Hardware-Supported Virtualization on ARM

Prashant Varanasi and Gernot Heiser
NICTA and University of New South Wales, Sydney
Virtualization for ARM Processors

• Virtualization is becoming prevalent in embedded systems
  – Open Kernel Labs (and others) have been selling hypervisors for years

• ARM announced hardware support in September 2010
  – Architecture specification released late 2010
  – Simulators released to partners in late 2010
  – Silicon samples expected late 2011
  – Products expected to ship in late 2012

• We built first complete hypervisor prototype for this architecture
  – … able to run unmodified Linux binaries
Pure vs Para-Virtualization

- Hypervisor must *emulate* sensitive instructions executed by OS
  
  ![Guest Code Example](image)

  Patch (“para-virtualize”) guest code to force traps

- On ARM (as on x86) not all sensitive instructions trap
  
  \[ \text{Guest Exception Hypervisor} \]

  - Para-virtualization
    - Significant engineering cost

- Hardware extensions support pure virtualization
  
  - Ability to run unmodified binaries
ARM Virtualization Extensions (1)

Hyp mode

- New privilege level
  - Strictly higher than kernel
  - Virtualizes or traps all sensitive instructions
  - Only available in ARM TrustZone “non-secure” mode

- Note: different from x86
  - VT-x “root” mode is orthogonal to x86 protection rings

“Non-Secure” world

User mode
Kernel modes
Hyp mode

“Secure” world

User mode
Kernel modes

Monitor mode
ARM Virtualization Extensions (2)

Configurable Traps

Native syscall

Virtual syscall

Can configure traps to go directly to guest OS

Trap to guest

© NICTA 2011

APSys’11
ARM Virtualization Extensions (3)

Emulation

1) Load faulting instruction
   • Compulsory L1-D miss!

2) Decode instruction
   • Complex logic

3) Emulate instruction
   • Usually straightforward

L1 I-Cache

IR

R2

mv CPU_ASID, r1

ld r1, (r0, ASID)

mv CPU_ASID, r1

ld sp, (r1, kern_stk)

L1 D-Cache

ld r1, (r0, ASID)

mv CPU_ASID, r1

ld sp, (r1, kern_stk)

L2 Cache

ld r1, (r0, ASID)

mv CPU_ASID, r1

ld sp, (r1, kern_stk)
Emulation Support

- HW decodes instruction
  - No L1 miss
  - No software decode
- SW emulates instruction
  - Usually straightforward

```
IR
mv CPU_ASID, r1

L1 I-Cache
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)

L1 D-Cache

R2
mv

R3
r1

L2 Cache
ld r1, (r0, ASID)
mv CPU_ASID, r1
ld sp, (r1, kern_stk)
```
Page table virtualization through shadow page tables

User
1d r0, adr

Virt PT ptr (Software)
Guest OS

Guest virtual address

(Virtual) guest page table

Shadow guest page table
Hypervisor mirrors guest's page table updates

Guest physical address

Hypervisor's guest memory map

PT ptr (Hardware)
Hypervisor

Physical address

Memory

data
Page table virtualization through 2-stage translation

- Hardware PT walker traverses both PTs
- Loads combined (guest-virtual to physical) mapping into TLB

User
1d r0, adr

Guest virtual address

1st PT ptr
(Hardware)

Guest OS

(Virtual) guest page table

Hypervisor's guest memory map

2nd PT ptr
(Hardware)

Hypervisor

Physical address

Memory

data
ARM Virtualization Extensions (5)

Virtual Interrupts

- ARM has 2-part IRQ controller
  - Global “distributor”
  - Per-CPU “interface”
- New H/W “virt. CPU interface”
  - Mapped to guest
  - Used by HV to forward IRQ
  - Used by guest to acknowledge
- Reduces hypervisor entries for interrupt virtualization
### Hypervisor Size

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>ISA</th>
<th>Type</th>
<th>Kernel</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKL4</td>
<td>ARMv7</td>
<td>para-virtualization</td>
<td>9.8 kLOC</td>
<td>0</td>
</tr>
<tr>
<td><em>This work</em></td>
<td>ARMv7</td>
<td>pure virtualization</td>
<td>6 kLOC</td>
<td>0</td>
</tr>
<tr>
<td>Nova</td>
<td>x86</td>
<td>pure virtualization</td>
<td>9 kLOC</td>
<td>27 kLOC</td>
</tr>
</tbody>
</table>

- Size (& complexity) reduced about 40% wrt to para-virtualization
- Much smaller than x86 pure-virtualization hypervisor
  - Mostly due to greatly reduced need for instruction emulation
## Overheads (Estimated)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pure virtualization</th>
<th>Para-virtualiz.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruct</td>
<td>Cycles (est)</td>
</tr>
<tr>
<td>Guest system call</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hypervisor entry + exit</td>
<td>120</td>
<td>650</td>
</tr>
<tr>
<td>IRQ entry + exit</td>
<td>270</td>
<td>900</td>
</tr>
<tr>
<td>Page fault</td>
<td>356</td>
<td>1500</td>
</tr>
<tr>
<td>Device emul.</td>
<td>249</td>
<td>1040</td>
</tr>
<tr>
<td>Device emul. (accel.)</td>
<td>176</td>
<td>740</td>
</tr>
<tr>
<td>World switch</td>
<td>2824</td>
<td>7555</td>
</tr>
</tbody>
</table>

- No overhead on regular (virtual) syscall – unlike para-virtualization
- Invoking hypervisor 500–1200 cycles (0.6–1.5 µs) more than para
- World switch in ~10 µs compared to 0.25 µs for para
  ⇒ Trade-offs differ
Experience

• Well-designed virtualization architecture ⇒ simple hypervisor
  – Emulation simplified by hardware support
• Especially IRQ handling simpler and much faster than on x86
• Hypervisor-entry/exit costs much higher than with para-virtualization
  – On-going need for para-virtualization, especially for device drivers
  – Para-virtualized drivers can co-exist with purely-virtualized guest
• World-switch is expensive
  – Due to large amount of extra privileged state
  – Performance challenge for shared drivers
  – However, this is highly sensitive to implementation of hardware
• More details in paper!

mailto:gernot@nicta.com.au
Google: “ertos”