Pre-virtualization: soft layering for virtual machines

Joshua LeVasseur[†] Volkmar Uhlig[§] 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

Para-virtualization ties a guest operating system and a hypervisor together, which restricts the system architecture; e.g., when Linux uses the Xen API, Linux 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 discards 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; combined with the performance of para-virtualization it becomes even more compelling. We show how to achieve both *together*. We offer a set of design principles, which we call *soft layering*, that govern the modifications made to the operating system. Additionally, our approach is highly automated, thus reducing 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). Many have proposed to modify the operating system (OS) for co-design with the hypervisor, i.e., para-virtualization [35], to improve performance and correctness; the possibilities seem unlimited, but the cost has been the emergence of many para-virtualization projects with incompatible and unreconcilable 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, such as Xen [2] vs. Linux-as-hypervisor [10,19] vs. hardware-accelerated para-virtualization — most para-virtualization interfaces hard code assumptions that prevent adaptation to these different scenarios. Modularity permits runtime hypervisor upgrades (via checkpointing or migration [6, 31] of the guest OS); in contrast, XenoLinux 2.6.9 is tied to Xen 2.0.2, and is unable to even run on the newer Xen 3 series – high-level source code modifications lock-in a particular hypervisor, obstructing 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, even if those features are unwelcome by the kernel developers themselves (as most are); in contrast, the Linux kernel team can veto the ideas of the Xen team in the current merge of XenoLinux into the mainstream Linux source. Modularity supports proliferation of hypervisors; prolif-

eration is unavoidable if the proliferation of operating systems is an indicator, whether for licensing reasons (proprietary [34] vs. open source [2]), or products from competing vendors (e.g., VMware, Microsoft, and Parallels), or products that focus on different feature sets (e.g., desktop [32] vs. server [2, 34, 35]). Modularity additionally supports stackable enhancements via recursive virtual machine construction [8, 13, 17].

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 prearrangement).

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

1.1 Strict layering

A traditional virtual machine enforces strict layering, in the manner of Djikstra's "THE" [9], with the hypervisor (lower) layer hiding resource multiplexing from the guest OS (upper) layer; the upper layer then focuses on its problems rather than those of the lower layers. This information hiding provides modularity: it permits the upper layer to use simpler abstractions (in regards to the information hidden in the lower layer) via its pre-existing use of the platform interface; and to remain independent of the lower-level's implementation, which permits substitution of a new lower layer without disturbing the upper layer. The virtual machine must provide the strict layering because its guest operating systems are oblivious to any resource multiplexing handled by the lower layers; an OS was written to perform the resource multiplexing, but not to participate in collaborative resource multiplexing with other OSes.

Design by layering involves choosing appropriate abstractions and cut points for the layer interfaces. Layering's abstractions provide benefits, but as explained by Parnas, the abstractions easily introduce inefficiencies due to the lack of transparency into the lower layers [26]; for example, the upper layer may have the impression that a resource is infinite, when the opposite is the case; or the upper layer may have an algorithm that maps well to a feature of the machine but which is hidden by the abstractions of the lower layer. Virtual machines have a problem with transparency, particularly since the guest OSes try to use the mechanisms of the hardware, but must instead use abstractions that result from the hypervisor multiplexing the hardware mechanisms. Many virtualization projects have addressed this issue by increasing the transparency between the guest OS and the virtual machine, by modifying the guest OS to interact with the internals of the layers below. These modifications introduce a dependency 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 VM recursion (a layer to be inserted between the guest OS and the virtual machine below it).

1.2 Soft layering

It is the power of the platform's primitive interface [12] that enables the rich solutions of traditional virtualization, and its neutrality that enables uncoordinated development [6], and we focus on this interface in our solution. The solution is simple: mostly honor strict layering, but move the hypervisor's virtualization routines into the protection domain of the guest kernel, thus enabling low-cost virtualization (see Figure 1). This is similar to the technique used by VMware, but we additionally prepare the guest kernel to help achieve the performance of para-virtualization.

