Multi-Hypervisor Virtual Machines: Enabling An Ecosystem of Hypervisor-level Services

Kartik Gopalan, Rohith Kugve, Hardik Bagdi, Yaohui Hu – Binghamton University

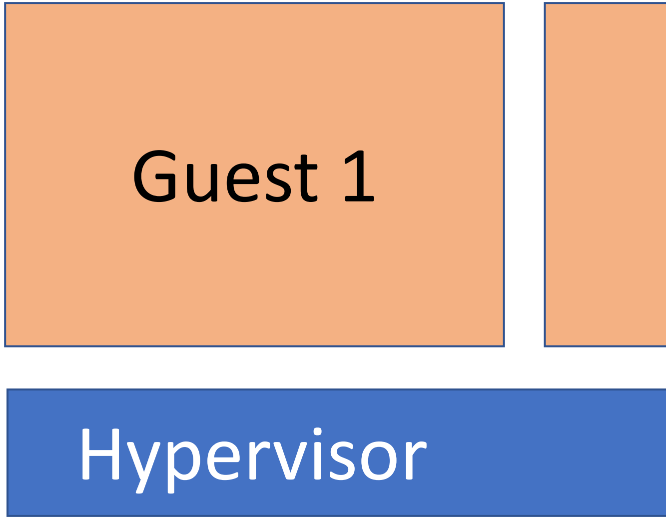
Dan Williams, Nilton Bila – IBM T.J. Watson Research Center

Funded by NSF

Hypervisors

- A thin and secure layer in the cloud

-- or --

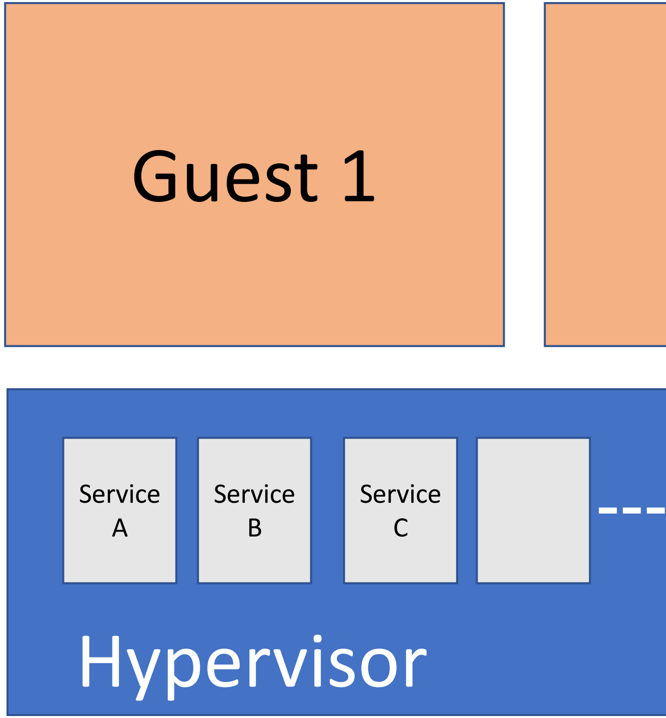


Hypervisors

- A thin and secure layer in the cloud

-- or --

- Feature-filled cloud differentiators
  - Migration
  - Checkpointing
  - High availability
  - Live Guest Patching
  - Network monitoring
  - Intrusion detection
  - Other VMI



