

The Turtles Project: Design and Implementation of Nested Virtualization

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

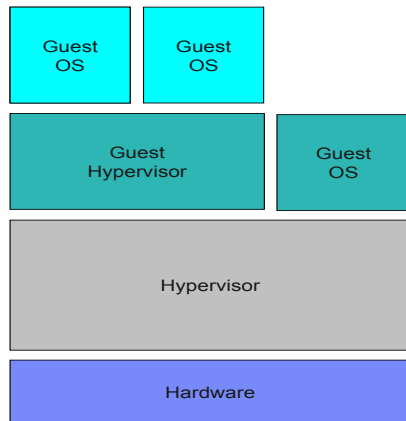
[†]IBM Research – Haifa

[‡]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



- 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](#)]



- 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](#)]



- 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](#)]



- 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?

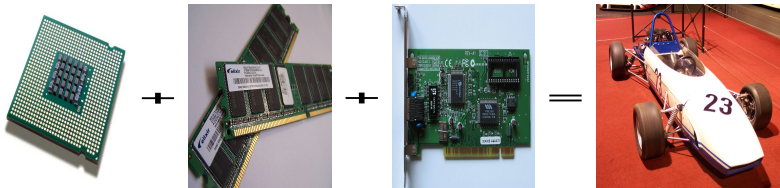


- [Efficient nested virtualization for Intel x86](#) based on KVM
- Runs multiple guest hypervisors and VMs: KVM, VMware, Linux, Windows, ...
- Code publicly available



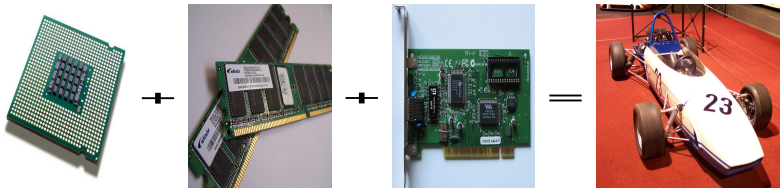
What is the Turtles project? (cont')

- 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



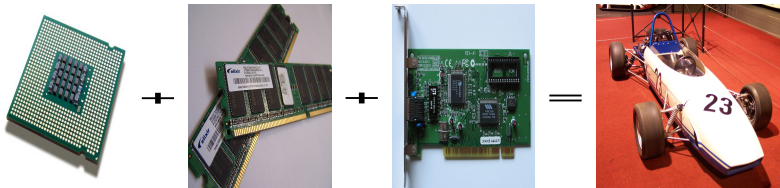
What is the Turtles project? (cont')

- 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



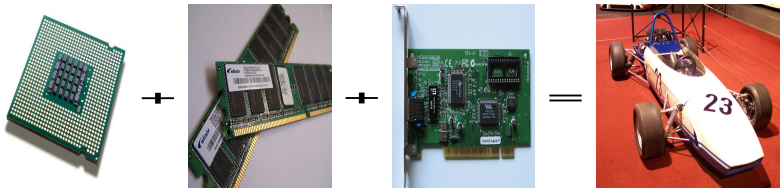
What is the Turtles project? (cont')

- 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



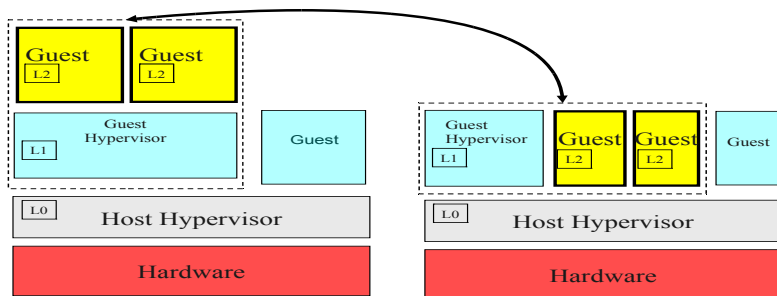
What is the Turtles project? (cont')

- 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] \Rightarrow it's all about the traps
- Single-level (x86) vs. multi-level (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 $n=2$)



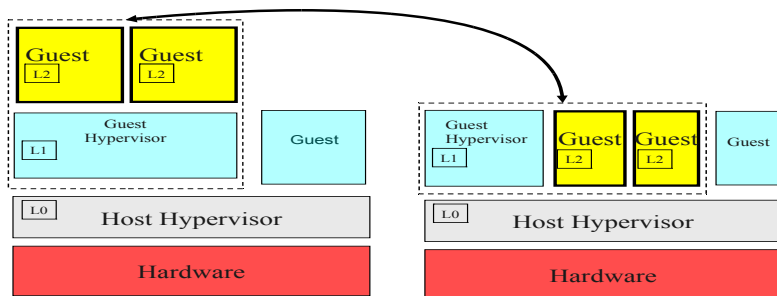
Multiple logical levels

Multiplexed on a single level



Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] \Rightarrow it's all about the traps
- Single-level (x86) vs. multi-level (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 $n=2$)



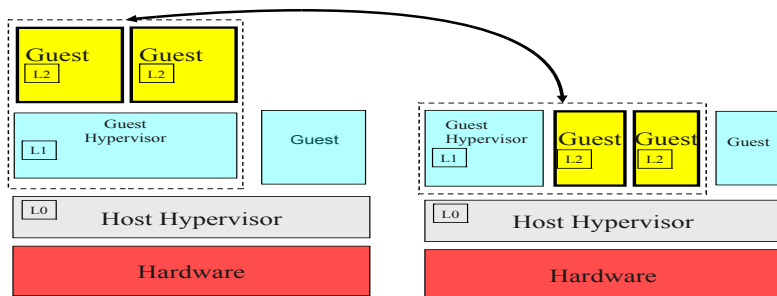
Multiple logical levels

Multiplexed on a single level



Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] \Rightarrow it's all about the traps
- Single-level (x86) vs. multi-level (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 $n=2$)