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 (i.e., the guest kernel's use of the privileged instructions and its internal design philosophies), it uses the neutral platform interface as the default interface (i.e., the OS will execute directly on raw hardware while the enhancements require activation), and the OS includes annotations that help the hypervisor optimize performance. Soft layering forbids changes to the guest OS that would interfere with correct execution on the neutral platform interface, it discourages hypervisor-specific changes, and it discourages changes that substantially penalize the performance of the neutral platform interface.

The decision to activate a specialized hypervisor interface happens at runtime when the hypervisor and guest kernel are joined together. The guest OS adds to its binary a description of its soft layer, and the hypervisor inspects the descriptor to determine whether they agree on the soft layer. If not, the hypervisor can abort load-

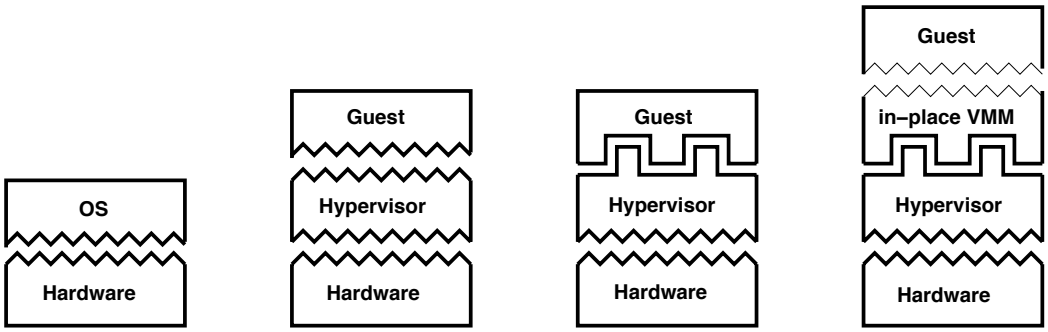

Figure 1: Comparison of virtualization strategies (left to right): (a) native execution — no virtual machine; (b) traditional virtualization — hypervisor uses neutral interface; (c) para-virtualization — hypervisor presents a changed API; (d) pre-virtualization — the in-place VMM maps the neutral interface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

ing, or activate a subset of the enhancements (to ignore extensions unequally implemented, or rendered unnecessary when using VM hardware acceleration [1, 20–22]).

For performance we must increase transparency to the hypervisor's internal abstractions without violating the second criterion of soft layering – that the interface must flexibly adapt to the algorithms provided by competitors (e.g., Linux as a hypervisor with Intel's VT, or Xen). We take advantage of how operating systems use the platform interface in a fairly predictable and consistent manner, and map this behavior to the hypervisor's efficient primitives, by permitting execution of mapping algorithms directly within the guest kernel's protection domain. 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 a different solution, such as a convention or an idealized device interface that all agree upon. 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, and have introduced specialized interfaces [2, 4, 10, 11, 18, 19, 24, 35] to solve their problems. These interfaces not only created strong dependencies between the guest kernel and hypervisor of each project, but often duplicated effort between projects due to incompatible specializations that implemented similar concepts.

Besides performance, para-virtualization addresses correctness, particularly for timing [2, 35], and occasionally for direct hardware access [14, 23]. Timing and direct hardware access fall outside the scope of virtualization [28], 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).

The interface specializations of para-virtualization fall into three categories for which we offer soft-layer alternatives:

Instruction-level modifications can extend the underlying architecture (e.g., Denali [35] introduced an idle with timeout instruction), or more importantly, replace privileged instructions with optimized code that avoids expensive trapping due to their execution at user level. Some architectures, especially x86, do not satisfy the requirements of virtualization [28, 30], and rely on substitution of calls to the hypervisor (hypercalls) for the virtualization-sensitive instructions, which leads to a hypervisor-specific solution. Instruction-level modifications generally 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 are very common for adding efficient networking and disk support. Also they are used to map guest address spaces to hypervisor address spaces, guest threads to hypervisor threads, and to provide for efficient memory copying between the guest ker-

nel and its applications. 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. These modifications are very intrusive to the guest kernel, and require specific knowledge of the guest kernel's internal abstractions.