Lots of third-party interest in hypervisor-level services

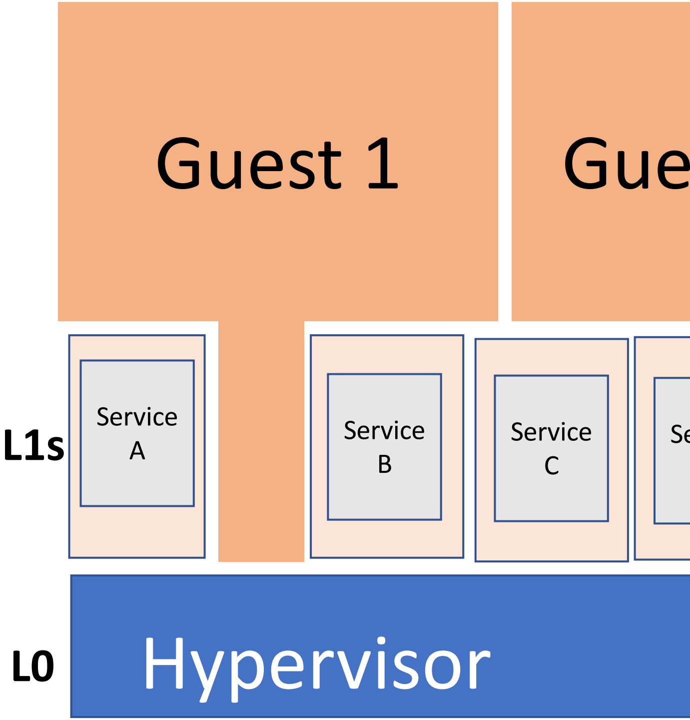
- Ravello
- Bromium
- XenBlanket
- McCafe DeepDefender
- Secvisor
- Cloudvisor
- And more…

- But limited support for third party services from base hypervisor.

How can a guest use multiple third-party hypervisor-level services?

- Our Solution: ***Span virtualization***

- One guest controlled by multiple coresident hypervisors.

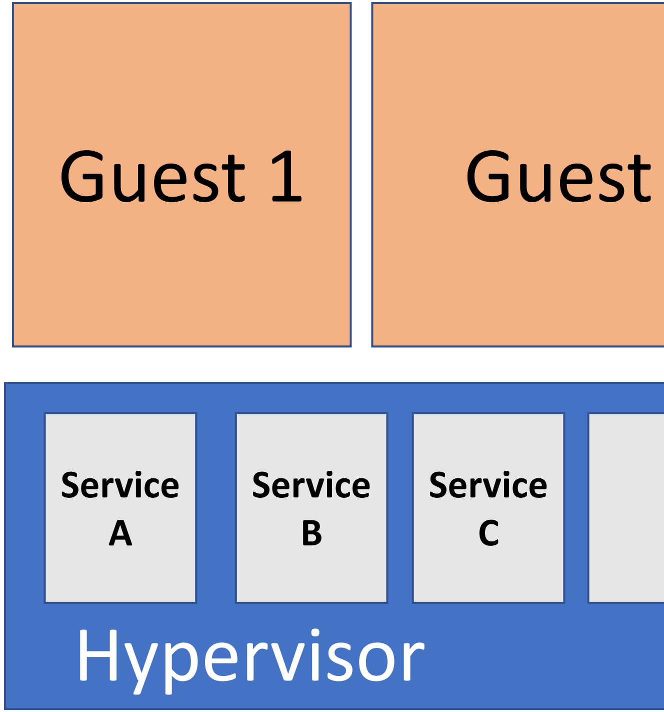


Outline

- Why multi-hypervisor virtual machines?
- Design of Span Virtualization
- Evaluations
- Related Work
- Conclusions and Future Work

