The Turtles Project: Design and Implementation of Nested Virtualization

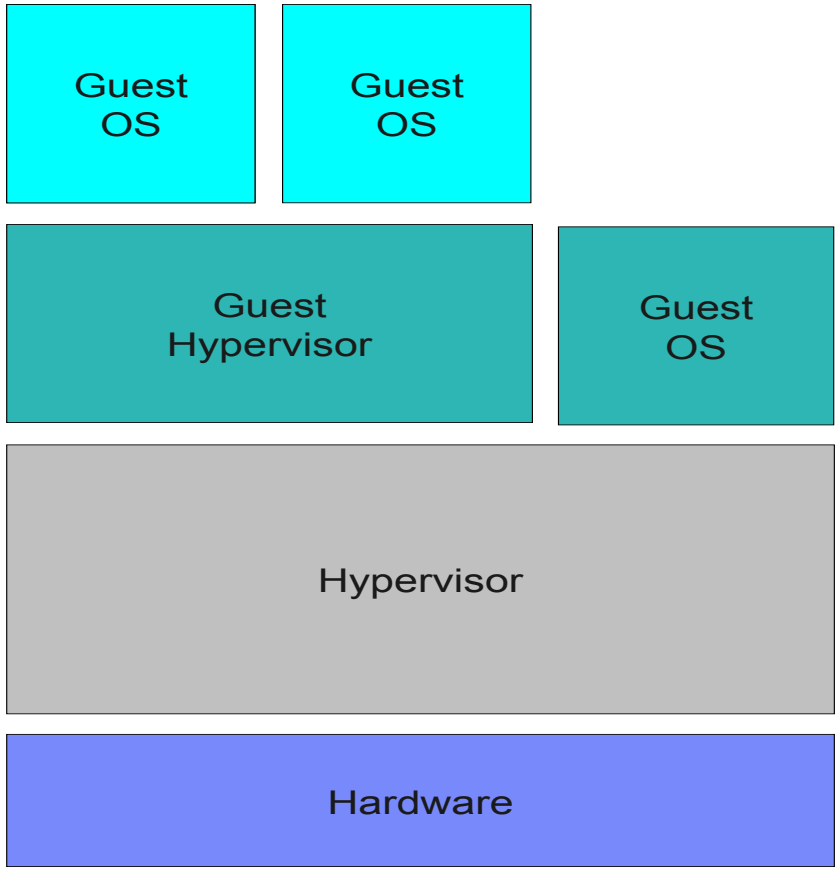
Muli Ben-Yehuda[†] Michael D. Day[‡] Zvi Dubitzky[†] Michael Factor[†]
Nadav Har'El[†] Abel Gordon[†] Anthony Liguori[‡] Orit Wasserman[†]
Ben-Ami Yassour[†]

[†]IBM Research – Haifa

[‡]IBM Linux Technology Center

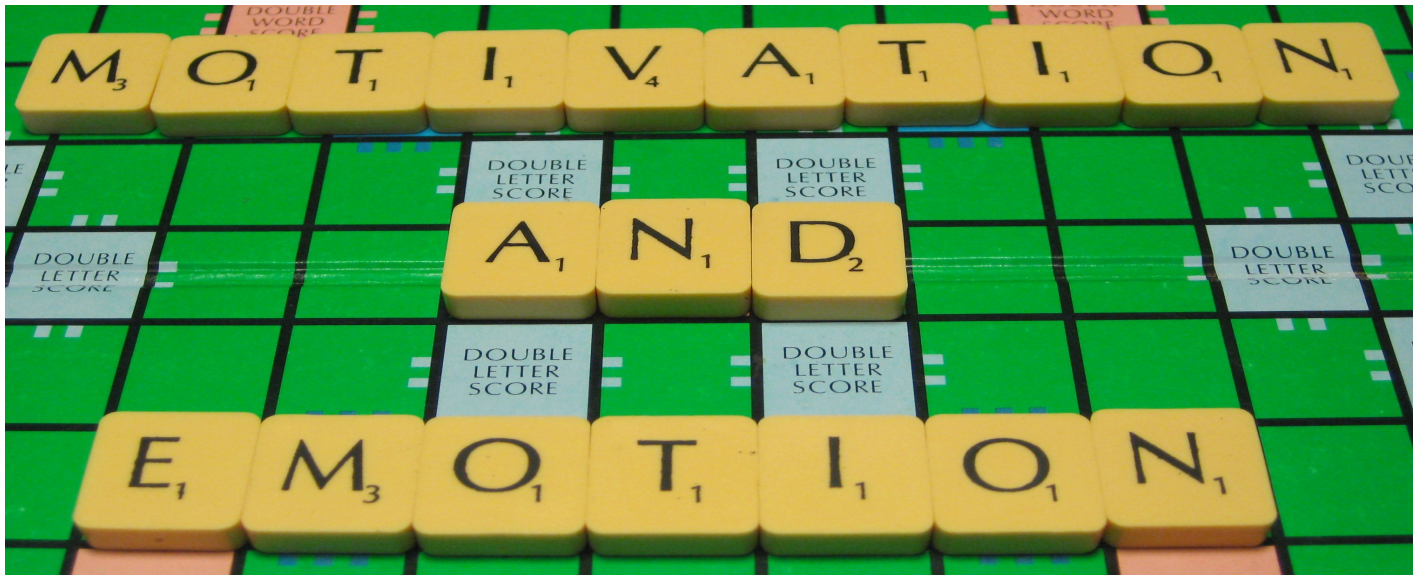
What is nested x86 virtualization?

- 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



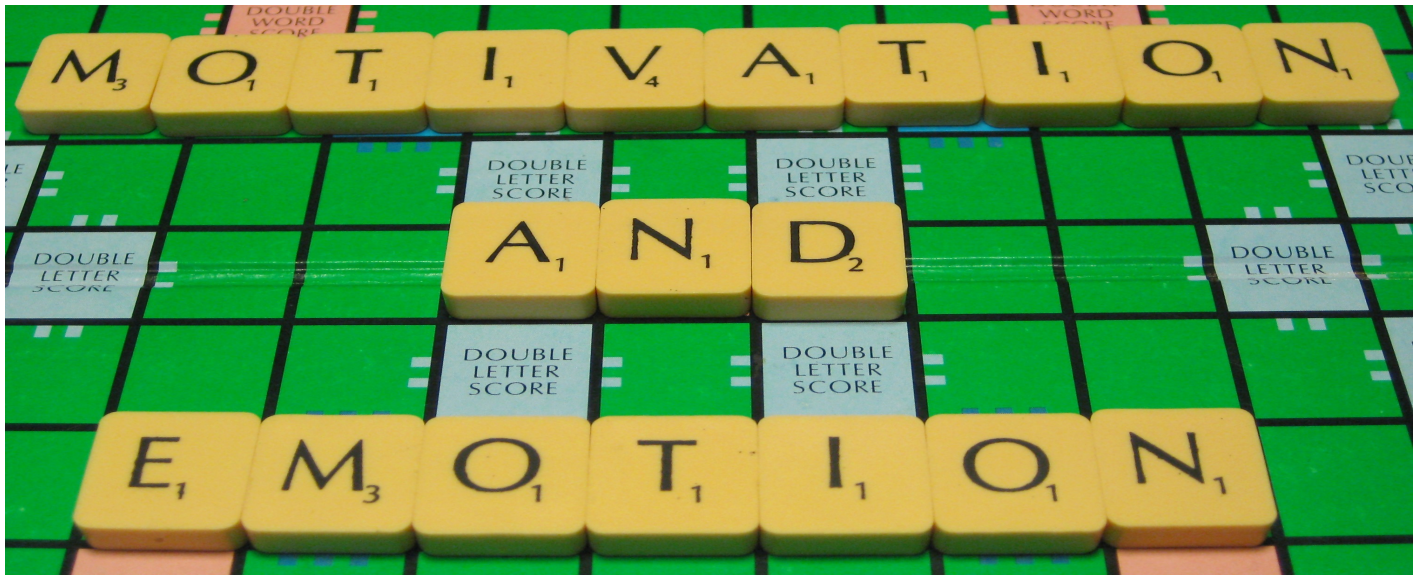
Why?

- 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



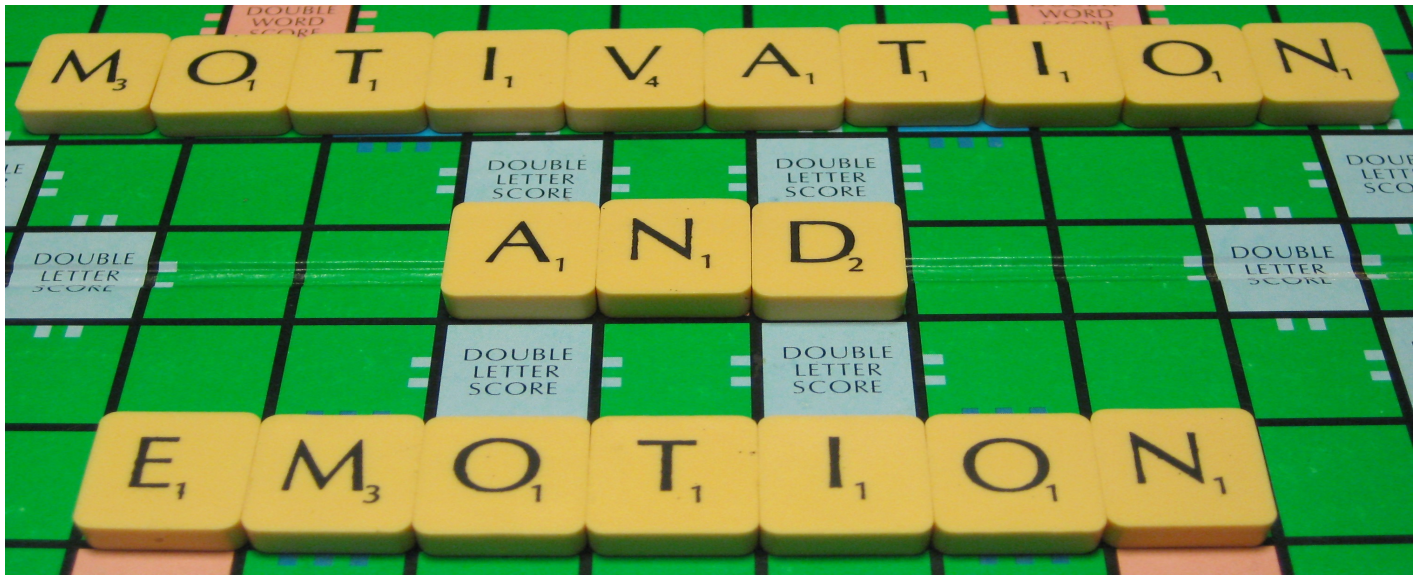
Why?

- 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



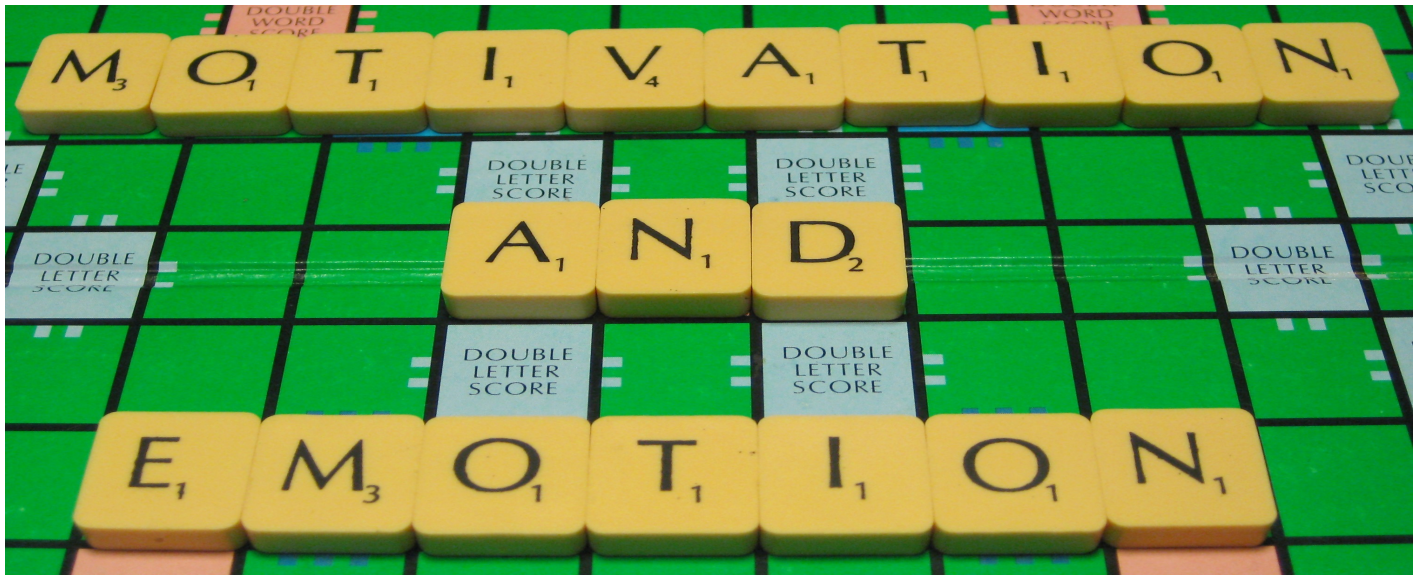
Why?

- 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



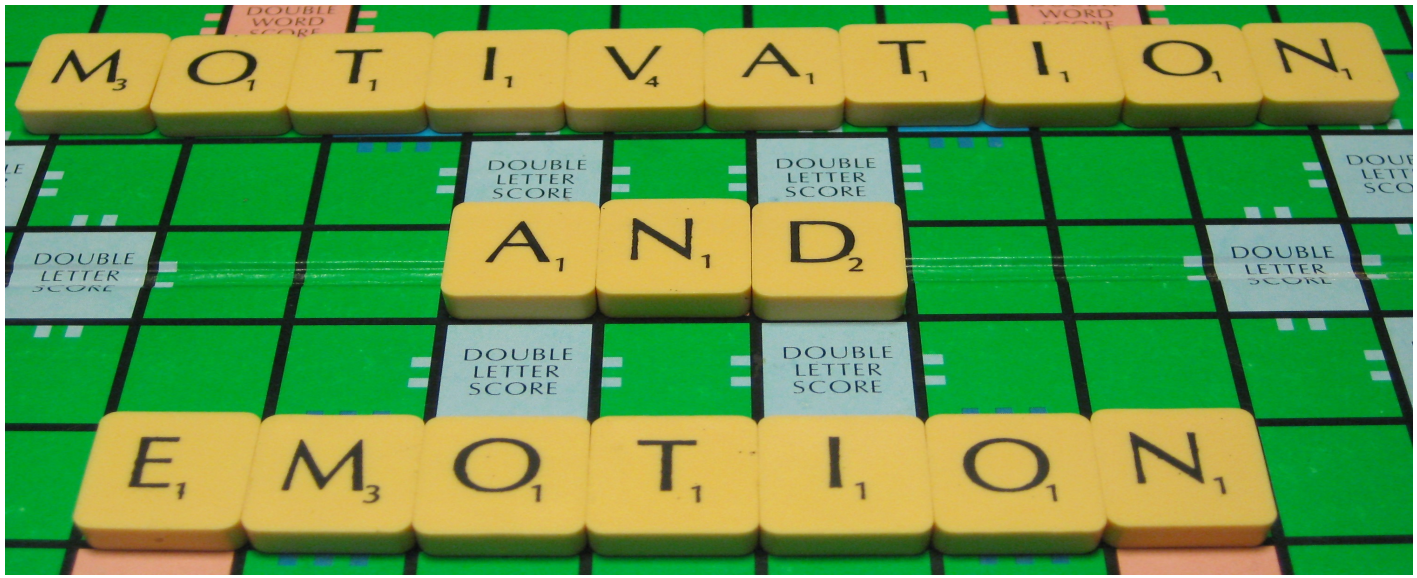
Why?

- 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



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]

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]

Related work

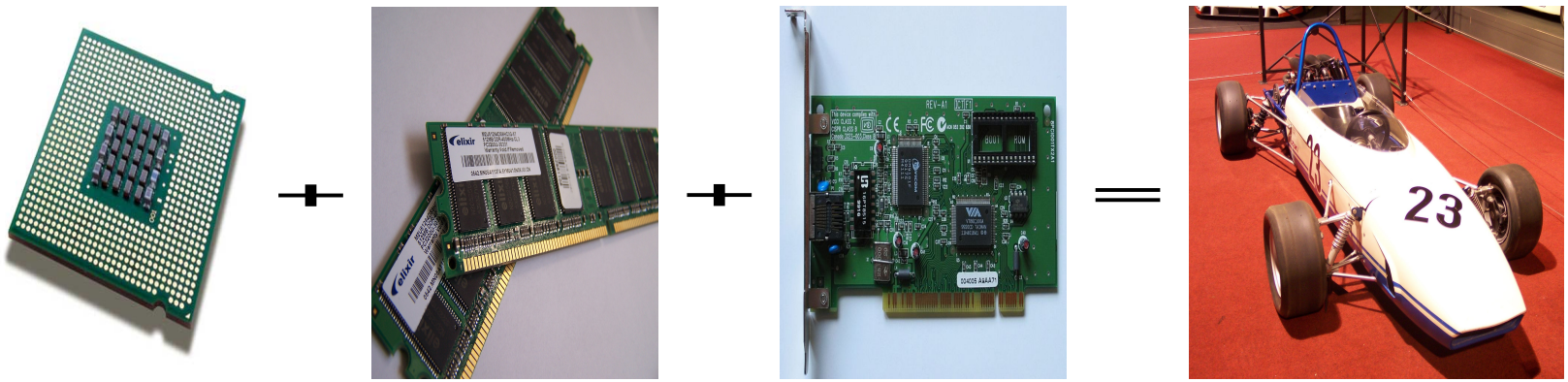
- First models for nested virtualization [PopekGoldberg74, BelpaireHsu75, LauerWyeth73]
- First implementation in the IBM z/VM; relies on architectural support for nested virtualization (`sie`)
- Microkernels meet recursive VMs [FordHibler96]: assumes we can modify software at all levels
- x86 software 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
- Runs multiple guest hypervisors and VMs: KVM, VMware, Linux, Windows, ...
- Code publicly available

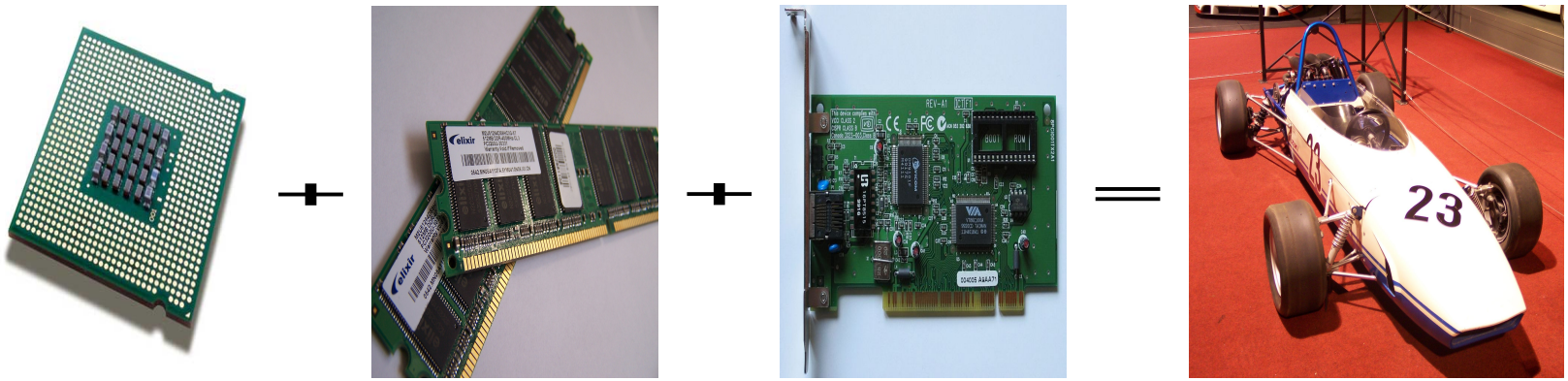
What is the Turtles project? (cont')

