Pre-virtualization: soft layering for virtual machines

Joshua LeVasseur[†] Volkmar Uhlig[§†] Yaowei Yang[†] Matthew Chapman[‡¶]
Peter Chubb[‡¶] Ben Leslie[‡¶] Gernot Heiser[‡¶]
[†]University of Karlsruhe, Germany [§]IBM T. J. Watson Research Center, NY
[‡]National ICT Australia [¶]University of New South Wales, Australia

Abstract

Despite its current popularity, para-virtualization has an enormous cost. Its deviation from the platform architecture abandons many of the benefits of traditional virtualization: stable and well-defined platform interfaces, hypervisor neutrality, operating system neutrality, and upgrade neutrality — in sum, modularity. Additionally, para-virtualization has a significant engineering cost. These limitations are accepted as inevitable for significantly better performance, and for the ability to provide virtualization-like behavior on non-virtualizable hardware such as x86.

Virtualization and its modularity solve many systems problems, and when combined with the performance of para-virtualization become 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 call* soft layering. *Additionally, our approach is highly automated and thus reduces the implementation and maintenance burden of para-virtualization, which is especially useful for enabling obsoleted operating systems. We demonstrate soft layering on x86 and Itanium: we can load a single Linux binary on a variety of hypervisors (and thus substitute virtual machine environments and their enhancements), while achieving essentially the same performance as para-virtualization with less effort.*

1. Introduction

Although many hypervisor implementers strive to build high-performance virtual machine (VM) environments, the constraints for supporting commodity operating systems are enormous and force costly optimizations (e.g., VMware's runtime binary translation [1]). Many have proposed to modify the operating system (OS) for co-design with the hypervisor, i.e., para-virtualization [27], to improve performance and correctness; the possibilities seem unlimited, but the cost has been the emergence of many para-virtualization projects with incompatible and irreconcilable architectures, yet overlapping maintenance efforts for modifications to the guest OSes, and customer lock-in. By using specialized interfaces rather than the neutral machine interface, para-virtualization discards the modularity of traditional virtual machines. Modularity via the neutral machine interface is a key ingredient of virtualization's benefits. It enables a guest OS to run on hypervisors with substantially different architectures, to achieve runtime hypervisor upgrades, and the vital capability of VMs to run obsoleted OSes alongside modern OSes. Modularity permits OS enhancements written outside the OS's kernel community to be added in layers [6], remaining independent of fast-paced changes of kernel internals. Modularity supports proliferation of hypervisors, and stackable enhancements via recursive virtual machine construction.

We show how to add modularity to para-virtualization, achieving high performance and many of the features of traditional virtualization. Our solution relies on constraining para-virtualization's modifications according to several principles, which we call *soft layering*. As originally proposed for layered network protocols [7], soft layering embraces co-design of neighboring software layers, but with some conditions:

1. it must be possible to degrade to a neutral interface, by ignoring the co-design enhancements (thus permitting execution on raw hardware and hypervisors that lack support for the soft layering);

2. the interface must flexibly adapt to the algorithms that competitors may provide (thus supporting arbitrary hypervisor interfaces without pre-arrangement).

Additionally, we use tools to apply the soft layer to a guest kernel (with substantial automation) to easily support obsoleted kernels.

1.1. Layering

Software layering promotes module simplification, although the resulting abstractions introduce inefficiencies due to the lack of transparency into the lower layers' implementation details [21]; 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.

Virtual machines have a problem with transparency, particularly since the guest OSes are oblivious to the internals of the layers below, and because their downcalls use hardware mechanisms (privileged instructions) rather than software mechanisms (function calls). Many virtualization projects have increased the transparency between the guest OS and the virtual machine by modifying the guest OS to directly interact with the internals of the layers below, and to use efficient downcalls. These modifications introduce a dependency between the guest OS and its lower layer, often breaking the modularity achieved by virtualization: the guest OS may no longer execute on raw hardware, within other virtual machine environments, or permit nested VMs.

1.2. Soft layering

To achieve modularity, soft layering mostly honors strict layering, but permits the hypervisor to inject efficient downcalls and virtualization logic into the protection domain of the guest kernel (see Figure 1). This is similar to the technique used by VMware's binary translation [1], but we additionally prepare the guest kernel to help achieve the performance of para-virtualization.

Soft layering forbids changes to the guest OS that would interfere with correct execution on the original platform interface (bare metal), it discourages 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 loads a guest kernel.

