Pre-virtualization internals

Joshua LeVasseur
3 March 2006

L4Ka.org

Universität Karlsruhe (TH)
Compile time overview

OS source code

Compiler

C code

Assembler code

Afterburner

Hand-written assembler

Assembler

Virtualization aware binary
Afterburner assembler parser

Built using an Antlr grammar
Instruction padding

movl %eax, %fs

Afterburner

.Virtualization aware binary

.L_sensitive_6:
  movl %eax, %fs
  nop
  nop
  nop
  nop
  nop

.popsection  .afterburn
  .balign 4
  .long   .L_sensitive_6
  .long   .L_sensitive_7
  .popsection
Address space

Xen x86

0

Guest application

Guest kernel

IPVMM

Hypervisor

4 GB
Address space

Xen x86

- Hypervisor
- IPVMM
- Guest kernel
- Guest application

4 GB

0

Ring 1

Ring 3

L4, Linux-on-Linux (generic)

- Hypervisor
- IPVMM
- Guest kernel
- Guest application

0
Loading the guest kernel

1. Virtualization aware binary

2. Guest kernel

3. Guest kernel

Hypervisor
In-Place VMM

mov eax, cr3  sti  cli  popf  hlt

frontend

Virtual CPU

backend

IPVMM

Map, unmap, TLB, timer, threads

Page faults, traps, system calls, interrupts

Hypervisor
Constraints

Code expansion:

• Timing?
• Interrupts?
• Simple state machine?
• Efficiency?
Constraints

- Guest kernel is a sequential process
  - Important: forward progress
  - Unimportant: rate of forward progress
- We provide a virtual CPU
  - Illusion of continuous time

Code expansion:
- Timing?
- Interrupts?
- Simple state machine?
- Efficiency?
Constraints

Guest kernel is a sequential process
- Important: forward progress
- Unimportant: rate of forward progress

We provide a virtual CPU
- Illusion of continuous time

Delay delivery
- Synchronous (sti, popf)
  - Optimize common case
- Asynchronous
  - Avoid IPVMM reentrance

Code expansion:
- Timing?
- Interrupts?
  - Simple state machine?
  - Efficiency?
Constraints

- Guest kernel is a sequential process
  - Important: forward progress
  - Unimportant: rate of forward progress
- We provide a virtual CPU
  - Illusion of continuous time

Delay delivery
  - Synchronous (sti, popf)
    - Optimize common case
  - Asynchronous
    - Avoid IPVMM reentrance

Code expansion:
- Timing?
- Interrupts?
  - Simple state machine?
  - Efficiency?

Thread model
- VM thread
- Interrupt thread
Threads

mov eax, cr3
sti
cli
popf
hlt

frontend
VM thread
backend

Page faults, traps, system calls

Hypervisor
Threads

mov eax, cr3
sti
cli
popf
hlt

Hypervisor

Interrupts

Page faults, traps, system calls

Hypervisor

frontend

backend

IRQ thread

VM thread
Frontend entry

Assembler trampoline

- Establish C calling conventions. **Future**: eliminate by Afterburning the IPVMM.
- **C++ frontend, thread**

mov eax, cr3

- **sti**
- **cli**
- **popf**
- **hlt**

Hypervisor

- **IRQ thread**
- **VM thread**
- **Interrupts**
- **Page faults, traps, system calls**

Future: eliminate by Afterburning the IPVMM.
Boundary transitions

Guest kernel

```
invlpg vaddr
pushl %eax
lea vaddr, %eax
call burn_invlp
popl %eax
```
Boundary transitions (generic)

Guest kernel

invlpg vaddr

pushl %eax
lea vaddr, %eax
call burn_invlpv
popl %eax

IPVMM trampoline

burn_invlpv:
// Preserve C clobbers.
pushl %eax
pushl %ecx
pushl %edx

// Build burn_clobbers_frame_t parameter.
pushl %esp
subl $8, 0(%esp)
call afterburn_cpu_invlpv_ext

popl %edx ; popl %ecx ; popl %eax
ret

invlpg vaddr

// Preserve C clobbers.
pushl %eax
pushl %ecx
pushl %edx

// Build burn_clobbers_frame_t parameter.
pushl %esp
subl $8, 0(%esp)
call afterburn_cpu_invlpv_ext

popl %edx ; popl %ecx ; popl %eax
ret
Boundary transitions (generic)

