The Turtles Project: Nested Virtualization

Muli Ben-Yehuda†  Michael D. Day‡  Zvi Dubitzky†  Michael Factor†
Nadav Har’El†  Abel Gordon†  Anthony Liguori‡  Orit Wasserman†
Ben-Ami Yassour†

†IBM Research – Haifa
‡IBM Linux Technology Center
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)
- To be able to run other hypervisors in clouds
- Security (e.g., hypervisor-level rootkits)
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- To be able to run other hypervisors in clouds
- Security (e.g., hypervisor-level rootkits)
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- To be able to run other hypervisors in clouds
- Security (e.g., hypervisor-level rootkits)
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- To be able to run other hypervisors in clouds
- Security (e.g., hypervisor-level rootkits)
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
Why?

- Operating systems are already hypervisors (Windows 7 with XP mode, Linux/KVM)
- To be able to run other hypervisors in clouds
- Security (e.g., hypervisor-level rootkits)
- Co-design of x86 hardware and system software
- Testing, demonstrating, debugging, live migration of hypervisors
Related work

- 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]
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]
Related work

- 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]
Related work

- First models for nested virtualization \cite{PopekGoldberg74, BelpaireHsu75, LauerWyeth73}
- First implementation in the IBM z/VM; relies on architectural support for nested virtualization \cite{sie}
- Microkernels meet recursive VMs \cite{FordHibler96}: assumes we can modify software at all levels
- x86 software based approaches (slow!) \cite{Berghmans10}
  - KVM \cite{KivityKamay07} with AMD SVM \cite{RoedelGraf09}
  - Early Xen prototype \cite{He09}
  - Blue Pill rootkit hiding from other hypervisors \cite{Rutkowska06}
Related work

- 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]
Related work

- 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]
Related work

- 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
- Multiple guest hypervisors and VMs: VMware, 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 (see paper)
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 (see paper)
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 (see paper)
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 (see paper)
Theory of nested CPU virtualization

- **Single-level** architectural support (x86) vs. **multi-level** architectural support (e.g., z/VM)
- Single level $\Rightarrow$ one hypervisor, many guests
- Turtles approach: $L_0$ multiplexes the hardware between $L_1$ and $L_2$, running both as guests of $L_0$—without either being aware of it
- (Scheme generalized for $n$ levels; Our focus is $n=2$)
Theory of nested CPU virtualization

- **Single-level** architectural support (x86) vs. **multi-level** architectural support (e.g., z/VM)
- **Single level** ⇒ one hypervisor, many guests
  - Turtles approach: \(L_0\) multiplexes the hardware between \(L_1\) and \(L_2\), running both as guests of \(L_0\)—without either being aware of it
  - (Scheme generalized for \(n\) levels; Our focus is \(n=2\))
Theory of nested CPU virtualization

- **Single-level** architectural support (x86) vs. **multi-level** architectural support (e.g., z/VM)
- **Single level** ⇒ one hypervisor, many guests
- Turtles approach: $L_0$ **multiplexes** the hardware between $L_1$ and $L_2$, running both as guests of $L_0$—without either being aware of it
- (Scheme generalized for $n$ levels; Our focus is $n=2$)

![Diagram of nested virtualization]

---

Ben-Yehuda et al. (IBM Research)  
The Turtles Project: Nested Virtualization  
OSDI '10
Theory of nested CPU virtualization

- **Single-level** architectural support (x86) vs. **multi-level** architectural support (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

- $L_0$ runs $L_1$ with $VMCS_{0\rightarrow1}$
- $L_1$ prepares $VMCS_{1\rightarrow2}$ and executes `vmlaunch`
- `vmlaunch` traps to $L_0$
- $L_0$ merges $VMCS$’s: $VMCS_{0\rightarrow1}$ merged with $VMCS_{1\rightarrow2}$ is $VMCS_{0\rightarrow2}$
- $L_0$ launches $L_2$
- $L_2$ causes a trap
- $L_0$ handles trap itself or forwards it to $L_1$
- ...
- eventually, $L_0$ resumes $L_2$
- repeat
Nested VMX virtualization: flow