To achieve our performance goals we increase transparency to the hypervisor's internal abstractions, 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.

In this paper we describe our soft layering approach, which we call *pre-virtualization*. We present reference implementations for several hypervisors on two architectures, and show that they offer modularity while sustaining the performance of para-virtualization.

2. Architecture

Many projects have improved the transparency of virtual machine layering via co-design of the hypervisor and OS (i.e., para-virtualization), and have introduced specialized interfaces [3,4,9,11,12,14–16,19,20,27] to solve their problems. These interfaces improved both performance and correctness.

Para-virtualization has three categories of interfaces between the hypervisor and guest OS, 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.

Structural modifications add efficient mappings between high-level abstractions in the guest kernel and hypervisor interfaces. These modifications are very intrusive to the guest kernel, and require specific knowledge of the guest kernel's internal abstractions. They are common for adding efficient networking and disk support. 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.

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 [28].

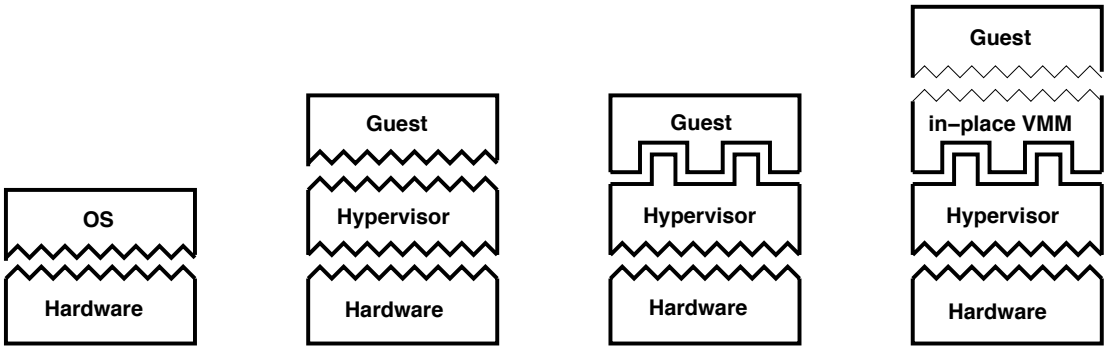

Figure 1. Comparison of virtualization strategies (left to right): (a) native execution — no virtual machine; (b) traditional virtualization — hypervisor uses neutral interface; (c) para-virtualization — hypervisor presents a changed API; (d) soft layering — the VAM maps the neutral interface to a para-virtualizing hypervisor API (thin lines indicate an interface without privilege change).

2.1. Instruction level

The performance of virtual machines relies on using bare-metal execution for the innocuous instructions, while introducing expensive emulation only for the infrequently executed virtualization-sensitive instructions [23]. 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 rewriting the source code to map virtualization-sensitive instructions into higher-level abstractions. For our approach to have performance comparable to para-virtualization, we must 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 pad each virtualization-sensitive instruction with a sequence of no-op instructions,[1] and annotate their locations to permit a hypervisor to rewrite the virtualization-sensitive instructions at runtime (see Figure 2). The rewriting process decodes the original instructions to determine intent and the locations of the instructions' parameters, and writes higher-performance downcalls over the scratch space provided by the no-op padding.

To map the low-level operations of individual instructions to the higher-level abstractions of the hypervisor, we collocate a mapping module within the address space of the guest kernel. The mapping module provides a virtual CPU and device models. The rewritten instructions directly access the mapping module via function calls or memory references. The mapping module defers interaction with the hypervisor until necessary by batching state changes, thus imitating the behavior of para-virtualization, and the hypervisor authorizes activity that could subvert security (note that a misbehaving guest could corrupt the mapping module). We term the mapping module the *virtualization-assist module (VAM)*. A VAM is specific to a hypervisor, and 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 OS kernel. Additionally, since the binding to the VAM takes place at runtime, the guest OS kernel can execute on a variety of hypervisors, and a running guest OS can migrate between hypervisors (all of which are especially useful across hypervisor upgrades).