- 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



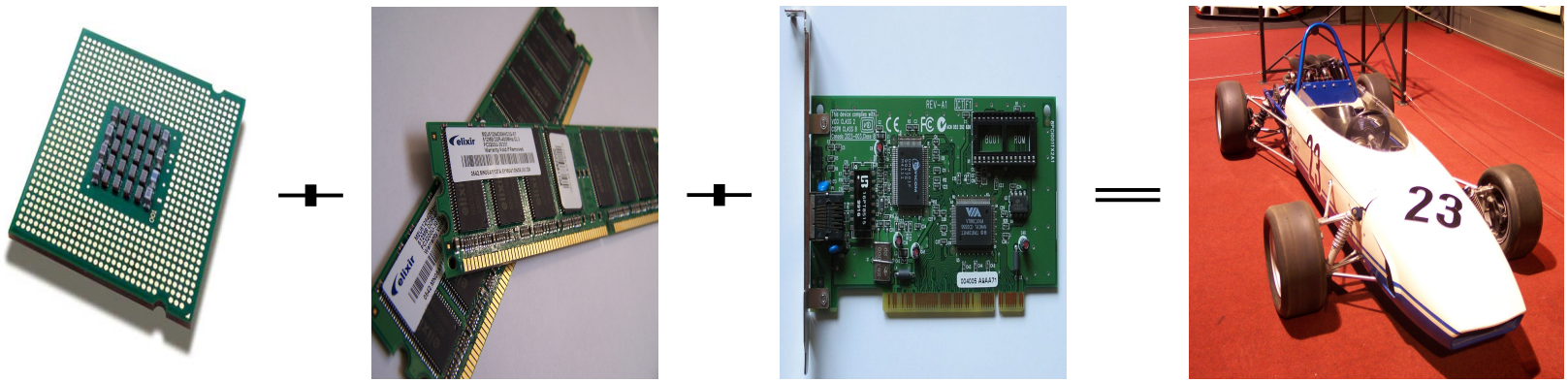
What is the Turtles project? (cont')

- 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



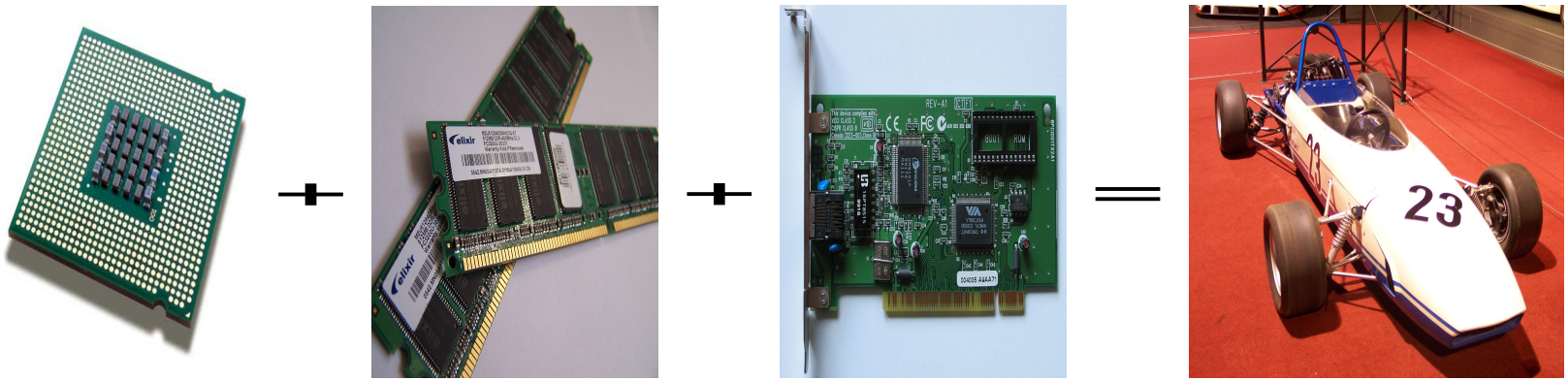
What is the Turtles project? (cont')

- 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



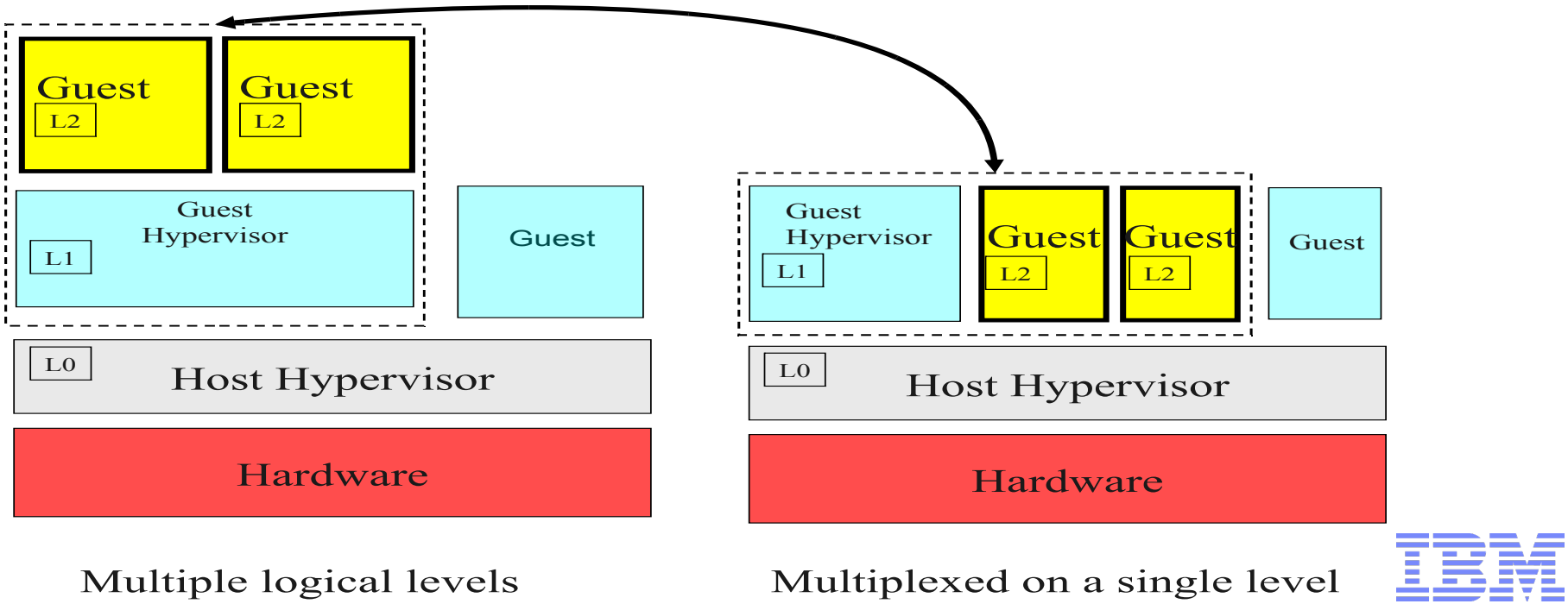
What is the Turtles project? (cont')

- 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



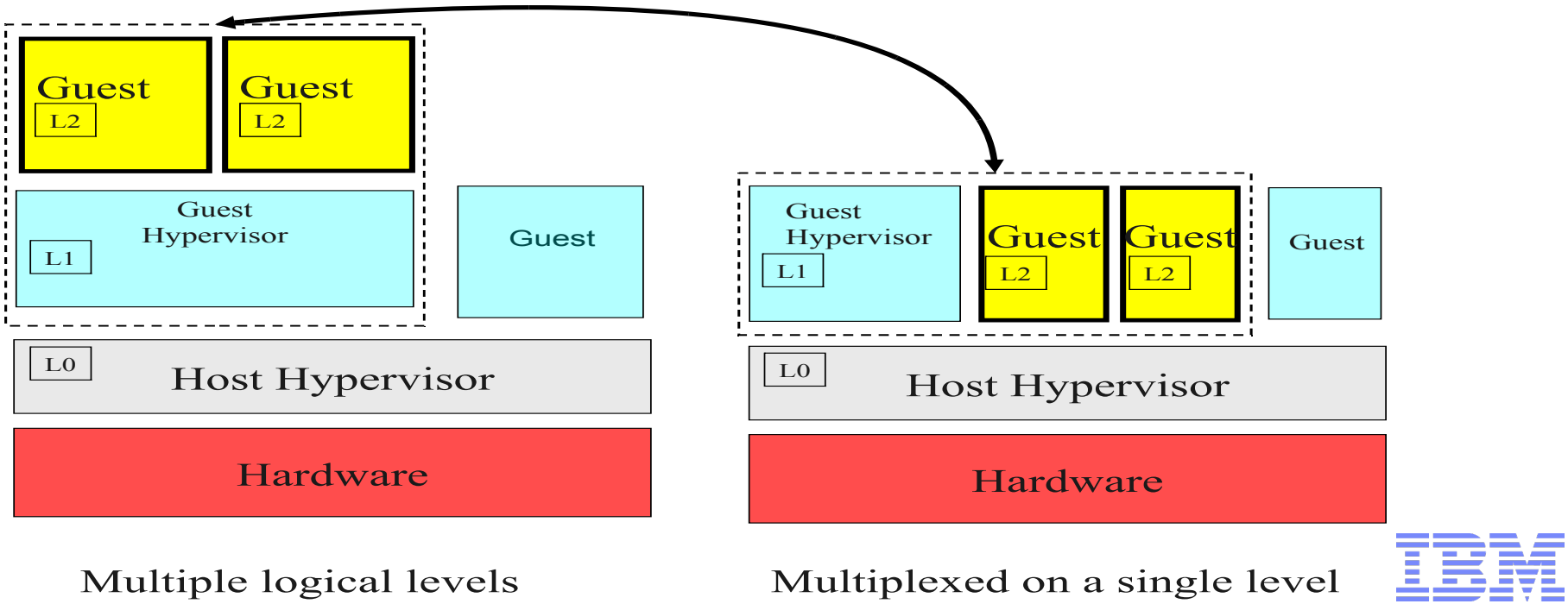
Theory of nested CPU virtualization

- Trap and emulate[PopekGoldberg74] ⇒ it's all about the traps
- Single-level (x86) vs. multi-level (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)



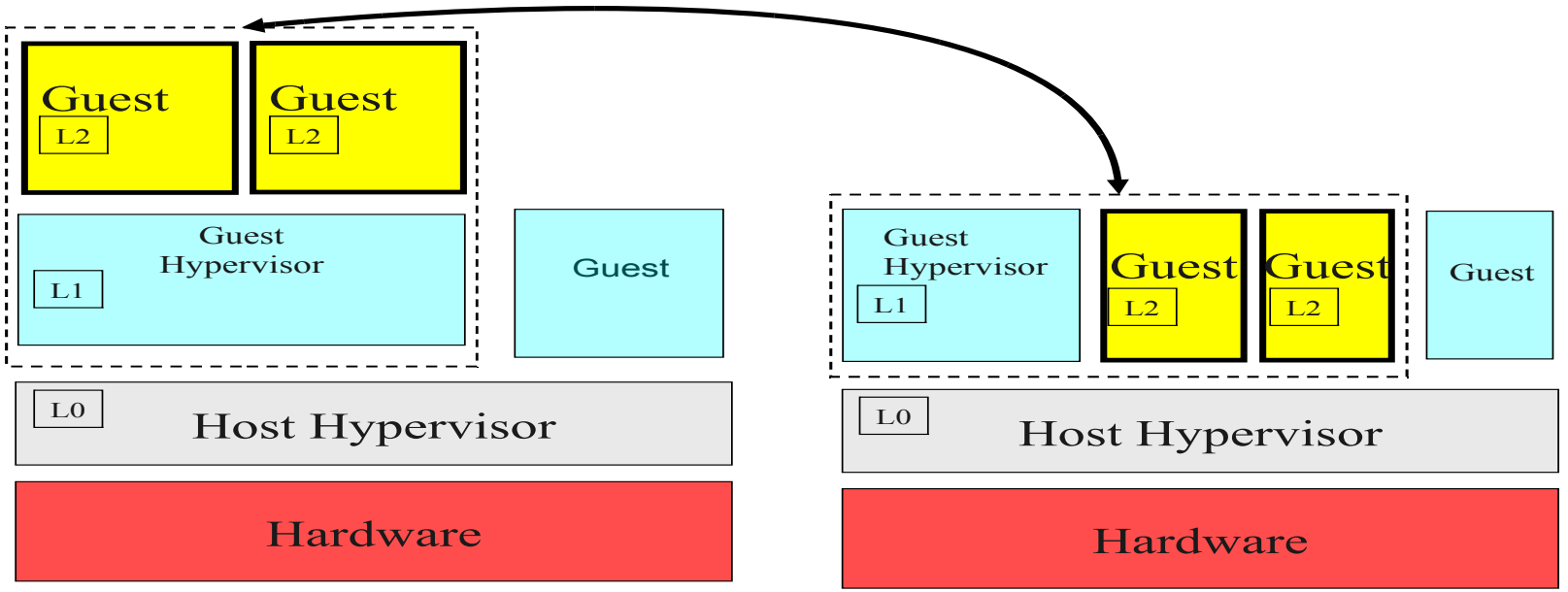
Theory of nested CPU virtualization

- 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: 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)



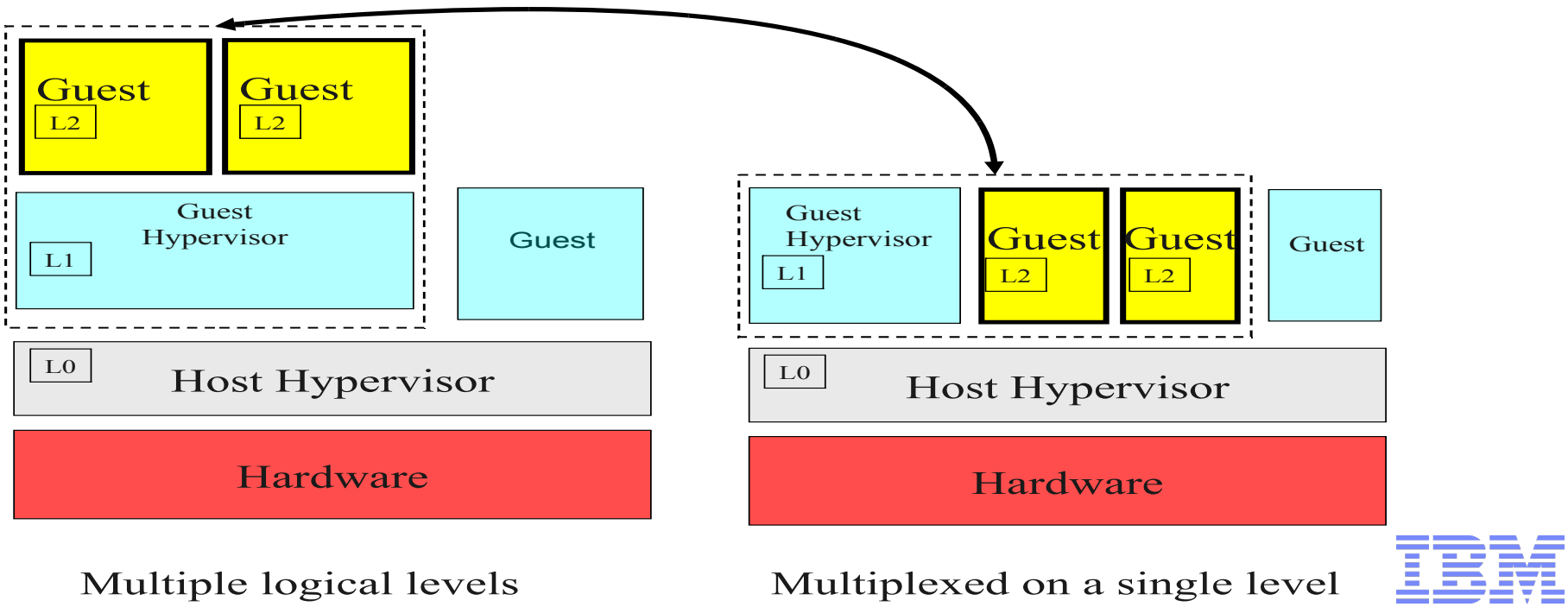
Theory of nested CPU virtualization

- 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: 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)



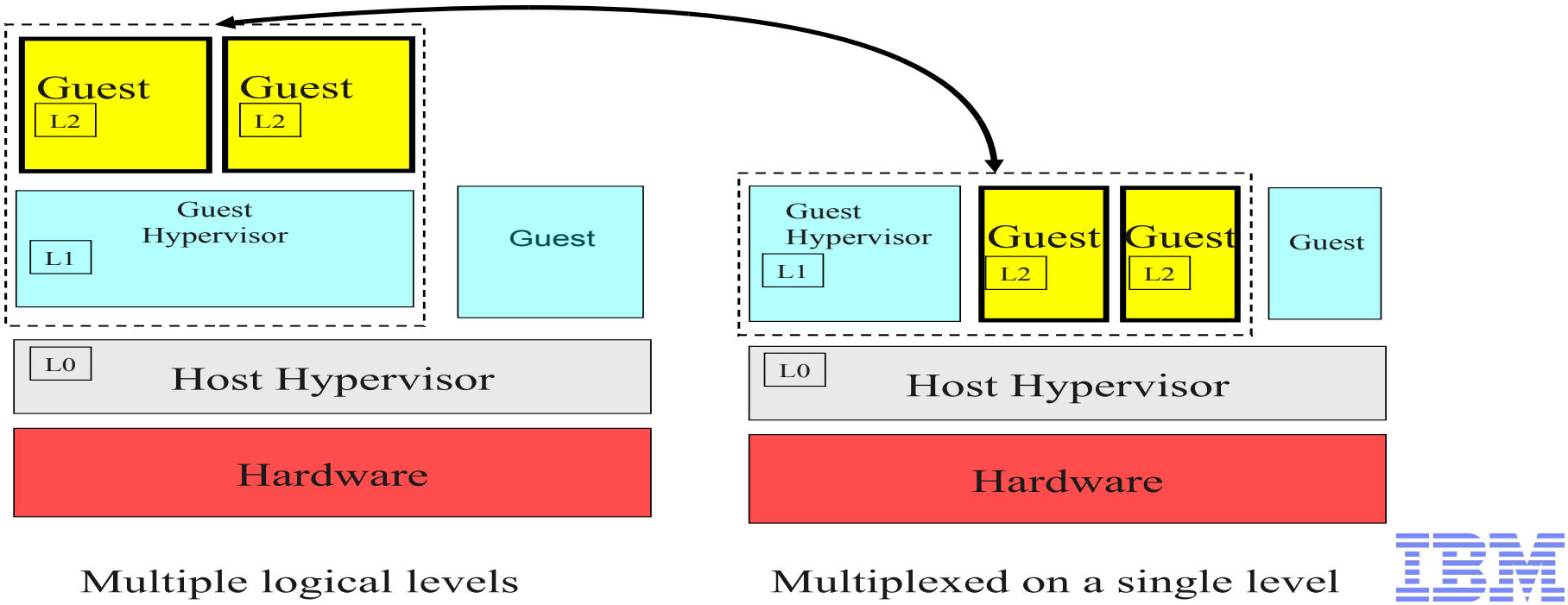
Theory of nested CPU virtualization

- 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: 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)



