send benchmark running variations of Linux 2.6.8.1 with direct device access. It compares para-virtualization (*L4Ka::Linux*) to pre-virtualization (*L4Ka afterburnt*), and to native Linux. The 95% confidence interval is at worst $\pm$ 0.3%

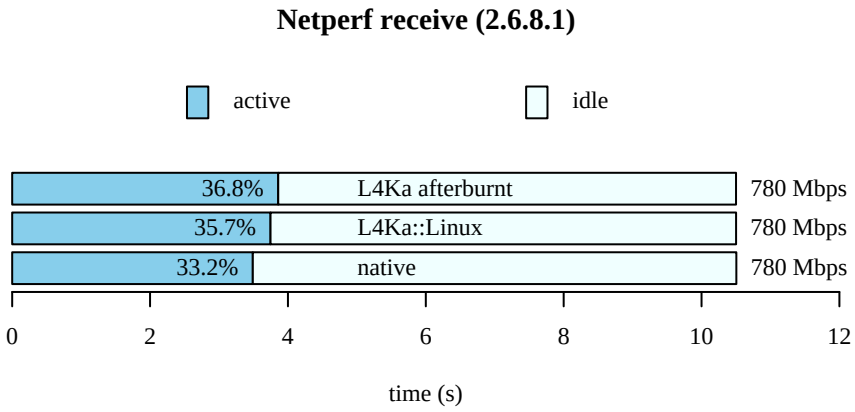



Figure 5.34: The Netperf-receive benchmark running variations of Linux 2.6.8.1 with direct device access. It compares para-virtualization (*L4Ka::Linux*) to pre-virtualization (*L4Ka afterburnt*), and to native Linux. The 95% confidence interval is at worst $\pm$ 0.2%

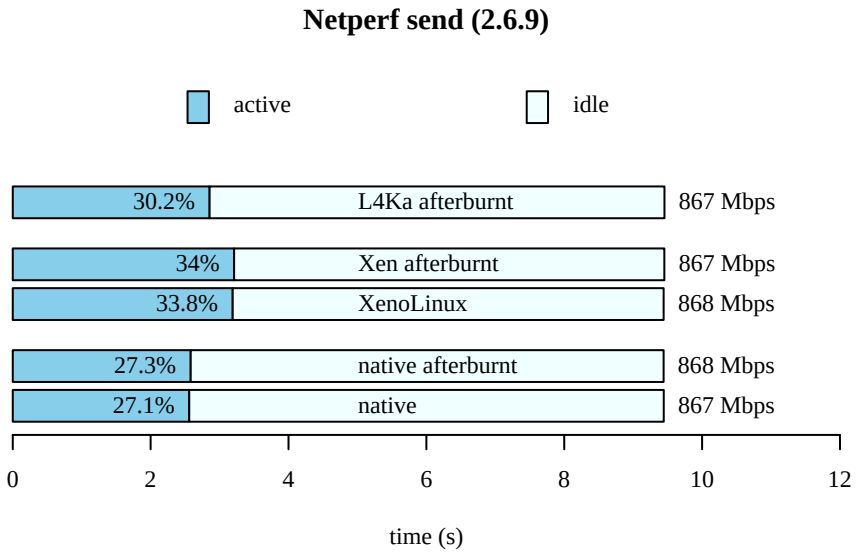



Figure 5.35: Netperf-send benchmark, Linux 2.6.9, with direct device access. Compares para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt* and *L4Ka afterburnt*), and native Linux (*native*) to a pre-virtualized binary on raw hardware (*native afterburnt*). The 95% confidence interval is at worst $\pm$ 0.2%