Furthermore, for the easy use of soft layering, we apply

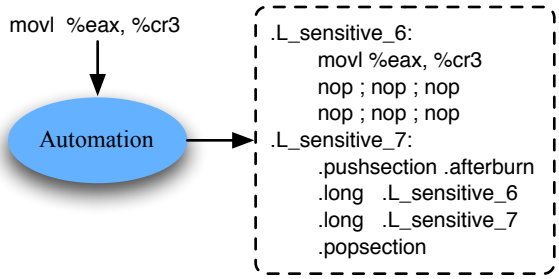

Figure 2. Example of assembler preparation for a sensitive x86 instruction — it adds scratch space via `nop` instructions, and adds assembler directives to record the location of the instruction.

[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.

the instruction-level changes automatically at the assembler stage [12]. 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.

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. To distinguish between memory types, we use another automated tool that applies data-type analysis to the guest kernel's source code; if the guest kernel lacks unique data types for the memory objects, then we manually apply the instruction no-op padding and annotations via C language macros or compiler primitives.

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 a standard is an attraction, for it enables custom solutions that reflect the expertise of the developers for their problem domains. Structural changes are compatible with the criteria of soft layering — soft layering only demands that the structural changes obey an interface convention. Potentially the structural changes are specific to a hypervisor or a guest kernel, and when they conform to soft layering then all others can ignore these structural changes (or others can later add support for them).

Our reference implementation adds function-call overloads to the guest kernel that we can rewrite to invoke methods in the VAM. Additionally, we support loadable kernel modules that link against the VAM as well as link against the guest kernel, which permits the runtime addition of new hypervisor-aware functionality.

2.3. The VAM

The VAM 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 [15, 18]). The VAM controls access to hypercalls, and intercepts the upcalls from the hypervisor to the guest kernel (for fault and interrupt emulation).

The hypervisor links together the guest OS and the VAM at load time by rewriting the virtualization-sensitive instructions and by hooking the function overloads. Where we replace the original, indivisible instructions of the guest kernel with emulation sequences of many instructions, we must respect their indivisibility in regards to faults and interrupts. The guest kernel could malfunction if exposed to VAM 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 VAM as an event processor: the guest kernel requests a service, and the VAM returns to the guest kernel only after completing the service; or it may roll back or forward 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.

2.4. Device emulation

Soft layering is unique for virtualizing real, standard devices with high performance; all other virtualization approaches depend on special device drivers for performance, which 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 [25]. We instead run the device model within the VAM, and convert these device register accesses into efficient downcalls via instruction-level soft layering. At runtime we rewrite the instructions to invoke the VAM. The VAM models the device, and batches state changes to minimize interaction with the hypervisor.

3. Implementation

We implemented a reference pre-virtualization environment according to the soft layering principles described in the prior sections, for x86 and Itanium. We describe the automation, the VAM, and a pre-virtualized network device model, all as used on x86. We describe implementations for two families of x86 hypervisors that have very different APIs, the L4 microkernel and the Xen hypervisor, to demonstrate the versatility of virtualizing at the neutral platform API. Our Itanium implementation supports three hypervisors, also with very different APIs: Xen/ia64 [24], vNUMA [5], and Linux. For the guest kernel, we used several versions of Linux 2.6 and 2.4 for x86, and Linux 2.6 on Itanium.

[2] The guest kernel is a sequential process concerned about forward progress but not the rate of forward progress [10].

3.1. Guest preparation

Soft layering involves modifications to the guest kernel, and we have different techniques for applying the modifications to sensitive instructions, sensitive memory instructions, and structural changes.

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 [22]; it builds an abstract syntax tree, walks and transforms the tree, and then emits new assembler code.

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 parser written in ANTLR, and redefines the assignment operator based on data type (similar to C++ operator overloading).

Structural: Our primary structural modification allocates a hole within the virtual address space of Linux for the VAM and hypervisor. The hole's size is currently a compile-time constant. 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 the L4 microkernel with decent performance, we added other function-call overloads. 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.

3.2. Runtime environment

We divide the VAM 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. Upcalls from the hypervisor (e.g., interrupt notifications) interact with the back-end, and propagate to the front-end.

Xen/x86 hypervisor back-end: The x86 Xen API resembles the hardware API, even using the hardware `iret` instruction to transition from kernel to user. Still, the VAM intercepts all privileged-instructions and upcalls to enforce the integrity of the virtualization. Interrupts, exceptions, and x86 traps are delivered to the VAM, which updates the virtual CPU state machine and then transitions to the guest kernel's handler. The VAM intercepts transitions to user-mode, 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.

