Pre-Virtualization with Compiler Afterburning

Joshua LeVasseur, Volkmar Uhlig
University of Karlsruhe, Germany
September 22, 2005

http://l4ka.org
Overview

- Virtualization background
- Basics of pre-virtualization
- Virtualization kit
- Analysis
Virtual Machine (VM)

- A software duplicate of the hardware
  - Indistinguishable from real hardware
    - Except for timing

- Statistically, most instructions execute directly on real CPU
  - Faster than full emulation
Basic VM Structure

Hypervisor + Virtual Machine Monitor

Guest OS
  App
  App
  App

Guest OS

Guest OS

Guest OS

User-level VMM

L4 Microkernel
Uses of VMs

Server consolidation:
- Quality of service
- Strict isolation
- Incompatible software
- Fine grained restart
Legacy Reuse

- VMs enhance legacy code
- Modular encapsulation
  - Guest OS is nearly a black box
  - Well defined interface (but sometimes buggy)
  - Communication with the black box via platform interfaces (network, disk, ...)
- Traditionally, enhancement is via loadable kernel modules

Current OS design is deficient.

The VM is a hack to fix the problems.
Virtualization Definitions

- **Sensitive instruction:**
  - Destroys the illusion of virtualization

- **Innocuous instruction:**
  - Safe to execute within a VM

- **Privileged instruction:**
  - No side effects when executed at user level; raises a fault

- **Virtualizable ISA:** all sensitive instructions are privileged
Pure Virtualization

Advantages:

- Trustworthy emulation
- Hypervisor diversity
  - Raw hardware
  - VMware
  - VirtualPC
- Guest OS diversity
  - 1 x engineering effort
  - N x guest OS
Pure Virtualization

Problems:
- Trapping is costly (cycles, pipe flush)
- x86 isn’t fully virtualizable

VMware’s solution:
- Dynamic code rewriting
- Difficult
Pure Virtualization

- Harsh requirements on hypervisor:
  - 4GB address space
  - Hide differences in CPUs [MMX vs. SSE2]
  - Burdensome legacy emulation

- High perf. devices:
  - Custom device drivers
Para-virtualization

- L4Linux, Denali, Xen
- Replace sensitive instructions with hypercalls
  - Avoids costs of trapping
  - Batch state changes into single hypercall
  - Relax requirements on hypervisor

Problems:
- Engineering effort
- Reduces trustworthiness of guest OS
- Ties guest OS directly to a single hypervisor

```
mov ecx, cr3
pushf
mov ecx, ebx
mov $1, eax
sysenter
push virtual_flags
```
Para-virtualization

Problems:

- Migrations
  - Only possible between ABI compatible hypervisors
  - MMX vs. SSE2
- No support for raw hardware
- High perf. devices:
  - Custom device drivers

```
mov ecx, cr3
pushf
```

```
mov ecx, ebx
mov $1, eax
sysenter
```

```
push virtual_flags
```
Transparent virtualization

VMware’s VMI
- Replace sensitive instructions with hooks
  - Avoids costs of trapping
  - Hypervisor diversity
  - Potential industry standard

Problems:
- Manual engineering effort
- Reduces trustworthiness of guest OS [unless done by kernel architects]
The Interface Difference

- **Pure virtualization**: lowest level interface
- **Para-virtualization**: high-level interface
  - Semantic assists
  - Higher performance
  - Supports unfriendly architectures
  - But hypervisor specific
- The semantic-assists cost modularity
  - The ISA is the modular interface
What is Virtualization About?

- **Customization:**
  - I want my features
  - You want your features

- **Decentralized development:**
  - Enhance a guest OS without the architect’s permission

- **Modularity:**
  - Layering
  - Software reuse

- **Conclusion:** *para-virtualization is inferior*
Why not standardize Para-virtualization?

- What is the perfect interface?
  - Ex: Xen changed their API: 2.0 → 3.0. But wasn’t 2.0 already the ultimate?