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 [32], 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. We needn't address para-virtualization's behavioral modifications, since they are a natural form of soft layering: they avoid interference with OS and hypervisor neutrality, and may achieve self activation (e.g., the kernel detects that certain operations require far more cycles to execute, and thus it changes behavior to match the more expensive operations [36]).

2.1 Instruction level

The performance of virtual machines relies on using bare-metal execution for the innocuous instructions, reserving expensive emulation for the virtualization-sensitive instructions [28]. 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 on the most frequently executed instructions [24]). Yet if the modifications substitute hypercalls for each instruction, the savings may be small, since hypercalls are usually expensive. Thus para-virtualization hypervisors such as Xen map the low-level instruction sequences into higher abstractions, via source code modifications, to reduce the number of hypercalls.

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. The rewriting process decodes the original instructions to determine intent and the locations of the instructions' parameters, and writes higher-performance alternatives 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. We term the mapping module the *in-place VMM*. The mapping module is specific to the 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 kernel. Additionally, since the binding to the mapping module takes place at runtime, the guest kernel can execute on a variety of hypervisors, and a running guest kernel can migrate between hypervisors (all of which are especially useful across hypervisor upgrades).

Furthermore, for the easy use of soft layering, we apply the instruction-level changes automatically at the assembler stage [11]. Thus we avoid many manual changes to the guest OS's source code, which reduces the manpower 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. This source code modularity is particularly useful for the fast-paced development of open source kernels, where people focus on editing the latest edition of a kernel while abandoning older kernels to intermediate states of development, and while the users 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).

A variety of memory objects are also virtualization-sensitive, such as memory-mapped device registers and x86's page tables. We apply the soft-layering technique to the instructions that access these memory objects (e.g., writes and reads to page table entries). To distinguish between memory types, our automated tool uses data-type analysis in 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 prim-

[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.

itives [3].

2.2 Structural

Many of para-virtualization's structural enhancements are ineligible for instruction-level substitution, and also ineligible for automated application, and thus require intimate knowledge of the guest kernel to apply the changes correctly. For these reasons, it is best to favor instruction-level solutions (which alone solve many of the performance problems) over structural changes.

Some structural changes permanently alter the design of the kernel, such as changes in the guest kernel's address space layout, which apply 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, such as memory ballooning [34] (which enables cooperative resource multiplexing with the hypervisor). The soft layer can add function-call overload points, or can rely on runtime installable kernel modules.

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.

The principal structural change of virtual machines concerns the collision of privileges and address spaces in the VM environment; most kernels use hardware to combine kernel and user privileges in a single address space to avoid an expensive address-space switch when transitioning between the privileges. The hypervisor requires a third privilege level. The hardware limits the number of concurrent privileges; many processor architectures provide only two privileges, and those that provide more may find the guest kernel using the additional privileges too. Even if the hardware provides additional privilege levels, the hypervisor may prohibit their use, such as when using Linux as a hypervisor. We virtualize a privilege level by providing it a dedicated address space, which can be performed transparently to the guest kernel; but for performance reasons, it requires overloading all places where the guest kernel accesses user memory at addresses in the user portion of the virtual address space. In all cases, the hypervisor and in-place VMM require a dedicated area in all virtual address spaces; thus soft layering must provide a convention for allocating such a hole.

2.3 The in-place VMM

Besides providing performance, the in-place VMM includes comprehensive virtualization code for mapping the instruction-level and device-level activity of the guest kernel into the high-level abstractions of the hypervisor (e.g., mapping disk activity into POSIX file operations for Linux-on-Linux). This is especially useful for repurposing general-purpose kernels as hypervisors (such as Linux on Linux, or Linux on the L4 microkernel). The in-place VMM not only controls access to hypercalls, it also captures the upcalls from the hypervisor to the guest kernel (for fault and interrupt emulation).

