Pre-Virtualization: Slashing the Cost of Virtualization

Joshua LeVasseur[†] Volkmar Uhlig[§] Matthew Chapman[‡¶] Peter Chubb[‡¶]
Ben Leslie[‡¶] Gernot Heiser[‡¶]

[†]University of Karlsruhe, Germany
[§]IBM T. J. Watson Research Center, New York
[‡]National ICT Australia[*]
[¶]University of New South Wales, Australia

Abstract

Despite its current popularity, para-virtualization has an enormous cost. Its diversion from the platform architecture abandons many of the benefits that come with pure virtualization (the faithful emulation of the platform API): stable and well-defined platform interfaces, single binaries for kernel and device drivers (and thus lower testing, maintenance, and support cost), and vendor independence. These limitations are accepted as inevitable for significantly better performance and the ability to provide virtualization-like behavior on non-virtualizable hardware, such as x86.

We argue that the above limitations are not inevitable, and present *pre-virtualization*, which preserves the benefits of full virtualization without sacrificing the performance benefits of para-virtualization. In a semi-automatic step an OS is *prepared* for virtualization. The required modifications are orders of magnitudes smaller than for para-virtualization. A virtualization module, that is collocated with the guest OS, transforms the standard platform API into the respective hypervisor API. The guest OS is still programmed against a common architecture, and the binary remains fully functional on bare hardware. The support of a new hypervisor or updated interface only requires the implementation of a single interface mapping. We validated our approach for a variety of hypervisors, on two very different hardware platforms (x86 and Itanium), with multiple generations of Linux as guests. We found that pre-virtualization achieves essentially the same performance as para-virtualization, at a fraction of the engineering cost.

1. INTRODUCTION

Virtual machines, originally introduced in the '70s [8], have recently gained immense popularity. The reason is that they provide an attractive approach to solving a variety of problems arising in a number of quite dissimilar contexts. These include server consolidation [2, 31], migration of users' complete operating environments between physical machines [9, 26], intrusion detection [14], debugging [15], secure computing platforms with strictly controlled information flow [4, 7, 20, 25], co-existence of realtime and non-realtime applications [19], and backward compatibility with older or out-of-production hardware.

Notwithstanding their popularity, virtual machine (VM) technologies have significant cost, and different approaches to virtualization differ primarily in how they trade off engineering cost against run-time overheads.

Pure virtualization, the classical approach, employs a VM API that is a faithful copy of the platform API (i.e., instruction set and devices). This approach has the great benefit that there is zero cost for porting and maintaining a guest operating system (OS) to run on the VM; the only engineering cost of this approach is for the development and maintenance of the VM itself. This cost is also minimized by virtue of the enormous stability of a typical platform API. In exchange, pure virtualization has a high run-time overhead, resulting from frequent switches between user mode and privileged mode, which severely limits the use of this technology. Furthermore, most contemporary instruction sets are not fully virtualizable, making pure virtualization extremely hard.

The popular alternative to pure virtualization is to replace the low-level platform API by a high-level API that requires far fewer mode switches when running a de-privileged guest OS — an approach called *para-virtualization* [31]. Para-virtualization mostly eliminates the run-time cost of the VM [2], at the expense of high engineering cost, which comes in several guises:

- the high cost of porting a guest OS to the virtual machine, and maintaining the port as the guest OS develops. Para-virtualization adds a new architecture (or several) to the guest for each hypervisor API. This cost is amplified by evolution of the hypervisor API;
- the cost of distributing more separate guest binaries for VM execution. While pure virtualization runs a standard guest binary (built for native execution), para-virtualization requires the distribution of separate binaries for each VM and thus extra testing, maintenance, and support. This is a particular deterrent for companies whose business is built on packaging and

[*]National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

Technical Report 2005-30, November 2005, Fakultät für Informatik, Universität Karlsruhe (TH)

distributing open-source operating systems and companies that only provide binary extensions;

- the cost for integrating new processor features into the hypervisor API is higher than for pure virtualization, as the hardware and hypervisor interfaces evolve independently. A bad design choice in the hypervisor API (including the binary interface) has long-term repercussions: In order to preserve backward compatibility to already deployed OSes — a prime benefit of virtualization — requires the hypervisor to support all previous interfaces. Besides the additional maintainance cost, interface backward compatibility may lead to structural (and performance) compromises in the hypervisor and increases the trusted computing base. A small code base is particularly critical for formal verification of a hypervisor [28].

While some of those costs can be mitigated by standardizing on a single platform API, the significant changes in successive versions of Xen [2, 22] (the at present arguably most popular para-virtualization hypervisor) indicate that the technology is far away from a single, stable API. In fact, we consider it unlikely that — in the foreseeable future — a single VM API will be able to support the wide range of uses of virtualization.

VMware's approach of rewriting the guest OS binary [29] (necessitated by the unvirtualizable x86 architecture) is somewhere in between: it eliminates the cost of porting to the VM, but cannot eliminate the run-time overhead to the same extent as with para-virtualization, still depends on assumptions or inside-knowledge about the guest implementation, and has high engineering cost for the VM supplier.

Ideally, one would like to combine the advantages of pure and para-virtualization by an approach that

- retains the low-level hardware API of pure virtualization, and thus avoids the engineering cost of porting to a hypervisor
- achieves the run-time performance of para-virtualization.

Such an approach dramatically reduces the overall cost of virtualization. In this paper we argue that what sounds like the holy grail of virtualization is not impossible. We present a constructive proof in the form of a new technique we call *pre-virtualization*, which comes very close to achieving the goal. Specifically we show that pre-virtualization mostly eliminates the guest-side engineering cost, yet matches the run-time performance of para-virtualization.

The remainder of this paper presents pre-virtualization, our implementation of it, an evaluation, and a discussion of its advantages and limitations.

2. PRE-VIRTUALIZATION

2.1 Overview

If pre-virtualization is to deliver on the above goals, it must be able to utilize powerful high-level hypervisor mechanisms in a similar fashion as para-virtualization. The requirement to retain the low-level hardware API, as seen by the guest, implies that the use of such high-level mechanisms must be transparent to the guest. In other words, pre-virtualization must perform a translation from the guest-visible platform API to a hypervisor API, in order to maintain overall API transparency.