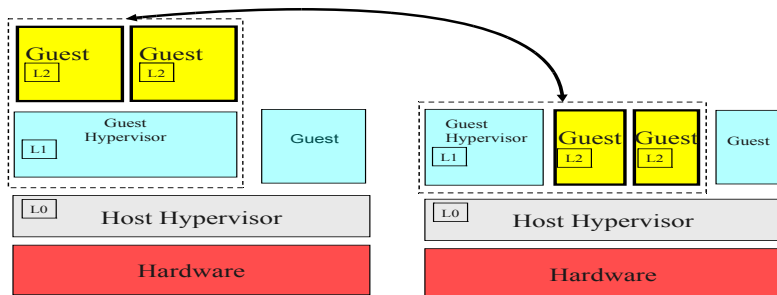
Multiple logical levels

Multiplexed on a single level



Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] \Rightarrow it's all about the traps
- Single-level (x86) vs. multi-level (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 $n=2$)



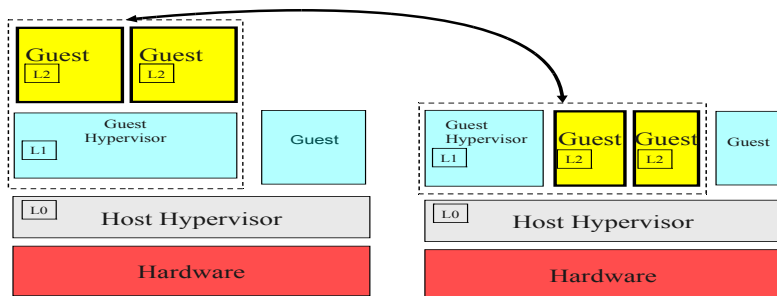
Multiple logical levels

Multiplexed on a single level



Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] \Rightarrow it's all about the traps
- Single-level (x86) vs. multi-level (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 $n=2$)