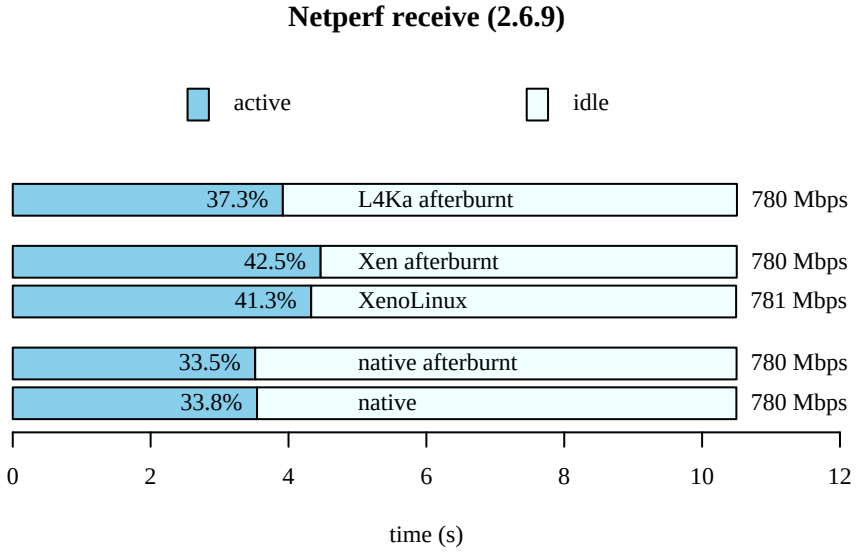



Figure 5.36: Netperf-receive benchmark, Linux 2.6.9, with direct device access. Compares para-virtualization (*XenoLinux*) to pre-virtualization (*Xen afterburnt* and *L4Ka afterburnt*), and native Linux (*native*) to a pre-virtualized binary on raw hardware (*native afterburnt*). 95% confidence interval is at worst $\pm$ 1.0%.

| type | null call | null I/O | stat | open close | sig inst | sig hndl | fork | exec | sh |
|---|---|---|---|---|---|---|---|---|---|
| raw | 0.46 | 0.53 | 1.34 | 2.03 | 0.89 | 2.93 | 77 | 310 | 5910 |
| NOP | 0.46 | 0.52 | 1.40 | 2.03 | 0.91 | 3.19 | 83 | 324 | 5938 |
| Xeno | 0.45 | 0.52 | 1.29 | 1.83 | 0.89 | 0.97 | 182 | 545 | 6711 |
| pre | 0.44 | 0.50 | 1.37 | 1.82 | 0.89 | 1.70 | 243 | 700 | 7235 |

Table 5.8: Partial LMbench2 results for Linux 2.6.9 (of benchmarks exposing virtualization overheads) in microseconds, smaller is better. *Raw* is native Linux on raw hardware, *NOP* is pre-virtualized Linux on raw hardware, *Xeno* is XenoLinux, and *pre* is pre-virtualized Linux on Xen. The 95% confidence interval is at worst $\pm$ 2%

### 5.2.5 Engineering effort

The first virtualization-assist module supported x86 and the L4 microkernel, and provided some basic device models (e.g., the XT-PIC). The x86 front-end, L4 back-end, device models, and initial assembler parser were developed over three person months. The Xen virtualization-assist module became functional with a further one-half person month of effort. Optimizations and heuristics involved further effort.

Table 5.9 shows the source code distribution for the individual x86 virtualization-assist modules and shared code for each platform. The DP83820 network device model is 1055 source lines of code, compared to 958 SLOC for the custom virtual network driver. They are very similar in structure since the DP83820 uses producer-consumer rings; they primarily differ in their interfaces to the guest OS.

In comparison to our past experience applying para-virtualization to Linux 2.2, 2.4, and 2.6 for the L4 microkernel, we observe that the effort of pre-virtualization is far less, and more rewarding. The Linux code was often obfuscated (e.g., behind untyped macros) and undocumented, in contrast to the well-defined and well-documented x86 architecture against which we wrote the virtualization-assist module. The pre-virtualization approach has the disadvantage that it must emulate the platform devices; occasionally they are complicated state machines.

After completion of the initial infrastructure, developed while using Linux 2.6.9, we pre-virtualized Linux 2.4 in a few hours, so that a single binary could boot on both the x86 Xen and L4 virtualization-assist modules.