The hypervisor links together the guest OS and the in-place VMM at load time, by rewriting the virtualization-sensitive instructions to invoke the appropriate emulation code. We replace the original, indivisible instructions of the guest kernel with emulation sequences of many instructions; the in-place VMM must respect the indivisibility of the original instructions in regards to faults and interrupts; the guest kernel should never have exposure to the in-place VMM state in an interrupt frame. To avoid reentrance, we structure the in-place VMM as an event processor: the guest kernel requests a service, and the in-place VMM reactivates 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.

The guest kernel may execute several of the virtualization-sensitive instructions frequently, e.g., Linux often toggles interrupt delivery. The performance of the in-place VMM depends on its algorithms and how it chooses between instruction expansion and hypercalls; for the frequent instructions we want to avoid both. For example, when running Linux on Linux, toggling interrupts via POSIX signal masking would add hypercall overheads; likewise, using Xen's interrupt toggle would involve too many instructions. Instead we rely on an algorithm: we use a virtual CPU within the in-place VMM 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. Since the in-place VMM captures the hypervisor's upcalls, it becomes aware of pending interrupts even when the guest kernel has interrupts disabled, in which case it queues the pending interrupt for later delivery when the guest kernel reactivates its interrupts via the virtual CPU.

The in-place VMM supports extension via loadable

guest-kernel modules; the modules dynamically link against both the guest kernel and the in-place VMM. We added a soft-layer structural hook to the guest OS for helping resolve module symbols against the in-place VMM. A memory-ballooning module could be implemented in this manner for example.

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 massive performance bottlenecks when emulated via traps [32]. Instead we use the instruction-level soft layering approach: we annotate and pad with no-ops the instructions that access the device registers, and then rewrite these instructions in the soft layer to invoke the in-place VMM. The in-place VMM models the device, and batches state changes to minimize interaction with the hypervisor.

Networking throughput is particularly sensitive to batching: if the batching adds too much delay to transmitted packets, then throughput deteriorates; if the in-place VMM 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 [32]. In past work with Linux, we used a callback executed immediately after the networking subsystem, which provided good performance [23]; 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.

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 in-place VMM, and a pre-virtualized network device model, all as used on x86. We describe implementations for two 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. We briefly describe our Itanium implementation, which supports three hypervisors, also with very different APIs: Xen/ia64 [29], 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.

3.1 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.

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 [27]; it builds an abstract syntax tree (AST), 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 ELF section the start and end addresses of the instruction and its no-op window. 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. Itanium Linux also has a situation that manually calculates an instruction pointer, which we fixed by modifying the assumptions of the Linux code.

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

Sensitive memory instructions: An automated solution for pre-virtualizing the memory instructions must disambiguate the sensitive from the innocuous. We have been implementing a data-type analysis engine 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 [27], and redefines the assignment operator based on data type (similar to C++ operator overloading). It is incomplete, and so we currently apply the soft layer manually by modifying the appropriate abstractions in Linux. Our modifications (1) force the operation to use an easily decodable memory instruction, and (2) add the no-op padding around the instruction.

We integrate the following memory operations within the soft layer: page table and page directory accesses, device register accesses (for the devices that we pre-virtualize), and DMA translations. The DMA translations support pass-through device access [14, 23] from within a VM; in the normal case, the DMA operations are no-ops.

Structural: Our primary structural modification allocates a hole within the virtual address space of Linux for the in-place VMM 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. This is the only structural modification necessary for running on the Xen hypervisor.

To support the L4 microkernel with decent performance, we additionally added several 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 in-place VMM. 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 in-place VMM 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.

Indivisible instructions: The in-place VMM preserves the boundary between itself and the guest kernel for correct emulation of the indivisible instructions of the guest kernel. It handles three interrupt conditions: (1) when the guest kernel enables interrupts and they are 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 for jumping to an interrupt handler when exiting the in-place VMM, which is used only if an interrupt is pending. Upon entry of the interrupt handler the register-file and stack contain the guest kernel's boundary 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 `iret`, we roll-back and restart the front-end, so that it follows the route for case 1. For idle we roll forward, to abort the idle hypercall, and then deliver the interrupt using the redirection frame.