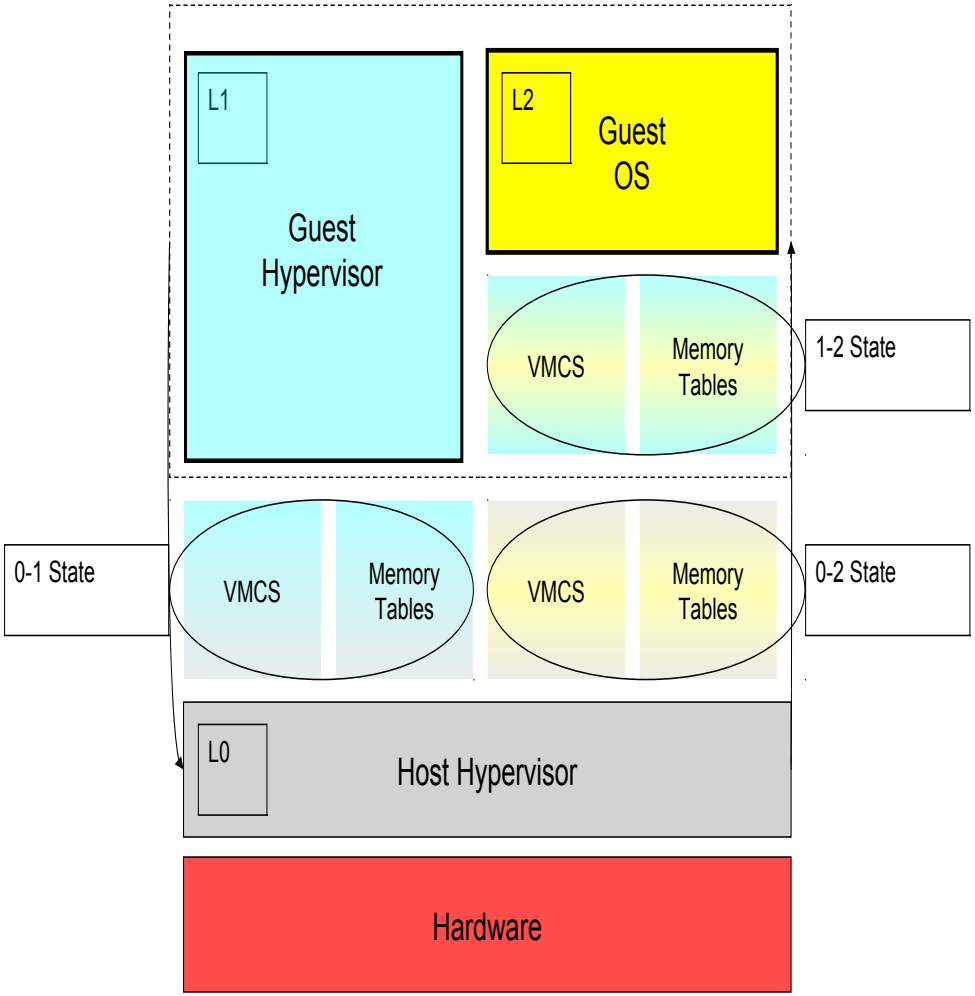
Theory of nested CPU virtualization

- 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: 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)



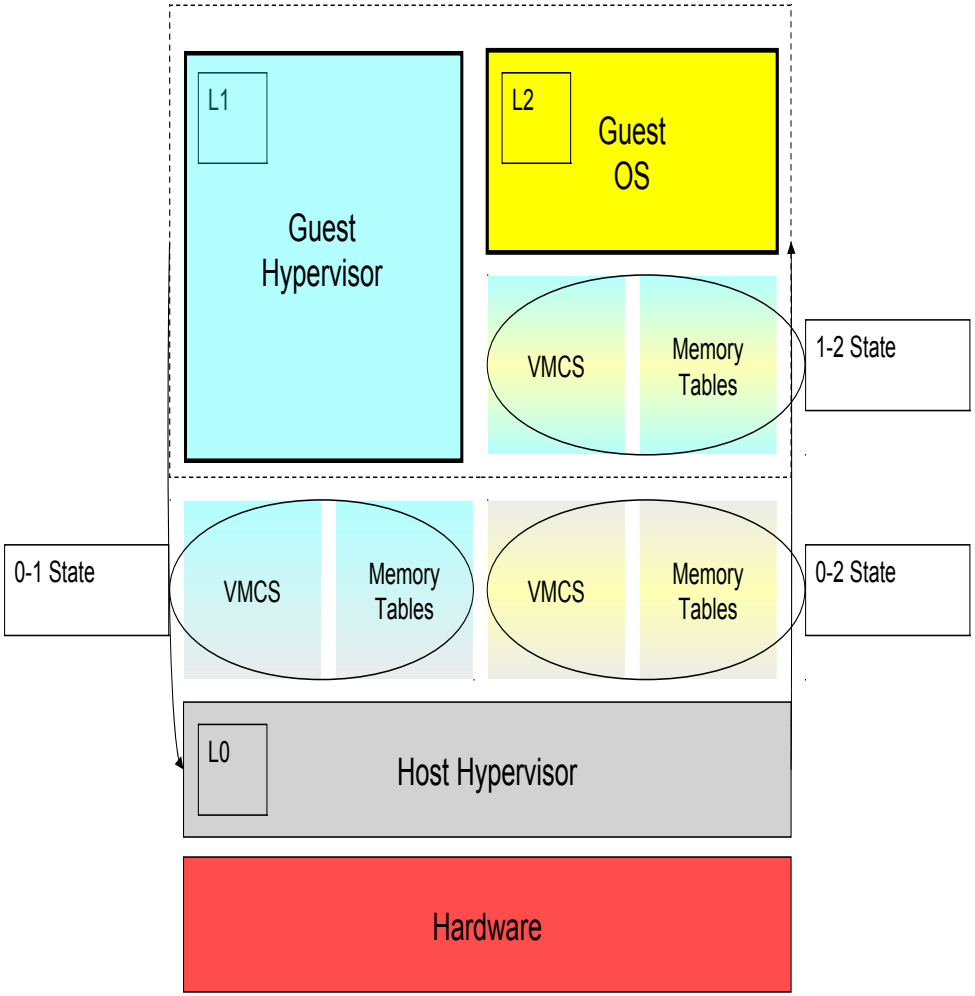
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



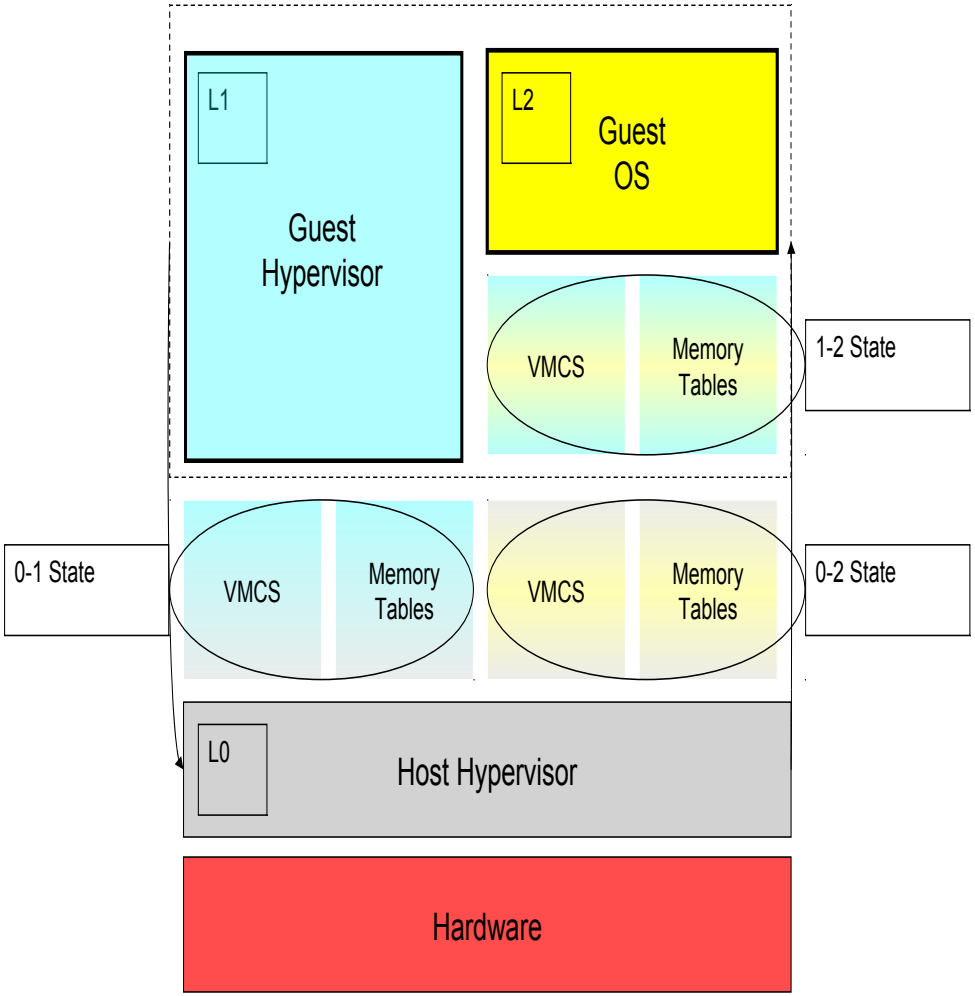
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



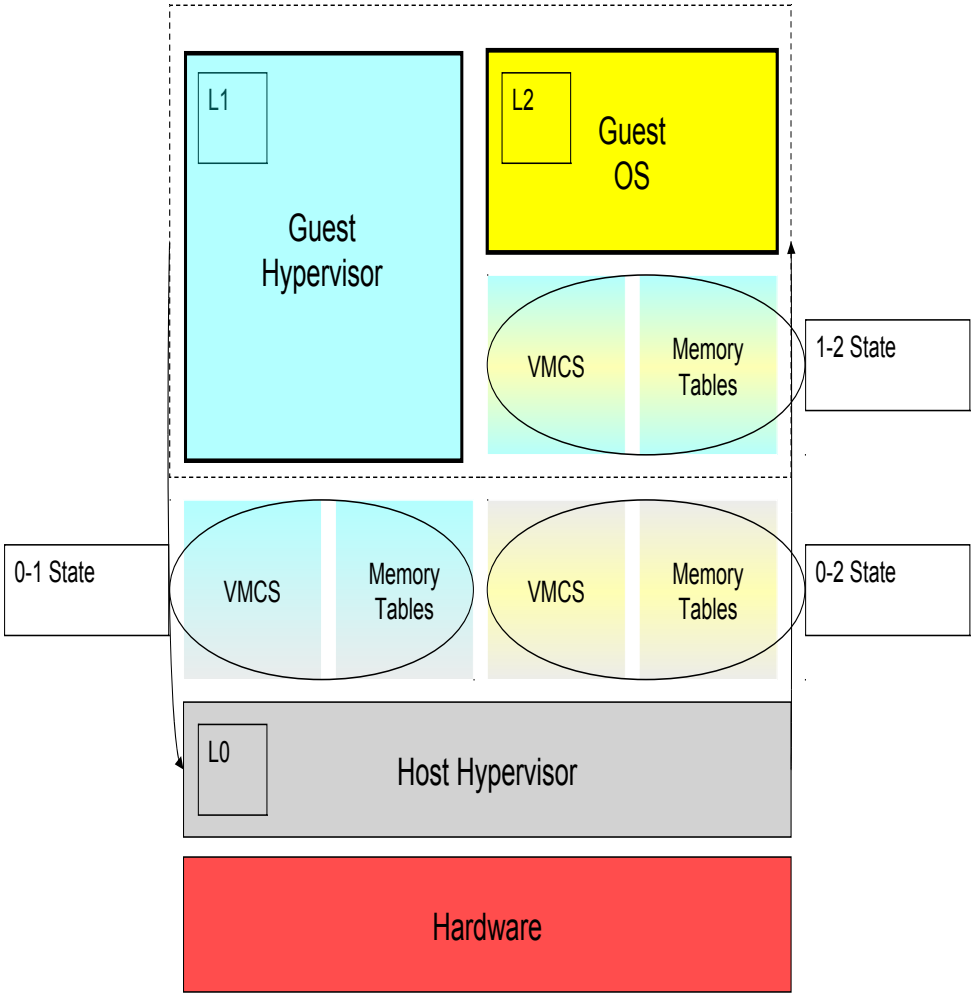
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



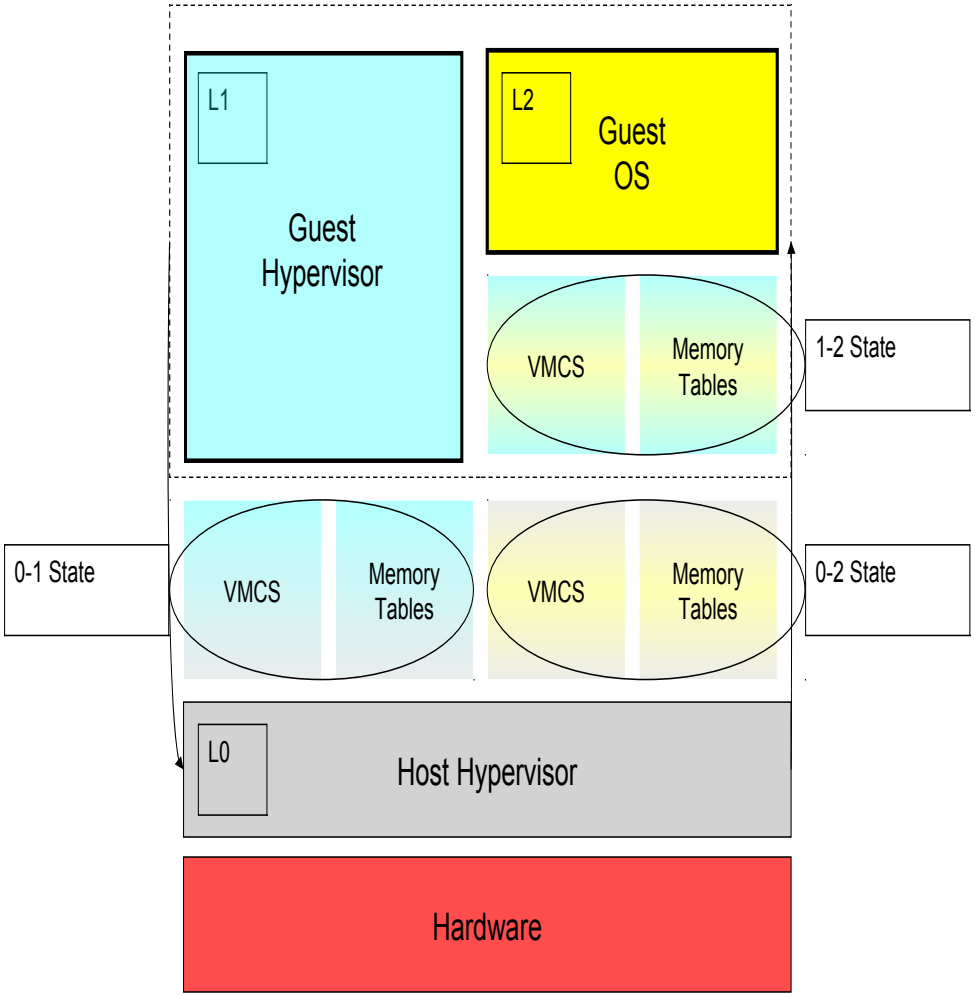
Nested VMX virtualization: flow

- L_0 runs L_1 with $VMCS_{0\to1}$
- L_1 prepares $VMCS_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges VMCS's: $VMCS_{0\to1}$ merged with $VMCS_{1\to2}$ is $VMCS_{0\to2}$
- 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



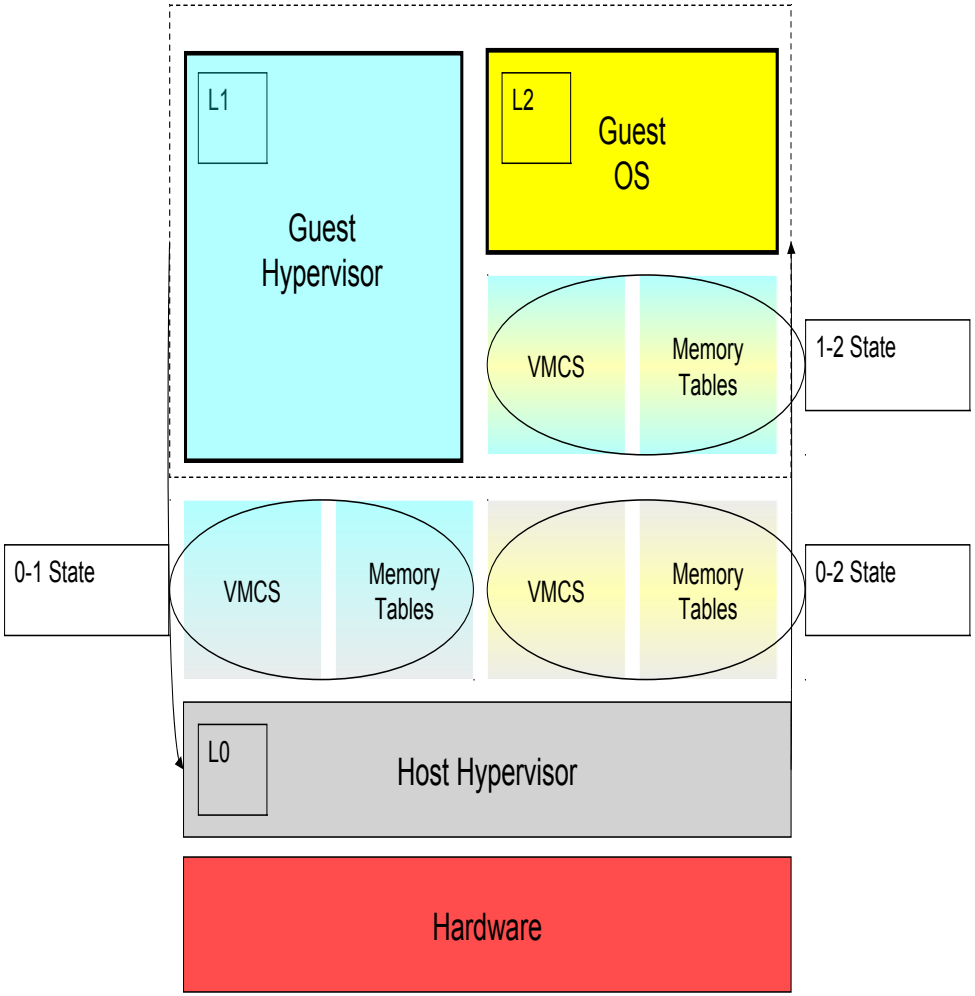
Nested VMX virtualization: flow

- L_0 runs L_1 with $VMCS_{0\to1}$
- L_1 prepares $VMCS_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges VMCS's: $VMCS_{0\to1}$ merged with $VMCS_{1\to2}$ is $VMCS_{0\to2}$
- 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



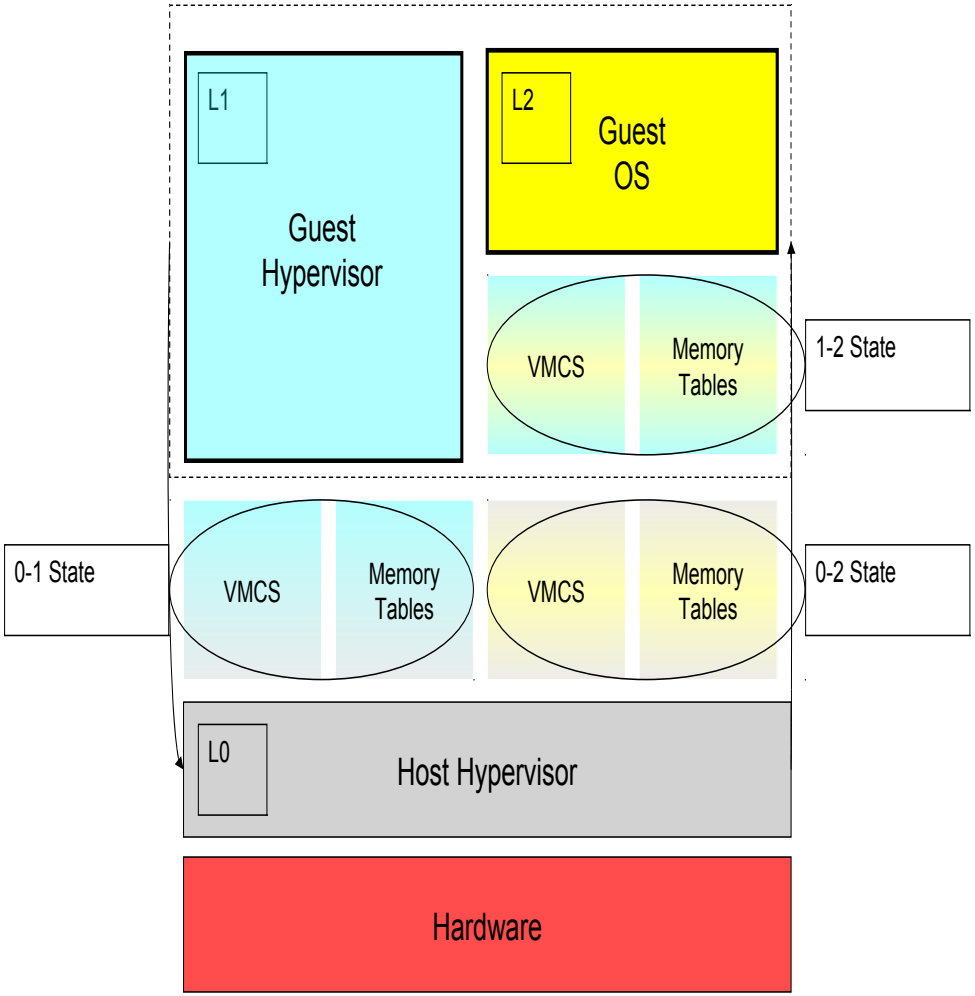
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