In both Linux 2.6 and 2.4 we applied manual annotations (we had not yet written the C source code transformation tool), relinked the kernel, added DMA translation support for direct device access, and added L4 performance hooks, as

| Type | Headers | Source |
|---|---|---|
| Common | 686 | 746 |
| Device | 745 | 1621 |
| x86 front-end | 840 | 4464 |
| L4 back-end | 640 | 3730 |
| Xen back-end | 679 | 2753 |

Table 5.9: The distribution of code for the x86 virtualization-assist modules, expressed as source lines of code, counted by SLOCcount.

| Type | Linux 2.6.9 | Linux 2.4.28 |
|---|---|---|
| Device and page table | 52 | 60 |
| Kernel relink | 18 | 21 |
| Build system | 21 | 16 |
| DMA translation hooks | 53 | 26 |
| L4 performance hooks | 103 | 19 |
| Loadable kernel module | 10 | n/a |
| Total | 257 | 142 |

Table 5.10: The number of lines of manual annotations, functional modifications, and performance hooks added to the Linux kernels.

described in Table 5.10, totaling 257 lines for Linux 2.6.9 and 142 lines for Linux 2.4.28. The required lines of modifications without support for pass-through devices and L4-specific optimizations are 91 and 97 respectively.

In contrast, in Xen [BDF+03], the authors report that they modified and added 1441 sources lines to Linux and 4620 source lines to Windows XP. In L4Linux [HHL+97], the authors report that they modified and added 6500 source lines to Linux 2.0. Our para-virtualized Linux 2.6 port to L4, with a focus on small changes, still required about 3000 modified lines of code [LUSG04].

## 5.3 Summary

We studied the behavior and performance of driver reuse in several scenarios. We reused Linux drivers, and compared their performance and overheads to native Linux drivers. Since we run the drivers in VMs at user level, we also compared the performance of network driver reuse to a user-level driver, to clarify sources of overheads. The results show that driver reuse achieves comparable performance to native drivers, as long as the workloads are I/O bound, as driver reuse consumes more CPU, memory, cache, and TLB.

We also studied the performance of pre-virtualization, by comparing to the

highest performing approach for VM construction, para-virtualization, for two different environments: L4 and Xen. We implemented pre-virtualization for both, and achieved the modularity of traditional virtualization, by enabling a single Linux executable to run on raw hardware, Xen, and L4. We found that the performance of pre-virtualization is comparable to para-virtualization, while reducing engineering effort.

# Chapter 6

# Conclusion

Device drivers account for a majority of today's operating system kernel code (in Linux 2.4 on x86, 70% of 1.6 million lines of kernel code implement device support [CYC+01]). A new operating system endeavor that aims for even a reasonable breadth of device support faces a major development and testing effort, or must support and integrate device drivers from a driver-rich OS (e.g., Linux or Windows). The availability of many drivers solely in binary format prevents known approaches based on source-code reuse. We have proposed a pragmatic approach for full reuse and strong isolation of legacy device drivers.

Our solution runs the reused device drivers with their original operating systems in virtual machines, along with minor infrastructure for interfacing with the rest of the system. By supporting the drivers with their original systems, we avoid implementing high-level replacement code, and instead supply low-level virtualization code for managing bulk resources — this reduces development effort, and allows us to focus on the highly-documented platform interface, thus supporting drivers from multiple operating systems with a single solution. Our solution insulates the drivers in a virtual machine, which can increase system dependability, and permits contradictory architectures for the new operating system and the driver's operating system. The merits of the system also limit its applicability:

- the virtual machines, and the drivers' operating systems running within them, can consume more CPU and memory resources compared to alternative driver approaches — this may make the solution undesirable for low-resource systems;
- running multiple operating systems adds additional management, monitoring, debugging, testing, and logging complexity — some operating systems

will provide less hassle than others for reusing their drivers;

- the reused drivers need to offer all of the functionality desired by the new system;
- and the CPU must be able to run a virtual machine.

