What is nested x86 virtualization?

- Running multiple unmodified hypervisors
- With their associated unmodified VM’s
- Simultaneously
- On the x86 architecture
- Which does not support nesting in hardware...
- …but does support a single level of virtualization
Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- **Security**: attack via or defend against hypervisor-level rootkits such as Blue Pill
- To be able to run other hypervisors in clouds
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
First models for nested virtualization [PopekGoldberg74, BelpaireHsu75, LauerWyeth73]

First implementation in the IBM z/VM; relies on architectural support for nested virtualization (sie)

Microkernels meet recursive VMs [FordHibler96]: assumes we can modify software at all levels

x86 software based approaches (slow!) [Berghmans10]

KVM [KivityKamay07] with AMD SVM [RoedelGraf09]

Early Xen prototype [He09]

Blue Pill rootkit hiding from other hypervisors [Rutkowska06]
What is the Turtles project?

- Efficient nested virtualization for Intel x86 based on KVM
- Runs multiple guest hypervisors and VMs: KVM, VMware, Linux, Windows, ...
- Code publicly available
What is the Turtles project? (cont’)

- Nested VMX virtualization for nested CPU virtualization
- Multi-dimensional paging for nested MMU virtualization
- Multi-level device assignment for nested I/O virtualization
- Micro-optimizations to make it go fast
Theory of nested CPU virtualization

- Trap and emulate [PopekGoldberg74] ⇒ it’s all about the traps
- Single-level (x86) vs. multi-level (e.g., z/VM)
- Single level ⇒ one hypervisor, many guests
- Turtles approach: L₀ multiplexes the hardware between L₁ and L₂, running both as guests of L₀—without either being aware of it
- (Scheme generalized for n levels; Our focus is n=2)
Nested VMX virtualization: flow

- L₀ runs L₁ with VMCS₀→₁
- L₁ prepares VMCS₁→₂ and executes vmlaunch
- vmlaunch traps to L₀
- L₀ merges VMCS’s: VMCS₀→₁ merged with VMCS₁→₂ is VMCS₀→₂
- L₀ launches L₂
- L₂ causes a trap
- L₀ handles trap itself or forwards it to L₁
- ...
- eventually, L₀ resumes L₂
- repeat
Exit multiplication makes angry turtle angry

- To handle a single L₂ exit, L₁ does many things: read and write the VMCS, disable interrupts, …
- Those operations can trap, leading to **exit multiplication**
- **Exit multiplication**: a single L₂ exit can cause 40-50 L₁ exits!
- Optimize: make a single exit fast and reduce frequency of exits
Introduction to x86 MMU virtualization

- x86 does page table walks in hardware
- MMU has one currently active hardware page table
- Bare metal $\Rightarrow$ only needs one logical translation, (virtual $\rightarrow$ physical)
- Virtualization $\Rightarrow$ needs two logical translations
  1. Guest page table: (guest virt $\rightarrow$ guest phys)
  2. Host page table: (guest phys $\rightarrow$ host phys)
- ... but MMU only knows to walk a single table!
Two logical translations compressed onto the shadow page table [DevineBugnion02]

- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table

Problem: traps are expensive
Two-dimensional paging: guest owns GPT, hypervisor owns EPT [BhargavaSerebrin08]
Unmodified guest OS updates GPT
Hypervisor updates EPT table controlling (guest phys → host phys) translations
MMU walks both tables
Nested MMU virt. via multi-dimensional paging

- **Three logical translations**: $L_2$ virt $\rightarrow$ phys, $L_2 \rightarrow L_1$, $L_1 \rightarrow L_0$
- **Only two tables in hardware with EPT**: virt $\rightarrow$ phys and guest physical $\rightarrow$ host physical
- **$L_0$ compresses** three logical translations onto two hardware tables

---

Ben-Yehuda et al. (IBM Research)  The Turtles Project: Nested Virtualization  dc9723 May, 2011  13 / 1
Baseline: shadow-on-shadow

- Assume no EPT table; all hypervisors use shadow paging
- Useful for old machines and as a baseline
- Maintaining shadow page tables is expensive
- **Compress:** three logical translations $\Rightarrow$ **one** table in hardware
Instead of one hardware table we have two

- **Compress**: three logical translations $\Rightarrow$ two in hardware
- Simple approach: $L_0$ uses EPT, $L_1$ uses shadow paging for $L_2$
- Every $L_2$ page fault leads to multiple $L_1$ exits
Best: multi-dimensional paging

- EPT table rarely changes; guest page table changes a lot
- Again, **compress three** logical translations ⇒ **two** in hardware
- L₀ **emulates** EPT for L₁
- L₀ uses EPT₀→₁ and EPT₁→₂ to construct EPT₀→₂
- End result: a lot less exits!
Introduction to I/O virtualization

- From the hypervisor’s perspective, what is I/O?
  - (1) PIO
  - (2) MMIO
  - (3) DMA
  - (4) interrupts

Device emulation [Sugerman01]

Para-virtualized drivers [Barham03, Russell08]