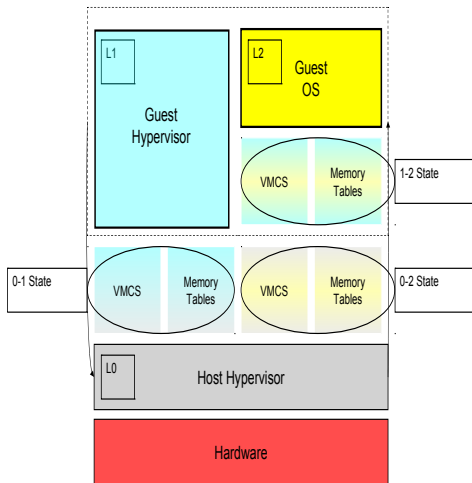
Multiple logical levels

Multiplexed on a single level



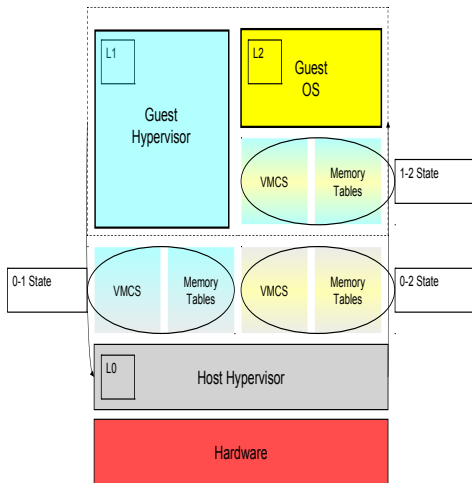
Nested VMX virtualization: flow

- 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



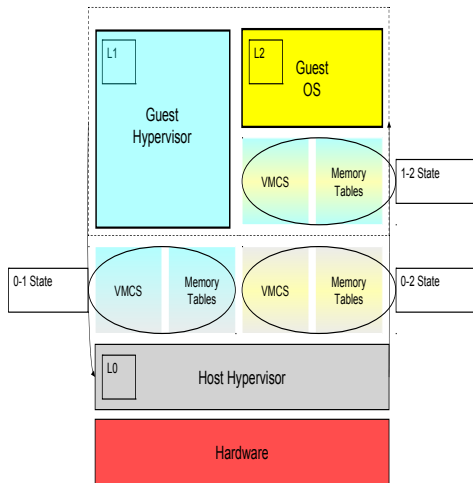
Nested VMX virtualization: flow

- 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



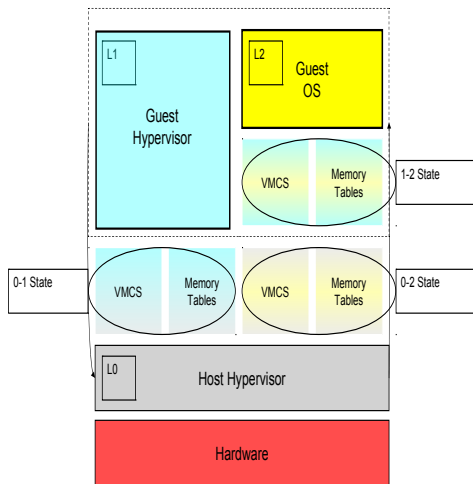
Nested VMX virtualization: flow

- 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



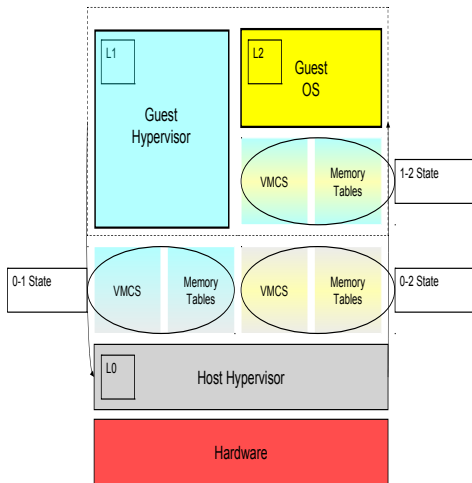
Nested VMX virtualization: flow

- 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



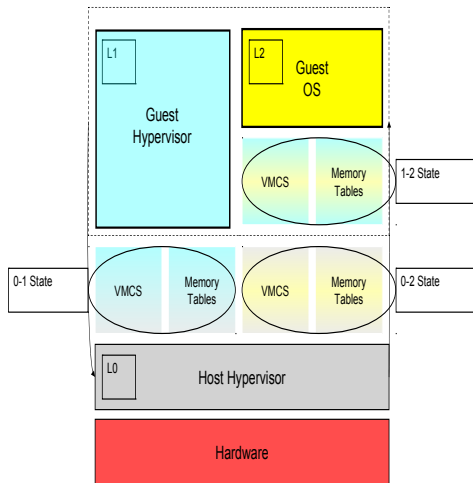
Nested VMX virtualization: flow

- 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



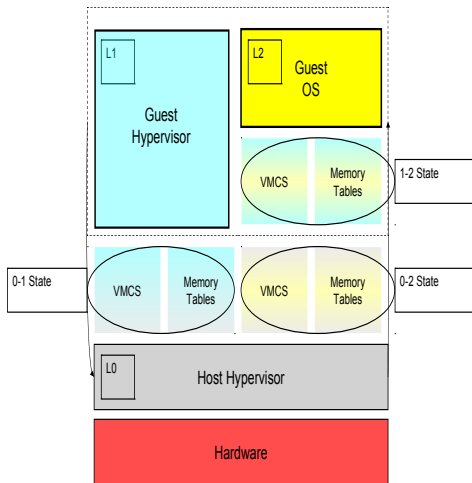
Nested VMX virtualization: flow

- 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



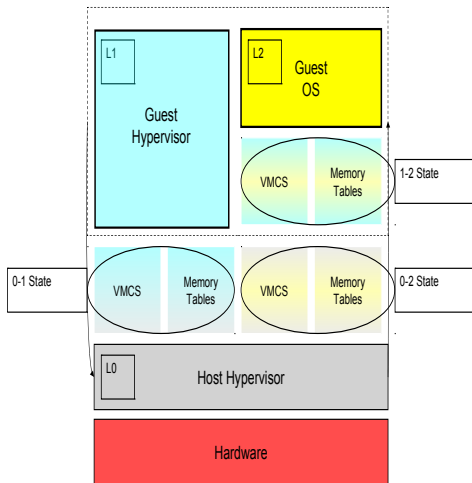
Nested VMX virtualization: flow

- 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



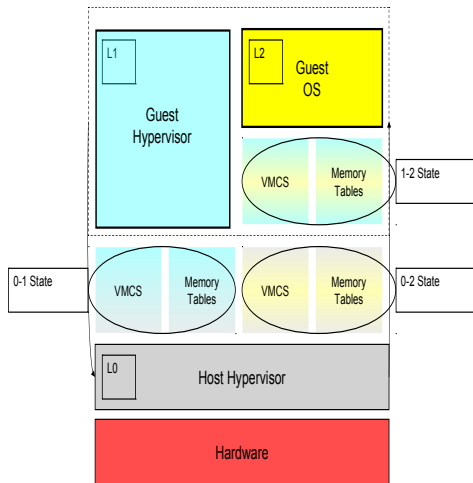
Nested VMX virtualization: flow

- 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



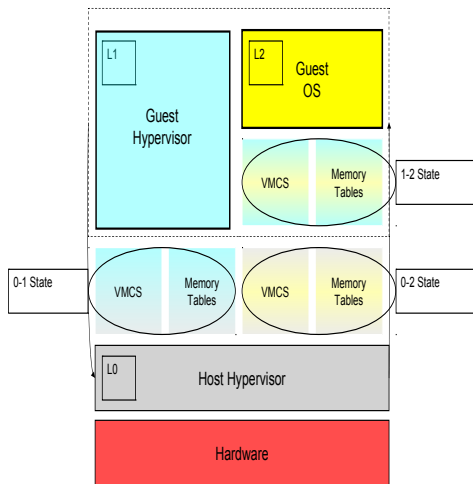
Nested VMX virtualization: flow

- 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



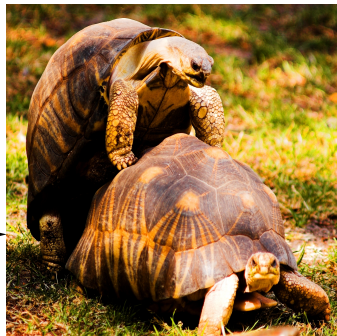
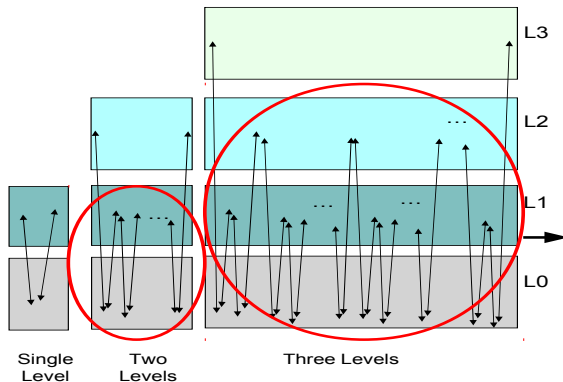
Nested VMX virtualization: flow

- 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



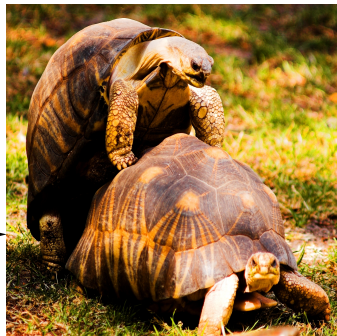
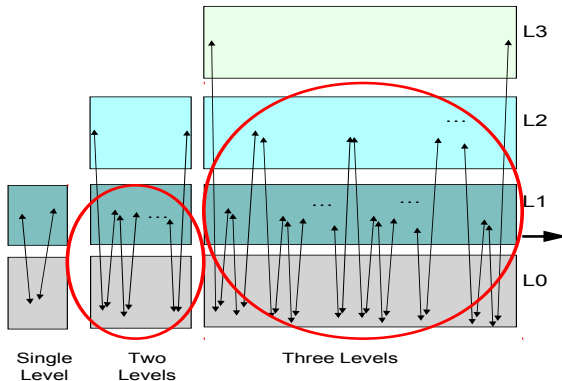
Exit multiplication makes angry turtle angry