- Domain specific trade-offs:
  - Classic: security vs. performance

- Different philosophies:
  - Ex: Run guest kernel at
    - user-level
    - intermediate privilege via x86’s segments

- Conclusion: a standardized para-virtualization interface is not possible
  - The platform ISA is a standard.
Uniting Two Worlds

Pure virtualization
- Trustworthiness
- Layered engineering
- Hypervisor diversity
- Guest OS diversity

Para-virtualization
- Performance
- Scalability
- Supports unfriendly architectures

Pre-virtualization
Pre-virtualization Overview

OS source code

Pre-virtualization

Instrumented OS binary

Raw hardware

VMware

Xen

L4

WinNT

Linux

Runtime migration
Pre-virtualization Basics

- Automated source modifications [based on Eiraku and Shinjo]

- **mov ecx, cr3**
- **pushf**
- **nop**
- **nop**
- **pushf**
- **nop**

- **mov ecx, ebx**
- **mov $1, eax**
- **sysenter**
- **push virtual_flags**

- **Runtime re-writing**
Pre-Virtualization: Eiraku and Shinjo

- **Static**
  - Replace sensitive ops with emulation code
  - Supports a single hypervisor

- **Trapping**
  - Replace sensitive ops with trapping instructions
  - Via trapping, hypervisor diversity
Pre-Virtualization: Dynamic Rewriting

- Runs on raw hardware
- Hypervisor diversity

1. Instruction padding
   - Pad sensitive ops with NOPs
   - Rewrite instruction at runtime
   - Special sequences: sti ; hlt

2. Basic-block padding
   - Pad basic-blocks with NOPs
   - Rewrite entire basic block at runtime
Pre-Virtualization: Dynamic Rewriting

3. Basic-block bypass
   - Write a `jump` instruction at beginning of basic-block, and jump to emulation code

4. Basic-block dynamic link
   - Rewrite all jump sources to the basic-block, so that they jump to the replacement basic-block
Rich annotations possible:
• Register liveness
• Basic block boundaries
• Function boundaries
Phase 1: Afterburner

OS source code → Compiler → OS assembler

Afterburner

Transformed assembler

Assembler
Phase 2: Profiler Feedback

- Virtualization-sensitive memory:
  - Page tables, tss, idt, gdt, etc.
  - Memory mapped devices
  - Important for performance

- Use a profile-feedback loop
  - Instrument and annotate the instructions that access sensitive memory
  - Or compiler data-flow analysis?
Phase 2: Profiler Feedback

OS source code → Compiler → OS assembler

Phase 1 afterburnt binary

Profiler

Locations of sensitive ops.

Afterburner

Afterburnt assembler

Assembler

Afterburnt binary
Profiler Problems

- Profiling is imperfect
  - Incomplete coverage
  - Captures only the common case

- In production environment:
  - Guest kernel could access sensitive memory
  - To detect sensitive accesses, use page faults
    - Easy to implement for devices
    - Suboptimal for guest’s page tables

- Long term solution: data flow analysis
Is Pre-virtualization Good Enough?

- Pre-virtualization:
  - Instruction-level transformations
  - At best, basic-block transformations

- Para-virtualization:
  - High-level interface
  - Semantic assists

We need more from pre-virtualization!
In-Place Virtualization: Approximating Para-virtualization

- The wedge creates a virtual CPU
- Invokes hypercalls only when necessary
- Frequent operations, such as cli/sti, emulated in the wedge
- Loose state consistency
- Use heuristics
- No licensing issues
In-Place Virtualization: Examples

- Interrupts: accumulate in wedge, delivered on guest’s iret, thus cheap masking
- PTE updates: synchronize page tables on TLB flushes
- PTE scanning: read referenced+dirty bits in batches
- fork() + exec(): heuristics
In-Place Virtualization: Problems

- Share the address space
  - May require support from guest kernel

- Large wedges:
  - User-level on Linux/BSD: 1GB
  - User-level on WinNT: 2GB
  - Requires relinking of the guest kernel