The API translation layer must be invoked by the guest OS with minimal overhead, in particular without mode or context switches. This implies that it must reside in the guest's protection domain. We call this layer, which translates between the guest-visible platform API and the underlying hypervisor API, the *in-place VMM*. Figure 1 compares the architectures of different approaches to virtualization.

In a sense, the in-place VMM operates in the opposite way of more familiar translation layers (e.g. system libraries), by mapping a low-level API used by the application to an underlying higher-level API.

The in-place VMM is unprivileged and part of the guest's address space. This means it can be invoked via a function-call interface, avoiding the overhead of protection-level (trap) or protection-domain switches, and supporting optimizations similar to para-virtualization. As it only performs an interface transformation, the unprivileged nature of the in-place VMM does not create a security problem.

The high-level view of pre-virtualization presented so far does not explain how the two critical steps of any virtual machine are performed — (1) locating and identifying virtualization-sensitive instructions, and (2) emulating the instructions.

2.2 Locating and Identifying Instructions

Pure virtualization locates sensitive instructions by relying on them being privileged [21], so their execution by the guest causes a trap into the hypervisor — the source of the high run-time overhead of pure virtualization. Para-virtualization manually locates sensitive instructions and replaces them (including potentially many non-sensitive instructions) by explicit hypervisor calls, a transformation at the source-code level which is responsible for the high engineering cost of para-virtualization.

A way to avoid this engineering cost is to perform an automated scan of the guest kernel for sensitive instructions. While VMware makes a heroic effort to perform this scan on the kernel at run time, we perform it when detailed knowledge of the source code is available: the assembler stage, as pioneered by Eiraku and Shinjo [6]. In contrast to binary scanning, the assembler files (whether written by hand or generated by the compiler) clearly differentiate between code and data, and the instructions are by definition parseable.

Additionally, the assembler files provide supplementary information, including basic block boundaries and function boundaries. For assembler files generated by the compiler, richer information is available such as register data flow between basic blocks (if the compiler is designed to share the information; gcc can emit supplementary information). This supplementary information is valuable for the later emulation of the instruction.

One problem is that some instructions which are not inherently virtualization-sensitive can be sensitive in certain contexts. This includes simple `mov` instructions used to access architecture-defined memory objects, such as page tables and device registers. In particular x86 has many types of memory objects, such as the TSS, the GDT, and the LDT. Updates to the memory objects often generate synchronous side effects, such as enabling an interrupt, or reading the

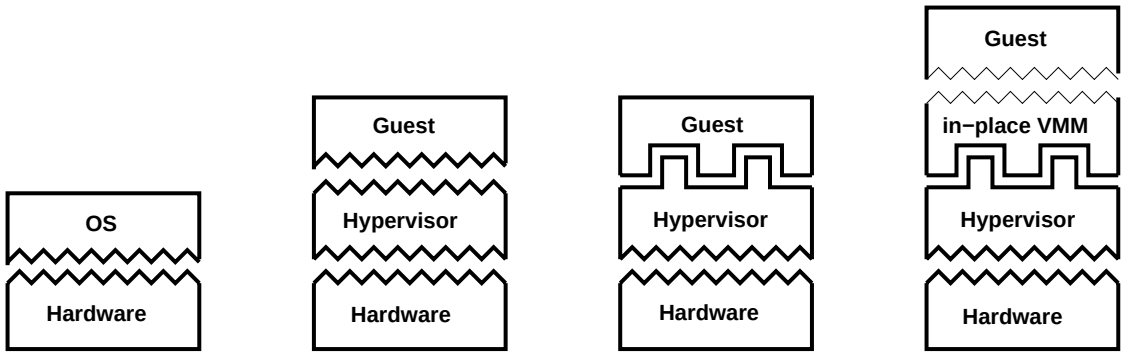

Figure 1: Comparison of virtualization strategies (left to right): (a) native execution — no virtual machine; (b) pure virtualization — VM/hypervisor is API-transparent; (c) para-virtualization — VM/hypervisor presents a changed API; (d) pre-virtualization — a para-virtualizing hypervisor API is mapped back to the hardware API by the in-place VMM, making the overall VM API-transparent (thin lines indicate an interface without privilege change).

status bits from a page table entry. While in native execution those side effects are generated by the hardware, under virtualization they must be emulated by the VMM. We call such operations *sensitive memory operations*, since the illusion of virtualization is sensitive to the emulation of such instructions.

Sensitive memory operations can be forced to trap by protecting the memory objects from access, but this creates significant overhead. Automated scanning for sensitive memory operations, e.g. via compiler data-flow analysis, might be possible but is certainly difficult. Instead we propose a profile-feedback loop: Execute the guest OS in a VM, stimulate it to generate a sufficient coverage of sensitive memory operations, emulate them by trapping, and record their instruction locations. The OS can then be re-compiled, using the profiling data to identify additional instructions that need to be prepared for virtualization. No harm is done by incomplete profile coverage, as sensitive memory operations which have been overlooked will just create traps and be emulated as normal. As long as the profiling run detected all frequently-executed sensitive memory operations, the performance impact will be minimal.

This approach supports a fully-automated discovery of sensitive operations. For guest OSes which abstract operations on memory objects through a function interface (which can be determined via a quick inspection of the source code), it is possible to identify the sensitive memory operations by adding annotations directly to the guest OS's abstractions. Note that such manual annotations do not interfere with the platform API as they only record the locations of the instructions. They are also not required for correctness, only for performance — a missed annotation will result in a trap leading to correct (but expensive) virtualization. The engineering cost of inserting those annotations is tiny compared to the invasive para-virtualization approach.

2.3 Instruction Emulation

In pre-virtualization it is possible to emit the instruction emulation code directly from the compiler, thus using the compiler to statically translate from one API to another [6]. However, this generates a different API, leading to an OS executable that will only run on a particular hypervisor, and hence incomplete virtualization. For complete virtualization the emulation must be enabled only during or after loading of the OS binary into the VM.

We rewrite the instructions with emulation code when the VMM loads the guest OS (and when a guest OS later loads a dynamic module). In order to avoid a VMware-style brute-force rewrite of the code, we require access to some of the original compiler state. We collect this state in a database during compilation. The database is stored in a separate ELF section of the binary and is provided to the VMM for translating the sensitive instructions from the original platform API to the target API.

