"You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

Presenter: Naser AlDuaij
The Turtles Project: Nested Virtualization

- Running multiple unmodified hypervisors/VMs
- Single level (non-nested) hypervisor is modified
- Battle against performance degradation
The Turtles Project: Motivation

- Popular OS have hypervisors built in
- Platforms with hypervisors embedded in f/w
- Live migration of hypervisors with their Vms
- Increased security
- Testing/Debugging/Benchmarking hypervisors
The Turtles Project: Making it work

- CPU virtualization: Forward traps to upper levels
- Memory virtualization: Multi-dimensional paging
- I/O virtualization: Multi-level device assignment
The Turtles Project: Architecture

- IBM Sys z: Multi-level arch. support for nested virtualization (each hypervisor deals with guest trap)
- VMX/SVM: Single-level arch. support for nested virtualization (all traps to level 0)
- Only level 0 can use VMX instructions (emulation)
- VMX transitions: VMEtry/VMExit for guest/root
The Turtles Project: Architecture

Figure 1: Nested traps with single-level architectural support for virtualization
The Turtles Project: Multiplexing

Figure 2: Multiplexing multiple levels of virtualization on a single hardware-provided level of support
The Turtles Project: CPU Nested Virtualization

- Only hypervisor (L0) runs in root mode
- VMCS/VMCB are control structures/blocks in memory
- VMCS divided into three groups:
  - Guest state (Virtual CPU registers)
  - Host state (Real CPU registers)
  - Control Data (To inject events into VMs)
The Turtles Project: CPU Nested Virtualization

- VMCS 1-->2 is never loaded into the processor but is used by L0 to emulate a VMX for L1
- VMCS 0-->2 constructed using VMCS 1-->2
The Turtles Project: CPU Nested Virtualization

- L1 uses VMX instructions to load L2
- Causes VMExits
- L0 traps and emulates VMX instructions by L1
- vmlaunch/vmresume to launch a [new] VM
- VMExit from L1 to L0, VMEntry from L0 to L2
The Turtles Project: MMU virtualization

- Implementations for page table translation
  - Shadow-on-shadow: Slowest, useful if system does not support 2D page tables
  - Shadow on EPT: Straightforward. Support only for single-level EPT. Still considerable overhead
  - EPT on EPT: Multi-dimensional page tables pure HW
The Turtles Project: MMU Virtualization

Figure 4: MMU alternatives for nested virtualization
The Turtles Project: I/O Virtualization

- Emulation: Emulates a known device- Guest uses an unmodified driver to interact with it
- Paravirtualization: Driver installed in guest
- Device Assignment: Direct access to the hardware
The Turtles Project: I/O Virtualization

- DMA is complicated with guest physical addresses
- IOMMU: HW component with a table of addresses from the hypervisor
- Allow multi-level virtualization by compressing multiple tables into one
- Better solution: Emulate IOMMU
The Turtles Project: Micro Optimizations

- Optimizations to VMCS merging code (not sig.)
  - Only perform a copy of VMCS if relevant values were modified
  - Copy multiple VMCS fields at once
- Exit-handling code slower due to additional exits
  - Reduce number of unnecessary traps
  - Direct Read/Write optimization (DRW)
The Turtles Project: Evaluation

- KVM
- kernbench: CPU, memory, and I/O intensive
- SPECjbb: Mainly CPU intensive
- Testing environments
  - Bare-metal (host), single-level, nested, nested DRW
The Turtles Project: Evaluation

### Table 2: kernbench and SPECjbb results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernbench</th>
<th>SPECjbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>324.3</td>
<td>90493</td>
</tr>
<tr>
<td>STD dev</td>
<td>1.5</td>
<td>1104</td>
</tr>
<tr>
<td>% overhead vs. host</td>
<td>9.5</td>
<td>7.6</td>
</tr>
<tr>
<td>% overhead vs. guest</td>
<td>25.3</td>
<td>14.8</td>
</tr>
<tr>
<td>%CPU</td>
<td>93</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kernbench</th>
<th>SPECjbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>% overhead vs. single-level guest</td>
<td>14.98</td>
<td>8.85</td>
</tr>
</tbody>
</table>

### Table 3: VMware Server as a guest hypervisor

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernbench</th>
<th>SPECjbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>355</td>
<td>83599</td>
</tr>
<tr>
<td>STD dev</td>
<td>10</td>
<td>1230</td>
</tr>
<tr>
<td>% degradation vs. host</td>
<td>6.7</td>
<td>14.8</td>
</tr>
<tr>
<td>% degradation vs. guest</td>
<td>391.5</td>
<td>7.8</td>
</tr>
<tr>
<td>%CPU</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 5: CPU cycle distribution
The Turtles Project: Evaluation

Figure 6: Cycle costs of handling different types of exits
The Turtles Project: I/O Evaluation

Figure 7: Performance of netperf in various setups
The Turtles Project: I/O Evaluation

Figure 8: Performance of `netperf` with interrupt-less network driver
The Turtles Project: MMU Evaluation – Page faults cause the exits

Figure 9: Impact of multi-dimensional paging
The Turtles Project: Evaluation (Running exits only)

Figure 10: CPU cycle distribution for cpuid
The Turtles Project: Cost of a nested exit

1. L₂ executes a cpuid instruction
2. CPU traps and switches to root mode L₀
3. L₀ switches state from running L₂ to running L₁
4. CPU switches to guest mode L₁
5. L₁ modifies VMCS₁→₂
   repeat n times:
   (a) L₁ accesses VMCS₁→₂
   (b) CPU traps and switches to root mode L₀
   (c) L₀ emulates VMCS₁→₂ access and resumes L₁
   (d) CPU switches to guest mode L₁
6. L₁ emulates cpuid for L₂
7. L₁ executes a resume of L₂
8. CPU traps and switches to root mode L₀
9. L₀ switches state from running L₁ to running L₂
10. CPU switches to guest mode L₂
The Turtles Project: Other performance issues

- Virtualization API improvements
- Traps occurring on a core are handled by it
- Cache pollution when switching from guest-hypervisor
The Turtles Project

Questions?