- Assumes valid guest stack
Device Access: Multiplexing

VMM
- dp83820 model
- e1000 driver

Guest OS
- dp83820 driver

Guest OS
- dp83820 driver

dp83820 model
- e1000 driver
Device Emulation: Brute Force

- Trapping is expensive
- Trapping is frequent
  - Many small state updates

```
  mov ecx, 0(device)
  mov edx, 4(device)
```
Device Emulation: Custom Drivers

- Traditional solution
- High performance

Problems:
- Specific to hypervisor
- Different config for hypervisor and hardware
- Unable to migrate between different hypervisors

```
mov ecx, ebx
mov edx, ecx
mov $10, eax
sysenter
```
Device Emulation: Pre-virtualization

- Rewrite sensitive ops
  - Replace with calls to the wedge

- Choose well behaved device interface:
  - network: dp83820
  - disk: SATA? SCSI?

- Can migrate between different hypervisors

- Good engineering leverage:
  - 1x engineering
  - Multiple guests
Device Emulation: Batching

- Device ops cause synchronization:
  - Hyper call
  - Potential address space switch
  - Expensive

- Solution:
  - Batching
  - Producer consumer rings

- Batching is difficult:
  - No semantic information
  - OS might send a file, but we see only packets
Device Emulation: Batching Heuristics

- We know:
  - When guest OS performs low-level ops
  - Map low-level ops to high-level ops?
    - High device activity in interrupt handlers
    - When interrupt handler completes, device activity is paused briefly
  - Use `hlt` and `iret`
Transparent Virtualization: Safety Net

- Pre-virtualization is unfinished
  - Use manual annotations for memory operations
- Some hypervisors need high-level hooks for performance:
  - L4, Linux/BSD/WinNT user level
- Performance problems:
  - Signals
  - Process delete
  - Access to user memory
  - Thread local data
Transparent Virtualization: Safety Net

- Bypass platform init
  - BIOS
  - 16-bit code
- Make guest kernel relocatable
- Limited address space
  - Linux/BSD/WinNT user
Extensibility

Wedge call:
- Management utilities
- Introspection
- Filters
- Enhancements

Hyper call
- IPC
- Management utilities
- Enhancements
Loadable Kernel Modules

- Kernel modules are pre-virtualized
  - When are they rewritten?
  - The rewriter needs the annotations …

A. User-level utility
  1. Rewrites the module via wedge calls
  2. Guest kernel loads the module

B. Transparent kernel hook
  - Unmodified user-level tool chain
  - Guest kernel invokes rewrite hook
Loadable Wedge Modules

- Efficient interaction with the wedge:
  - Permit kernel modules to link against the wedge
  - Avoids expensive hyper calls
- Requirement:
  - Wedge symbol resolution
- Uses:
  - Memory ballooning module
  - Virtual device drivers
Evaluation

- Hypervisors:
  - L4Ka::Pistachio microkernel
  - Xen 2.0.2
- Guest OS: Linux 2.6.9
- Comparison:
  - afterburnt Linux on hypervisors
  - para-virtualized Linux: L4Linux, XenoLinux
  - native Linux
  - afterburnt Linux on hardware

(all execute with direct device access)
Implementation

- Simple implementation in C++
  - Contrast para-virtualization: must use language of the guest OS
- Reserved 64MB of address space
- Hooks added to guest OS:
  - page table
  - DMA translation
  - thread-local-storage
  - thread exit
  - user access
  - manual annotations
  - manual hooks
  - L4 optimizations
Benchmark: Netperf

- **System-under-test:**
  - 2.8 GHz Pentium 4
  - gigabit ethernet
  - 256 MB of RAM, 64 MB RAM disk

- **Client system**
  - 1.4 GHz Pentium 4
  - gigabit ethernet
  - native Linux 2.4
  - 256 MB of RAM, 64 MB RAM disk