Virtual machines have a performance overhead, which many have addressed via para-virtualization. But para-virtualization adds developer burden (which we are trying to reduce) and forfeits modularity (which we desire for running different types of operating systems for driver reuse). We have thus proposed the pre-virtualization technique for building virtual machines, which combines the performance of para-virtualization with the modularity of traditional virtualization. We still apply the performance enhancements of para-virtualization, but guided by a set of principles that preserves modularity, and with automation. Our solution particularly helps the x86 architecture, which poses problems to efficient virtualization. Yet we have limitations for our approach:

- we assume that we can automate the replacement of virtualization-sensitive instructions — it is possible to write source code that interferes with the automation (e.g., no abstractions to identify sensitive-memory operations), but we expect that most operating systems are well structured and thus suitable for our automation;
- we assume that we can find heuristics that match the performance of traditional para-virtualization modifications — this may not always be the case, but so far we have been successful;
- and we require access to the source code of the OS.

Additionally, many processors now include hardware-virtualization acceleration that may reduce the benefit of our approach — we did not evaluate these hardware extensions and thus have not demonstrated that our approach is universally beneficial, but Adams and Agesen [AA06] demonstrated that executing virtualization logic within a guest OS improves the performance of hardware virtualization, which could be easily implemented via pre-virtualization.

## 6.1 Contributions of this work

This dissertation proposes solutions for reusing device drivers in a manner to promote novel operating system construction. We focus on insulating the novel oper-

ating system from the invariants and faults of the reused device drivers. Primary and supporting contributions of this dissertation:

- We propose a set of principles for facilitating driver reuse (Section 3.1).
- We run the reused drivers, along with their reuse infrastructure, in virtual machines with the drivers' original operating systems. We describe solutions for full virtualization and para-virtualization.
- We can support binary-only drivers via full virtualization or pre-virtualization (for pre-virtualization, the driver binaries must be pre-virtualized by their vendors).
- We present an architecture for isolating each driver in a dedicated VM, yet with the ability to support each other (e.g., one driver uses a bus driver), even if from different operating systems.
- We show how to integrate and interface the reused drivers with the client operating system.
- We describe resource management techniques.
- We present a reference implementation for reusing Linux device drivers, on the L4Ka::Pistachio microkernel.

To improve the quality of our driver-reuse solution, we also propose the technique of *pre-virtualization* for constructing virtual machines, as an alternative to traditional virtualization and para-virtualization. Contributions of this dissertation:

- We show how to achieve the modularity of traditional virtualization with performance approaching that of para-virtualization.
- Our solution is based on para-virtualization, but we propose a set of principles called *soft layering* for guiding the application of para-virtualization to an operating system. These principles are:
    1. it must be possible to degrade to the platform interface, by ignoring the para-virtualization enhancements (thus permitting execution on native hardware and hypervisors that lack support for soft layering);
    2. the interface must flexibly adapt at runtime to the algorithms that competitors may provide (thus supporting arbitrary hypervisor interfaces without pre-arrangement).

- We describe how to include the major features of para-virtualization: instruction-level modifications, structural modifications, and behavioral modifications.
- Para-virtualization has a substantial development cost, and we show how to automate most modifications to significantly reduce the development burden.
- We present a solution for modular, high-speed device virtualization, based on our soft layering principles. This solves one of the banes of virtualization: having to ship hypervisor-specific drivers to the guest operating system for performance.
- We show how to design the modular runtime support of our approach, particularly with a focus on reliability and performance.
- We provide reference implementations for running Linux on the Xen hypervisor and the L4Ka::Pistachio microkernel. These are available for download at `http://l4ka.org/projects/virtualization/afterburn`.

We demonstrated driver reuse and pre-virtualization by running Linux drivers in virtual machines constructed from pre-virtualization, on a research operating system based on the L4Ka::Pistachio microkernel.

## 6.2 Future work

We see several areas for future work:

We would like to see future virtualization hardware that enables virtualization logic to execute *within* the virtual machine, particularly to virtualize memory-mapped devices. Finding the best techniques requires some exploration.

The applicability of our approach to general legacy reuse needs more exploration.

Although we proposed solutions for using traditional virtualization, we did not validate them. Nor did we test and benchmark the latest hardware that supports virtualization acceleration, particularly for DMA redirection.

We also did not investigate alternative approaches for building the proxy modules that reuse device drivers, particularly for implementing them as more portable user-level code of the guest operating system.