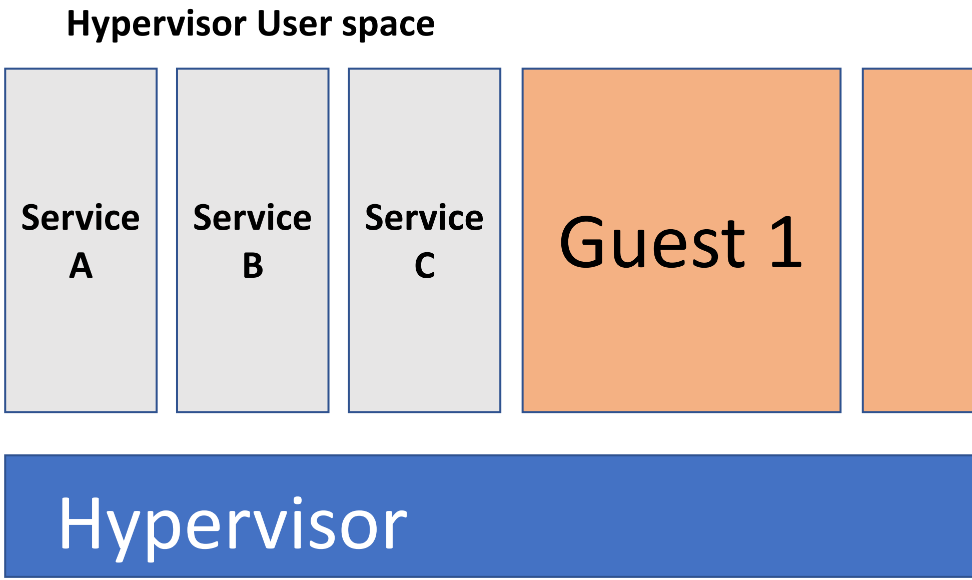
Option 1: Fat hypervisor

- All services run at the most privileged level.
- But…*hypervisor cannot trust third-party services in privileged mode.*



Option 2: Native user space services

- Services run natively in the user space of the hypervisor
- Services control guest indirectly via the hypervisor
- E.g. QEMU with KVM, uDenali
- But…Potentially large user-kernel interface
  - event interposition and system calls

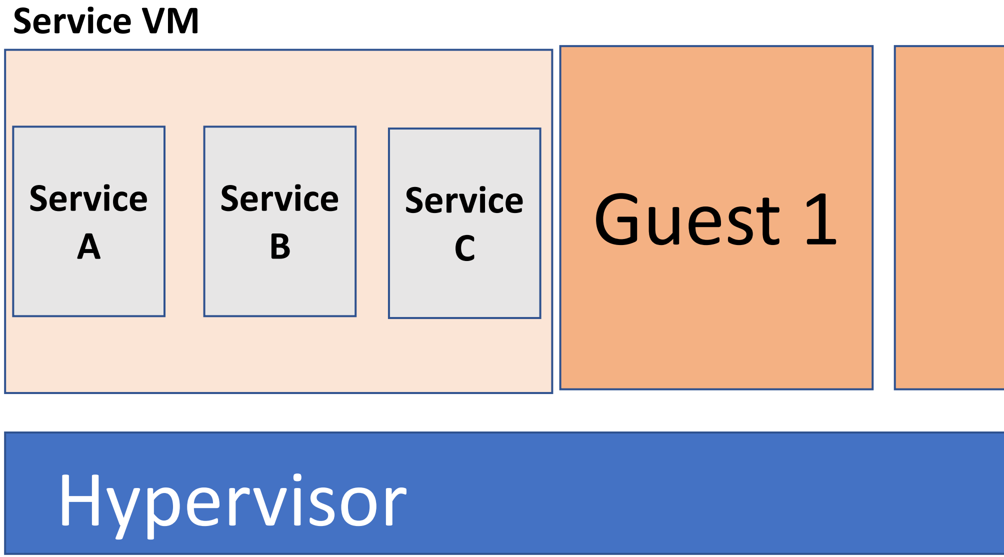


Cloud providers reluctant to run third-party services natively, even if in user space.