In general it is difficult to rewrite instructions *in situ*, in particular for architectures with variable instruction length. We therefore use the compiler to prepare the output binary for later rewriting and in a manner that maintains the platform API: Sensitive instructions are padded with innocuous NOPs.[1] The NOP instructions provide space to write the emulation instructions, or at least enough space to branch to emulation code in the in-place VMM.

2.4 Performance Optimizations

We observed that the quality of the emulation code greatly influences the performance of the OS. The instruction emulation leads to code expansion: We replace a single sensitive instruction with several, or perhaps many, emulation instructions. Linux executes a small subset of x86 sensitive instructions very frequently, and we observe on kernel-intensive benchmarks (such as Netperf) a noticeable increase in CPU utilization with careless code expansion. Such performance-critical instructions include enabling and disabling interrupts and reading and writing segment registers.

Para-virtualization addresses this issue via normal compiler optimizations. We instead employ heuristics to reduce the code expansion, by ensuring that performance-critical sensitive instructions are replaced by just one or two emulation instructions.

Another performance issue arises from the need of in-

[1]Instruction padding must not violate sequence-sensitive transitions on raw hardware, such as x86's interrupt window or branch delay slots.

struction emulation to interface with a VMM state machine. In pure virtualization, each individual sensitive instructions causes a trap, leading to poor performance. Paravirtualization replaces whole instruction sequences with macro operations involving a single hypervisor call. In pre-virtualization we keep emulation of individual instructions separate, but replace traps by function calls to the in-place VMM's state machine. The in-place VMM applies heuristics for converting the individual emulation operations into the equivalent of accumulated hypervisor calls of paravirtualization. In the following we summarize the most important heuristics.

2.4.1 Code Inlining

To minimize instruction expansion, we convert the high-frequency operations into inlined emulation code. For x86, the critical instructions are those that manipulate segment registers, and enable and disable interrupts. The segment register operations are simple to inline within the original instruction stream: The emulated code directly accesses the virtual CPU of the in-place VMM with a `mov` instruction, or a `mov` and `push/pop`. The `cli` instruction is also easy to inline, with a simple `btr` instruction for disabling interrupts in the virtual CPU. The `sti` instruction, which enables interrupts, is a problem because on real hardware it immediately delivers pending interrupts, requiring substantial emulation code. We replace `sti` with a single `bts` instruction, to perform the opposite of `cli`, and use a heuristic to deliver pending interrupts.

2.4.2 Heuristic: Interrupt Enable

Since the `sti` instruction is critical to performance, we optimize for the common case and thus avoid delivery of pending interrupts when `sti` is executed. It is pretty rare to encounter a pending interrupt at the time of `sti`, and thus the instruction expansion is unjustifiable.[2] Instead we have found it sufficient to deliver all pending interrupts during the idle loop, when transitioning to user mode, and when returning from an interrupt handler. It turns out that these are excellent times to deliver synchronous interrupts, since the kernel is completely finished with its other operations and the delivery of an interrupt will not preempt other kernel activity. Our heuristic may increase interrupt latency, but running within a VM environment already increases the latency due to arbitrary preemption of the virtual CPU.

To support efficient virtualization of the interrupt flag, our in-place VMM *always* accepts asynchronous interrupts from the hypervisor, even if the guest OS disabled interrupt delivery. The interrupts are buffered by the in-place VMM, and asynchronous traps are delivered to the guest OS only if the virtual interrupt flag is enabled. Otherwise, the interrupts are delivered synchronously, as described.

2.4.3 Heuristic: Page Table Operations

Page table operations often happen in batches, such as at `fork()` and `exec()`. The updates are batchable and can be delayed, since page table updates have TLB semantics: Their side effects are enabled at the time of a page fault, or when explicitly invalidating the TLB. Thus `fork()` and `exec()` can be fairly efficient. The problem is when the OS reads from a page table entry, since it may want to examine access bits. Reads therefore need to be synchronous operations, but the access bits can be prefetched. Access-bit checks tend to be infrequent operations, though.

The Xen hypervisor [2] does not provide virtualized page tables, instead it relies on cooperative page table sharing between the hypervisor and the guest OS. For pass-through page tables we use completely different heuristics. Most writes must be synchronous, since the guest OS may immediately afterward read the updated page-table entry (a read-after-write hazard). In order to enable batching for `fork()` and `exec()` on Xen, we disconnect page tables from the page directory, so that they are no longer active, and are permitted to be efficiently read and written by the guest OS. We reconnect the page table upon a page fault in the virtual address range covered by the page table, and restart the faulting instruction. We avoid disconnecting page tables if the guest OS appears to be updating a page table in response to a normal page fault, since the appropriate Xen hypercall is cheaper.

2.4.4 Heuristic: Device Emulation

Pre-virtualization is unique for its ability to efficiently model a *real device*. All other virtualization approaches require special device drivers for performance, which inherently tie the guest OS to a particular hypervisor. We avoid this problem by emulating standard devices, thus obeying the platform API in line with our goal of providing the full benefits of virtualization.

Device drivers are notorious for frequent execution of sensitive operations (device register accesses), leading to a massive performance bottleneck when emulated via traps [27]. Pre-virtualization avoids those traps by replacing the device-sensitive instructions with emulation instructions. We convert the individual updates into function calls, and then batch state changes. For example, we add outbound network packets to a producer/consumer ring, so if the guest OS is sending a file via the network, we batch all the packets for the file into the ring, and occasionally inform the hypervisor that pending packets are ready for transmission. Choosing the proper time for packet delivery is *critical* for network throughput and latency, and we have found returning from interrupt and entering user mode are perfect times for this. These are typically times when the kernel is finished processing packets and has nothing more to send.

3. IMPLEMENTATION

We implemented a pre-virtualization environment according to the principles and architecture described in the prior sections on x86, and a slight variant on Itanium. We describe in detail virtualization automation, our in-place VMM, and our pre-virtualized network device model as used on x86. We describe implementations for two x86 hypervisors that have very different APIs, the L4Ka::Pistachio microkernel and the Xen 2.0 hypervisor, to demonstrate the versatility of virtualizing at the platform API. We also describe the Itanium versions, where we support three hypervisors, also with very different APIs: Xen/ia64 [22], vNUMA [3] and Linux.