We used only Linux as a guest OS, and did not explore alternative kernels. Future work should attempt mixing different kernels, and explore the consequences of reusing drivers from different donor systems.

# Bibliography


[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *The 12th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2006. doi:10.1145/1168857.1168860.

[AAD+02] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, August 2002.

[AAH+06] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. VMI: An interface for paravirtualization. In *Proceedings of the Linux Symposium*, pages 363–378, Ottawa, Canada, July 2006.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–117, April 1991. doi:10.1145/106972.106984.

[AMD05] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual*, May 2005.

[BCG73] Jeffrey P. Buzen, Peter P. Chen, and Robert P. Goldberg. A note on virtual machines and software reliability. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 228–235, Cambridge, MA, March 1973. doi:10.1145/800122.803963.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, October 2003. doi:10.1145/945445.945462.

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 143–156, Saint Malo, France, October 1997. doi:10.1145/268998.266672.

[BFHW75] 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 Fifth Symposium on Operating System Principles*, pages 82–88, Austin, TX, November 1975. doi:10.1145/800213.806525.

[Bro95] Kraig Brockschmidt. *Inside OLE (2nd ed.)*. Microsoft Press, Redmond, WA, 1995.

[BS04] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. Technical Report CMU-CS-04-113, Carnegie Mellon University, Pittsburgh, PA, February 2004.

[CCB99] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[CF01] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, pages 125–132, Elmau/Oberbayern, Germany, May 2001. IEEE Computer Society Press. doi:10.1109/HOTOS.2001.990072.

[CIM97] Compaq, Intel, and Microsoft. *Virtual Interface Architecture Specification, Draft Revision 1.0*, December 1997.

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In

*Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, San Francisco, CA, December 2004.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 171–180, Orcas Island, WA, December 1985. doi:10.1145/323647.323645.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems*, page 133, Elmau/Oberbayern, Germany, May 2001. doi:10.1109/HOTOS.2001.990073.

[Coo83] Geoffrey Howard Cooper. An argument for soft layering of protocols. Technical Report TR-300, Massachusetts Institute of Technology, Cambridge, MA, April 1983.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Canada, October 2001. doi:10.1145/502034.502042.

[DD67] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in Multics. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 12.1 – 12.8, Gatlinburg, TN, October 1967. doi:10.1145/800001.811668.

[Dij68] Edsger Wybe Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968. doi:10.1145/363095.363143.

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[dPSR96] François Barbou des Places, Nick Stephen, and Franklin D. Reynolds. Linux on the OSF Mach3 microkernel. In *Conference on Freely Distributable Software*, Boston, MA, February 1996.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 75–88, Seattle, WA, November 2006.

[EG04] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In P. Yew and J. Xue, editors, *Advances in Computer Systems Architecture: 9th Asia-Pacific Computer Systems Architecture Conference*, volume 3189, Beijing, China, September 2004.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *The Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. doi:10.1109/HOTOS.1995.513459.

[ERW05] Úlfar Erlingsson, Tom Roeder, and Ted Wobber. Virtual environments for unreliable extensions — a VEXE'DD retrospective. Technical Report MSR-TR-05-82, Microsoft Research, Redmond, WA, June 2005. `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-82.pdf`.

[ES03] Hideki Eiraku and Yasushi Shinjo. Running BSD kernels as user processes by partial emulation and rewriting of machine instructions. In *Proceedings of BSDCon '03*, San Mateo, CA, September 2003.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Saint-Malo, France, October 1997. doi:10.1145/268998.266642.

[FGB91] A. Forin, D. Golub, and Brian N. Bershad. An I/O system for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163–176, Monterey, CA, November 1991.

[FHN+04a] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, August 2004.

[FHN+04b] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, October 2004.

[GD96] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 65–74, San Diego, CA, January 1996.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, et al. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000. doi:10.1145/566726.566751.

[Gol74] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6), 1974.

[GPC+03] 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 ACM Symposium on Operating System Principles*, Bolton Landing, NY, October 2003. doi:10.1145/945445.945464.

[GSRI93] David B. Golub, Guy G. Sotomayor, Jr., and Freeman L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proceedings of the USENIX Mach III Symposium*, pages 153–171, Sante Fe, NM, April 1993.