Option 3: Service VMs

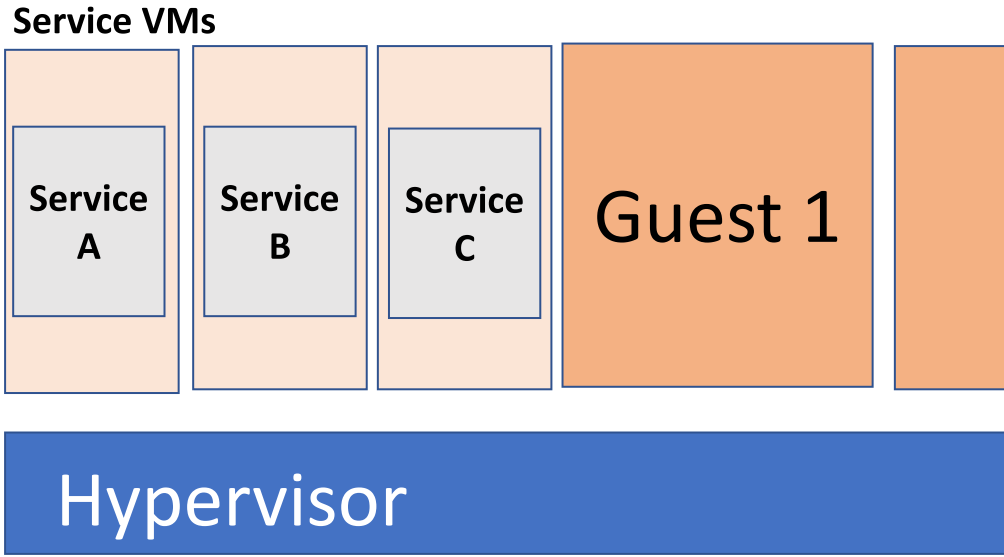
- Run services inside deprivileged VMs
- Services control guest indirectly via hypercalls and events
- **Single trusted Service VM**
  - Runs all services
  - E.g. Domain0 in Xen

-- or --



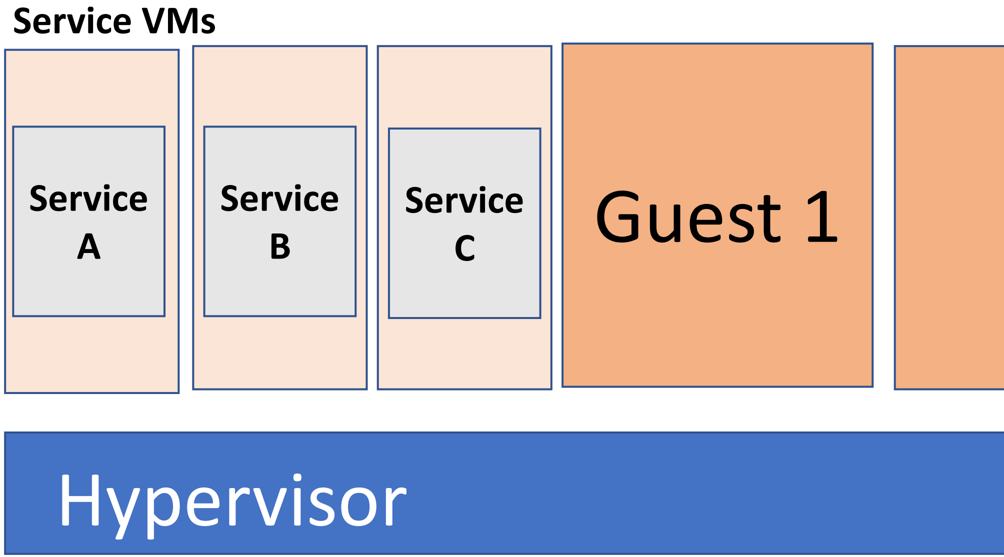
Option 3: Service VMs

- Run services inside deprivileged VMs
- Services control guest indirectly via hypercalls and events
- **<u>Multiple service VMs</u>**
  - One per service
  - Deprivileged and restartable
  - E.g. Service Domains in Xoar



Option 3: Service VMs

- Run services inside deprivileged VMs
- Services control guest indirectly via hypercalls and events
- **Multiple service VMs**
  - One per service
  - Deprivileged and restartable
  - E.g. Service Domains in Xoar