3.1 Instruction Scanning

We use macro replacement at the assembler level to locate and transform sensitive instructions, with one macro defined

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

for each sensitive instruction. The macro replacement is performed on the intermediate output of gcc independent of whether the file is compiled C code or hand written assembler. In contrast to para-virtualization, there is no danger of missing a sensitive instruction via this automated process; bugs are only possible if the macros and tools are misconfigured. The assembler macros limit us to extracting only the locations of the instructions; they do not support extraction of further supplementary information, such as basic block boundaries.

Some guest kernels compile code that is mapped into and executed by the guest's applications, such as Linux 2.6's system call trampoline page; we apply a different set of macros for these files, permitting optimization of the application's system call path.

The macros append a sufficient number of NOP instructions to the sensitive instruction for later rewriting, and record the instruction's location in a dedicated ELF section of the kernel's binary; see Figure 2 for an example macro. The additional NOPs are compatible with the OS code, because symbols are still relative at the assembler phase, and thus the NOPs expand the size of their surrounding basic blocks.

One issue is that certain basic blocks may be limited to an absolute size. This is the case for interrupt vectors on some CPU architectures such as Itanium. In practice, by carefully implementing the macros used in such contexts to minimize code expansion, we were able to avoid overflowing those blocks. A related issue is that hand-written assembly code may make assumptions about code layout. For instance, on x86 a software interrupt handler typically decrements the caller IP by two bytes to restart a system call; this can be handled by careful code construction. In one instance, an IP-relative calculation in the Itanium kernel produced incorrect results; it was possible to modify the code to eliminate the assumption, without adversely impacting the function or performance of the kernel on real hardware.

3.2 Annotations for Sensitive Memory

In some OSes, such as Linux, the sensitive memory operations are adequately abstracted, enabling simple manual annotations of the sensitive memory instructions; i.e., we manually apply our annotation as inlined assembler to the macro that wraps the operation. Our annotations also force the memory operation to use an easily decodable instruction.

For Linux 2.4 and 2.6, we use manual annotations, which avoids enabling the runtime support required by profiling to supplement incomplete profiles. We have not yet completed our profiler support; currently we have only enhanced the assembler to apply macros based on profiler feedback.

3.3 Instruction Rewriting

At boot time, and when loading dynamic kernel modules, the VMM rewrites the sensitive instructions with emulation code. The VMM obtains the locations of the sensitive instructions from the ELF sections added by the assembler to the guest binary.

The rewriter decodes each sensitive instruction to determine intention, scratch registers, and register data flow, for writing optimized substitute code. For sensitive memory operations, the rewriter decodes the instruction to determine register flow, but the intention of the operation was originally determined by the person that added the annotation (and is easily determined by a profiler too), and is thus read from the ELF section (i.e., does the memory operation read a page table entry or access the interrupt controller?). We could keep the instruction decoder quite simple, [3] since our manual annotations force the use of an easily decodable instruction.

```
.macro lldt seg
push \seg
call emul_lldt
add $4,%esp
.endm
```

```
.macro lldt seg
lldt \seg
nop
nop
.endm
```

Figure 2: Example of assembler macros for virtualization of a sensitive x86 instruction (`lldt`). Static translation (left) replaces the instruction by a jump to emulation code, while dynamic translation (right) only adds sufficient *NOP* instructions to leave space for boot-time substitution.

3.4 In-Place VMM Implementation

In contrast to para-virtualization, the structure of the in-place VMM is independent of guest OS source code. The in-place VMM can use different coding techniques, an alternative language (we use C++ for x86, assembler for Itanium), and avoids licensing conflicts (e.g., can use a proprietary license while running a GPL-ed guest OS).

The in-place VMM is divided into a front-end and a back-end. The front-end emulates the platform architecture; the sensitive instructions of the guest OS are rewritten to interact with the front-end. The back-end translates platform operations to the hypervisor API, when visible side effects become necessary, such as invalidating a TLB entry. The back-end also translates asynchronous events from the hypervisor, such as interrupts, into calls to the front-end. The performance of pre-virtualization relies on the division between front-end and back-end; the in-place VMM maps the micro operations of the platform API into the macro operations typical of para-virtualization.

The modularity permits easy adaptation of the in-place VMM to new architectures and hypervisors. A front-end can service different back-ends, so that adding a new hypervisor only requires a new back-end. And a back-end is useable across a variety of front-ends, e.g., the L4Ka::Pistachio API is portable across many architectures and so the back-end is also portable across architectures.

All of our x86 in-place VMMs execute the same, pre-virtualized guest OS binary.

3.4.1 x86 Xen In-Place VMM

We support Xen on two different platforms, x86 and Itanium. We only discuss the x86-specifics here, and leave Itanium to Section 3.4.3.

The x86 Xen API closely resembles the hardware API. Still, the in-place VMM must intercept all Xen API interactions to enforce the actual virtualization of the guest OS. Interrupts, exceptions, and x86 traps are delivered to the in-place VMM which updates the virtual CPU state machine and then transitions to the installed handler of the guest OS.

[3] At this point, dynamic instruction rewriting is only supported on x86; on Itanium we use static binding of in-place VMM and guest OS.

When the guest OS transitions to user-mode, the in-place VMM intercepts the operation, updates the virtual CPU, and then completes the transition.

Xen's API for constructing page mappings uses the guest OS's page table as the actual x86 hardware page table. The in-place VMM virtualizes the hardware page table for the guest OS, and thus intercepts all accesses to the page table. This is the most complicated aspect of the API, since Xen must ensure that its applications never insert mappings that could compromise the Xen hypervisor. The in-place VMM transparently write-protects security-vulnerable mappings to protect the hardware page tables.

3.4.2 L4Ka::Pistachio In-Place VMM

The L4Ka::Pistachio API consists of a set of portable microkernel abstractions, and thus are high-level. The API is very different from Xen's x86-specific API, yet pre-virtualization works in both cases for mapping the platform API to the hypervisor API, and the same x86 front-end is used for both in-place VMMs.

For performance reasons, an address space switch of the guest OS is mapped to an address space switch in L4. The in-place VMM associates one L4 address space with each guest address space; L4 is shadowing the page tables of the guest OS.