- 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**



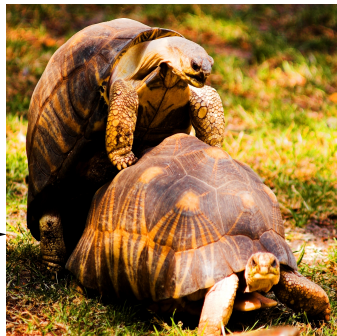
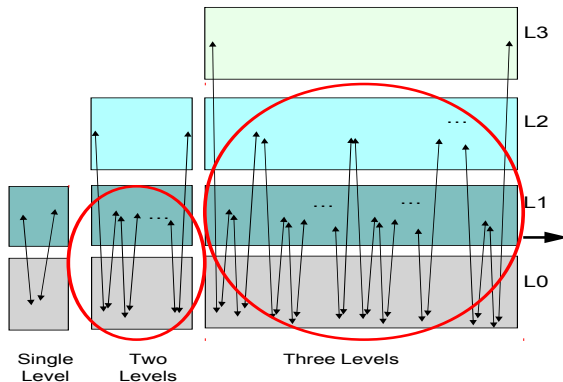
Exit multiplication makes angry turtle angry

- 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**



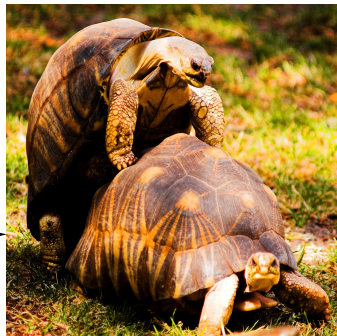
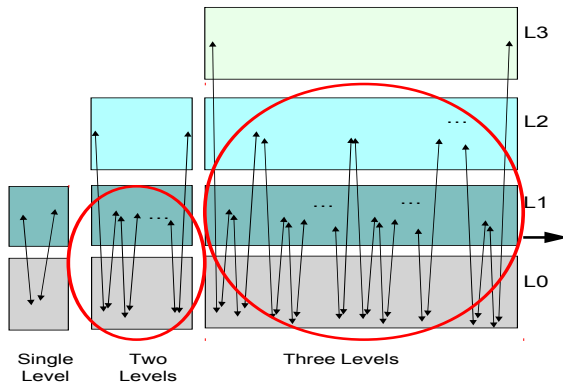
Exit multiplication makes angry turtle angry

- To handle a single 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**



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!



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!



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!



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!

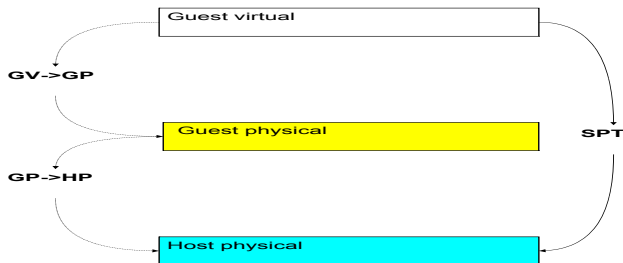


Introduction to x86 MMU virtualization

- 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!



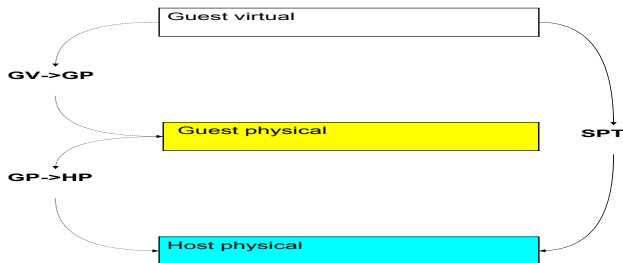
Software MMU virtualization: shadow paging



- 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](#)



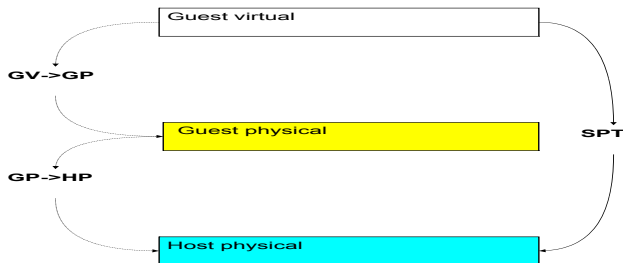
Software MMU virtualization: shadow paging



- 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](#)



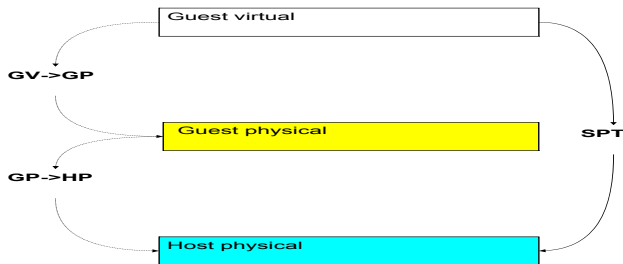
Software MMU virtualization: shadow paging



- 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](#)



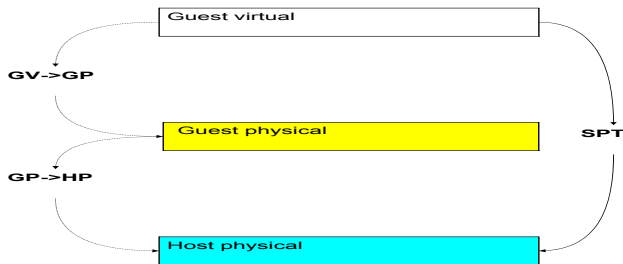
Software MMU virtualization: shadow paging



- 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](#)