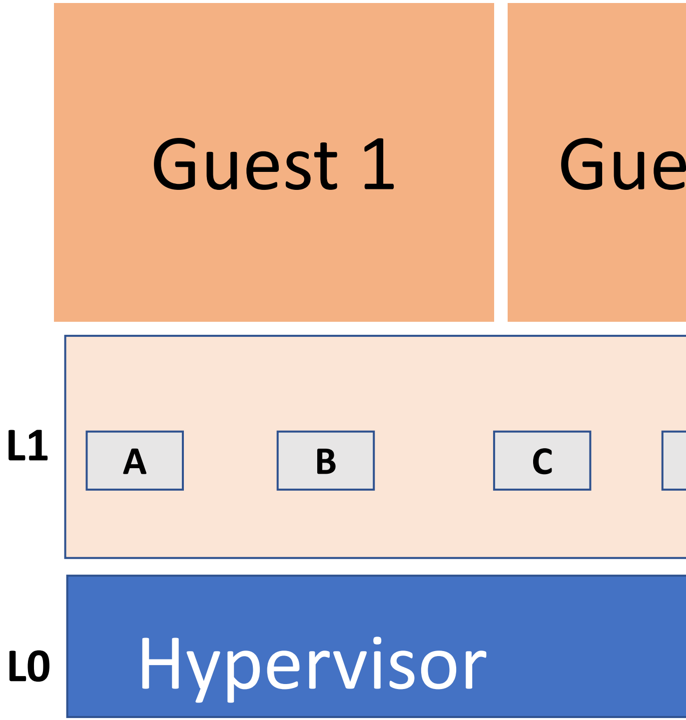


Lack direct control over ISA-level guest state

- Memory mappings, VCPU scheduling, port-mapped I/O, etc.

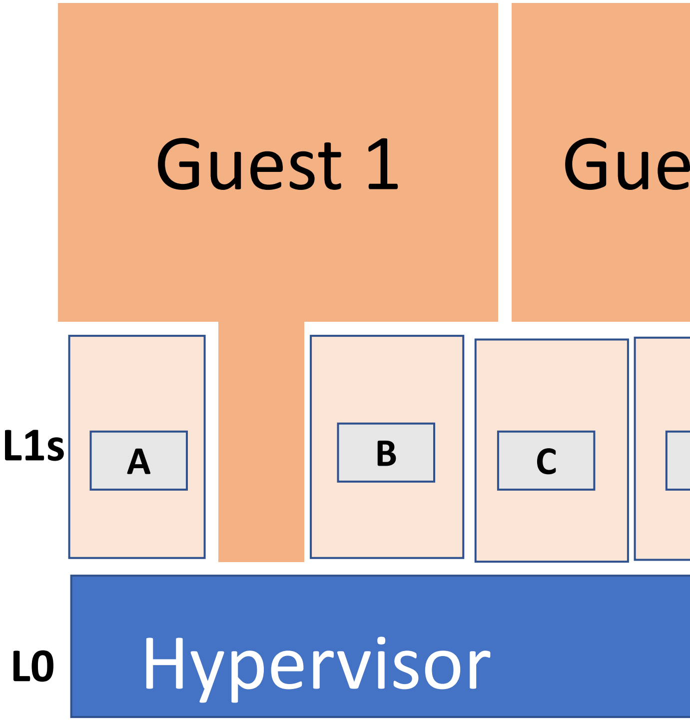
Option 4: Nested Virtualization

- Services run in a deprivileged L1 hypervisor, which runs on L0.
- Services control guest at virtualized ISA level.
- But … multiple services must reside in the same L1, i.e. fat L1.
- <u>Vertically</u> Stack L1 hypervisors?
  - More than two levels of nesting is inefficient.