The shadow page tables are updated lazily: Mappings are created upon a miss and are destroyed when the guest OS overwrites the page tables and updates the TLB. Shadow page tables resemble the classical virtual machine model where memory can be revoked transparently. This model differs from Xen, where page tables are shared between guest and hypervisor and therefore requires a cooperating guest OS for memory revocation.

L4 does not support asynchronous signal delivery but requires a rendezvous of two threads via IPC; hardware interrupts and timer events are mapped 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 VM thread or updates the virtual CPU model accordingly (e.g., register a pending interrupt when interrupts are disabled).

As described in Section 5.3, we added several transparent hooks to the guest OS, to accommodate inefficiencies in the L4 API for virtualization.

3.4.3 Itanium In-Place VMMs

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

One very useful feature of the Itanium architecture is the `epc` instruction (*enter privileged code*). `epc` raises the privilege level without a stall or pipeline flush, and thus enters the hypervisor in a single cycle. This enables efficient hypercalls, and greatly reduces the need for complicated batching logic. Since it does require hypervisor co-design, this mechanism is currently only used in the vNUMA backend.

On the other hand, the RISC nature of the Itanium architecture makes implementing a transparent in-place VMM somewhat more complicated than on 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 is to virtualize a subset of the machine registers that are rarely used by compiler-generated code, 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 a single, non-interruptible 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.

3.5 Network Device Emulation

A prime advantage of pre-virtualization is the ability to efficiently model a real device, and conform to the platform API. We implemented a device model for the DP83820 gigabit network card. The DP83820 has a thoroughly documented interface, which is simple to emulate efficiently. Additionally, the device is designed for high-throughput packet streams, and thus uses producer-consumer rings to transfer packets, where packets are guaranteed to be pinned in memory for the DMA operation. We chose to model this device with the intention of matching the performance of the typical custom network drivers used in VM environments. A drawback is that only recent operating systems, such as Linux 2.4 and Linux 2.6, support this gigabit card; Linux 2.2 does not.

To pre-virtualize the guest OS's DP83820 driver, we annotate the memory-sensitive instructions of the guest OS that manipulate the DP83820 device registers, and rewrite the instructions when loading the guest OS in the VMM.

The DP83820 model is also split between a front-end and a back-end. The front-end models the device registers, while the back-end translates device operations into the networking API of the hypervisor.

The driver's micro updates to the device registers are rewritten into function calls to the in-place VMM's front-end, thus avoiding the trap bottleneck common in virtualization. The front-end applies heuristics to determine when to transmit packets via the back-end. Network devices asynchronously receive packets; the back-end receives packets from the hypervisor, and passes them to the front-end, which inserts them into the DP83820's producer-consumer ring.

Outbound network packet delivery requires synchroniza-

tion of state with an external process, adding overhead to the packet delivery due to the protection-domain crossing (even if only performing a hypercall). The standard approach is to amortize the cost of the protection domain crossing across multiple packets, by batching outgoing packets. With performance analysis, we have found that good points to synchronize are when returning from the guest kernel's interrupt handler, transitioning to user, and entering idle.

4. EVALUATION

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

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; the baseline is a para-virtualized device driver reuse environment [16], running two virtual machines, one providing device services to the other. Our pre-virtualization network device evaluation used the same architecture.

The benchmark setups used identical configurations as much as possible, in order 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 the XT-PIC. 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, thus exercising `fork()` and `exec()` and the normal page fault handling code, 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.

System	Time [s]	CPU util	O/H [%]
Linux 2.6.9 x86			
native, raw	211.5	98.6%	
NOPs, raw	209.5	98.5%	-0.98%
XenoLinux	218.8	97.8%	3.44%
Xen in-place VMM	226.2	98.4%	6.96%
L4Ka::Linux	235.9	97.9%	11.5%
L4Ka in-place VMM	239.6	98.7%	13.3%
Linux 2.4.31 x86			
native, raw	206.8	98.6%	
NOPs, raw	206.8	98.7%	0.00%
Xen in-place VMM	224.0	98.8%	8.31%
Linux 2.6.12 Itanium			
native, raw	434.7	99.6%	
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 ± 0.43%.

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. We do not include data for XenoLinux 2.4, because Xen has discontinued its support.

The 3.4% performance difference between XenoLinux and the Xen in-place VMM for Linux 2.6.9 seems to be due mainly to execution of more page table hypercalls when running a pre-virtualized Linux. For example, according to the microbenchmarks in Section 4.1.4, the process creation overhead is 693 cycles higher than in XenoLinux, which alone contributes about 3s to the entire kernel build benchmark. We have pending optimizations to batch more operations across hypercalls, which will hopefully address this issue.

The annotated and padded binary showed a performance anomaly: When running the padded binary on bare hardware, the performance slightly improved over the original binary. We account this effect to the Pentium 4's microarchitecture, specifically the tightly integrated trace cache and branch-prediction unit.

We have not yet investigated the reason for the additional 1.35% performance penalty of Linux 2.4.31 running on the Xen in-place VMM, compared to Linux 2.6.9 running on the Xen in-place VMM.

4.1.2 Netperf

We used the Netperf send and receive network benchmarks to analyze I/O performance. Our benchmark transfered a gigabyte of data at standard Ethernet packet size, with 256kB socket buffers. These are I/O-intensive bench-

System	Xput [Mb/s]	CPU util	cyc/B
Linux 2.6.9 x86			
native, raw	866.1	28.8%	7.10
NOPs, raw	867.7	27.3%	6.73
XenoLinux	867.6	33.8%	8.32
Xen in-place VMM	866.2	33.8%	8.35
L4Ka::Linux	775.7	34.5%	9.50
L4Ka in-place VMM	866.5	30.2%	7.45
Linux 2.4.31 x86			
native, raw	779.6	39.7%	10.88
NOPs, raw	779.7	39.6%	10.85
Xen in-place VMM	778.3	44.8%	12.29

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 ± 0.25%.

System	Xput [Mb/s]	CPU util	cyc/B
Linux 2.6.9 x86			
native, raw	780.9	35.2%	9.64
NOPs, raw	780.2	33.5%	9.17
XenoLinux	780.7	41.3%	11.29
Xen in-place VMM	778.7	41.1%	11.28
L4Ka::Linux	780.1	35.7%	9.77
L4Ka in-place VMM	779.8	37.3%	10.22
Linux 2.4.31 x86			
native, raw	740.8	36.0%	10.39
NOPs, raw	740.8	36.4%	10.49
Xen in-place VMM	739.6	43.2%	12.48

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

marks, 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. The first rows apply to Linux 2.6.9. They compare native Linux on raw hardware, a pre-virtualized Linux (with NOPs) on raw hardware, XenoLinux and pre-virtualized Linux on Xen 2.0.2, and para-virtualized Linux (L4Ka::Linux) and pre-virtualized Linux on the L4Ka::Pistachio microkernel. The three last rows repeat several of the experiments for Linux 2.4.31.