- \( L_0 \) runs \( L_1 \) with \( \text{VMCS}_{0\rightarrow 1} \)
- \( L_1 \) prepares \( \text{VMCS}_{1\rightarrow 2} \) and executes \text{vmlaunch}
- \text{vmlaunch} traps to \( L_0 \)
- \( L_0 \) merges VMCS’s:
  - \( \text{VMCS}_{0\rightarrow 1} \) merged with \( \text{VMCS}_{1\rightarrow 2} \)
  is \( \text{VMCS}_{0\rightarrow 2} \)
- \( L_0 \) launches \( L_2 \)
- \( L_2 \) causes a trap
- \( L_0 \) handles trap itself or forwards it to \( L_1 \)
- \( \ldots \)
- eventually, \( L_0 \) resumes \( L_2 \)
- repeat
Nested VMX virtualization: flow

- \( L_0 \) runs \( L_1 \) with \( \text{VMCS}_{0 \rightarrow 1} \)
- \( L_1 \) prepares \( \text{VMCS}_{1 \rightarrow 2} \) and executes \( \text{vmlaunch} \)
- \( \text{vmlaunch} \) traps to \( L_0 \)
- \( L_0 \) merges VMCS’s: \( \text{VMCS}_{0 \rightarrow 1} \) merged with \( \text{VMCS}_{1 \rightarrow 2} \) is \( \text{VMCS}_{0 \rightarrow 2} \)
- \( L_0 \) launches \( L_2 \)
- \( L_2 \) causes a trap
- \( L_0 \) handles trap itself or forwards it to \( L_1 \)
- ... 
- eventually, \( L_0 \) resumes \( L_2 \)
- repeat
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
Nested VMX virtualization: flow

- $L_0$ runs $L_1$ with VMCS$_{0\rightarrow 1}$
- $L_1$ prepares VMCS$_{1\rightarrow 2}$ and executes `vmlaunch`
- `vmlaunch` traps to $L_0$
- $L_0$ merges VMCS’s:
  - VMCS$_{0\rightarrow 1}$ merged with VMCS$_{1\rightarrow 2}$ is VMCS$_{0\rightarrow 2}$
- $L_0$ launches $L_2$
  - $L_2$ causes a trap
  - $L_0$ handles trap itself or forwards it to $L_1$
  - ...
  - eventually, $L_0$ resumes $L_2$
  - repeat
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
Nested VMX virtualization: flow

- $L_0$ runs $L_1$ with $VMCS_{0 \rightarrow 1}$
- $L_1$ prepares $VMCS_{1 \rightarrow 2}$ and executes `vmlaunch`
- `vmlaunch` traps to $L_0$
- $L_0$ merges $VMCS$’s:
  - $VMCS_{0 \rightarrow 1}$ merged with $VMCS_{1 \rightarrow 2}$ is $VMCS_{0 \rightarrow 2}$
- $L_0$ launches $L_2$
- $L_2$ causes a trap
- $L_0$ handles trap itself or forwards it to $L_1$
  - ...
  - eventually, $L_0$ resumes $L_2$
  - repeat
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
Nested VMX virtualization: flow

- $L_0$ runs $L_1$ with $\text{VMCS}_{0\to1}$
- $L_1$ prepares $\text{VMCS}_{1\to2}$ and executes $\text{vmlaunch}$
- $\text{vmlaunch}$ traps to $L_0$
- $L_0$ merges $\text{VMCS}$’s:
  - $\text{VMCS}_{0\to1}$ merged with $\text{VMCS}_{1\to2}$ is $\text{VMCS}_{0\to2}$
- $L_0$ launches $L_2$
- $L_2$ causes a trap
- $L_0$ handles trap itself or forwards it to $L_1$
- $\ldots$
- eventually, $L_0$ resumes $L_2$
- repeat
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
Exit multiplication makes angry turtle angry

- To handle a single L$_2$ exit, L$_1$ does many things: read and write the VMCS, disable interrupts, …
- Those operations can trap, leading to exit multiplication
- Exit multiplication: a single L$_2$ exit can cause 40-50 L$_1$ exits!
- Optimize: make a single exit fast and reduce frequency of exits
Exit multiplication makes angry turtle angry

- To handle a single L\(_2\) exit, L\(_1\) does many things: read and write the VMCS, disable interrupts, . . .
- Those operations can trap, leading to exit multiplication
- **Exit multiplication**: a single L\(_2\) exit can cause 40-50 L\(_1\) exits!
- Optimize: make a single exit fast and reduce frequency of exits
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
MMU virtualization 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
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 ⇒ one table in hardware
Better: shadow-on-EPT