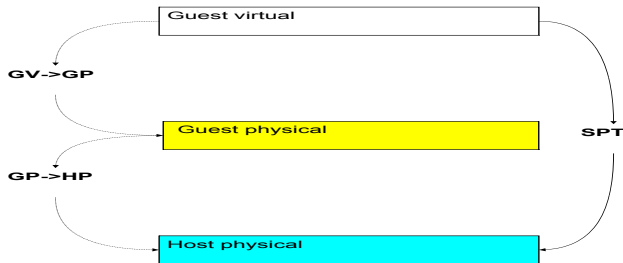
Software MMU virtualization: shadow paging



- 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](#)



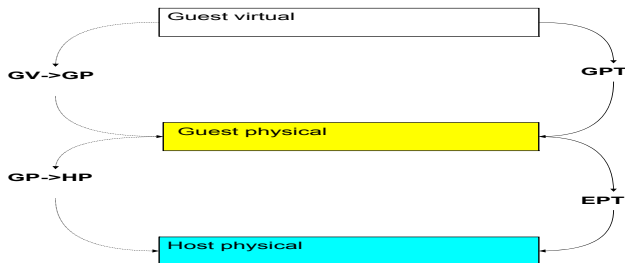
Software MMU virtualization: shadow paging



- 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](#)



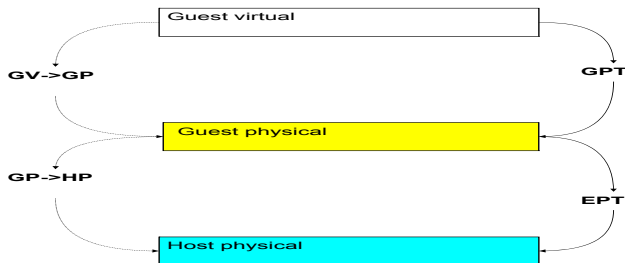
Hardware MMU virtualization: Extended Page Tables



- 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



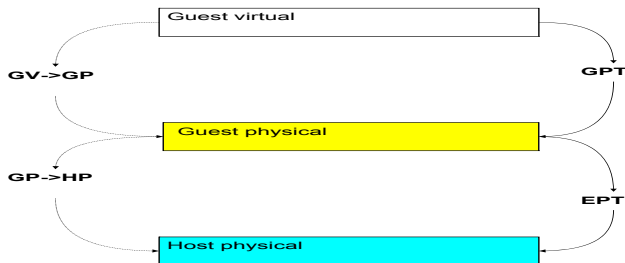
Hardware MMU virtualization: Extended Page Tables



- 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



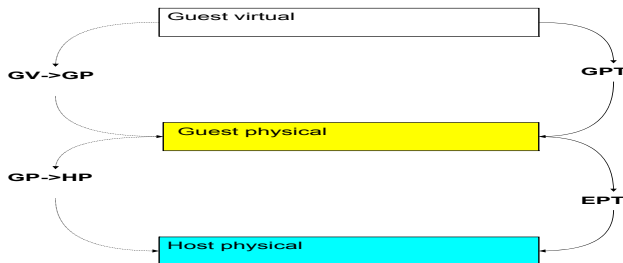
Hardware MMU virtualization: Extended Page Tables



- 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



Hardware MMU virtualization: Extended Page Tables

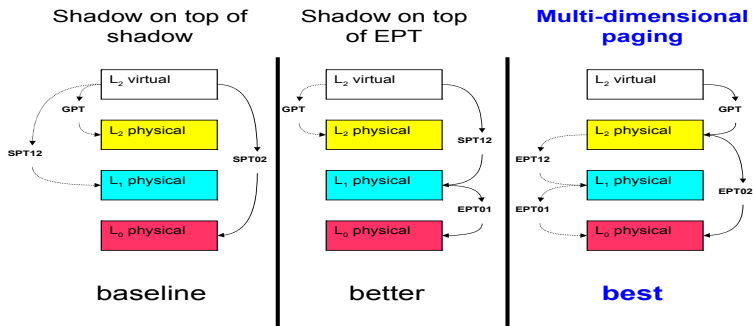


- Two-dimensional paging: guest owns GPT, hypervisor owns EPT [BhargavaSerebrin08]
- Unmodified guest OS updates GPT
- Hypervisor updates EPT table controlling (guest phys \rightarrow host phys) translations
- MMU walks both tables

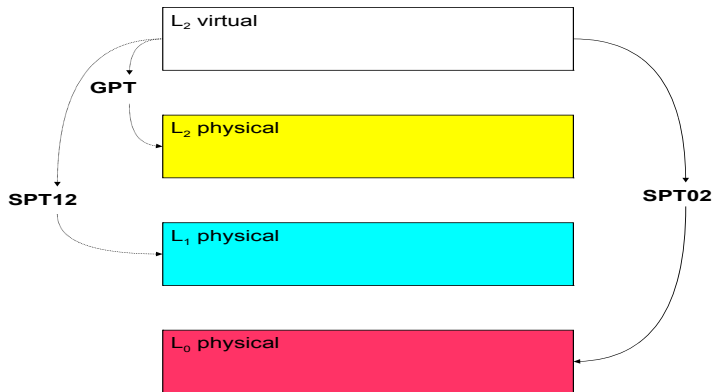


Nested MMU virt. via multi-dimensional paging

- **Three logical translations:** $L_2 \text{ virt} \rightarrow \text{phys}$, $L_2 \rightarrow L_1$, $L_1 \rightarrow L_0$
- 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



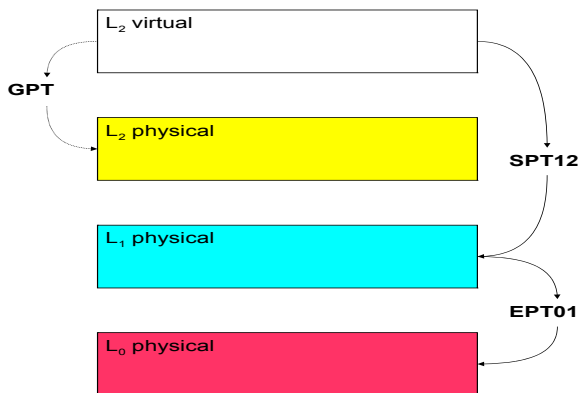
Baseline: shadow-on-shadow



- 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 \Rightarrow one table in hardware