[Han99] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 73–86, New Orleans, LA, February 1999.

[HAW+01] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg, and

Michael Stumm. Position summary: Supporting hot-swappable components for system software. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, May 2001. doi:10.1109/HOTOS.2001.990086.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a highly dependable operating system. In *Proceedings of the Sixth European Dependable Computing Conference*, pages 3–12, October 2006. doi:EDCC.2006.7.

[HBG+07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 41–50, June 2007. doi:10.1109/DSN.2007.46.

[HBS02] Hans-Jörg Höxer, Kerstin Buchacker, and Volkmar Sieh. Implementing a user-mode Linux with minimal changes from original kernel. In *Proceedings of the 9th International Linux System Technology Conference*, pages 72–82, Cologne, Germany, September 2002.

[HHF+05] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the First International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, CA, December 2005. doi:10.1109/COLCOM.2005.1651218.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu$-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 66–77, Saint-Malo, France, October 1997. doi:10.1145/268998.266660.

[HLM+03] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08-Juli-2003, Technische Universität Dresden, Dresden, Germany, July 2003.

[HLP+00] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st USENIX Workshop on Industrial Experiences with Systems Software*, pages 31–38, San Diego, CA, October 2000.

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *The 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[IBM05] IBM. *PowerPC Operating Environment Architecture, Book III*, 2005.

[Int05a] Intel Corp. *Intel Vanderpool Technology for Intel Itanium Architecture (VT-i) Preliminary Specification*, 2005.

[Int05b] Intel Corporation. *Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification*, 2005.

[Int06] Intel Corporation. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual — 82540EP/EM, 82541xx, 82544GC/EI, 82545GM/EM, 82546GB/EB, and 82547xx*, June 2006.

[Int07] Intel Corporation. *Intel Virtualization Technology for Directed I/O Architecture Specification*, May 2007.

[JKDC05] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Brighton, UK, October 2005. doi:10.1145/1095809.1095820.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 71–84, June 2003.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–15, Annaheim, CA, April 2005.

[KM02] Simon Kågström and Rickard Molin. A device driver framework for multiserver operating systems with untrusted memory servers. Master's thesis, Lund University, August 2002.

[KZB+91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, November 1991. doi:10.1109/32.106971.

[LBB+91] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a $\mu$-kernel based OS. *ACM SIGOPS Operating Systems Review*, 25(2):51–62, April 1991. doi:10.1145/122120.122124.

[LCFD+05] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[Les02] Ben Leslie. Mungi device drivers. BE Thesis, University of New South Wales, November 2002.

[LFDH04] Ben Leslie, Nicholas FitzRoy-Dale, and Gernot Heiser. Encapsulated user-level device drivers in the mungi operating system. In *Proceedings of the Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, Australia, January 2004.

[LH03] Ben Leslie and Gernot Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School of Computer Science and Engineering, UNSW, March 2003.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 175–188, Asheville, NC, December 1993. doi:10.1145/168619.168633.

[Lie95a] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical

Report 933, GMD — German National Research Center for Information Technology, November 1995.

[Lie95b] Jochen Liedtke. On $\mu$-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, December 1995. doi:10.1145/224057.224075.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996. doi:10.1145/234215.234473.

[LU04] Joshua LeVasseur and Volkmar Uhlig. A sledgehammer approach to reuse of legacy device drivers. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 131–136, Leuven, Belgium, September 2004. doi:10.1145/1133572.1133617.

[LUE+99] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, and Yoonho Park. How to schedule unlimited memory pinning of untrusted processes, or, provisional ideas about service-neutrality. In *7th Workshop on Hot Topics in Operating Systems*, pages 153–159, Rio Rico, AR, March 1999. doi:10.1109/HOTOS.1999.798393.

[LUSG04] 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*, pages 17–30, San Francisco, CA, December 2004.

[Mac79] R. A. MacKinnon. The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines. *IBM Systems Journal*, 18(1):18–46, 1979.

[Mar99] Kevin Thomas Van Maren. The Fluke device driver framework. Master's thesis, University of Utah, December 1999.