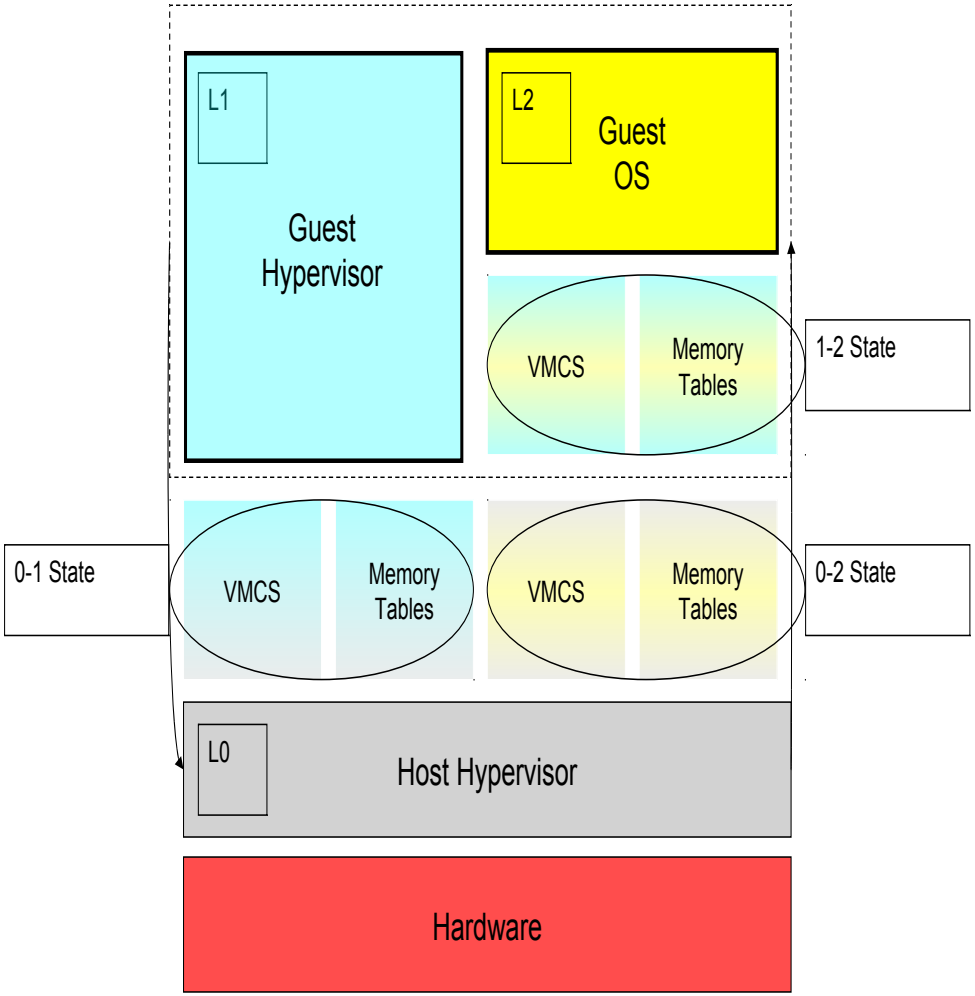
Nested VMX virtualization: flow

- L_0 runs L_1 with $VMCS_{0\to1}$
- L_1 prepares $VMCS_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges VMCS's: $VMCS_{0\to1}$ merged with $VMCS_{1\to2}$ is $VMCS_{0\to2}$
- 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



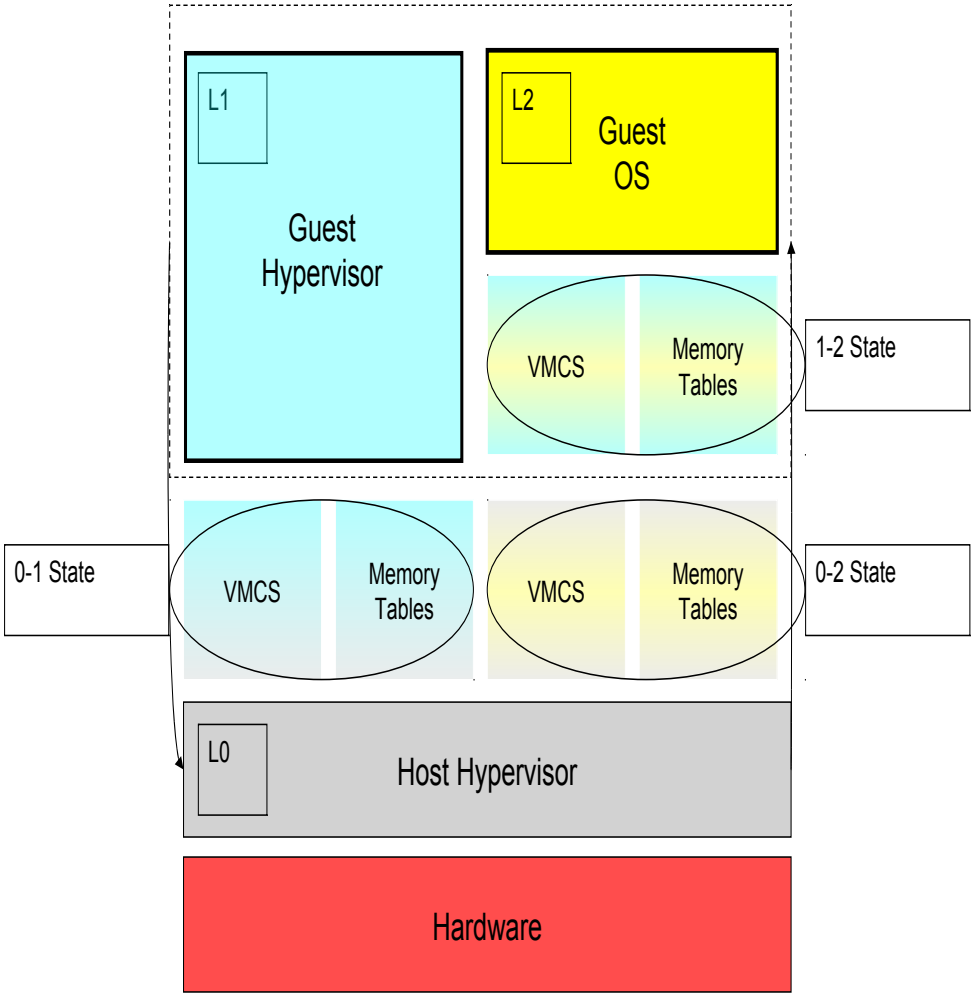
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



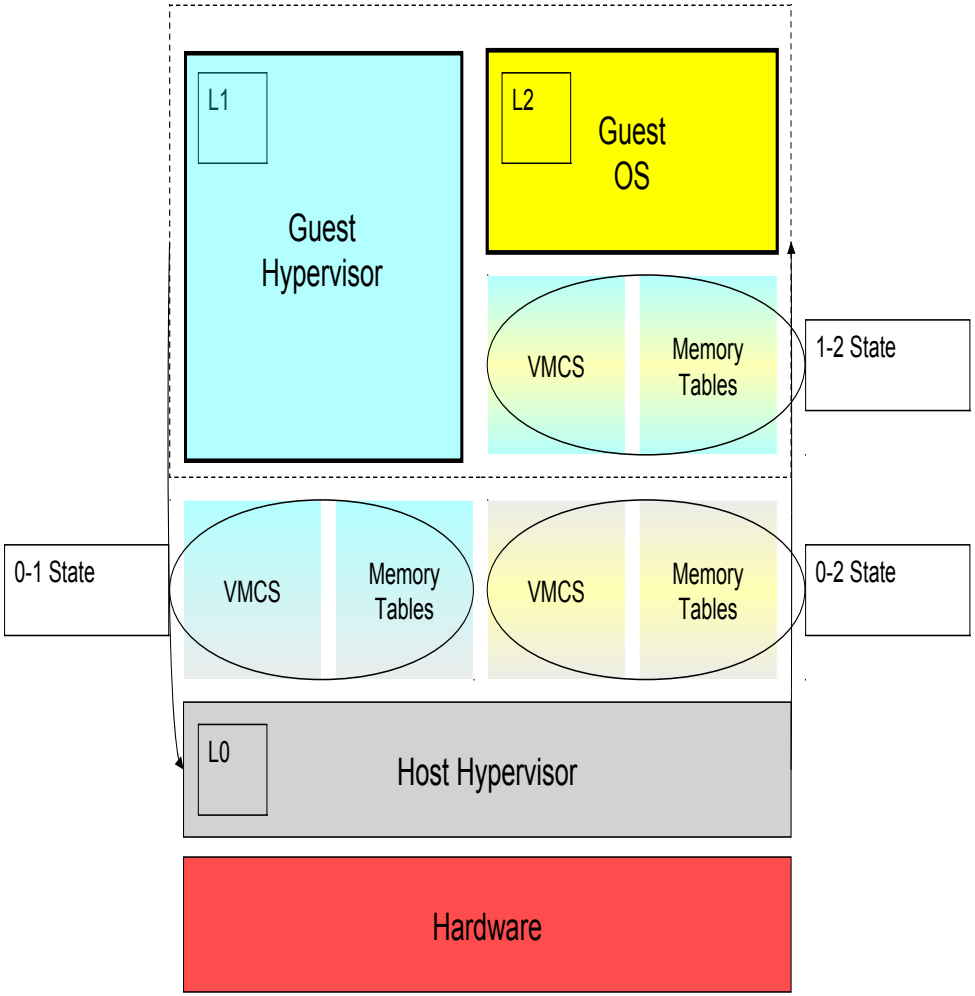
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



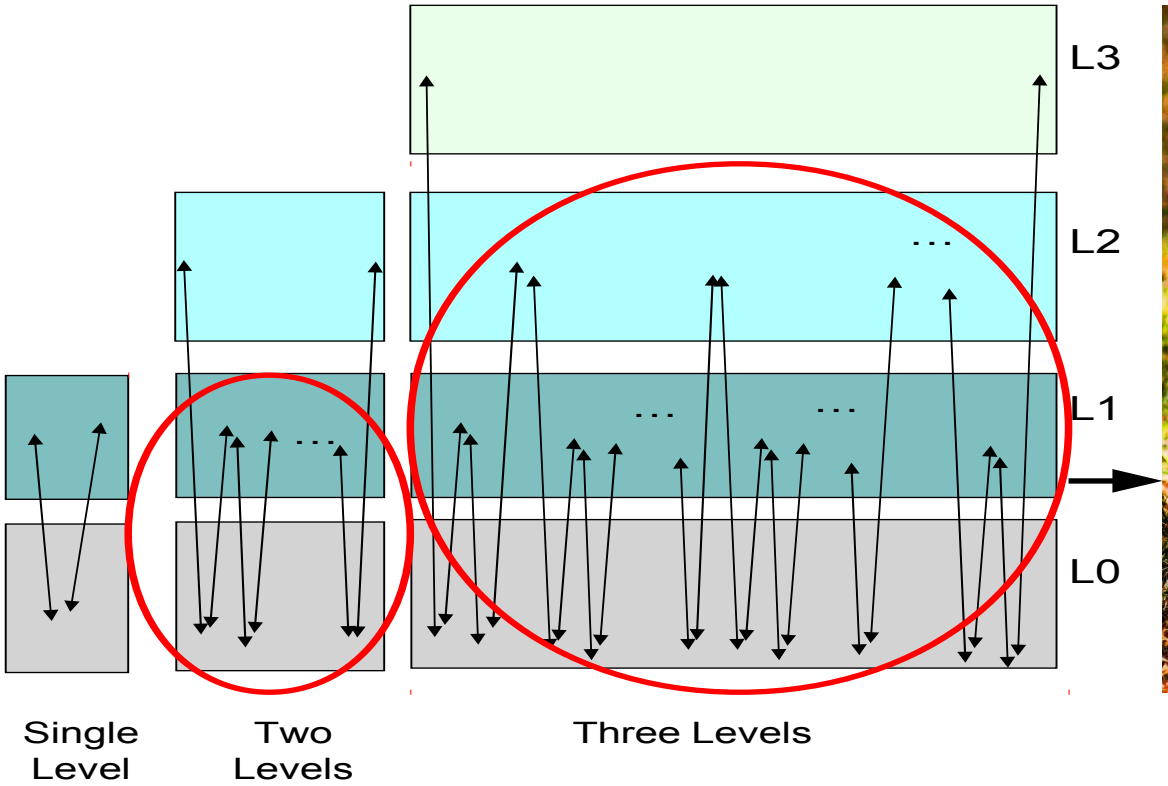
Nested VMX virtualization: flow

- L_0 runs L_1 with $\text{VMCS}_{0\to1}$
- L_1 prepares $\text{VMCS}_{1\to2}$ and executes `vmlaunch`
- `vmlaunch` traps to L_0
- L_0 merges 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
- ...
- eventually, L_0 resumes L_2
- repeat



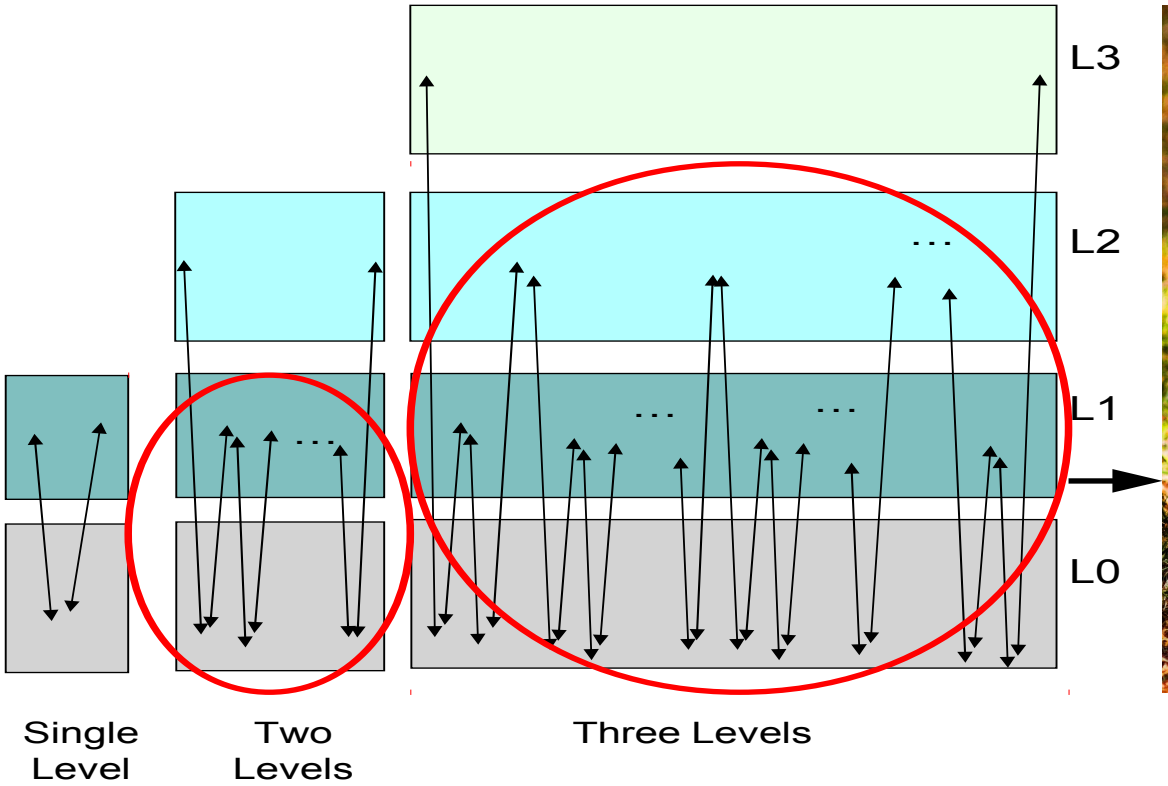
Exit multiplication makes angry turtle angry

- 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



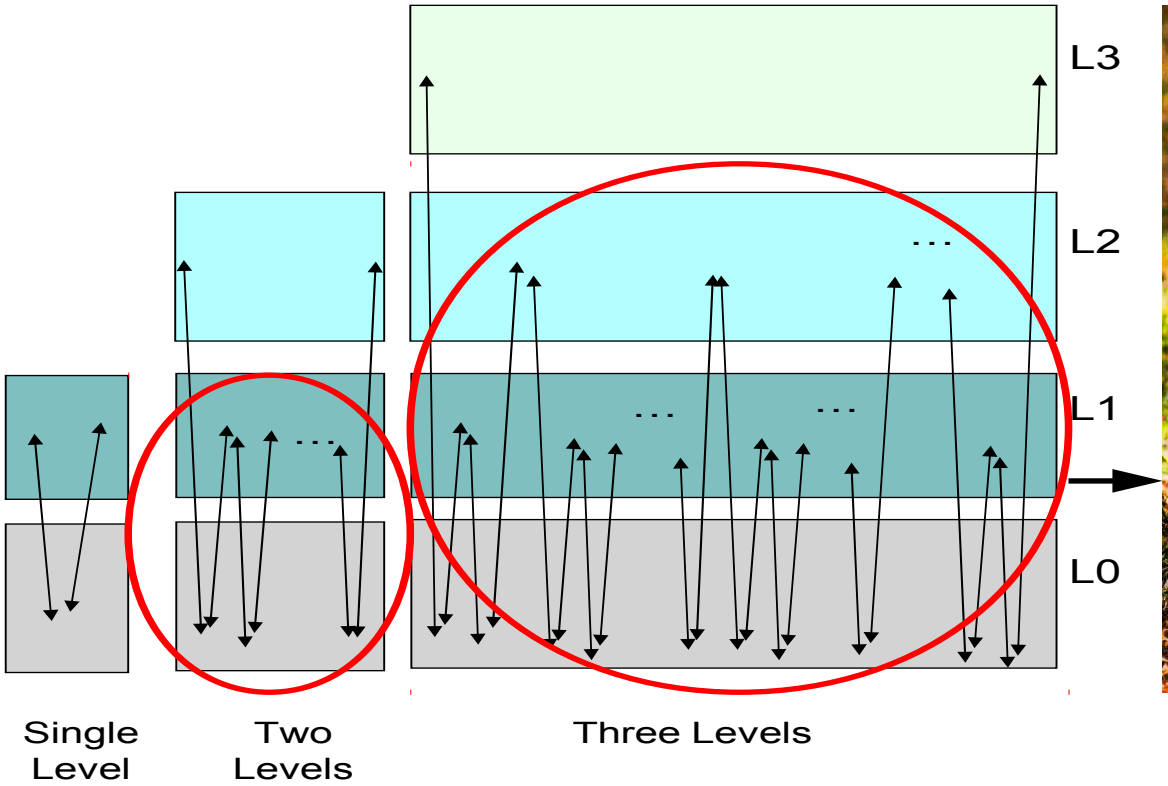
Exit multiplication makes angry turtle angry

- 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



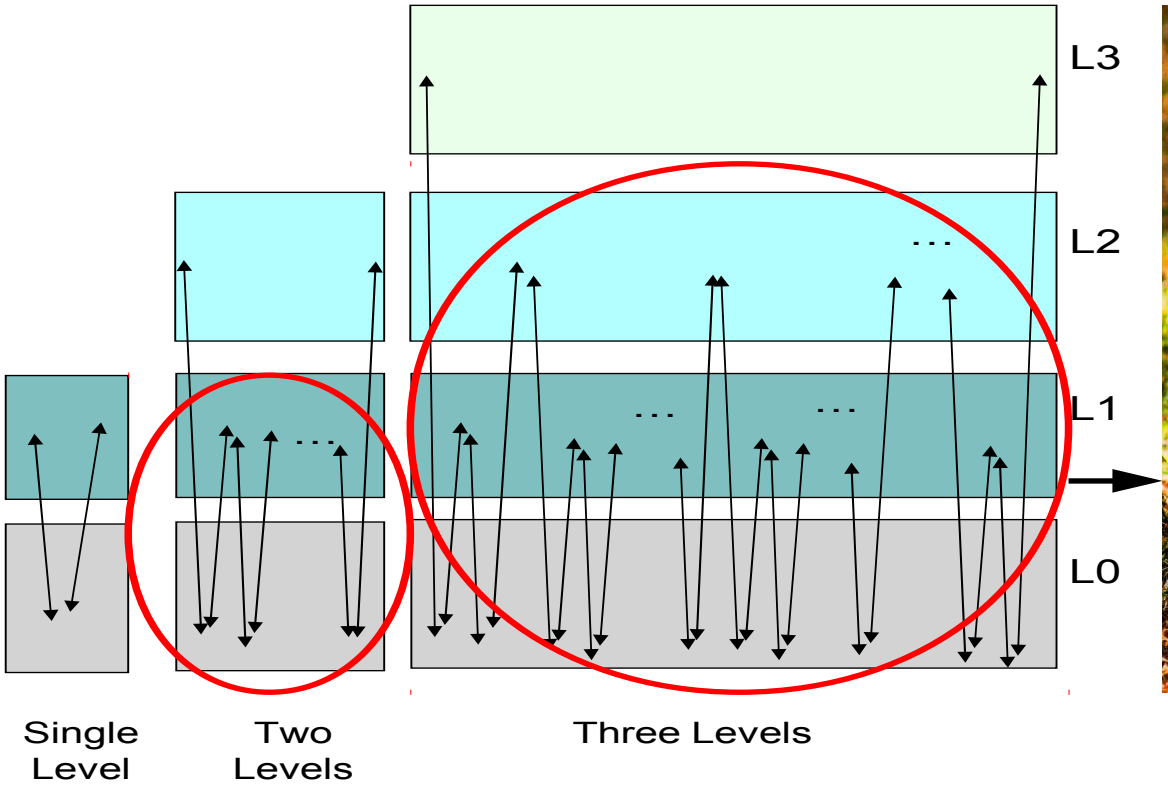
Exit multiplication makes angry turtle angry

- 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 ⇒ only needs one logical translation, (virtual → physical)
- Virtualization ⇒ needs two logical translations
  1. Guest page table: (guest virt → guest phys)
  2. Host page table: (guest phys → 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 ⇒ only needs one logical translation, (virtual → physical)
- Virtualization ⇒ needs two logical translations
  1. Guest page table: (guest virt → guest phys)
  2. Host page table: (guest phys → 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 ⇒ only needs one logical translation, (virtual → physical)
- Virtualization ⇒ needs two logical translations
  1. Guest page table: (guest virt → guest phys)
  2. Host page table: (guest phys → host phys)
- . . . but MMU only knows to walk a single table!