Xen's API for constructing page mappings uses the guest OS's page tables as the actual x86 hardware page tables. The VAM 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 VAM 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 VAM permits our Linux binaries to execute on both Xen 2 and Xen 3.

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, and we use the same x86 front-end for both.

For performance reasons, we associate one L4 address space with each guest address space. We switch the L4 address spaces upon privilege changes (such as at `iret`). The VAM can update the shadow L4 space optimistically or lazily, since it has TLB semantics.

Network device emulation: We implemented a device model for the DP83820 gigabit network card, for we predicted that its features would support high-speed batching between the guest and 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.

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 implemented a back-end for the L4 environment. The back-end forms the network client in the L4 device driver reuse environment [17].

4. Evaluation

We assessed the performance and engineering costs of our implementation, and compare to high-performance para-virtualization projects that use the same hypervisors. We also compare the performance of our pre-virtualized binaries running on raw hardware to the performance of native binaries running on raw hardware.

On x86, the hypervisors are the L4Ka::Pistachio microkernel and the Xen 2.0.2 hypervisor. The para-virtualized OSes are L4Ka::Linux 2.6.9, XenoLinux 2.6.9, and XenoLinux 2.4.28.

On Itanium, the hypervisors are Xen/ia64, vNUMA, and Linux 2.6.14. The para-virtualized OS is XenoLinux 2.6.12.

4.1. Performance

We perform a comparative performance analysis, using the guest OS running natively on raw hardware as the baseline. The comparative performance analysis requires similar configurations across benchmarks. Since the baseline ran a single OS on the hardware, with direct device access, we generally used a similar configuration for the hypervisor environments: a single guest OS ran on the hypervisor, and had direct device access.

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 the x86 systems to use a 100Hz timer, and the XT-PIC (our APIC model is incomplete). Additionally, on x86 we used the slow legacy `int` system call invocation, as required by some virtualization environments. On Itanium, there was no problem using the `epc` fast system call mechanism, which is the default when using a recent kernel and C library.

The x86 test machine was a 2.8GHz Pentium 4, constrained to 256MB of RAM, and ran Debian 3.1 from the local SATA disk. The Itanium test machine was a 1.5Ghz Itanium 2, constrained to 768MB of RAM, running a recent snapshot of Debian 'sid' from the local SCSI disk.

Most performance numbers are reported with an approximate 95% confidence interval, calculated using Student's t distribution with 9 degrees of freedom (i.e., 10 independent benchmark runs).

4.1.1. Linux kernel build

We used a Linux kernel build as a macro benchmark. Each kernel build started from a freshly unpacked archive of the source code, to normalize the buffer cache.

Table 1 shows the results for both Linux 2.6 and 2.4. The baseline for comparison is native Linux running on raw hardware (native, raw). Also of interest is comparing pre-virtualized Linux (Xen VAM) to para-virtualized Linux (XenoLinux), and comparing a pre-virtualized binary on raw hardware (NOPS, raw) to the native Linux binary running on raw hardware.

System	Time [s]	CPU util	O/H [%]
Linux 2.6.9 x86			
native, raw	209.2	98.4%	
NOPs, raw	209.5	98.5%	0.15%
XenoLinux	218.8	97.8%	4.61%
Xen VAM	220.6	98.8%	5.48%
L4Ka::Linux	235.9	97.9%	12.8%
L4 VAM	239.6	98.7%	14.6%
Linux 2.4.28 x86			
native, raw	206.4	98.9%	
NOPs, raw	206.6	98.9%	0.11%
XenoLinux	215.6	98.6%	4.45%
Xen VAM	219.5	98.9%	6.38%
Linux 2.6.12 Itanium			
native, raw	434.7	99.6%	
NOPs, raw	435.4	99.5%	0.16%
XenoLinux	452.1	99.5%	4.00%
Xen VAM	448.7	99.5%	3.22%
vNUMA VAM	449.1	99.4%	3.31%
Linux 2.6.14 Itanium			
native, raw	435.1	99.5%	
Linux VAM	635.0	98.4%	45.94%

Table 1. Linux kernel build benchmark. The "O/H" column is the performance penalty relative to the native baseline for the respective kernel version. Data for x86 have a 95% confidence interval of no more than $\pm$ 0.43%.

The performance degradation for the Xen VAM is due to more page-table hypercalls. The performance degradation of the L4 VAM is due to fewer structural modifications compared to L4Ka::Linux. On raw hardware, performance differences between the annotated and padded binaries were statistically insignificant.