[MC04] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimized paravirtualization for the Itanium Processor Family. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, page 6, San Jose, CA, May 2004.

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, May 2006.

[MD74] Stuart E. Madnick and John J. Donovan. *Operating Systems*, pages 549–563. McGraw-Hill Book Company, 1974.

[MHSH01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. RTLinux with address spaces. In *Proceedings of the 3rd Real-Time Linux Workshop*, Milano, Italy, November 2001.

[Mol03] Ingo Molnar. 4g/4g split on x86, 64 GB RAM (and more) support. The linux-kernel mailing list, available at `http://lwn.net/Articles/39283/`, July 2003.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 279–294, San Diego, CA, January 1996.

[MS00] Robert Meushaw and Donald Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 9, Fall 2000.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. doi:10.1145/361598.361623.

[PBB+02] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, U.C. Berkely Computer Science, March 2002.

[PG73] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Proceedings of the Fourth Symposium on Operating System Principles*, Yorktown Heights, New York, October 1973.

[PGR06] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks.

In *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, San Jose, CA, May 2006.

[PPTH72] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield. Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2):99–130, 1972.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: a predicated-$LL(k)$ parser generator. *Software—Practice & Experience*, 25(7):789–810, July 1995.

[PS75] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975. doi:10.1145/360881.360913.

[RI00] 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 9th USENIX Security Symposium*, Denver, Colorado, August 2000. USENIX.

[RN93] D. Stuart Ritchie and Gerald W. Neufeld. User level IPC and device management in the Raven kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, pages 111–126, San Diego, CA, September 1993.

[Rog97] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, USA, 1997.

[RPC97] *DCE 1.1: Remote Procedure Call*, chapter Appendix A. The Open Group, 1997. `http://www.opengroup.org/onlinepubs/9629399/apdxa.htm`.

[SABL04] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 1–16, San Francisco, CA, December 2004.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. doi:10.1145/361011.361067.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, Bolton Landing, NY, October 2003. doi:10.1145/1047915.1047919.

[SCP+02] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[Sha01] Jonathan Shapiro. User mode drivers in EROS 2.0. Design note at `http://eros-os.org/design-notes/UserDrivers.html`, 2001.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984. doi:10.1145/357401.357402.

[Sun06] Sun Microsystems. *UltraSPARC Virtual Machine Specification (The sun4v architecture and Hypervisor API specification)*, January 2006.

[SVJ+05] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert van Doorn, John Linwood Griffin, and Stefan Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM T.J. Watson Research Center, Yorktown Heights, NY, February 2005.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[TCP81] Transmission control protocol. RFC 793, Information Sciences Institute, September 1981.

[TKH05] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification – now! In *The 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, June 2005.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*, pages 362–364. Prentice Hall, 2002.

[UDS+02] Volkmar Uhlig, Uwe Dannowski, Espen Skoglund, Andreas Haeberlen, and Gernot Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-01, Fakultät für Informatik, Universität Karlsruhe, 2002.

[UK98] Ronald C. Unrau and Orran Krieger. Efficient sleep/wake-up protocols for user-level IPC. In *Proceedings of the International Conference on Parallel Processing*, pages 560–569, Minneapolis, MN, August 1998. doi:10.1109/ICPP.1998.708530.

[ULSD04] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 2004.

[VMI06] VMware, `http://www.vmware.com/vmi`. *Virtual Machine Interface Specification*, 2006.

[VMw03] VMware. *VMware ESX Server I/O Adapter Compatibility Guide*, January 2003.

[Wal02] Carl Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, December 2002.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, MA, December 2002. doi:10.1145/844128.844147.

[Yan07] Yaowei Yang. Automating pre-virtualization for memory objects. Master's thesis, University of Karlsruhe, Germany, February 2007.

[You73] Carl J. Young. Extended architecture and hypervisor performance. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 177–183, Cambridge, MA, March 1973. doi:10.1145/800122.803956.

[ZBG+05] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 2–12, Chicago, IL, June 2005. doi:10.1145/1064979.1064983.

[ZCA+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 45–60, Seattle, WA, November 2006.