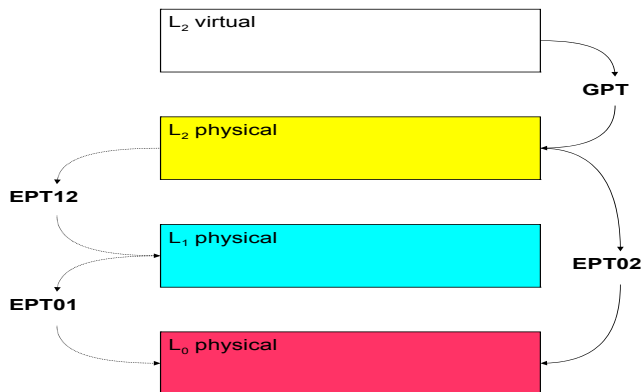
Better: shadow-on-EPT



- Instead of one hardware table we have two
- **Compress**: **three** logical translations \Rightarrow **two** in hardware
- Simple approach: L₀ uses EPT, L₁ uses shadow paging for L₂
- Every L₂ page fault leads to multiple L₁ exits



Best: multi-dimensional paging



- EPT table rarely changes; guest page table changes a lot
- Again, **compress three** logical translations \Rightarrow **two** in hardware
- L₀ **emulates** EPT for L₁
- L₀ uses EPT_{0 \rightarrow 1} and EPT_{1 \rightarrow 2} to construct EPT_{0 \rightarrow 2}
- End result: a lot less exits!

Introduction to I/O virtualization

- From the hypervisor's perspective, what is I/O?



Introduction to I/O virtualization

- From the hypervisor's perspective, what is I/O?
- (1) PIO



Introduction to I/O virtualization

- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO



Introduction to I/O virtualization

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



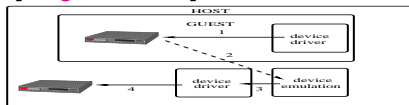
Introduction to I/O virtualization

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



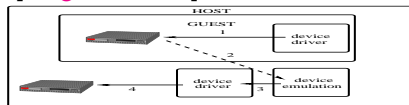
Introduction to I/O virtualization

- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]

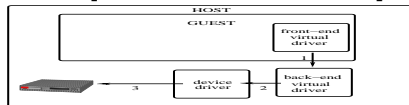


Introduction to I/O virtualization

- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]

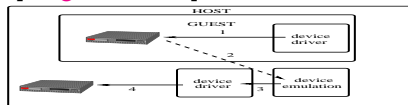


- Para-virtualized drivers [Barham03, Russell08]

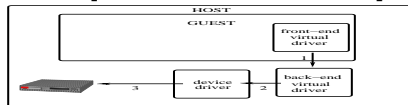


Introduction to I/O virtualization

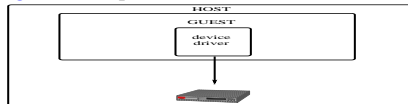
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]

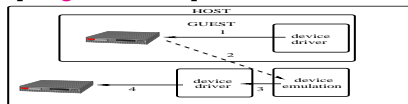


- Direct device assignment [Levasseur04, Yassour08]

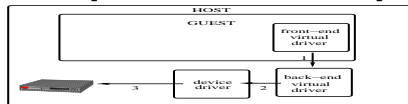


Introduction to I/O virtualization

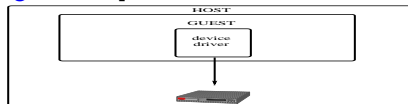
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]



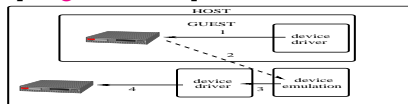
- Direct device assignment [Levasseur04, Yassour08]



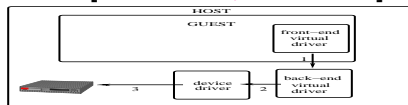
- Direct assignment **best performing option**

Introduction to I/O virtualization

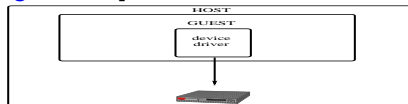
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]



- Direct device assignment [Levasseur04, Yassour08]

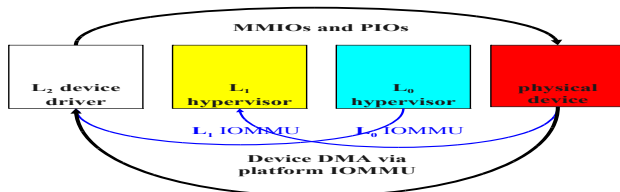


- Direct assignment **best performing option**
- Direct assignment **requires IOMMU** for safe DMA bypass



Multi-level device assignment

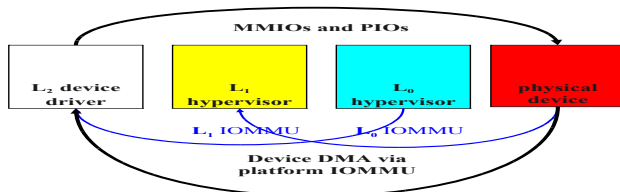
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0 's devices, safely bypassing both L_0 and L_1



- 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 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

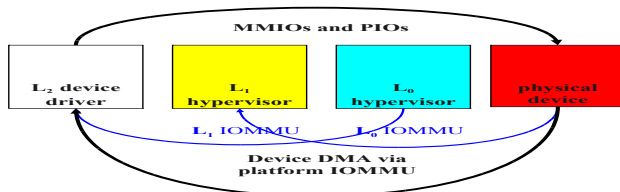


- How? L_0 emulates an IOMMU for L_1 [Amit10]
- L_0 compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA's into L_2 memory space directly