Case 3 requires manual inspection of pending interrupts. If an interrupt is pending, we alter the in-place VMM'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.

Instruction rewriting: When loading the guest kernel, the in-place VMM parses the ELF headers of the guest kernel to locate the soft layer annotations. Via the annotations, the in-place VMM 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 in-place VMM 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 in-place VMM. 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). Instead we use a heuristic to deliver pending interrupts at a later time:[2] 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.

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 in-place VMM intercepts all Xen API interactions to enforce the integrity of the virtualization. Interrupts, exceptions, and x86 traps are delivered to the in-place

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

VMM, which updates the virtual CPU state machine and then transitions to the guest kernel's handler. The in-place VMM 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 in-place VMM 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 in-place VMM 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 in-place VMM 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 by mapping the neutral platform API to the hypervisor API, and we use the same x86 front-end for both.

For performance reasons, we map an address-space switch of the guest OS to an address-space switch in L4. The in-place VMM associates one L4 address space with each guest address space (shadow page tables). The in-place VMM can update the shadow page tables optimistically or lazily, since they have TLB semantics.

L4 lacks asynchronous event delivery; it requires a rendezvous of two threads via IPC; we map hardware interrupts and timer events onto IPC. Within the in-place VMM, we instantiate an additional L4 thread that receives asynchronous event notifications and either directly manipulates the state of the L4 VM thread or updates the virtual CPU model (e.g., register a pending interrupt when interrupts are disabled).

As described in Section 3.1, we added several structural hooks to the guest OS, to accommodate virtualization inefficiencies in the L4 API.

Network device emulation: We implemented a device model for the DP83820 gigabit network card. 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. 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 implemented a back-end for the L4 environment. The back-end forms the network client in the L4 device driver reuse environment [23].

3.3 Itanium

We implemented multiple in-place VMMs for Itanium: for Xen, for vNUMA, and for Linux as a hypervisor.

The RISC nature of the Itanium architecture complicates the construction of a transparent in-place VMM compared to an architecture such as x86. It is not possible to load or store to memory without first loading the address into a register. Nor is it possible to simply save and restore registers on the stack, since the stack pointer may be invalid in low-level code. This makes it both necessary and difficult to find temporary registers for the in-place VMM. For sensitive instructions with a destination-register operand, the destination register can be considered scratch until the final result is generated. However, many instructions do not have a destination-register operand. It would also be preferable to have more than one scratch register available, to avoid costly saving and restoring of further needed registers.

Our solution virtualizes a subset of the machine registers that the compiler rarely uses, specifically `r4-r7` and `b2`. We replace all references to these registers with memory-based emulation code and save and restore them when transitioning in and out of the pre-virtualized code.

Instruction rewriting replaces an indivisible instruction with an instruction sequence; interruptions of the sequence may clobber the scratch register state. We avoid this problem by a convention: the emulation block uses one of the scratch registers to indicate a roll-back point in case of preemption. The last instruction of the sequence clears the roll-back register.

Xen/IA64 and vNUMA are both designed so that the hypervisor can be hidden in a small architecturally-reserved portion of the address space. This is not the case for Linux, which assumes that the whole address space is available. Thus, to run Linux-on-Linux it is necessary to modify one of the kernels to avoid address-space conflicts with the other. In our case we relocate the guest so that it lies wholly within the user address space of the host, which requires a number of non-trivial source changes. This precludes using the same pre-virtualized Linux kernel both as a host and a guest.

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 used a similar configuration for the hypervisor environments: A single guest OS ran on the hypervisor, and had direct device access. The exception is the evaluation of our network device model; it virtualizes the network device, and thus had indirect 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. 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, 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.

Each kernel build started from a freshly unpacked archive of the source code, to normalize the buffer cache. The build used a predefined Linux kernel configuration.

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 in-place VMM) 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.