In general, the performance of the pre-virtualized setups matched that of the para-virtualized setups. In regards to system behavior, and mapping of micro operations to macro operations, our L4 system offers kernel event counters covering a variety of events such as interrupts, protection domain crossings, and traps caused by guest OSes. The event-counter signature of the para-virtualized Linux on L4 was nearly identical to the even-counter signature for the pre-virtualized Linux on L4.

4.1.3 Network Device Model

We also used Netperf to evaluate the DP83820 network device model, and in this case, the Netperf VM had to have only indirect access to the network hardware. Thus we used a device driver reuse environment [16] based on the L4Ka::Pistachio microkernel, to connect the Netperf VM 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 hosted a kernel module communicating via L4 IPC with the Netperf VM, to convert the DP83820 device requests into Linux internal network operations.

In the baseline case, the Netperf VM used the para-virtualized L4Ka::Linux, and a custom Linux network driver

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

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

to provide virtualized network access. To evaluate the 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. 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, for x86 and Itanium.

On x86, the null system call for pre-virtualized Linux is $0.11\mu s$ more costly than on raw hardware, and is reflected in many of the micro benchmarks. It executes more instructions at two code paths: when the trap vector for a system call is delivered, and when returning to user mode. The delivery updates virtual CPU state, and is optimized. The return is described in Section 4.2.2, and is unoptimized. The extra expense for `fork()`, `exec()`, and for starting `/bin/sh` seem to be due to an excessive number of hypercalls; as noted earlier we are working on addressing this.

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 *opti-*

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	1.38	2.00	0.91	3.01	77	312	5958
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	0.57	0.64	1.39	2.07	1.03	1.73	221	696	7404
	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. All are microseconds, and 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 ± 1.59%.

mised para-virtualization [18]). 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 Pre-Virtualization Mechanisms

Pre-virtualization has two steps for achieving virtualization: locating the sensitive instructions, and rewriting the instructions to invoke emulation code. We evaluated the result of locating the sensitive operations, and the code rewriting quality.

4.2.1 Annotations

The assembler automatically added annotations for the sensitive instructions, and we added manual annotations to the sensitive memory instructions in the C code that were then instantiated by the compiler. Our pre-virtualized Linux 2.6.9 for x86 had the following annotations: 5181 sensitive instructions (including port accesses to the XT-PIC and other devices via the `in` and `out` instructions), 17 page directory writes, 33 page table entry (PTE) writes, 8 PTE read-and-clears via the `xchg` instruction, 5 PTE test-and-clears via the `btr` instruction, and 103 device register accesses for the DP83820 driver model.

Our pre-virtualized Linux 2.4.31 for x86 had the following annotations: 3072 sensitive instructions (including port accesses), 20 page directory writes, 30 PTE writes, 6 PTE read-and-clears, 3 PTE test-and-clears, and 111 device register access for the DP83820 driver model.

4.2.2 Code Expansion

Table 6 lists the most frequently executed sensitive instructions during the Netperf receive benchmark. Of the instructions, `cli`, `sti`, `pushf`, and `popf` are by far the most frequently executed and therefore most performance-critical virtualizations.

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 clobbers 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.

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 6: 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.

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 less frequently-executed instructions are primarily a concern for micro benchmarks, such as LMbench2, and do not execute often enough in our macro benchmarks to warrant severe optimization (for example, the system call overhead using the in-place VMM on Xen is 0.23% of the total kernel build time on raw Linux, and `iret` is partially responsible for this overhead). Of interest are those instructions which expand significantly during virtualization, which are `iret`, `hlt` and `out`. The remaining instructions, for reading and writing segment registers, translate into one or two instructions for manipulating the virtual CPU.

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. Thus the single `iret` instruction expands considerably.

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

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

Our optimizations avoid code expansion problems for the critical instructions. The less critical instructions do not execute frequently enough to penalize system performance in macro workloads. In several cases, we substitute faster instructions for the privileged instructions (e.g., replacing `sti`/`cli` with bit-set and bit-clear instructions).

Type	Headers	Source
Common	686	746
Device	745	1621
x86 front-end	840	4464
L4 back-end	640	3730
Linux back-end	168	4271
Xen back-end	679	2753

Table 7: 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 2.4.31
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 8: The number of lines of manual annotations, functional modifications, and performance hooks added to the Linux kernels.

4.3 Engineering Effort

The first in-place VMM supported x86 and the L4Ka::Pistachio microkernel, and provided some basic device models (e.g., the XT-PIC). The x86 front-end, L4 back-end, device models, assembler macros, and assembler enhancements 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 7 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 L4Ka::Pistachio 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, defined by hard-to-obtain specifications.

After completion of the initial infrastructure, developed while using Linux 2.6.9, we pre-virtualized Linux 2.4.31 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, and added L4 performance hooks, as described in Table 8, totalling 257 lines for Linux 2.6.9 and 142 lines for Linux 2.4.31. 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 (while their publicly available XenoLinux 2.6.9 sparse source tree contains a maintenance nightmare of 28104 lines of code), and 4620 source lines to Windows XP. In L4Linux [10], the authors report that they modified and added 6500 source lines to Linux 2.0. Our para-virtualized Linux 2.6 port to L4Ka::Pistachio with a focus on small changes still required about 3000 modified lines of code [16].

5. DISCUSSION

5.1 Achievements

5.1.1 Performance

Our macro benchmarks show that for Linux as a guest, pre-virtualization roughly matches the performance of highly-optimized para-virtualized approaches on both the x86 and Itanium architectures. A small (3.5%) performance gap for kernel compiles in the case of the Xen hypervisor shows that minor gains are possible with (highly invasive) para-virtualization, but we expect that this can be addressed by improved annotations or virtualization expansions, once the reasons are identified. Profiling so far at least indicates that the problem does not seem to be structural.