Multi-level device assignment

- With nested 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

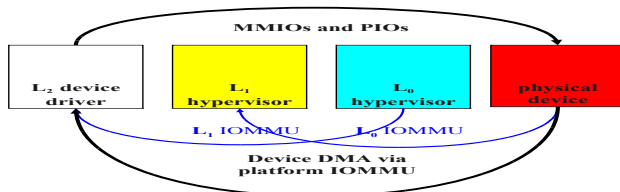


- How? L_0 emulates an IOMMU for L_1 [Amit10]
- L_0 compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA's into L_2 memory space directly



Multi-level device assignment

- With nested 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

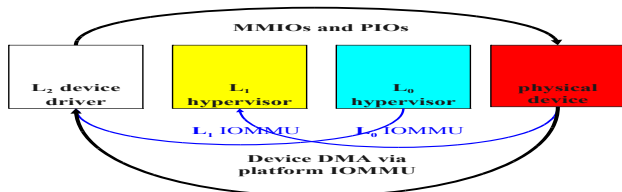


- How? L_0 emulates an IOMMU for L_1 [Amit10]
- L_0 compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA's into L_2 memory space directly



Multi-level device assignment

- With nested 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

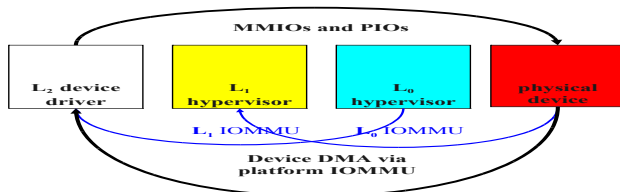


- How? L_0 emulates an IOMMU for L_1 [Amit10]
- L_0 compresses multiple IOMMU translations onto the single hardware IOMMU page table
- L_2 programs the device directly
- Device DMA's into L_2 memory space directly



Multi-level device assignment

- With nested 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

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?



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?



- 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?



Windows XP on KVM L₁ on KVM L₀



Linux on VMware L₁ on KVM L₀

The screenshot displays a QEMU virtual machine environment. The main window, titled 'Ubuntu-7.10-server-amd64 - localhost', shows a terminal session where the user 'oritw' is running various commands to verify the system's configuration and capabilities. The terminal output includes the Ubuntu warranty disclaimer, a list of system files, and the results of the 'lscpu' command, which confirms that the system is running on KVM hardware. A secondary window, titled 'Virtual machine communication interface', is open, showing a menu with options like 'Remote Console' and 'Devices'. The background window shows the QEMU application window with a menu bar and a list of virtual machine settings.

```
oritw@localhost:~  
File Edit View Terminal Tabs Help  
Virtual machine communication interface [OK]  
Ubuntu-7.10-server-amd64 - localhost  
Remote Console Devices  
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.  
To access official Ubuntu documentation, please visit:  
http://help.ubuntu.com/  
ubuntu@ubuntu64:~$ echo "Hello!! I am an ubuntu 64 running on top of vmware on t  
top of kvm"  
echo "Hello sudo halt I am an ubuntu 64 running on top of vmware on top of kvm"  
Hello sudo halt I am an ubuntu 64 running on top of vmware on top of kvm  
ubuntu@ubuntu64:~$ ls -la  
total 24  
drwxr-xr-x 2 ubuntu ubuntu 4096 2009-07-27 17:57 .  
drwxr-xr-x 3 root root 4096 2009-07-27 17:56 ..  
-rw-r--r-- 1 ubuntu ubuntu 215 2009-07-29 13:03 .bash_history  
-rw-r--r-- 1 ubuntu ubuntu 220 2009-07-27 17:56 .bash_logout  
-rw-r--r-- 1 ubuntu ubuntu 3115 2009-07-27 17:56 .bashrc  
-rw-r--r-- 1 ubuntu ubuntu 675 2009-07-27 17:56 .profile  
-rw-r--r-- 1 ubuntu ubuntu 0 2009-07-27 17:57 .sudo_as_admin_successful  
ubuntu@ubuntu64:~$ echo "Hello"  
Hello  
ubuntu@ubuntu64:~$ echo "Hello, I am an ubuntu 64 Running on top of vmware on to  
p of kvm"  
Hello, I am an ubuntu 64 Running on top of vmware on top of kvm  
ubuntu@ubuntu64:~$  
ubuntu@ubuntu64:~$  
To release input, press Ctrl+Alt  
drwxr-xr-x 2 oritw oritw 4096 2009-08-06 23:34 .wapi  
-rw-r--r-- 1 oritw oritw 115 2008-03-25 22:23 .Xauthority  
-rw-r--r-- 1 oritw oritw 1075 2009-08-06 23:34 .xsession-errors  
[oritw@localhost ~]$ ./vmware_ubuntu64 mode  
Aug 06 23:34:59.383: vmx| HV Settings: virtual exec = 'hardware'; virtual mmu =  
'hardware'  
[oritw@localhost ~]$
```

Experimental Setup

- Running Linux, [Windows](#), KVM, [VMware](#), SMP, ...
- Macro workloads:
 - kernbench
 - SPECjbb
 - netperf
- Multi-dimensional paging?
- Multi-level device assignment?
- KVM as L_1 vs. VMware as L_1 ?
- See paper for full experimental details and more benchmarks and analysis



Macro: SPECjbb and kernbench

| kernbench | | | | |
|----------------------|-------|-------|--------|-----------------------|
| | Host | Guest | Nested | Nested _{DRW} |
| Run time | 324.3 | 355 | 406.3 | 391.5 |
| % overhead vs. host | - | 9.5 | 25.3 | 20.7 |
| % overhead vs. guest | - | - | 14.5 | <u>10.3</u> |