4.1.2. Netperf

We used the Netperf send and receive network benchmarks to stress the I/O subsystems. Our benchmark transferred a gigabyte of data at standard Ethernet packet size, with 256kB socket buffers.

Table 2 shows the send performance and Table 3 the receive performance for Netperf. In general, the performance of the pre-virtualized setups matched that of the para-virtualized setups. Our L4 system provides event counters which allow us to monitor kernel events such as interrupts, protection domain crossings, and traps caused

System	Xput [Mb/s]	CPU util	cyc/B
Linux 2.6.9 x86			
native, raw	867.5	27.1%	6.68
NOPs, raw	867.7	27.3%	6.73
XenoLinux	867.6	33.8%	8.32
Xen VAM	866.7	34.0%	8.37
L4Ka::Linux	775.7	34.5%	9.50
L4 VAM	866.5	30.2%	7.45
Linux 2.4.28 x86			
native, raw	779.4	39.3%	10.76
NOPs, raw	779.4	39.4%	10.81
XenoLinux	778.8	44.1%	12.10
Xen VAM	779.0	44.4%	12.17

Table 2. Netperf send performance of various systems. The column "cyc/B" represents the number of non-idle cycles necessary to transfer a byte of data, and is a single figure of merit to help compare between cases of different throughput. Data have a 95% confidence interval of no more than $\pm$ 0.25%.

by guest OSes. Using those we found the event-counter signature of the para-virtualized Linux on L4 to be nearly identical to that of the pre-virtualized Linux on L4.

4.1.3. Network device model

We also used Netperf to evaluate the virtualized DP83820 network device model. A virtualized driver, by definition, has indirect access to the hardware. The actual hardware was an Intel 82540, driven by a device driver reuse environment [17] based on the L4 microkernel. In this configuration, the Netperf VM sent network requests to a second VM that had direct access to the network hardware. The second VM used the Linux e1000 gigabit driver to control the hardware, and communicated via L4 IPC with the Netperf VM, to convert the DP83820 device requests into requests for the Intel 82540.

In the baseline case, the Netperf VM used a custom Linux network driver to communicate with the VM controlling the Intel 82540.

Table 4 shows the Netperf send and receive results. Performance is similar, although the pre-virtualized device model required slightly less CPU resource, confirming that it is possible to match the performance of a customized virtual driver, by rewriting fine-grained device register accesses into efficient downcalls. The number of device register accesses during Netperf receive was 551k (around 48k/s), and during Netperf send was 1.2M (around 116k/s).

System	Xput [Mb/s]	CPU util	cyc/B
Linux 2.6.9 x86			
native, raw	780.4	33.8%	9.24
NOPs, raw	780.2	33.5%	9.17
XenoLinux	780.7	41.3%	11.29
Xen VAM	780.0	42.5%	11.65
L4Ka::Linux	780.1	35.7%	9.77
L4 VAM	779.8	37.3%	10.22
Linux 2.4.28 x86			
native, raw	772.1	33.5%	9.26
NOPs, raw	771.7	33.7%	9.33
XenoLinux	771.8	41.8%	11.58
Xen VAM	771.3	41.2%	11.41

Table 3. Netperf receive performance of various systems. Throughput numbers have a 95% confidence interval of $\pm$ 0.12%, while the remaining have a 95% confidence interval of no more than $\pm$ 1.09%.

System	Xput [Mb/s]	CPU util	cyc/B
Send			
L4Ka::Linux	772.4	51.4%	14.21
L4 VAM	771.4	49.1%	13.59
Receive			
L4Ka::Linux	707.5	60.3%	18.21
L4 VAM	707.1	59.8%	18.06

Table 4. Netperf send and receive performance of device driver reuse systems.

4.2. Engineering effort

The implementation effort is in the same order of magnitude as para-virtualization for a single kernel version, but the VAM is reusable across many guest operating systems and their many versions. Table 5 shows a breakdown of lines of source code for the individual x86 VAMs 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 contrast, in Xen [3], the authors report that they modified and added 1441 sources lines to Linux and 4620 source lines to Windows XP. In L4Linux [15], 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

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. The breakdown of lines of source code for the x86 VAMs, counted by SLOCcount.

on small changes, still required about 3000 modified lines of code [17].