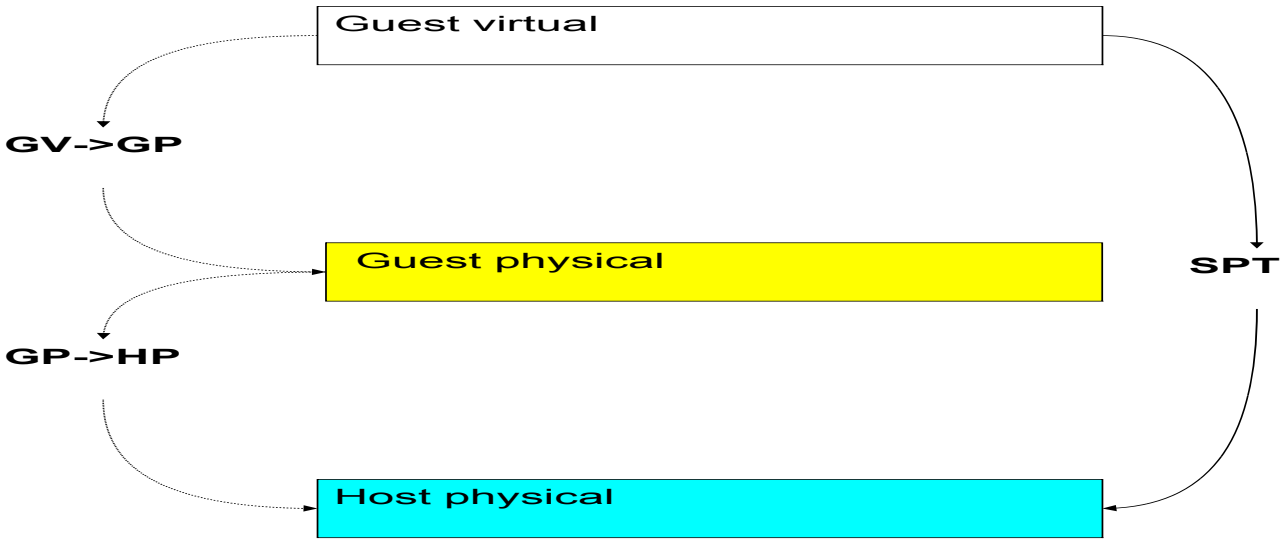
Introduction to x86 MMU virtualization

- 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

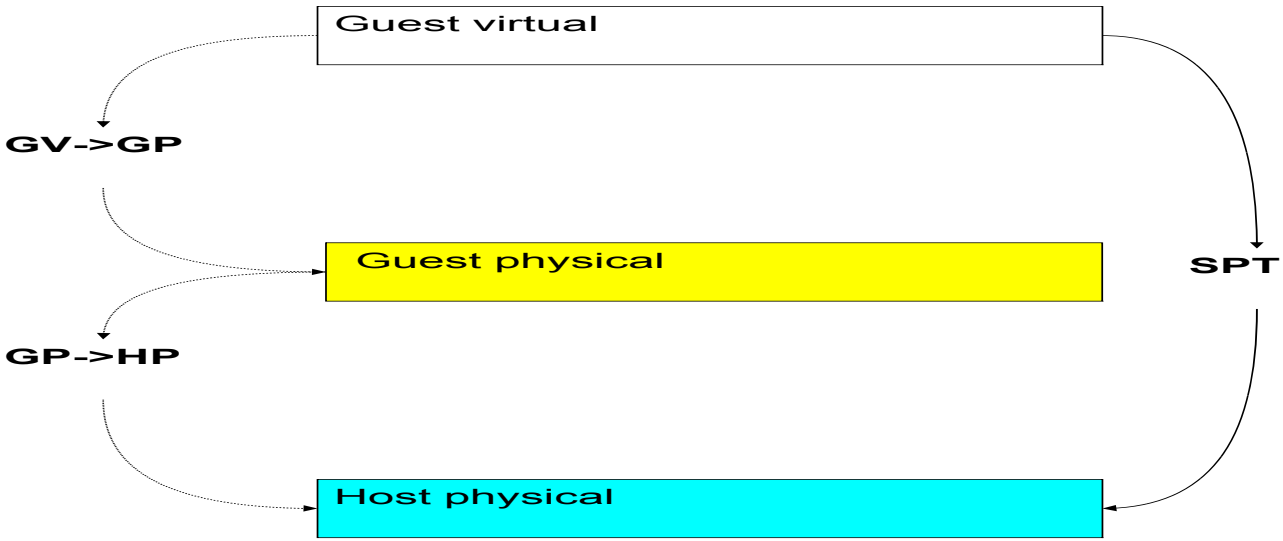
- x86 does page table walks in hardware
- MMU has one currently active hardware page table
- Bare metal $\Rightarrow$ only needs one logical translation, (virtual $\rightarrow$ physical)
- Virtualization $\Rightarrow$ needs two logical translations
  1. Guest page table: (guest virt $\rightarrow$ guest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

Software MMU virtualization: shadow paging



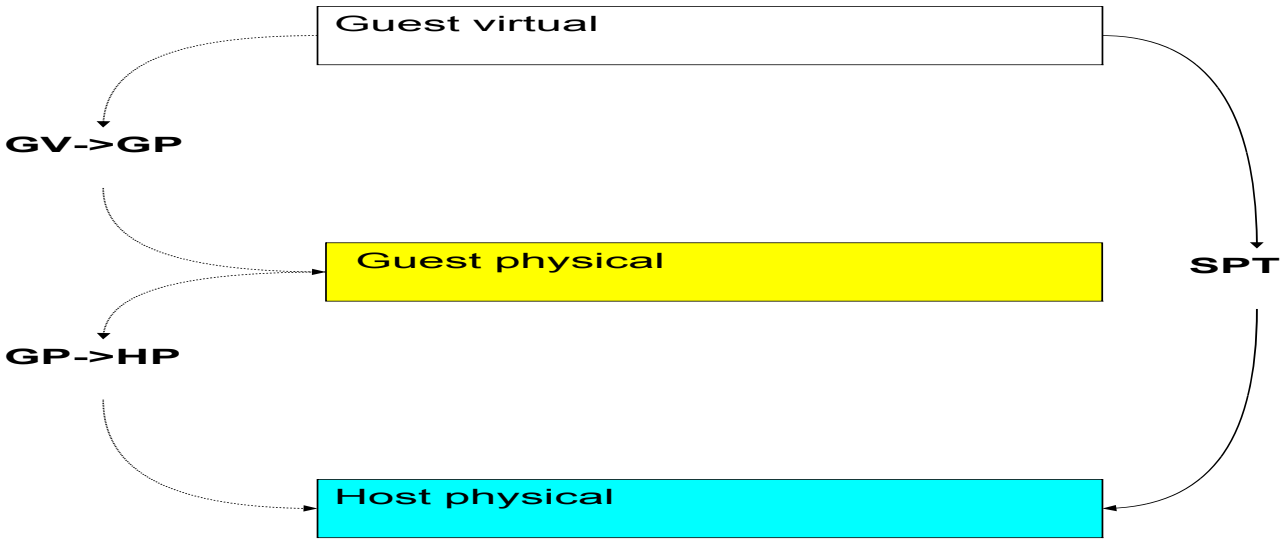
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Software MMU virtualization: shadow paging



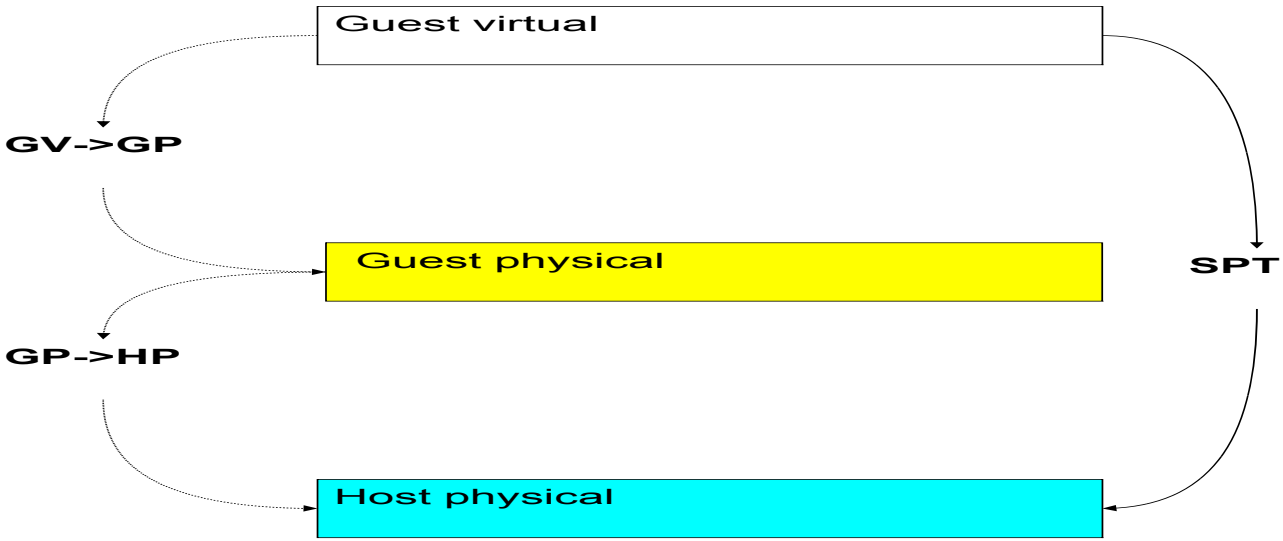
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Software MMU virtualization: shadow paging



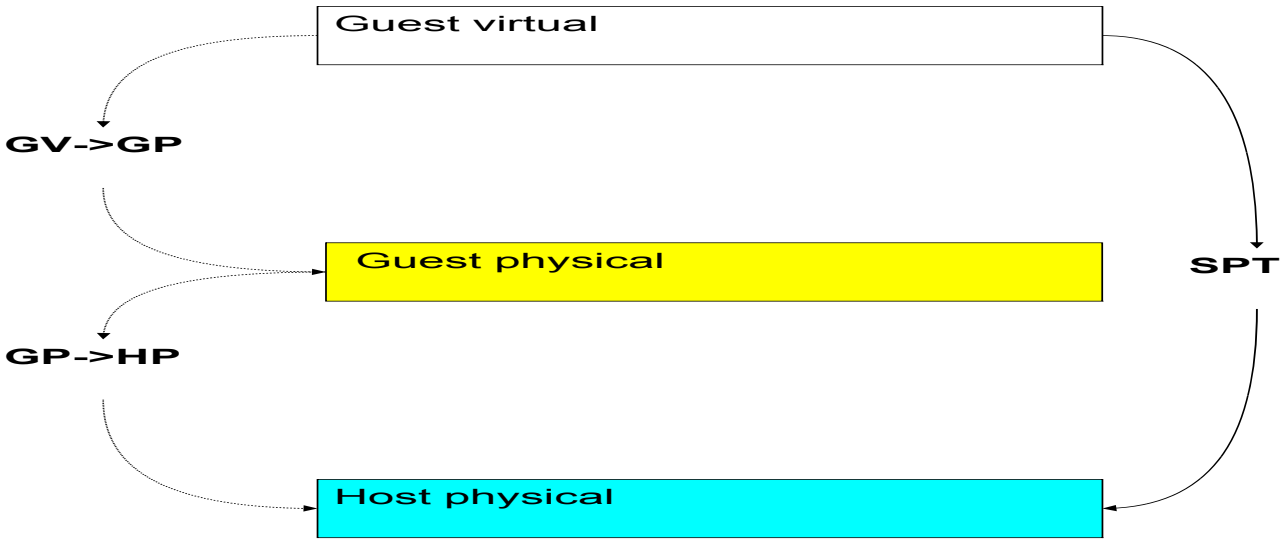
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Software MMU virtualization: shadow paging



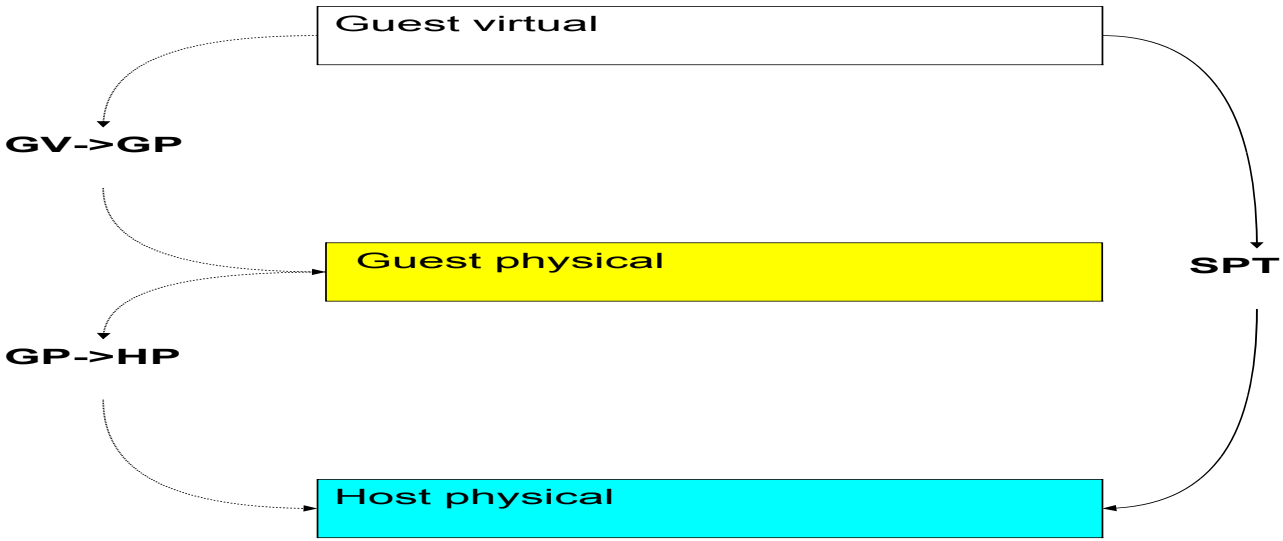
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Software MMU virtualization: shadow paging



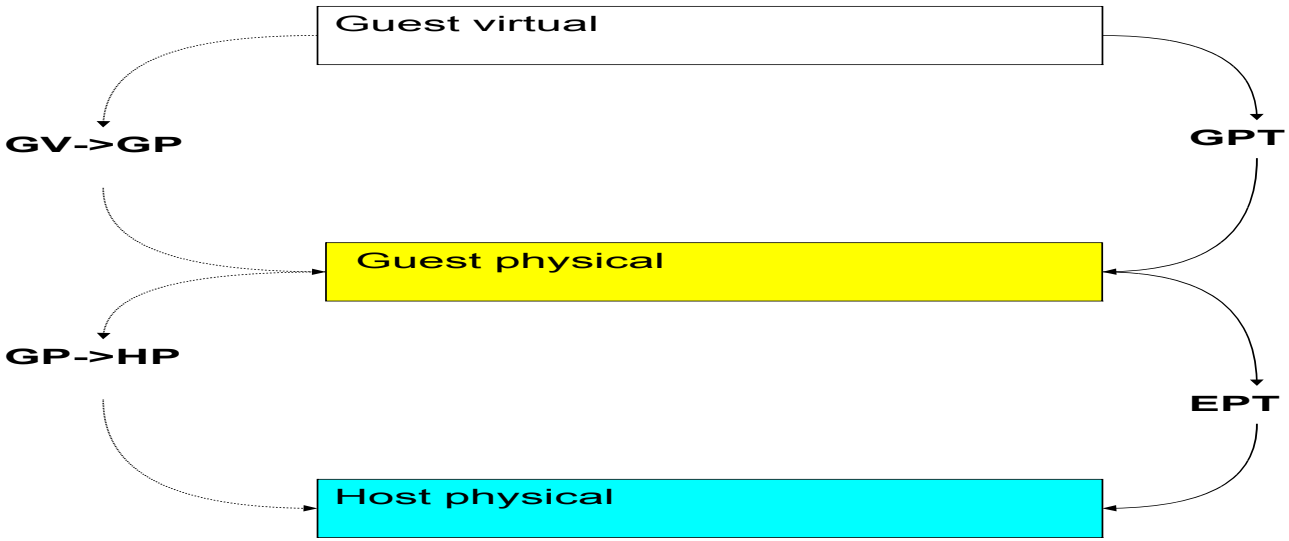
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Software MMU virtualization: shadow paging