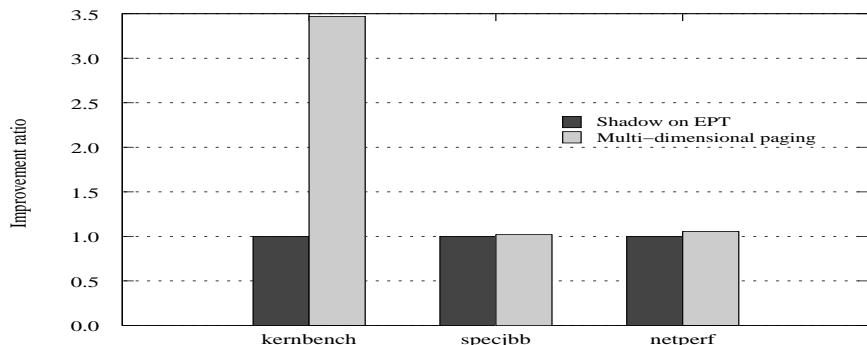
| SPECjbb | | | | |
|-------------------------|-------|-------|--------|-----------------------|
| | Host | Guest | Nested | Nested _{DRW} |
| Score | 90493 | 83599 | 77065 | 78347 |
| % degradation vs. host | - | 7.6 | 14.8 | 13.4 |
| % degradation vs. guest | - | - | 7.8 | <u>6.3</u> |

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!



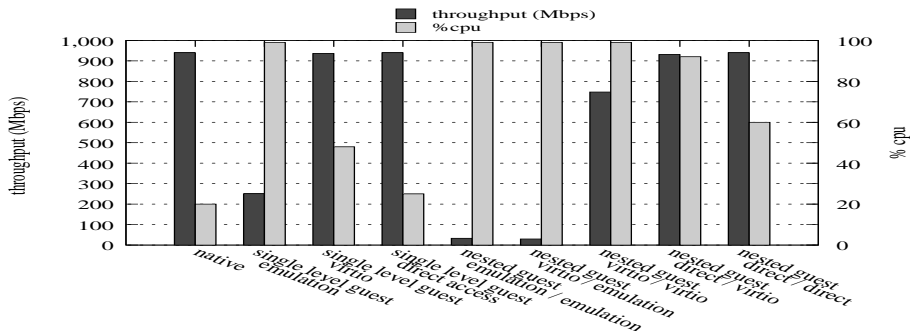
Macro: multi-dimensional paging



- 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



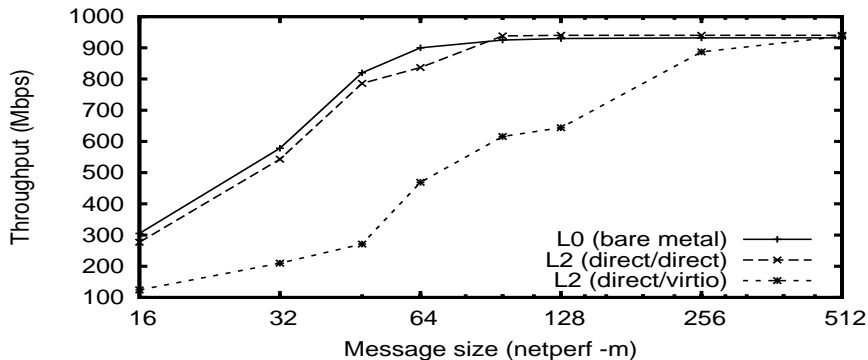
Macro: multi-level device assignment



- 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



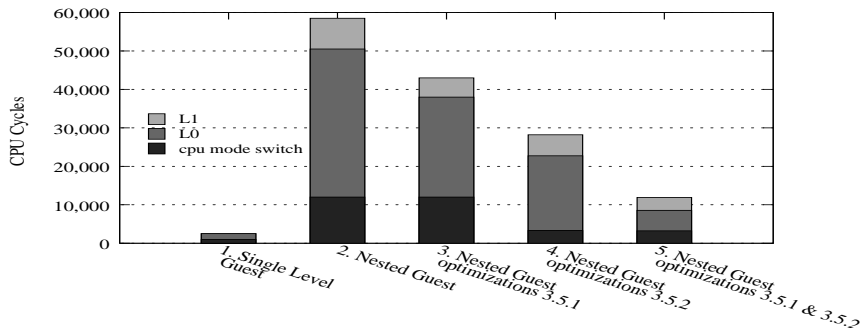
Macro: multi-level device assignment (sans interrupts)



- What if we could deliver device interrupts directly to L₂?
- Only 7% difference between native and nested guest!



Micro: synthetic worst case CPUID loop

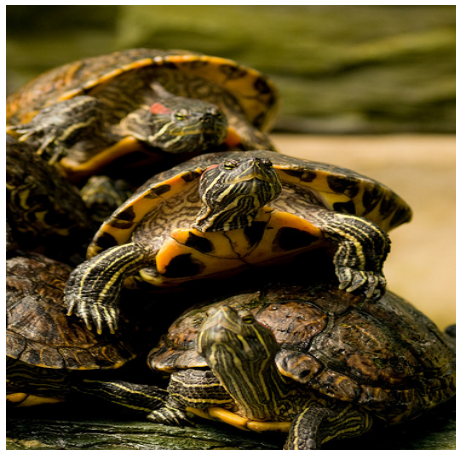


- CPUID running in a tight loop is not a real-world workload!
- Went from 30x worse to “only” 6x worse
- A nested exit is still expensive—minimize both single exit cost and frequency of exits



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



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



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



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



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



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?



IBM Research – Haifa is hiring

Systems Software Research in Haifa

- Contributions to IBM products, cutting-edge research
 - Virtualization, Cloud Computing, Data Storage
- Influencing European cloud research agenda
 - Consulting to European Commission, working through industrial groups, leading projects
- Papers, patents, image for IBM and Israel
- We are Hiring!
 - Researchers and Engineers



SYSTOR 2009
The Israeli Experimental Systems Conference
May 4-6, 2009
Organized by IBM R&D Labs in Israel



IBM Haifa Research Lab and IBM Systems and Technology Group Lab, Israeli academia, are organizing SYSTOR 2009, a successor to the high workshops on systems and storage held at the IBM Haifa Research Lab. The conference is to promote systems research and to foster stronger ties worldwide systems research communities and industry.



SYSTOR 2010
The 3rd Annual Haifa Experimental Systems Conference
May 24-26, 2010
Haifa, Israel