- 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
- \( L_0 \) emulates EPT for \( L_1 \)
- \( L_0 \) uses \( \text{EPT}_{0\rightarrow 1} \) and \( \text{EPT}_{1\rightarrow 2} \) to construct \( \text{EPT}_{0\rightarrow 2} \)
- End result: a lot less exits!
Introduction to I/O virtualization

- Device emulation [Sugerman01]

![Diagram of device emulation]

Para-virtualized drivers [Barham03, Russell08]

Direct device assignment [Levasseur04, Yassour08]

Direct assignment best performing option

Direct assignment requires IOMMU for safe DMA bypass
Introduction to I/O virtualization

- Device emulation [Sugerman01]

- Para-virtualized drivers [Barham03, Russell08]
Introduction to I/O virtualization

- **Device emulation** [Sugerman01]

![Diagram of device emulation](image)

- **Para-virtualized drivers** [Barham03, Russell08]

![Diagram of para-virtualized drivers](image)

- **Direct device assignment** [Levasseur04, Yassour08]

![Diagram of direct device assignment](image)
Introduction to I/O virtualization

- **Device emulation** [Sugerman01]
  
  ![Device emulation diagram]

- **Para-virtualized drivers** [Barham03, Russell08]
  
  ![Para-virtualized drivers diagram]

- **Direct device assignment** [Levasseur04, Yassour08]
  
  ![Direct device assignment diagram]

- Direct assignment best performing option
Introduction to I/O virtualization

- **Device emulation** [Sugerman01]

  ![Diagram of Device Emulation]

  Direct assignment best performing option
  Direct assignment requires IOMMU for safe DMA bypass

- **Para-virtualized drivers** [Barham03, Russell08]

  ![Diagram of Para-virtualized Drivers]

- **Direct device assignment** [Levasseur04, Yassour08]

  ![Diagram of Direct Device Assignment]
Multi-level device assignment

- With nested 3x3 options for I/O virtualization (L₂ ⇔ L₁ ⇔ L₀)
- Multi-level device assignment means giving an L₂ guest direct access to L₀’s devices, safely bypassing both L₀ and L₁

![Diagram showing multi-level device assignment]

- L₀ compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L₂ programs the device directly
- Device DMA’s into L₂ memory space directly
Multi-level device assignment

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

How? $L_0$ emulates an IOMMU for $L_1$ [Amit10]
- $L_0$ 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
Multi-level device assignment

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

How? $L_0$ emulates an IOMMU for $L_1$ \[\text{[Amit10]}\]

- $L_0$ 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
Multi-level device assignment

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

How? $L_0$ emulates an IOMMU for $L_1$ [Amit10]
$L_0$ 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
Multi-level device assignment

- With nested $3 \times 3$ options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- **Multi-level device assignment** means giving an $L_2$ guest direct access to $L_0$’s devices, safely **bypassing both $L_0$ and $L_1$**

How? $L_0$ emulates an IOMMU for $L_1$ \cite{Amit10}
- $L_0$ **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
Multi-level device assignment

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

**How?** $L_0$ emulates an IOMMU for $L_1$ [Amit10]
- $L_0$ 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
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, more benchmarks and analysis, including worst case synthetic micro-benchmark
**Macro:** SPECjbb and kernbench

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

<table>
<thead>
<tr>
<th>SPECjbb</th>
<th>Host</th>
<th>Guest</th>
<th>Nested</th>
<th>Nested_{DRW}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>90493</td>
<td>83599</td>
<td>77065</td>
<td>78347</td>
</tr>
<tr>
<td>% degradation vs. host</td>
<td>-</td>
<td>7.6</td>
<td>14.8</td>
<td>13.4</td>
</tr>
<tr>
<td>% degradation vs. guest</td>
<td>-</td>
<td>-</td>
<td>7.8</td>
<td>6.3</td>
</tr>
</tbody>
</table>

**Table:** kernbench and SPECjbb results

- Exit multiplication effect not as bad as we feared
- **Direct vmread and 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: $\#(\text{EPT violations}) \ll \#(\text{page faults})$
- Multi-dimensional paging huge win for page-fault intensive kernbench
Macro: multi-level device assignment

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 L₂?

Only 7% difference between native and nested guest!
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
- It’s turtles all the way down
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
- It’s turtles all the way down
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
- It’s turtles all the way down

Ben-Yehuda et al. (IBM Research)  The Turtles Project: Nested Virtualization  OSDI ’10  21 / 22
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
  - It’s turtles all the way down
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
- It’s turtles all the way down