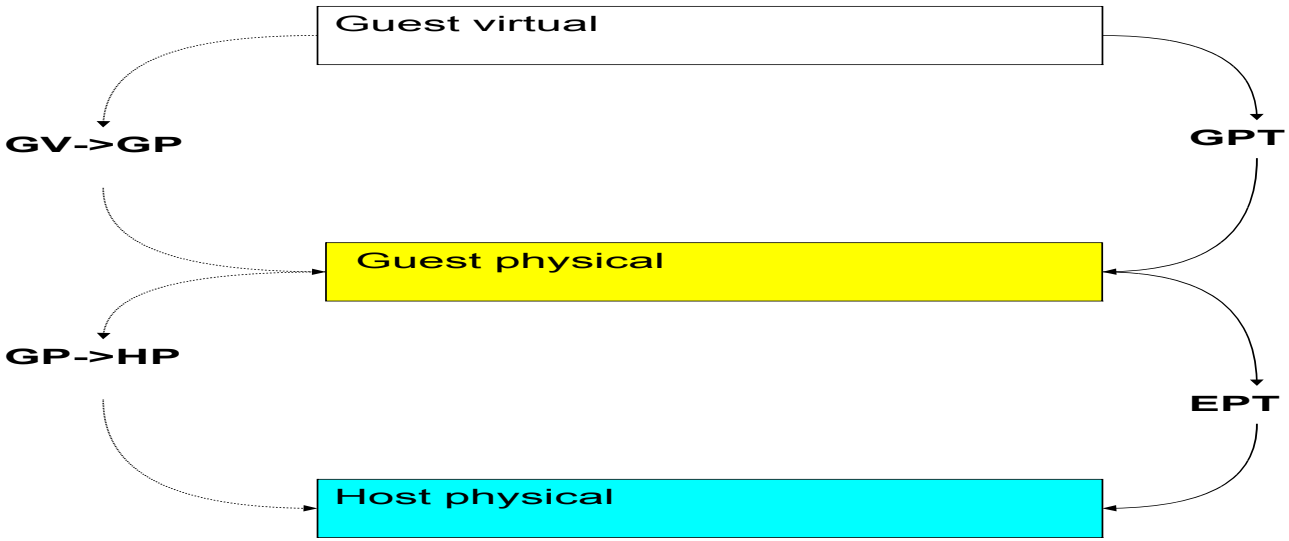
- Two logical translations compressed onto the shadow page table [DevineBugnion02]
- Unmodified guest OS updates its own table
- Hypervisor traps OS page table updates
- Hypervisor propagates updates to the hardware table
- MMU walks the table
- Problem: traps are expensive

Hardware MMU virtualization: Extended Page Tables



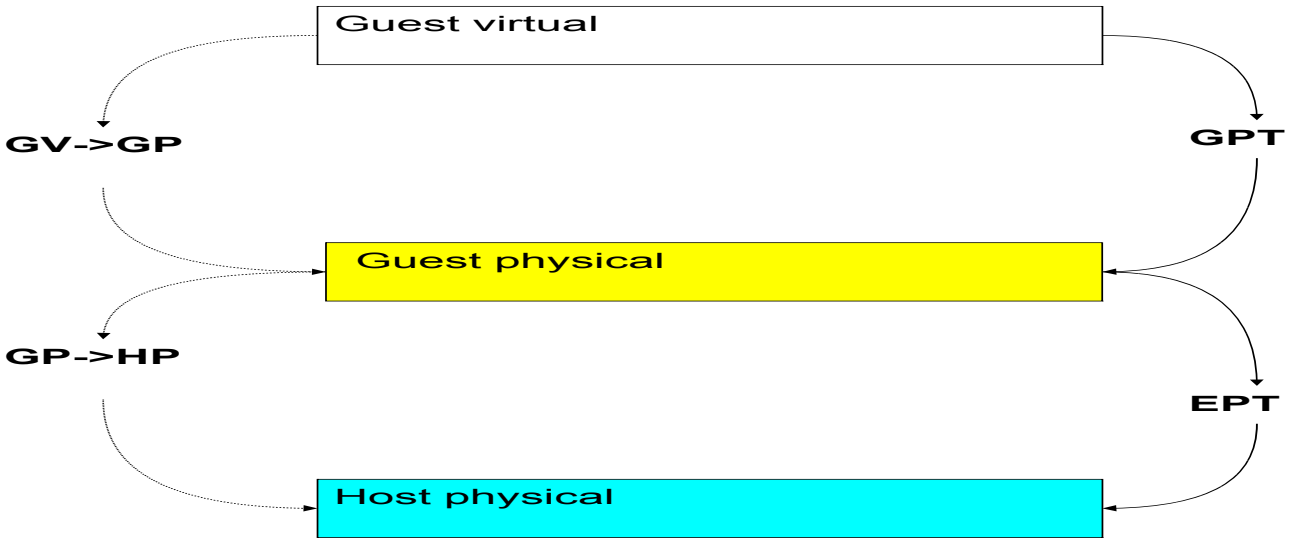
- Two-dimensional paging: guest owns GPT, hypervisor owns EPT [BhargavaSerebrin08]
- Unmodified guest OS updates GPT
- Hypervisor updates EPT table controlling (guest phys → host phys) translations
- MMU walks both tables

Hardware MMU virtualization: Extended Page Tables



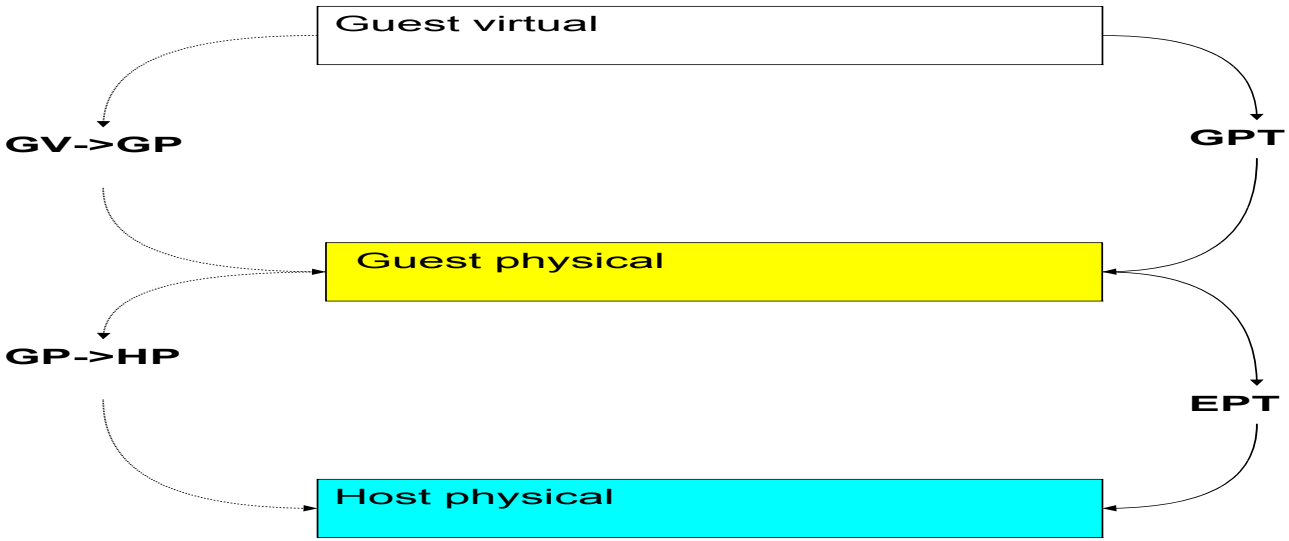
- Two-dimensional paging: guest owns GPT, hypervisor owns EPT [BhargavaSerebrin08]
- Unmodified guest OS updates GPT
- Hypervisor updates EPT table controlling (guest phys → host phys) translations
- MMU walks both tables

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

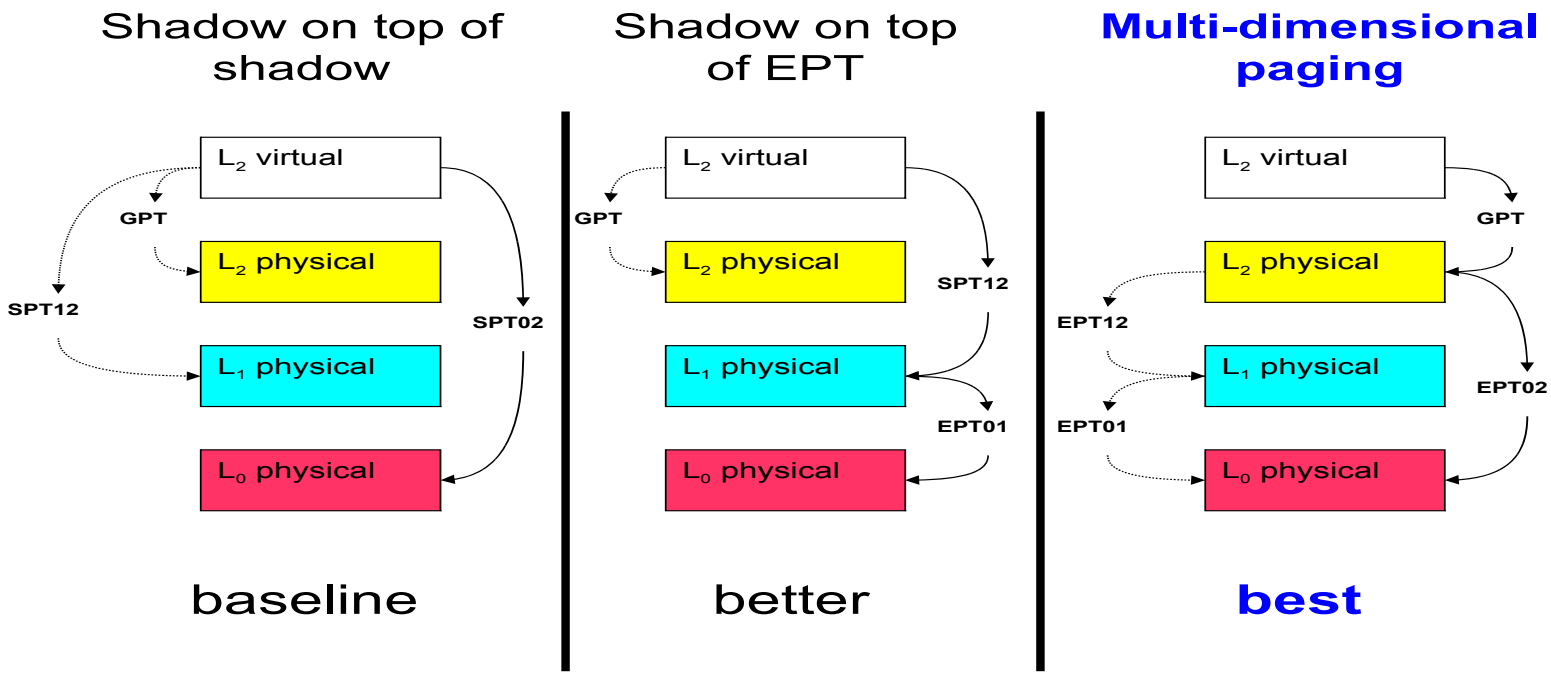
Hardware MMU virtualization: Extended Page Tables



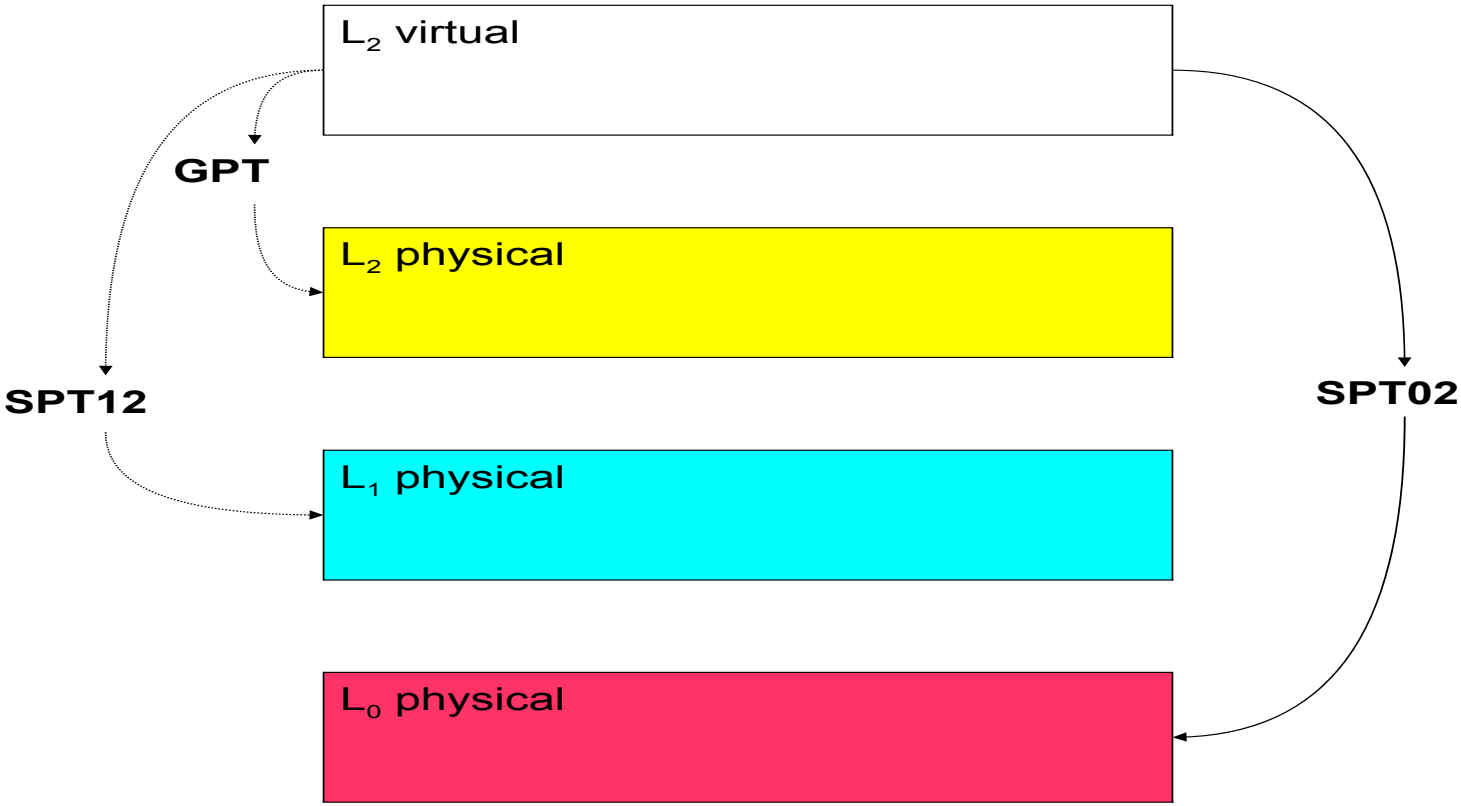
- Two-dimensional paging: guest owns GPT, hypervisor owns EPT [BhargavaSerebrin08]
- Unmodified guest OS updates GPT
- Hypervisor updates EPT table controlling (guest phys → host phys) translations
- MMU walks both tables

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

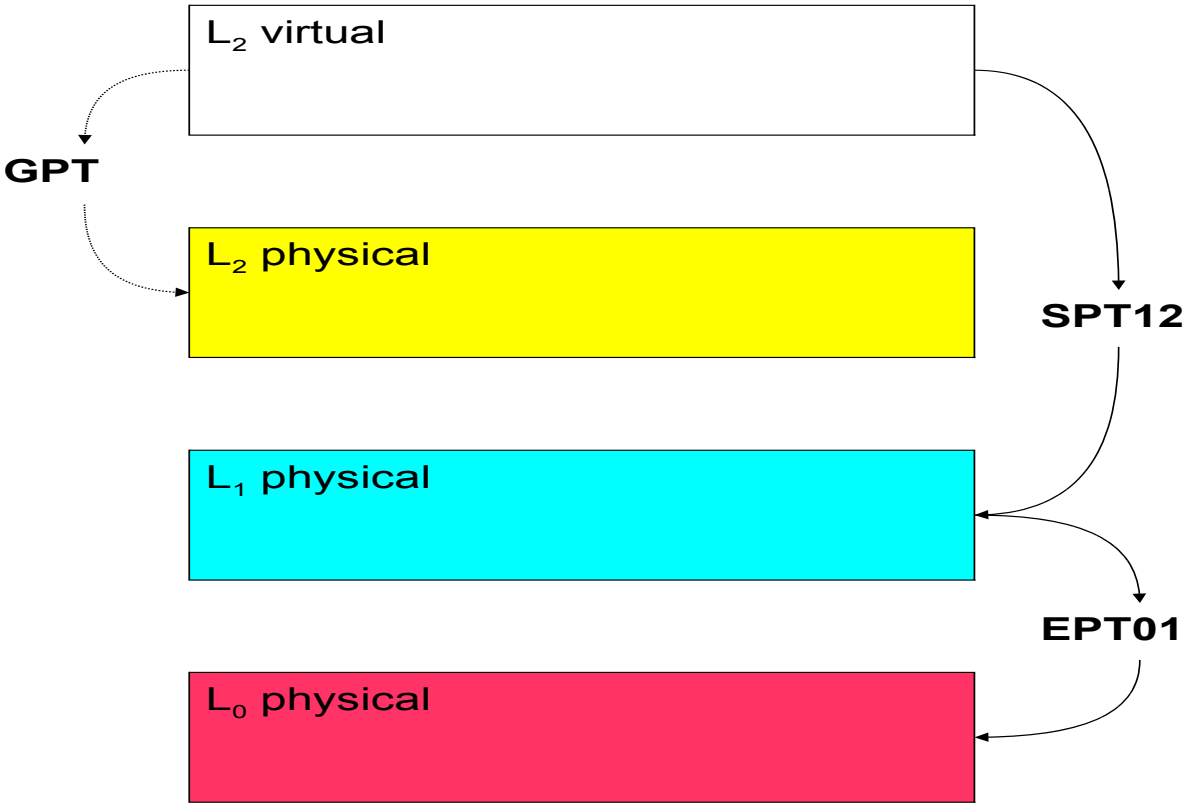


Baseline: shadow-on-shadow



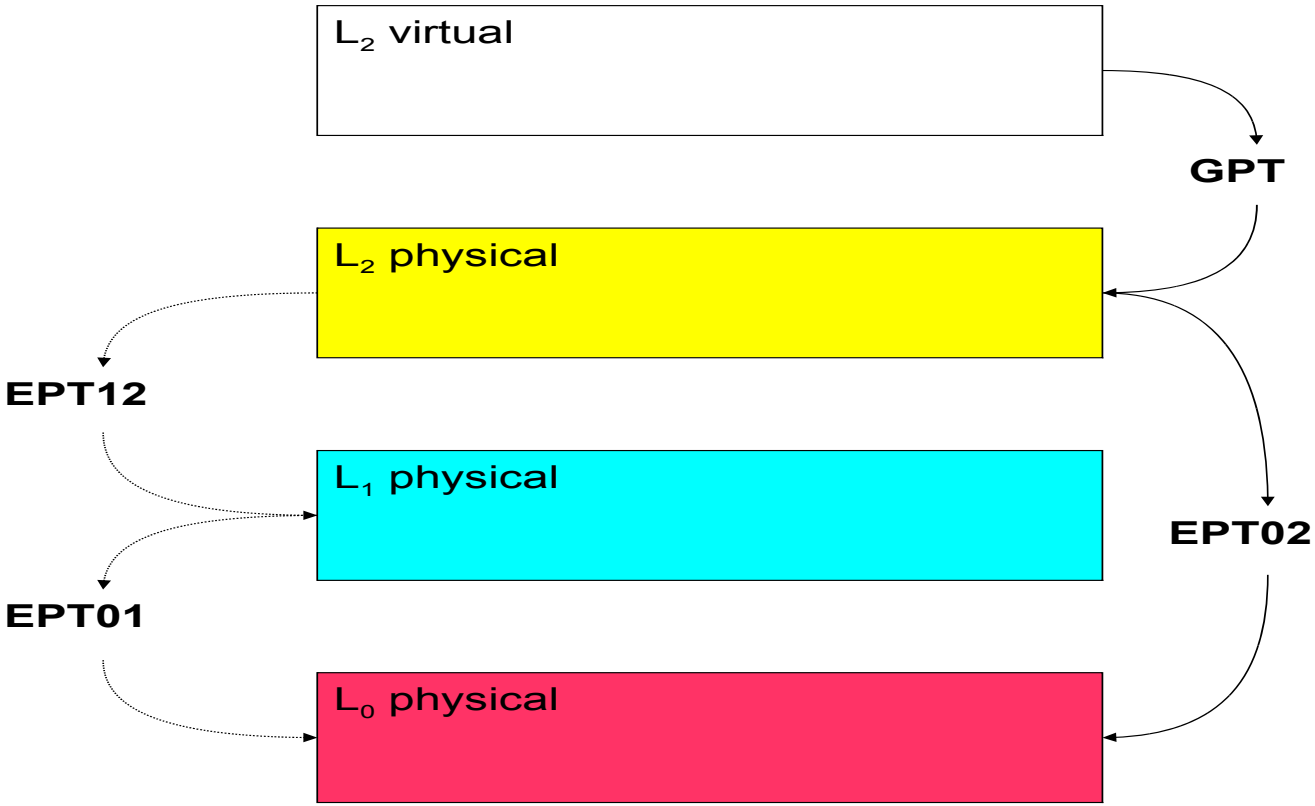
- Assume no EPT table; all hypervisors use shadow paging
- Useful for old machines and as a baseline
- Maintaining shadow page tables is expensive
- Compress: three logical translations $\Rightarrow$ one table in hardware