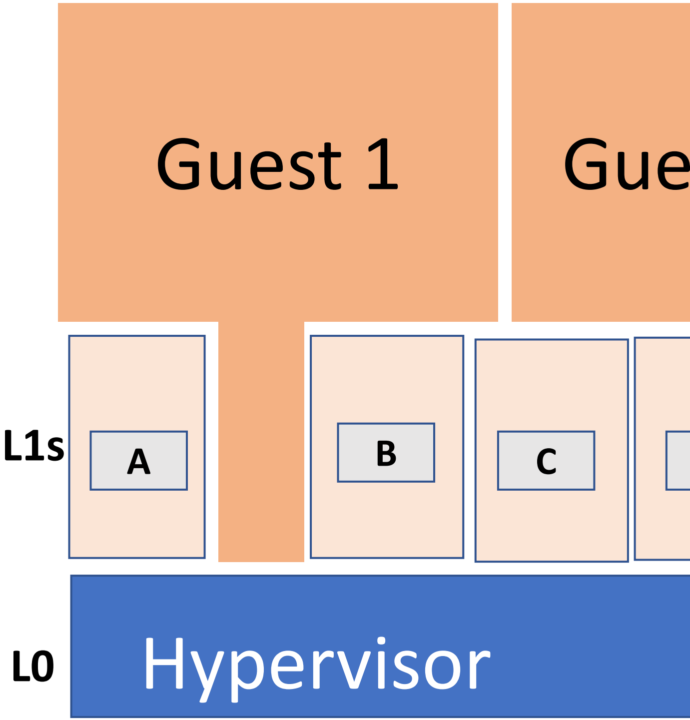
Our solution: *Span Virtualization*

- Allow multiple coresident L1s to concurrently control a common guest
  - i.e. Horizontal layering of L1 hypervisors
- Guest is a *multi-hypervisor virtual machine*
- Each L1
  - Offers guest services that augment L0's services.
  - Controls one or more guest resources



Design Goals of Span Virtualization

- Guest Transparency
  - Guest remains unmodified
- Service Isolation
  - L1s controlling the same guest are unaware of each other.

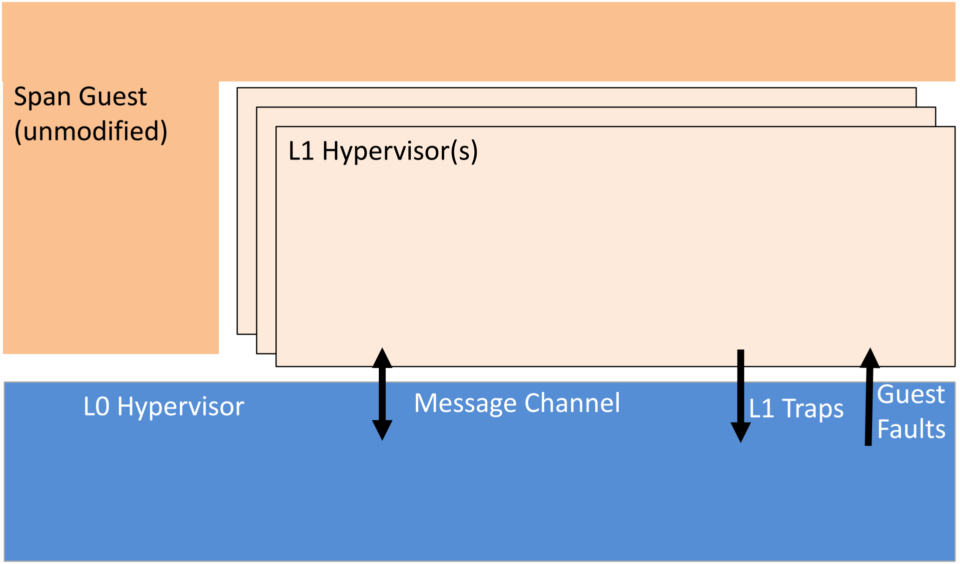


Guest Control operations

- L0 supervises which L1 controls which Guest resource
  - Memory, VCPU and I/O

- L0 and L1s communicate via Traps/Faults (implicit) and Messages (explicit)

- Operations:
  - **<u>Attach</u>** an L1 to a specified guest resource
  - **<u>Detach</u>** an L1 from a guest resource
  - **<u>Subscribe</u>** an attached L1 to receive guest events (currently memory events)
  - **<u>Unsubscribe</u>** an L1 from a subscribed guest event