- Netperf send + receive: one gigabyte
## Netperf Results

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MB/s</td>
<td>CPU</td>
</tr>
<tr>
<td>hardware</td>
<td>108.3</td>
<td>28.8%</td>
</tr>
<tr>
<td>afterburnt hardware</td>
<td>108.5</td>
<td>27.4%</td>
</tr>
<tr>
<td>XenoLinux</td>
<td>108.4</td>
<td>33.8%</td>
</tr>
<tr>
<td>afterburn Xen</td>
<td>107.9</td>
<td>33.5%</td>
</tr>
<tr>
<td>L4Ka::Linux</td>
<td>96.98</td>
<td>34.4%</td>
</tr>
<tr>
<td>afterburnt L4</td>
<td>108.3</td>
<td>30.1%</td>
</tr>
</tbody>
</table>

95% confidence interval is ± 0.21%
## Netperf Instruction Expansion

<table>
<thead>
<tr>
<th>operation</th>
<th>count</th>
<th>per interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>cli</td>
<td>6772992</td>
<td>74</td>
</tr>
<tr>
<td>sti</td>
<td>1572769</td>
<td>17</td>
</tr>
<tr>
<td>pushf</td>
<td>6715828</td>
<td>73</td>
</tr>
<tr>
<td>popf</td>
<td>5290060</td>
<td>58</td>
</tr>
<tr>
<td>hlt</td>
<td>91528</td>
<td>1</td>
</tr>
<tr>
<td>iret</td>
<td>184760</td>
<td>2</td>
</tr>
<tr>
<td>write ds</td>
<td>369520</td>
<td>4</td>
</tr>
<tr>
<td>write es</td>
<td>369520</td>
<td>4</td>
</tr>
<tr>
<td>read ds</td>
<td>184760</td>
<td>2</td>
</tr>
<tr>
<td>read es</td>
<td>184760</td>
<td>2</td>
</tr>
<tr>
<td>read fs</td>
<td>182866</td>
<td>2</td>
</tr>
<tr>
<td>read gs</td>
<td>182866</td>
<td>2</td>
</tr>
<tr>
<td>port out</td>
<td>278737</td>
<td>3</td>
</tr>
</tbody>
</table>
Device Pre-Virtualization Performance

Client Linux

Linux

dp83820

Device driver

Net

dp83820 model

Netperf benchmark

Cycles per byte

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>afterburnt Linux</td>
<td>18.1</td>
</tr>
<tr>
<td>L4Linux</td>
<td>18.2</td>
</tr>
</tbody>
</table>

Intel gigabit

2.8 GHz
Implementation: Afterburner

- Assembler macros
  - Walks the AST
  - But too inflexible (ex: sti ; hlt )

- Added new assembler operator: !
  - Prevents recursive macro expansion

```
.macro pushfl
  9998:   !pushfl ; nop ; nop ; nop ; nop
  9999:
.mpushsection .afterburn
.long 9998b
.long 9999b
.mpopsection
.endm
```
Implementation: Architecture

- **Front-ends:**
  - Architecture specific
  - x86, ia32e, Itanium, PowerPC

- **Back-ends:**
  - Hypervisor specific
  - L4, Xen, Linux-user, WinNT-user

- Code reuse
## Lines of Code

<table>
<thead>
<tr>
<th>type</th>
<th>headers</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>common</td>
<td>686</td>
<td>1380</td>
</tr>
<tr>
<td>device</td>
<td>650</td>
<td>1476</td>
</tr>
<tr>
<td>x86</td>
<td>815</td>
<td>4436</td>
</tr>
<tr>
<td>L4</td>
<td>615</td>
<td>3655</td>
</tr>
<tr>
<td>Xen</td>
<td>677</td>
<td>2670</td>
</tr>
<tr>
<td>Linux</td>
<td>168</td>
<td>3405</td>
</tr>
</tbody>
</table>
Join the Fun

- Lots of pre-virtualization projects
- http://l4ka.org
- Source code available on the web
  - includes a ‘make world’
    - retrieves all packages
    - configures all packages
    - builds everything
    - prepares for QEMU emulator