**Guest kernel**
- `invlpg vaddr`
  - `pushl %eax`
  - `lea vaddr, %eax`
  - `call burn_invlpv`
  - `popl %eax`

**IPVMM trampoline**
- `burn_invlpv`:
  - // Preserve C clobbers.
  - `pushl %eax`
  - `pushl %ecx`
  - `pushl %edx`
  - // Build `burn_clobbers_frame_t` parameter.
  - `pushl %esp`
  - `subl $8, 0(%esp)`
  - `call afterburn_cpu_invlpv_ext`
  - `popl %edx ; popl %ecx ; popl %eax`
  - `ret`

**IPVMM C++ frontend**
- `struct burn_clobbers_frame_t`{
  - `word_t burn_ret_address`;
  - `word_t frame_pointer`;
  - `word_t edx`;
  - `word_t ecx`;
  - `word_t eax`;
  - `word_t guest_ret_address`;
  - `word_t params[0];`;
}
- `extern "C" void afterburn_cpu_invlpv_ext( burn_clobbers_frame_t *frame )`
  - `backend_flush_vaddr( frame->eax );`
Atomic instructions

Interrupts
- During IPVMM code?
- During a hypercall?
Thread reschedule

mov eax, cr3

VM thread
C++ code

ret

VM thread
C++ code:
IRQ dispatch
Race conditions

VM thread

```c
if ( IRQ pending )
    reschedule();
else
    iret();
```
Race conditions

Time

VM thread

`iret_restart`:
```
if ( IRQ pending )
    reschedule();
else
    iret();
```

IRQ thread

```
if ( VM thread in iret )
    rollback( iret_restart );
...
```
Modules

Interrupt delivery to guest

Instruction set emulation

Device memory accesses

Port accesses

Memory paging

Device switch

Network

Disk

PIT

Serial port

Serial port

Interrupt dispatch

PCI

RTC

Hypervisor

Synchronous events

Asynchronous events
Adding a device model

1. Define a device class
2. Define its interfaces:
   - Port accesses
   - Memory-mapped registers
3. Define its PCI registers:
   - Static structure created at compile-time
4. Raise interrupts with the intlogic_t class
5. In some cases, define a virtual IRQ handler
Productivity-focused coding

Minimize errors:

- **Simple code**
  - Avoid purely quantitative increase in work
- Use assertions
- Avoid reentrancy
- **C++**
  - Modularity (quickly understood code)
  - Compile-time features only
  - Fast (world's fastest kernel, L4Ka::Pistachio, is written in C++)
- No dynamic memory allocation
  - Construct data structures at compile time
    - Ex: big switch() statement for port access, PCI device access
- **Heavily typed**
Productivity-focused coding

Code maintenance:
  • Minimize C Preprocessor
    - We want structured code
  • Use CML2 configuration system
    - Feature management

Code reuse:
  • Abstract frontends and backends
  • Reuse frontend across multiple backends
Legacy devices

CPU → XT-PIC primary

Keyboard → XT-PIC primary

UART → XT-PIC secondary

RTC

PIT

PC speaker

Channel 0

Channel 2
Legacy devices: IDT vector

- CPU
- XT-PIC primary
- XT-PIC secondary
- Keyboard
- UART
- PIT
- PC speaker
- RTC

ICW2

Interrupts:
- 0: Channel 0
- 1: INTR
- 2: INTR
- 3: INTR
- 4: INTR
- 8: INTR

IDT vector
Legacy devices: Ports

- XT-PIC primary
  - CPU
  - 0: Channel 0
  - 1: Keyboard
  - 2: PIT
  - 3, 4: UART
  - 8
- XT-PIC secondary
  - 2
  - 3, 4
  - 8
- RTC
  - 70h, 71h
  - 00h, A1h
- PIT
  - 60h-64h
  - 61h
- PC speaker
  - 61h
- PC speaker
  - 61h
- UART
  - 2
  - 3, 4
- Keyboard
  - 1
  - 2
  - 3, 4
- XT-PIC ICW2
  - 3F8h, 2F8h, 3E8h, 2E8h
  - IRQ
  - IDT vector
  - 20h, 21h
  - 20h, 21h
  - 60h, 64h
  - 60h-64h
  - 61h
  - 40h-43h
  - 3F8h, 2F8h, 3E8h, 2E8h
Further information

http://l4ka.org/projects/virtualization/

Publications:

• Full paper: *Pre-Virtualization: Slashing the Cost of Virtualization*

• Quick read: *Pre-Virtualization: Uniting Two Worlds*
• Quick read: White paper