The performance degradation for the Xen in-place VMM is due to more page-table hypercalls. We have not yet determined the reason for the increased number of hypercalls. The performance degradation of the L4 in-place VMM 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. These are I/O-intensive benchmarks, producing around 82k device interrupts while sending, and 93k device interrupts while receiving — 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. The client machine was a 1.4GHz Pentium 4, configured for 256MB of RAM, and ran Debian 3.1 from the local disk. Each machine used an Intel 82540 gigabit network card, connected via a gigabit network switch.

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 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.

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 in-place VMM	220.6	98.8%	5.48%
L4Ka::Linux	235.9	97.9%	12.8%
L4 in-place VMM	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 in-place VMM	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 in-place VMM	448.7	99.5%	3.22%
vNUMA in-place VMM	449.1	99.4%	3.31%
Linux 2.6.14 Itanium			
native, raw	435.1	99.5%	
Linux in-place VMM	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%.

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 [23] 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 the original, para-virtualized device driver reuse environment: L4Ka::Linux with virtualized network access via a custom Linux network driver. To evaluate our DP83820 network device model, we ran Netperf in a VM using a pre-virtualized Linux 2.6.

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 function calls to emulation code.

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 in-place VMM	866.7	34.0%	8.37
L4Ka::Linux	775.7	34.5%	9.50
L4 in-place VMM	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 in-place VMM	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%.

The number of device register accesses during Netperf receive was 551k (around 48k/s), and during Netperf send was 1.2M (around 116k/s).

4.1.4 LMbench2

Table 5 summarizes the results from several of the LMbench2 micro benchmarks (updated from the original lmbench [25]), for x86 and Itanium.

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

On Itanium, our pre-virtualized Linux has a clear advantage over the manually para-virtualized XenoLinux. The reason is that Itanium XenoLinux is not completely para-virtualized; only certain sensitive or performance critical paths have been modified (a technique referred to as *optimized para-virtualization* [24]). The remaining privileged instructions fault and are emulated by the hypervisor, which is expensive (as can be seen from the pure virtualization results shown in the same table). In contrast, pre-virtualization can replace *all* of the sensitive instructions in the guest kernel.

4.2 Code expansion

Pre-virtualization potentially introduces code expansion everywhere it emulates a sensitive instruction, which can degrade performance. The cost of the emulation code for an instruction i on a benchmark b is $C_i(b) = F_i(b)E_i$ where $F_i(b)$ is the invocation count for the instruction, and E_i is the code expansion factor (including instruction

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 in-place VMM	780.0	42.5%	11.65
L4Ka::Linux	780.1	35.7%	9.77
L4 in-place VMM	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 in-place VMM	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 in-place VMM	771.4	49.1%	13.59
Receive			
L4Ka::Linux	707.5	60.3%	18.21
L4 in-place VMM	707.1	59.8%	18.06

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

type	null call	null I/O	stat	open close	sig inst	sig hndl	fork	exec	sh
	Linux 2.6.9 on x86								
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
	Linux 2.6.12 on Itanium								
raw	0.04	0.27	1.10	1.99	0.33	1.69	56	316	1451
pure	0.96	6.32	10.69	20.43	7.34	19.26	513	2084	7790
Xeno	0.50	2.91	4.14	7.71	2.89	2.36	164	578	2360
pre	0.04	0.42	1.43	2.60	0.50	2.23	152	566	2231

Table 5: Partial LMbench2 results (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. For Itanium, we also show a minimally modified Linux on Xen, which models pure virtualization (*pure*). The 95% confidence interval is at worst $\pm$ 1.59%.

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 6: The categories of annotations, and the number of annotations (including automatic and manual), for x86. *PDE* refers to page directory entries, and *PTE* refers to page table entries.

scheduling and latency). We informally evaluate this cost for the Netperf receive benchmark.

Although the Linux kernel has many sensitive instructions that we emulate (see Table 6), the Netperf benchmark executes only several of them frequently, as listed in Table 7. 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.

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 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.

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

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

Table 7: Execution profile of the most popular sensitive instructions during the Netperf receive benchmark. Each column lists the number of invocations, where the *count* column is for the entire benchmark.

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).