The Netperf results are particularly encouraging, as they show that even device drivers, which are characterized by very frequent execution of sensitive instructions, can be pre-virtualized very efficiently. Furthermore, the benchmarks of the virtual-device model show that pre-virtualization efficiently supports the use of standardized virtual devices.

Our benchmarks also established that pre-virtualization does indeed allow us to run a single OS binary on bare hardware as well as on several supported hypervisors; in this sense, pre-virtualization behaves exactly like pure virtualization.

5.1.2 Engineering cost

Our experience shows that the engineering cost of pre-virtualization is orders of magnitude less than that of para-virtualization. While we are quite aware that development times are notoriously difficult to measure in a research environment, and there is a strong tendency among researchers to under-estimate and under-report their development cost, there is a combination of independent evidence to support this assertion:

- the number of lines of guest code touched by our annotations is several orders of magnitude less than the patches to Linux required for para-virtualizing on Xen or L4;

- the annotations apply to internal abstractions of the guest, rather than implementation details. This not only means that far less understanding of the guest internals is required for applying them, but also that there is dramatically less maintenance cost. This is clearly supported by the ease of pre-virtualizing earlier versions of Linux once a recent version had been successfully pre-virtualized, compared to the Xen team discontinuing support for the (still widely-used) Linux 2.4 kernels, apparently due to the high maintenance cost;

- the same guest binary runs on all supported hypervisors as well as on bare hardware;
- pre-virtualization outperformed para-virtualization of Linux on Xen on Itanium (where Xen is less mature than on x86), which indicates how much easier it is to get good performance from pre-virtualization;
- device virtualization supports the unmodified re-use of existing device drivers for virtual devices.

This dramatic reduction in engineering cost was achieved with the present semi-automated approach based on automatic macro expansion by the assembler and aided by manual annotations to identify sensitive memory operations (otherwise innocuous instructions). A great deal more automation is possible with the profiling approach discussed in Section 2.2, and if VMware's recent VMI proposal for virtualization-friendly internal OS interfaces [30] is adopted, it may completely eliminate the need for manual interference with the guest source. Hence, a further significant reduction of engineering effort is possible, to the point where the engineering cost approximates that of pure virtualization.

5.2 Implications

We believe that our results make a compelling cost case in favor of pre-virtualization: the performance of para-virtualization is achieved at a fraction of the engineering cost. Let us examine what this buys us.

5.2.1 Single binary distribution

A very obvious consequence of our results is that operating-system vendors, who at present typically distribute a single OS version per architecture (built for native execution) can continue to do so, but support virtual machines as well. The only change is that they would distribute a pre-virtualized OS. As we found, this can be run on bare hardware without loss of performance.

We imagine that this is an attractive proposition particularly for vendors of open-source operating systems, for whom the distribution of versions of additional platforms must be a significant part of their overheads. Here the manual annotation (as opposed to the profiling-based) approach to pre-virtualization enables users to build their own kernels from the sources used by the distributeos, and to generate the same final binary.

5.2.2 Hypervisor neutrality and support for VM innovation

A very powerful feature of pre-virtualization is that it makes the guest OS (source *and* binary) independent of the hypervisor it is to run on. Given the changes the Xen API went through over the last two years (e.g., splitting the MMU hypercall into two new hypercalls), it seems obvious that there is massive benefit from insulating the guest from evolution of the hypervisor, by means of the API indirection provided by the in-place VMM.

However, it goes much further. Each hypervisor represents a trade-off between many conflicting requirements, and is inherently optimized for particular application scenarios. It is unlikely that a single hypervisor API will be able to adequately serve all likely application scenarios, particularly given the fact that innovative uses of VM technology are being discovered all the time. Pre-virtualization makes it possible for hypervisors with different APIs, serving different application domains, to co-exist without an increase of cost to the guest OS developer/supplier. This avoids stifling VM research by premature API standardization.

5.2.3 Improved legacy reuse and support for OS innovation

Besides the specific case of virtual machines, pre-virtualization has the potential of helping OS research in general. One of the toughest practical problems facing experimental operating systems is the lack of support for applications, devices and user environments. Our experiences with using multiple hypervisors, and in particular the extreme example of using Linux itself as a hypervisor, show that it is possible to use pre-virtualization to run a guest OS on almost any OS kernel, even one that was never meant to be used as a hypervisor. This provides immediate access to applications and user environments on a new kernel.

Furthermore, our experiences with device virtualization show that it is possible to use pre-existing legacy device drivers to efficiently drive virtual devices, and to map those to real devices: The guest kernel only needs to be provided with one virtual device driver for each class of devices. This dramatically lowers the threshold for making experimental OS kernels useful.

5.3 Lessons Learned

While pre-virtualization significantly lowers the engineering effort for adapting an operating system to a hypervisor, achieving a well-performing system remains a complex task and the quality of the inserted code fragments has a strong impact on overall performance.

We found that a small set of explicit transparent callbacks provided by the guest OS significantly simplify resource tracking within the in-place VMM. For the L4 microkernel we identified two important hooks: (1) thread and process exits, and (2) functions for setting and retrieving an opaque pointer that is associated with the OS's abstraction of threads. These hooks help hypervisors that have their own notion of threads and need to map guest threads to host threads (e.g., L4, Linux, and Windows). Additionally, for such hypervisors, a third hook to copy data between the guest kernel and its applications is important for performance. The hooks do not interfere with execution on raw hardware, nor with migrations between incompatible hypervisors at runtime.

Manual annotation of page table and device accesses provides 100% coverage for all sensitive memory operations, and avoids the additional overhead of the supplementary trapping model for uncovered corner cases. Most operating systems already encapsulate device accesses in special functions and thus only require minimal code changes.

Supporting dynamically loadable kernel modules requires that the in-place VMM rewrites the kernel modules at load time, supported by access to the annotations in the module's ELF binary. Thus dynamic kernel modules require loading support from the guest system, either via a guest user application, or a guest kernel hook. For Linux 2.6, we use a hook, which points the in-place VMM to an in-kernel copy of the ELF file. Linux 2.4 likely requires a user-level solution, which executes system calls directly to the in-place VMM.