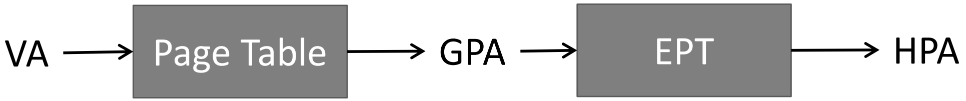


Control over Guest Resources

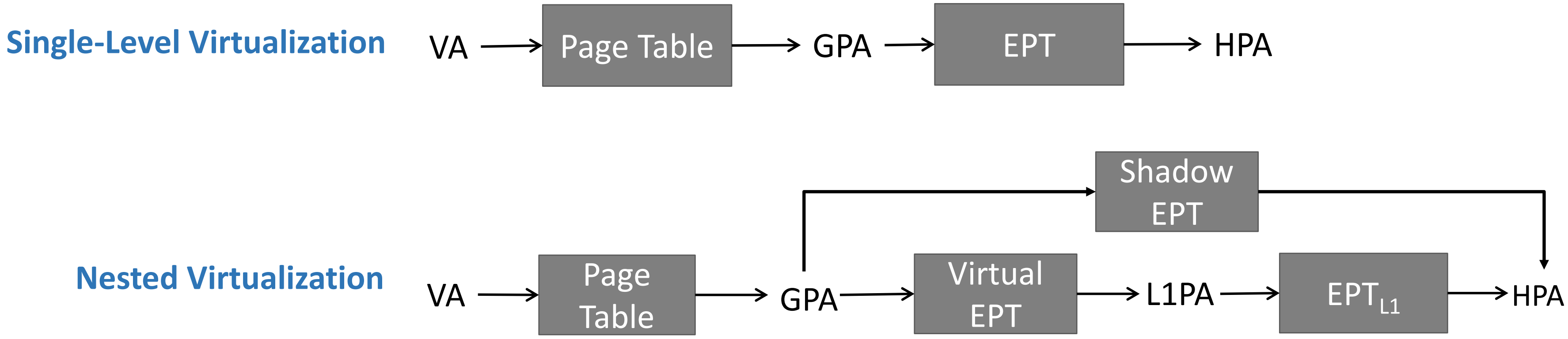
- **Guest Memory**
  - Shared: All hypervisors have the same consistent view of guest memory
- **Guest VCPUs**
  - Exclusive: All guest VCPUs are controlled by one hypervisor at any instant
- **Guest I/O devices**
  - Exclusive: Different virtual I/O devices of a guest may be controlled by different hypervisors
- **Control Transfer**
  - Control over guest VCPUs and I/O devices can be transferred from one L1 to another via L0.