4.3 Engineering effort

The first in-place VMM 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 in-place VMM became functional with a further one-half person month of effort. Optimizations and heuristics involved further effort.

Table 8 shows the source code distribution for the individual x86 in-place VMMs 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 documented x86 architecture against which we wrote the in-place VMM. The pre-virtualization approach has the disadvantage that it must emulate the platform devices; occasionally they are complicated state machines.

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 8: The distribution of code for the x86 in-place VMMs, 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 9: The number of lines of manual annotations, functional modifications, and performance hooks added to the Linux kernels.

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 in-place VMMs.

In both Linux 2.6 and 2.4 we applied manual annotations, relinked the kernel, added DMA translation support for direct device access, and added L4 performance hooks, as described in Table 9, 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 [2], the authors report that they modified and added 1441 sources lines to Linux and 4620 source lines to Windows XP. In L4Linux [18], 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 [23].

5 Related work

Co-design of a hypervisor and an OS has existed since the dawn of VMs [8, 15]. Several microkernel projects have used it with high-level changes to the guest kernels [16, 18]. Recent hypervisor projects have called it para-virtualization [2, 24, 35].

Some VM projects have added strict virtualization to architectures without such support. Eiraku and Shinjo [11] offer a mode that prefixes every sensitive x86 instruction with a trapping instruction. vBlades [24] and

Denali [35] substitute alternative, trappable instructions for the sensitive instructions.

All major processor vendors announced virtualization extensions to their processor lines: Intel's Virtualization Technology (VT) for x86 and Itanium [21, 22], AMD's Pacifica [1], and IBM's LPAR for PowerPC [20]. 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. The 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 [19]. Thus UMLinux can substitute different ROM implementations for the same Linux binary. 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 recent proposal [33] 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 explicitly designed to be installed 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 has evolved it into an implementation of soft layering, with no-op instruction padding, along with 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.

Binary rewriting and native execution of OS code are usually imperfect and use trapping on sensitive or tagged operations. Sugerman et al. [32] report that more than 77 percent of the overall execution for NIC processing is attributed to VMware's virtualization layer. The typical solution are para-virtualized drivers in the guest OS which communicate directly with the hypervisor, avoiding the trap overhead. However, those drivers are tied to a particular guest and hypervisor, and as such abandon the platform API and increase engineering cost.

6 Discussion

The relationship between guest OSes and hypervisors has many similarities to the relationship between applications and general purpose OSes. General purpose OSes have rich environments of applications created by independent developers, glued together by strong interfaces and conventions, and people expect a certain amount of longevity for their applications; the applications should continue to function (in concert) in spite of OS upgrades. The goal of soft layering is to also provide a fertile environment that promotes collaborative and uncoordinated development, and with longevity: an old guest kernel should function on the latest version of a hypervisor.

General-purpose OSes have a history of providing standardized and abstracted interfaces to their applications. Some will propose the same for para-virtualization — a standard high-level interface used by all hypervisors and guest OSes — but this approach is deficient principally because abstractions lock-in architectural decisions [12, 26], 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, even transparently supporting the hardware acceleration for virtualization of the latest processor generations (para-virtualization's abstractions interfere with transparent access to the new hardware features). Soft layering supports additional abstractions 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 in-place VMM (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 that it replaces 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, rather than system architecture, thus facilitating standardization since it is unnecessary to enshrine proprietary hypervisor enhancements within the standard (e.g., dependence on x86's segments for an additional privilege level). 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.

7 Conclusion and future work

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 to the architectures 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.

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 in-place VMM. 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.

References

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

[2] 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.

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

[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.

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

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

[11] 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.

[12] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *The Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995.

[13] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996.

[14] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. 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, Oct. 2004.

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

[16] 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.

[17] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of the 1996 USENIX Annual Technical Conference*, pages 279–294, San Diego, CA, Jan. 1996.

[26] 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.

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

[28] 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.

[29] 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.

[30] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proc. of the 9th USENIX Security Symposium*, Denver, Colorado, Aug. 2000. USENIX.

[31] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[32] 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.

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

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

[35] 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.

[36] 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.