Code replacements may violate self-modifying and self-analyzing code. While for pre-virtualization this property

is more apparent, self-modifying code is a general problem of all impure virtualization approaches. In Linux we found a case where code makes assumptions about the instruction length of the system call instruction in order to restart a syscall after signal delivery. Careful encoding of the rewritten instruction preserves the semantics, but requires detailed knowledge of the guest's assumptions.

6. RELATED WORK

Hardware virtualization is a well-known technique, and lately found wide attention in research and industry. Pre-virtualization combines techniques from multiple areas that we contrast here.

6.1 Code Transformation

Code transformation is a common technique for automatically enhancing and optimizing code. For example, MPTrace [5] incorporates data flow analysis of the compiler whereas Etch [23] allows for binary rewriting of x86 libraries. However, most rewriting tools target application binaries and lack support for the sensitive instructions of an OS kernel. SimOS [24] uses a binary translator for system code on the MIPS architecture. A similar code translator for x86 is productized by VMware [29].

Eiraku and Shinjo [6] use a static rewriting model similar to our assembler-level instruction substitution, to permit a guest OS to run on BSD. They either prefix every sensitive x86 instruction with a trapping instruction to permit pure virtualization, or they replace the sensitive instruction with a subroutine call.

6.2 Virtualization Optimizations

Binary rewriting and native execution of OS code are usually imperfect and use trapping on sensitive or tagged operations. Sugerman et al. [27] report more than 77 percent of the overall execution time for NIC processing 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.

The vBlades [18] project attempts to address the engineering problems related to para-virtualization by using a technique they call *optimized para-virtualization*. In order to minimize modifications to the guest OS they use an iterative process, using benchmarks to identify performance bottlenecks which are then para-virtualized. Unprivileged sensitive instructions are dealt with by techniques proposed by Denali [31], manually substituting alternative and trappable instructions for the sensitive instructions. vBlades is able to use a single binary for both native and virtualized execution, by employing run-time checks at each virtualization point. Pre-virtualization avoids run-time checks.

Lowell et al. [17] propose a technique for installing a virtualization layer under a running OS, and to then devirtualize the OS when the services of the hypervisor are no longer necessary. The technique is applicable to the problem of site maintenance, where the hypervisor is used only temporarily, under constrained circumstances, and cooperatively. It achieves its goals primarily via resource partitioning rather than virtualization. We can achieve similar goals by rewriting our annotated instructions.

6.3 Software Structuring

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.

VMware's recent proposal [30] for a virtual machine interface (VMI) introduces a similar indirection for x86's sensitive instructions. Instead of using a function vector, VMI uses a fixed code page that gets replaced with a hypervisor specific implementation. VMI has similarities to pre-virtualization, except that it does not support inline code replacements. While VMI is aimed at manual modifications, we believe that it will be beneficial to pre-virtualization, as it would allow us to mechanize some of our sensitive memory annotations.

6.4 Hardware Support

All major processor vendors announced virtualization extensions to their processor lines: Intel's Virtualization Technology (VT) for x86 and Itanium [12,13], AMD's Pacifica [1], and IBM's LPAR for PowerPC [11]. These proposals fix the unvirtualizable features of the x86 and Itanium architectures and reduce the virtualization overheads.

The hardware extensions are not a substitute for pre-virtualization: They still require expensive trapping, and require modifications to the hypervisor to support each virtualization model. Sophisticated state machines executed within the domain of the guest OS allow for better heuristics without requiring hypervisor intervention. Furthermore, the extensions only address virtualization of processor operations and thus still require hypervisor intervention for all other virtualized hardware.

7. CONCLUSIONS AND FUTURE WORK

We presented pre-virtualization, an approach to virtualizing operating systems based on automatic re-writing of sensitive operations. We have demonstrated that our pre-virtualization technique achieves nearly the same performance as established para-virtualization approaches. However, it does so at significantly reduced engineering effort, and without requiring intimate knowledge of the guest OS. This not only reduces the cost of virtualizing a system, it also maintains confidence in the correctness of the virtualized OS, and makes it easy to keep the virtualized guest in sync with on-going development.

Besides cost, pre-virtualization has a number of benefits which are a direct result of preserving the hardware platform API. These are the ability to run the same binary on all supported hypervisors as well as on bare hardware, making OS development independent of specific hypervisor APIs, and supporting re-use of legacy OS environments, thereby fostering OS innovation. The decoupling of the OS from the hypervisor supports the development of a variety of hypervisors based on domain-specific trade-offs.

We were able to demonstrate the feasibility of pre-virtualization by supporting a variety of very dissimilar hypervisors with the same approach and infrastructure. We believe that pre-virtualization will enable 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 canon-

ical format. The target hypervisor would rewrite the annotated instructions, and then restore the CPU and device state, using its own in-place VMM. For a fully transparent in-place VMM, the state serialization is obviously trivial, but it needs to be tested in cases where resources are not fully virtualized, such as Xen's page tables.

We also see promise for applying techniques similar to pre-virtualization to the task of translating operating system source code between CPU architectures.

8. 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] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. In *USENIX Annual Technical Conference*, Anaheim, CA, USA, Apr. 2005.

[4] 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.

[5] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *In Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, 1990. ACM.

[6] 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.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th ACM Symposium on Operating System Principles*, Oct. 2003.

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

[9] J. G. Hansen and E. Jul. Self-migration of operating systems. In *The 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[10] 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*, Saint-Malo, France, Oct. 1997.

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

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

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

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. of the 20th ACM Symposium on Operating System Principles*, Brighton, UK, Oct. 2005.

[15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Usenix Annual Technical Conference*, pages 1–15, Annaheim, CA, USA, Apr. 2005.

[16] 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*, San Francisco, CA, Dec. 2004.

[17] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *The 11th Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[18] 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.

[19] F. Mehnert, M. Hohmuth, S. Schönberg, and H. Härtig. RTLinux with address spaces. In *Proc. of the 3rd Real-Time Linux Workshop*, Milano, Italy, Nov. 2001.

[20] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. *Tech Trend Notes*, 9, Fall 2000.

[21] 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.

[22] 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.

[23] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proc. of the USENIX Workshop on Windows NT*, Aug. 1997.

[24] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, (4), 1995.

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

[26] 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.

[27] 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.

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

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

[30] VWware, `http://www.vmware.com/vmi`. *Virtual Machine Interface Specification*.

[31] 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*, Boston, MA, Dec. 2002.