Direct device assignment [Levasseur04, Yassour08]

- Direct assignment best performing option
- Direct assignment requires IOMMU for safe DMA bypass
Multi-level device assignment

- With nested \textbf{3x3} options for I/O virtualization (L_2 \leftrightarrow L_1 \leftrightarrow L_0)
- \textbf{Multi-level device assignment} means giving an L_2 guest direct access to L_0’s devices, safely \textbf{bypassing both L_0 and L_1}

How? L_0 emulates an IOMMU for L_1 \cite{Amit11}

- L_0 \textbf{compresses} multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA’s into L_2 memory space directly
Micro-optimizations

- Goal: reduce world switch overheads
- Reduce cost of single exit by focus on VMCS merges:
  - Keep VMCS fields in processor encoding
  - Partial updates instead of whole-sale copying
  - Copy multiple fields at once
  - Some optimizations not safe according to spec
- Reduce frequency of exits—focus on vmread and vmwrite
  - Avoid the exit multiplier effect
  - Loads/stores vs. architected trapping instructions
  - Binary patching?
This is the tenth iteration of the nested VMX patch set. Improvements in this version over the previous one include:

- Fix the code which did not fully maintain a list of all VMCSs loaded on each CPU. (Avi, this was the big thing that bothered you in the previous version).

- Add nested-entry-time (L1→L2) verification of control fields of vmcs12 - procbased, pinbased, entry, exit and secondary controls - compared to the capability MSRs which we advertise to L1.

[many other changes trimmed]

This new set of patches applies to the current KVM trunk (I checked with 6f1bd0daae731ff07f4755b4f56730a6e4a3c1cb).

If you wish, you can also check out an already-patched version of KVM from branch "nvmx10" of the repository:

git://github.com/nyh/kvm-nested-vmx.git
Windows XP on KVM L₁ on KVM L₀
Linux on VMware $L_1$ on KVM $L_0$

Ben-Yehuda et al. (IBM Research)

The Turtles Project: Nested Virtualization

dc9723 May, 2011 22 / 1
Experimental Setup

- Running Linux, Windows, KVM, VMware, SMP, ...
- Macro workloads:
  - kernbench
  - SPECjbb
  - netperf
- Multi-dimensional paging?
- Multi-level device assignment?
- KVM as L₁ vs. VMware as L₁?

See paper for full experimental details and more benchmarks and analysis.
Macro: \texttt{SPECjbb and kernbench}

<table>
<thead>
<tr>
<th></th>
<th>kernbench</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Host</td>
<td>Guest</td>
<td>Nested</td>
<td>Nested\textsubscript{DRW}</td>
<td></td>
</tr>
<tr>
<td>Run time</td>
<td>324.3</td>
<td>355</td>
<td>406.3</td>
<td>391.5</td>
<td></td>
</tr>
<tr>
<td>% overhead vs. host</td>
<td>-</td>
<td>9.5</td>
<td>25.3</td>
<td>20.7</td>
<td></td>
</tr>
<tr>
<td>% overhead vs. guest</td>
<td>-</td>
<td>-</td>
<td>14.5</td>
<td>10.3</td>
<td></td>
</tr>
</tbody>
</table>

|                   | SPECjbb         |               |               |               |               |
|                   | Host            | Guest         | Nested        | Nested\textsubscript{DRW} |
| Score             | 90493           | 83599         | 77065         | 78347         |
| % degradation vs. host | -               | 7.6           | 14.8          | 13.4          |
| % degradation vs. guest | -               | -             | 7.8           | 6.3           |

**Table: kernbench and SPECjbb results**

- Exit multiplication effect not as bad as we feared
- Direct \texttt{vmread} and \texttt{vmwrite} (DRW) give an immediate boost
- Take-away: each level of virtualization adds approximately the same overhead!
Impact of multi-dimensional paging depends on rate of page faults

Shadow-on-EPT: every $L_2$ page fault causes $L_1$ multiple exits

Multi-dimensional paging: only EPT violations cause $L_1$ exits

EPT table rarely changes: $\#(EPT\ violation) << \#(page\ faults)$

Multi-dimensional paging huge win for page-fault intensive kernbench
Benchmark: netperf TCP_STREAM (transmit)

- Multi-level device assignment best performing option
- But: native at 20%, multi-level device assignment at 60% (x3!)
- Interrupts considered harmful, cause exit multiplication
What if we could deliver device interrupts directly to $L_2$?

- Only 7% difference between native and nested guest!
CPUID running in a tight loop is not a real-world workload!
Went from 30x worse to “only” 6x worse
A nested exit is still expensive—minimize both single exit cost and frequency of exits
Conclusions

- Efficient nested x86 virtualization is challenging but feasible
- A whole new ballpark opening up many exciting applications—security, cloud, architecture, ...
- Current overhead of 6-14%
  - Negligible for some workloads, not yet for others
  - Work in progress—expect at most 5% eventually
- Code is available
- Why Turtles?
  It’s turtles all the way down
Questions?