Better: shadow-on-EPT



- Instead of one hardware table we have two
- Compress: three logical translations $\Rightarrow$ two in hardware
- Simple approach: 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 $\Rightarrow$ two in hardware
- L_0 emulates EPT for L_1
- L_0 uses $EPT_{0\rightarrow1}$ and $EPT_{1\rightarrow2}$ to construct $EPT_{0\rightarrow2}$
- 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

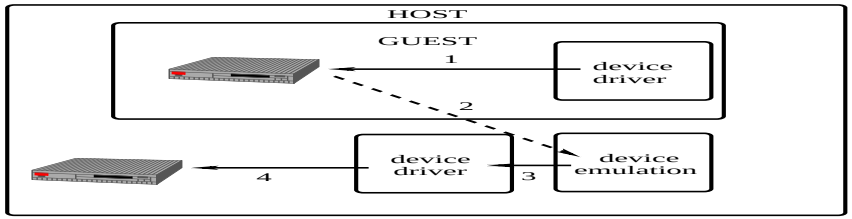
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA

Introduction to I/O virtualization

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

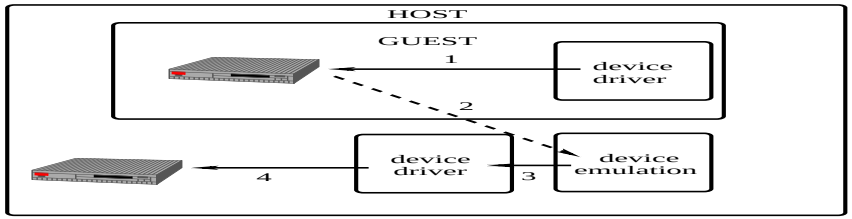
Introduction to I/O virtualization

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

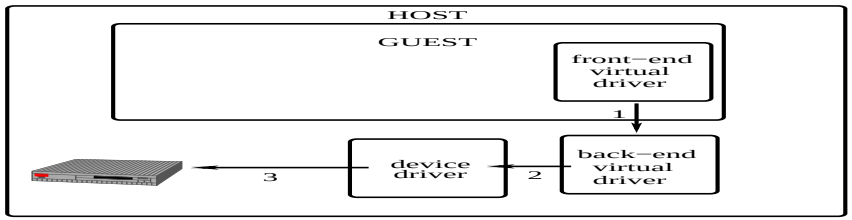


Introduction to I/O virtualization

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

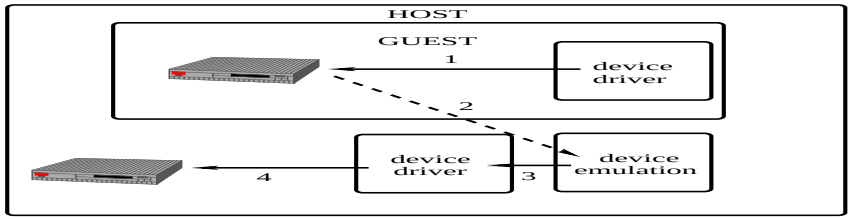


- Para-virtualized drivers [Barham03, Russell08]

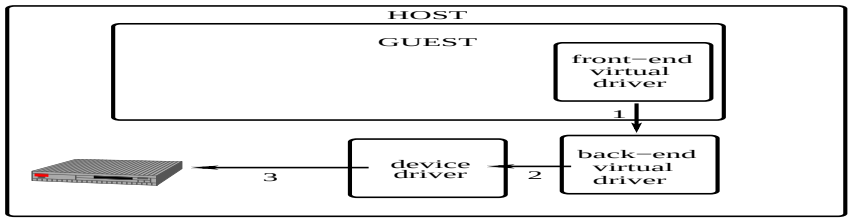


Introduction to I/O virtualization

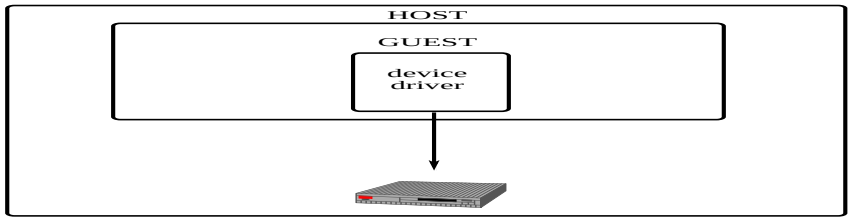
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]

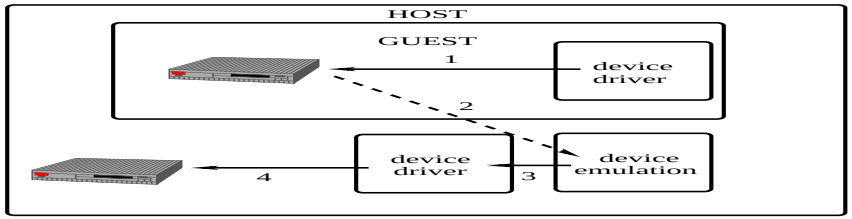


- Direct device assignment [Levasseur04,Yassour08]

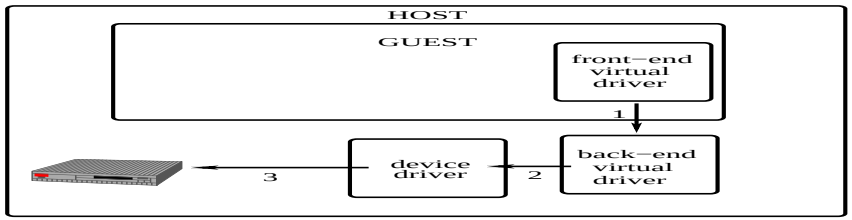


Introduction to I/O virtualization

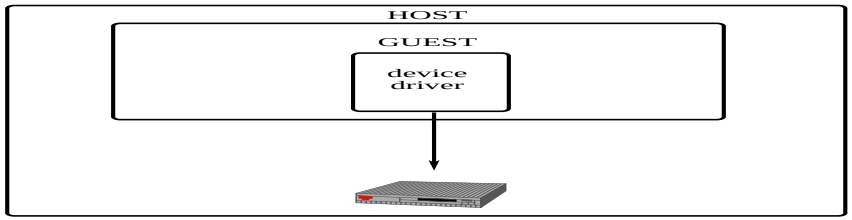
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]



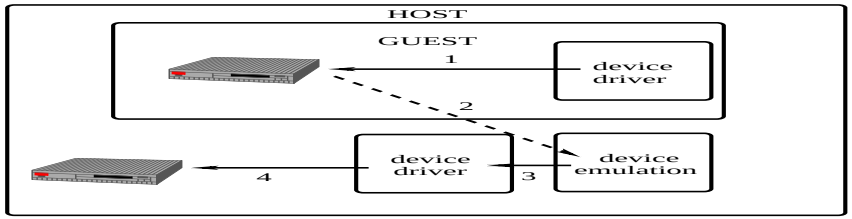
- Direct device assignment [Levasseur04,Yassour08]



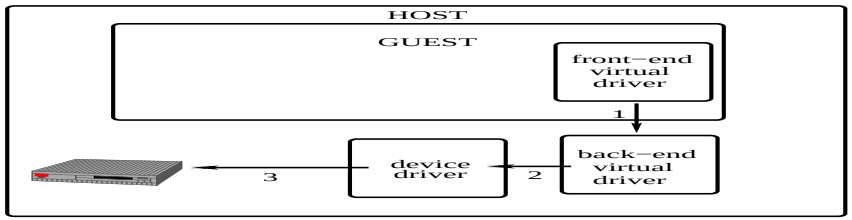
  - Direct assignment best performing option

Introduction to I/O virtualization

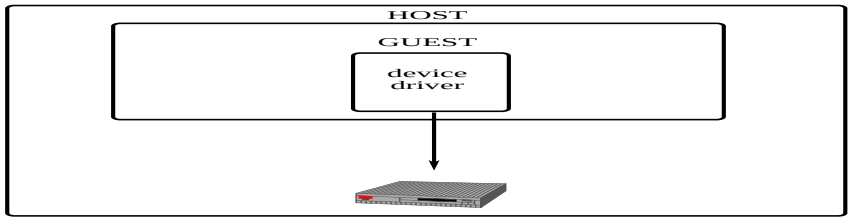
- From the hypervisor's perspective, what is I/O?
- (1) PIO (2) MMIO (3) DMA (4) interrupts
- Device emulation [Sugerman01]



- Para-virtualized drivers [Barham03, Russell08]



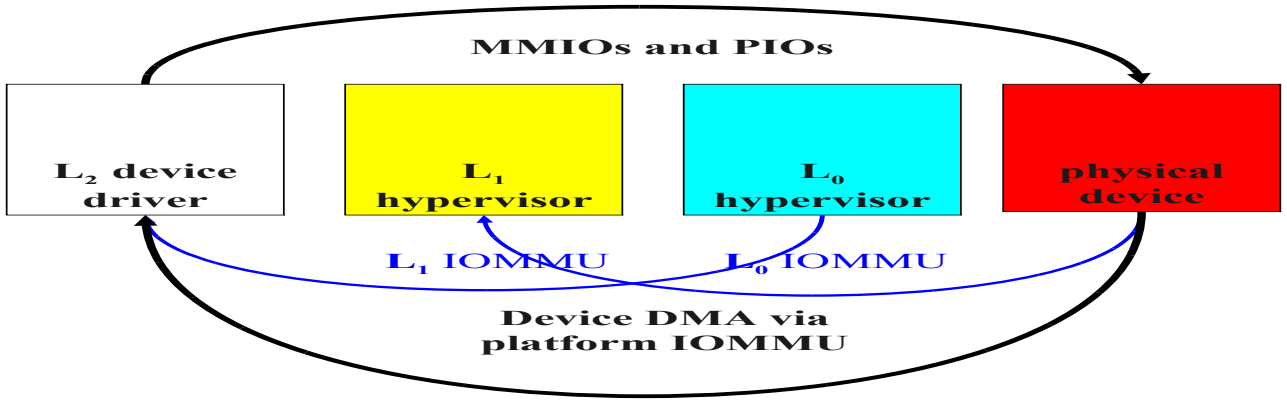
- Direct device assignment [Levasseur04,Yassour08]



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

Multi-level device assignment

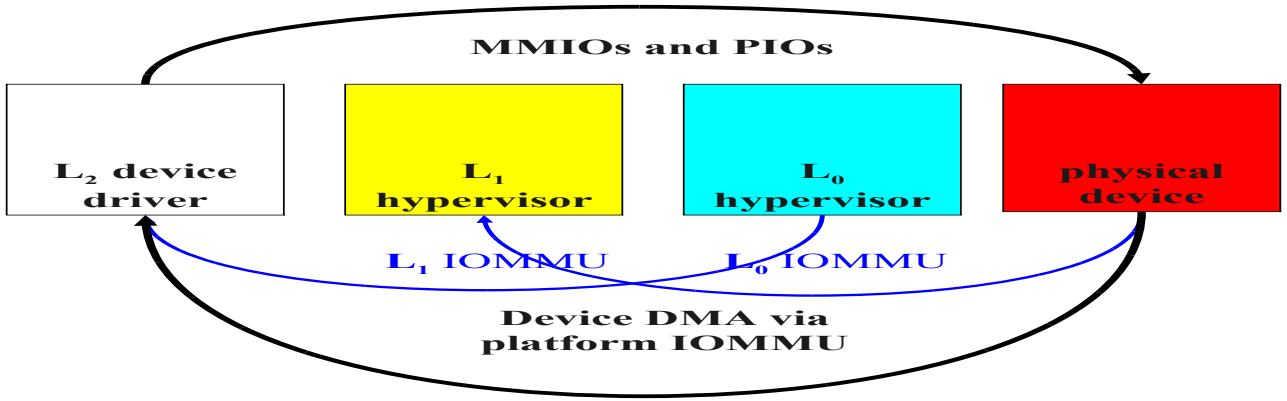
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- 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

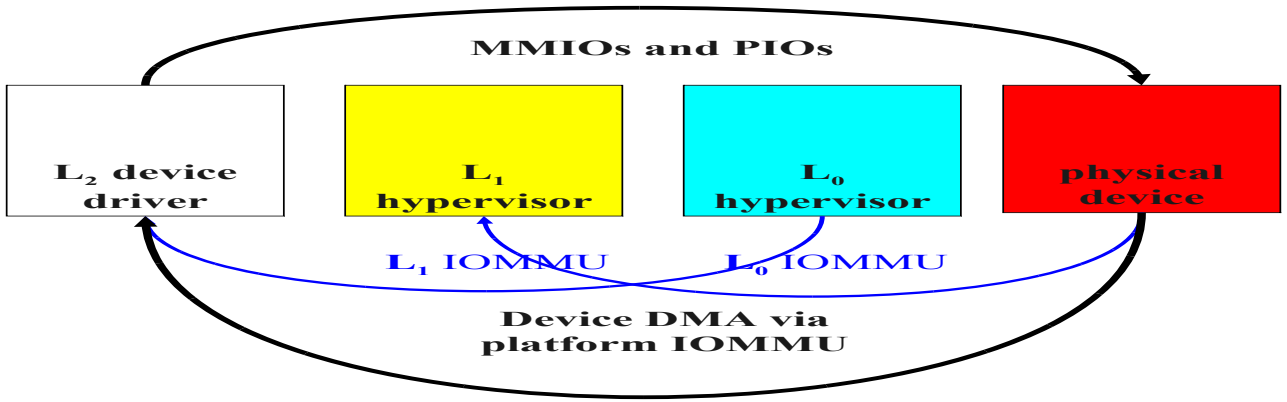
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- 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

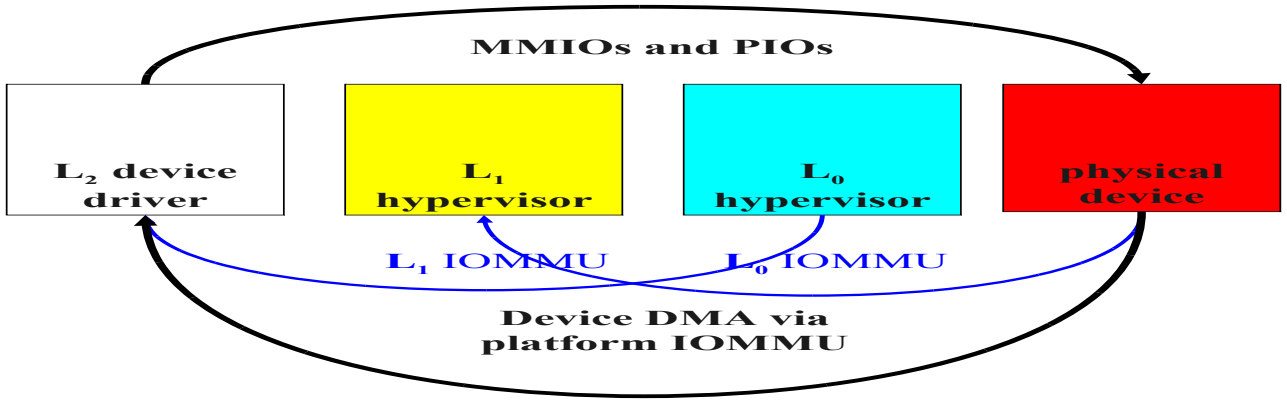
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- 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

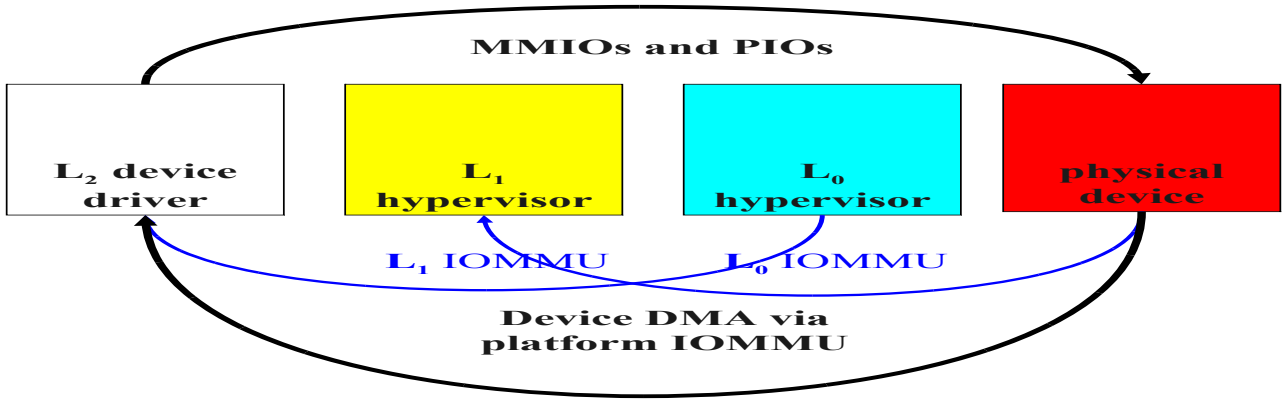
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- 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

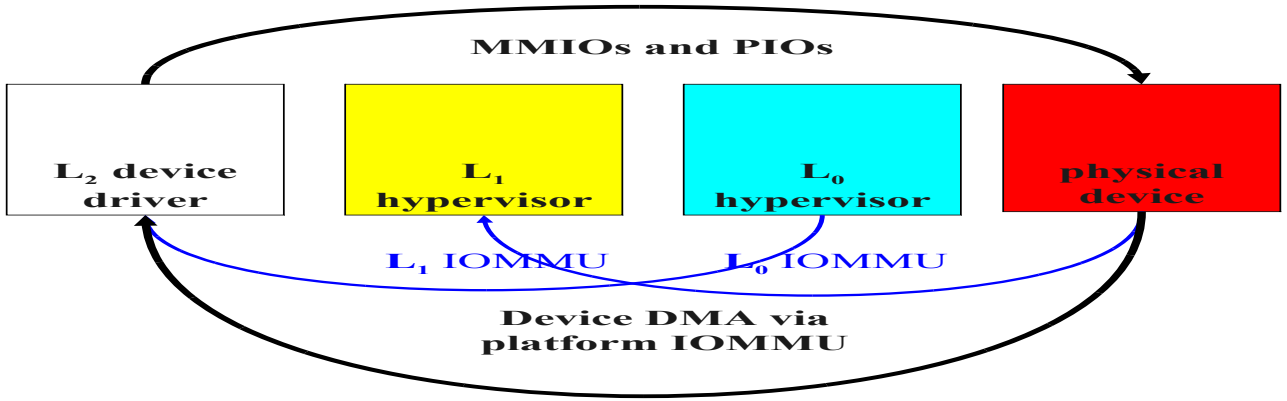
- With nested 3x3 options for I/O virtualization ($L_2 \Leftrightarrow L_1 \Leftrightarrow L_0$)
- Multi-level device assignment means giving an L_2 guest direct access to L_0's devices, safely bypassing both L_0 and L_1



- 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?