5. Related work

Co-design of a hypervisor and an OS has existed since the dawn of VMs [8, 13]. Several microkernel projects have used it with high-level changes to the guest kernels [14,15]. Recent hypervisor projects have called it paravirtualization [3,20,27].

Some VM projects have added strict virtualization to architectures without such support. Eiraku and Shinjo [12] offer a mode that prefixes every sensitive x86 instruction with a trapping instruction. vBlades [20] and Denali [27] substitute alternative, trappable instructions for the sensitive instructions.

All major processor vendors support virtualization extensions in their processor lines, yet performance benefit comes from executing virtualization logic within the guest kernel [1]. Soft layering permits sophisticated state machines to execute within the domain of the guest OS, especially for devices (which are unaddressed by the extensions). The hardware extensions can transform general purpose OSes into full-featured hypervisors, creating even more demand for the modularity of soft layering.

Customization of software for alternative interfaces is a widely used technique, e.g. PowerPC Linux uses a function vector that encapsulates and abstracts the machine interface. This manually introduced indirection allows running the same kernel binary on bare hardware and on IBM's commercial hypervisor.

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 [16]. Additionally, 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 proposal [2, 26] for a virtual machine interface (VMI) shares many of our goals. The first version used a ROM invoked by function calls as in the UMLinux project, but designed for dynamic linking at load time with emulation code specific to the hypervisor (including direct execution on hardware). After the first VMI version became public, we started contributing to its design, and VMware evolved it into an implementation of soft layering with no-op instruction padding and function entry points in the ROM. VMI deviates from the base instruction set more than our reference implementation. 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. The Linux community has reduced the scope of VMI, creating a new interface called `paravirt_ops` (to which we also participated). `paravirt_ops` requires all hypervisor mapping modules to be available at Linux compile time (which discards the modularity benefits of virtualization, such as upgrading the hypervisor to a new interface). Yet VMI can be one of those mapping modules, thus permitting the advantages of soft layering for Linux.

6. Conclusion

We presented the soft layering approach to hypervisor-OS co-design, which provides the modularity of traditional virtualization, while achieving nearly the same performance as established para-virtualization approaches. 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 reference implementation, called pre-virtualization, also reduces the effort of para-virtualization via automation. We demonstrated the feasibility of pre-virtualization by supporting a variety of very dissimilar hypervisors with the same approach and infrastructure.

References

[1] K. Adams and O. 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, Oct. 2006.

[2] Z. Amsden, D. Arai, D. Hecht, A. Holler, and P. Subrahmanyam. VMI: An interface for paravirtualization. In *Proc. of the Linux Symposium*, pages 363–378, Ottawa, Canada, July 2006.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating System Principles*, Bolton Landing, NY, Oct. 2003.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors.

In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 143–156, Saint Malo, France, Oct. 1997. doi:10.1145/268998.266672.

[5] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proc. of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, Apr. 2005.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, May 2001.

[7] G. H. Cooper. An argument for soft layering of protocols. Technical Report TR-300, Massachusetts Institute of Technology, Cambridge, MA, Apr. 1983.

[8] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, Sept. 1981.

[9] F. B. des Places, N. Stephen, and F. D. Reynolds. Linux on the OSF Mach3 microkernel. In *Conference on Freely Distributable Software*, Boston, MA, Feb. 1996.

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

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

[12] H. Eiraku and Y. Shinjo. Running BSD kernels as user processes by partial emulation and rewriting of machine instructions. In *Proc. of BSDCon '03*, San Mateo, CA, Sept. 2003.

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

[14] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proc. of the USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[15] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 66–77, Saint-Malo, France, Oct. 1997.

[16] H.-J. Höxer, K. Buchacker, and V. Sieh. Implementing a user-mode Linux with minimal changes from original kernel. In *Proc. of the 9th International Linux System Technology Conference*, pages 72–82, Cologne, Germany, Sept. 2002.

[17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, Dec. 2004.

[18] J. Liedtke. On μ-kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[19] 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.

[20] D. J. Magenheimer and T. W. Christian. vBlades: Optimized paravirtualization for the Itanium Processor Family. In *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[21] 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.

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

[23] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Proc. of the Fourth Symposium on Operating System Principles*, Yorktown Heights, New York, Oct. 1973.

[24] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the art of virtualization. In *Proc. of the 2005 Ottawa Linux Symposium*, Ottawa, Canada, July 2005.

[25] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

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

[27] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, MA, Dec. 2002.

[28] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. 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.