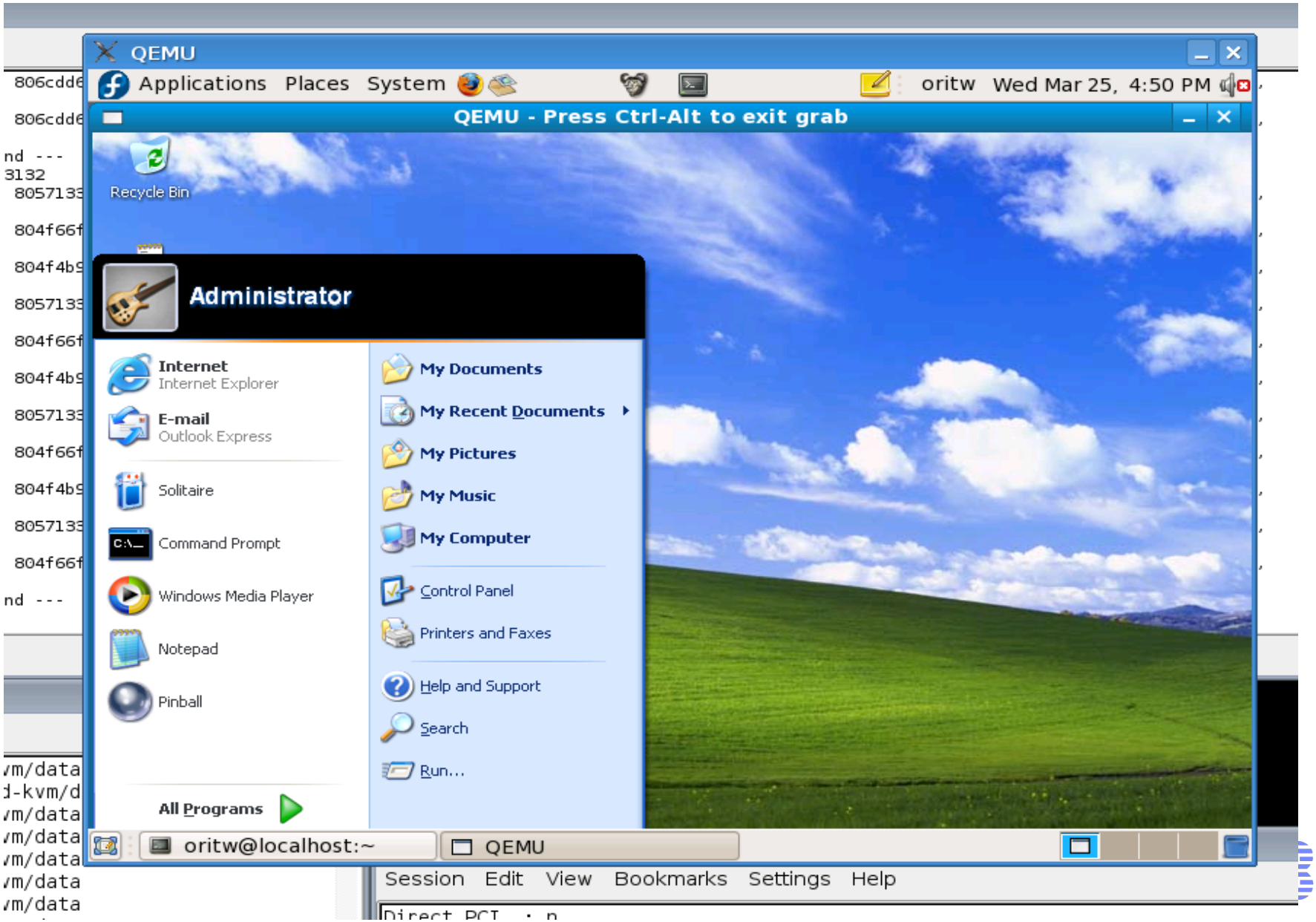
Micro-optimizations

- 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?

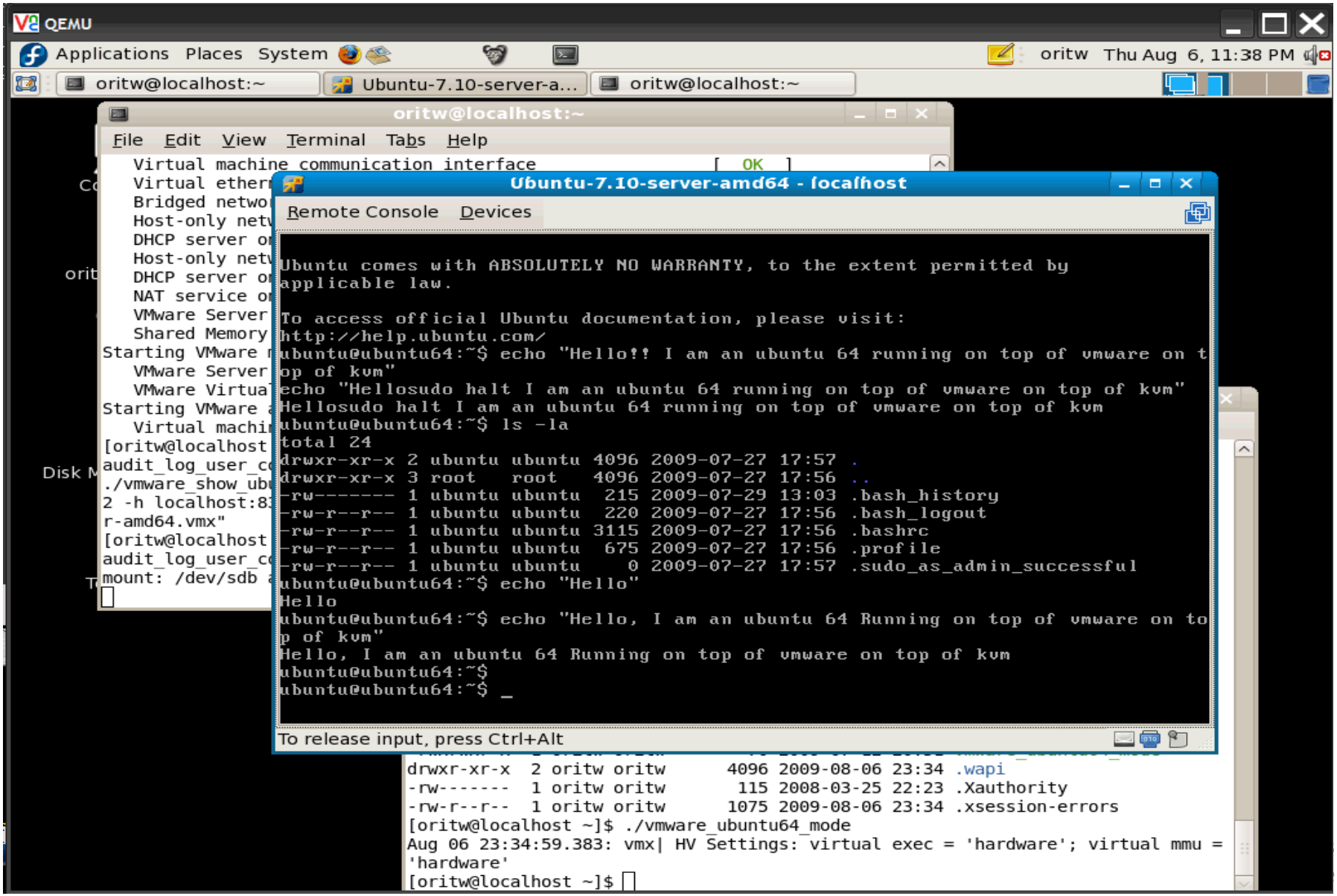
Micro-optimizations

- 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?

Windows XP on KVM L_1 on KVM L_0



Linux on VMware L_1 on KVM L_0



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

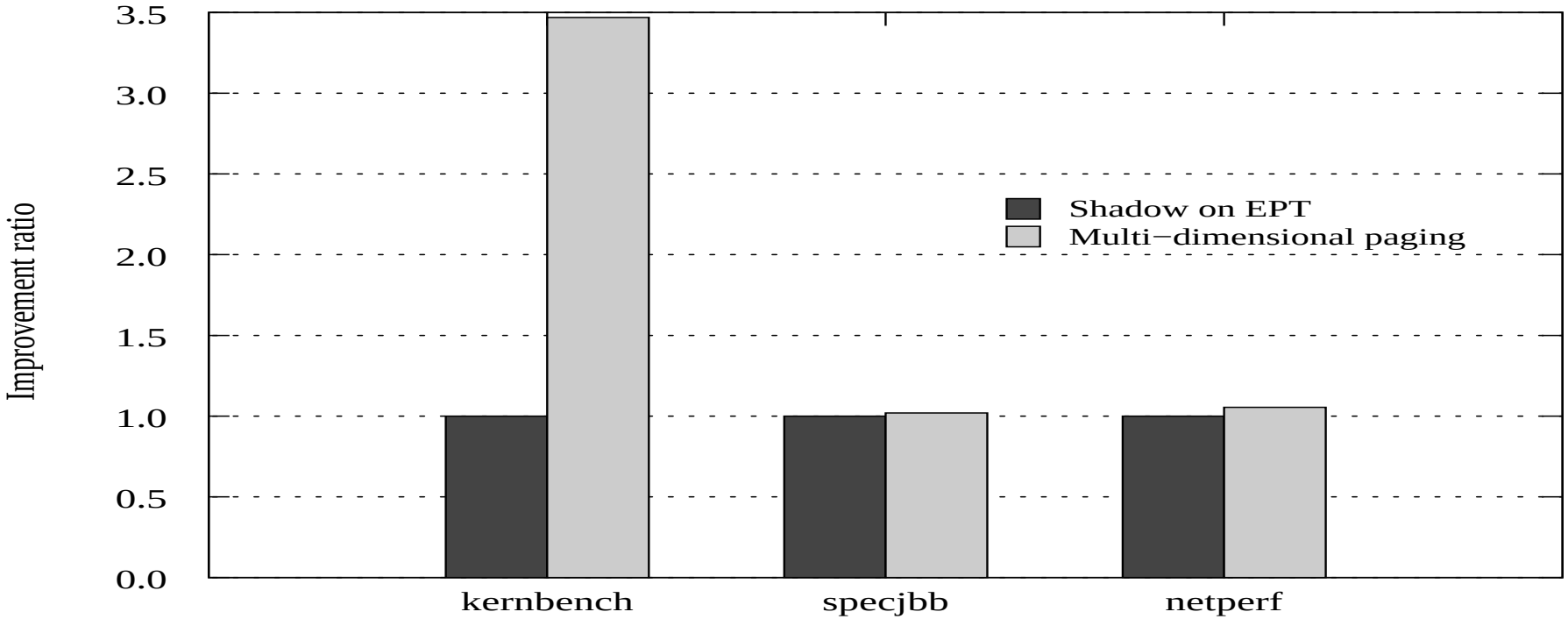
kernbench				
	Host	Guest	Nested	Nested$_{DRW}$
Run time	324.3	355	406.3	391.5
% overhead vs. host	-	9.5	25.3	20.7
% overhead vs. guest	-	-	14.5	10.3

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	6.3

Table: kernbench and SPECjbb results

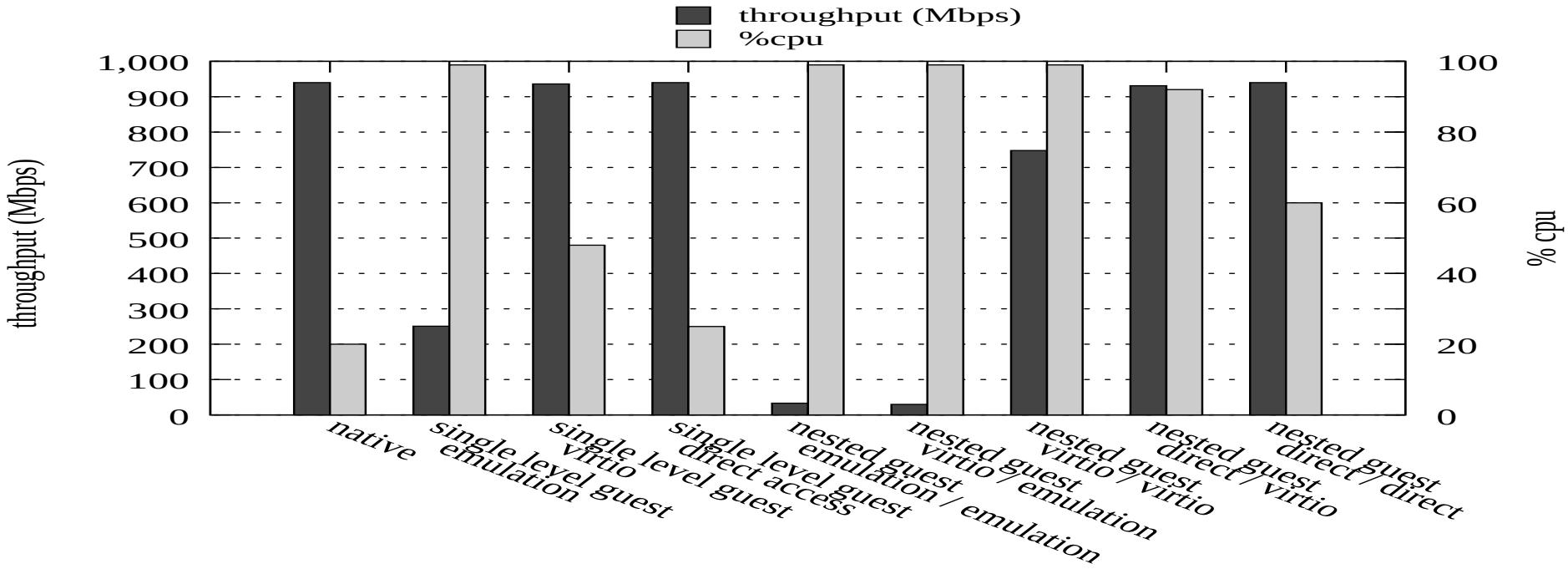
- Exit multiplication effect not as bad as we feared
- Direct vmread and vmwrite (DRW) give an immediate boost
- Take-away: each level of virtualization adds approximately the same overhead!

Macro: multi-dimensional paging



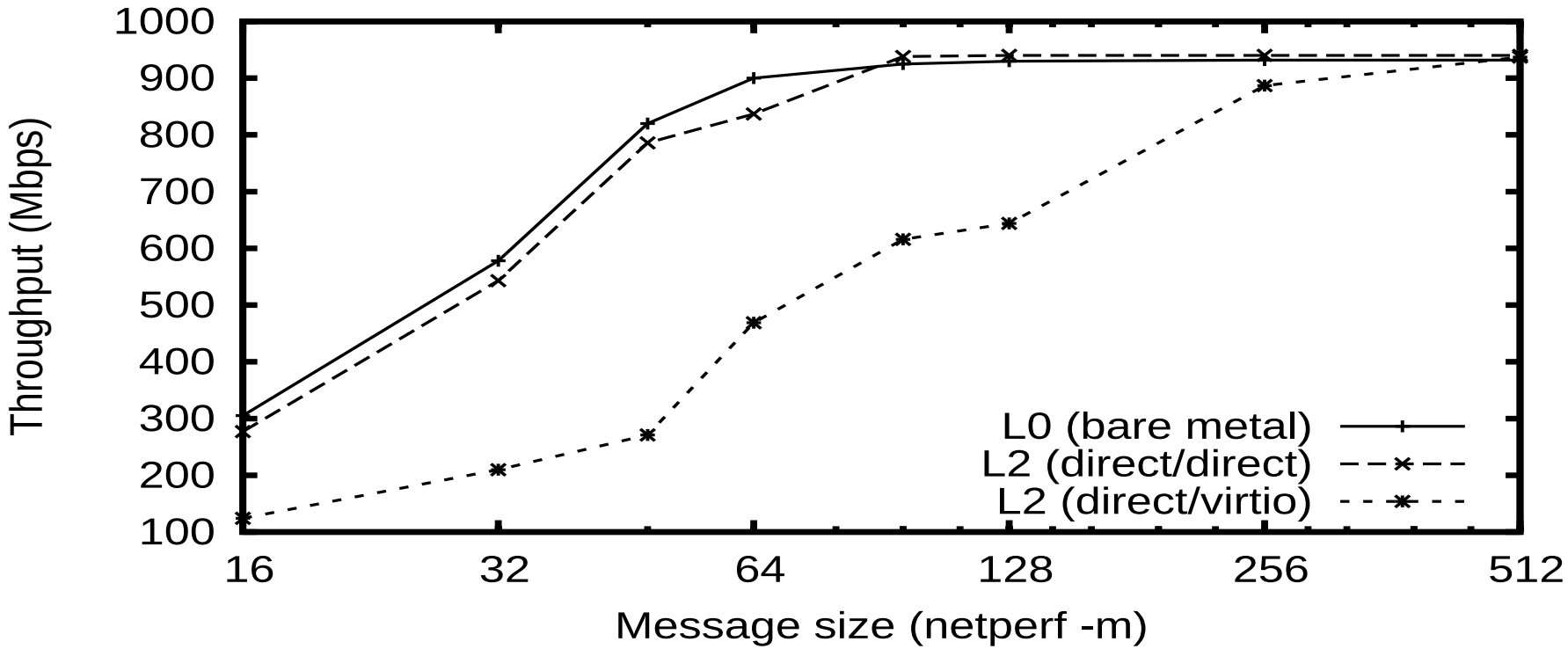
- Impact of multi-dimensional paging depends on rate of page faults
- Shadow-on-EPT: every L_2 page fault causes L_1 multiple exits
- Multi-dimensional paging: only EPT violations cause L_1 exits
- EPT table rarely changes: $\#(\text{EPT violations}) << \#(\text{page faults})$
- Multi-dimensional paging huge win for page-fault intensive `kernbench`

Macro: multi-level device assignment



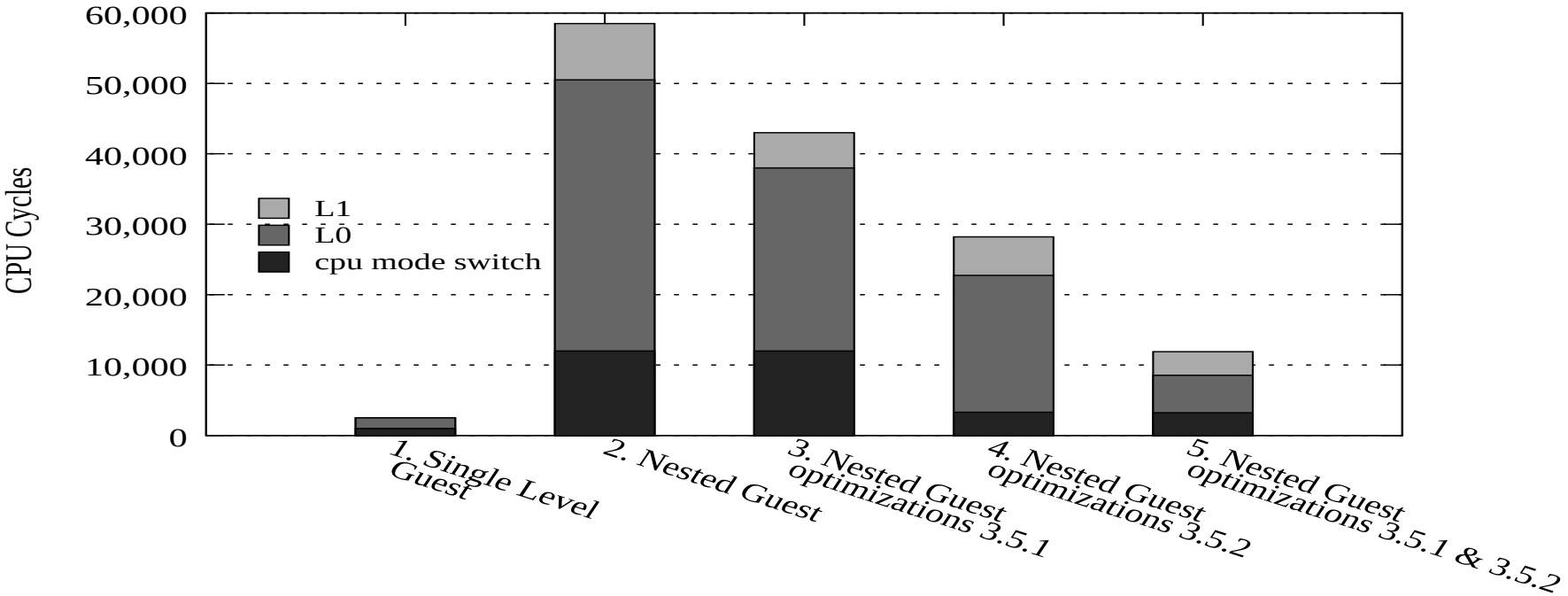
- Benchmark: `netperf TCP_STREAM` (transmit)
- Multi-level device assignment best performing option
- But: native at 20%, multi-level device assignment at 60% (x3!)
- Interrupts considered harmful, cause exit multiplication

Macro: multi-level device assignment (sans interrupts)



- What if we could deliver device interrupts directly to L_2?
- 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

IBM

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

IBM

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

IBM

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

IBM

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

IBM

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

IBM

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