Memory Translation

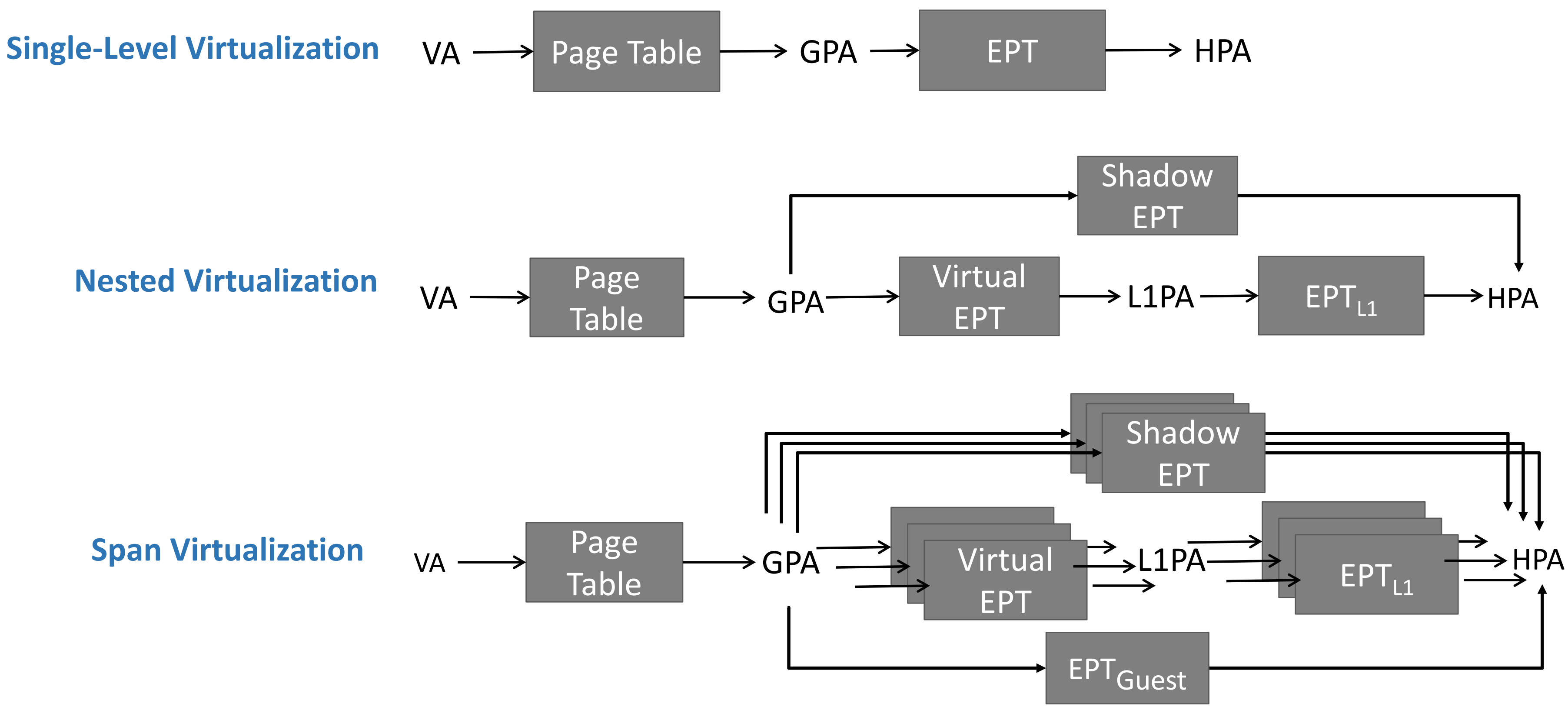
Single-Level Virtualization



Memory Translation

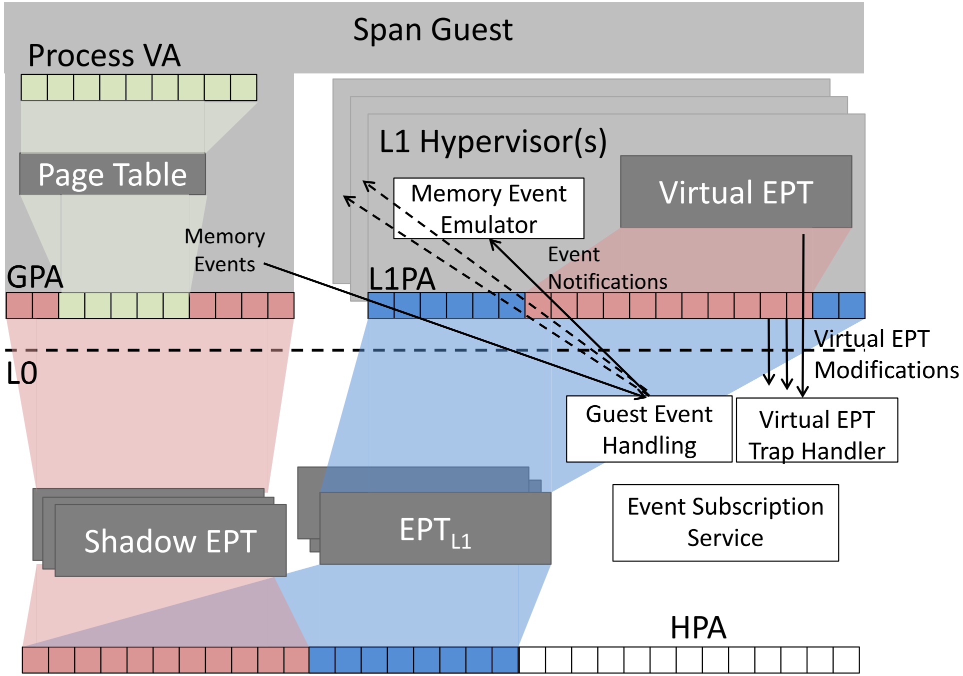


Memory Translation



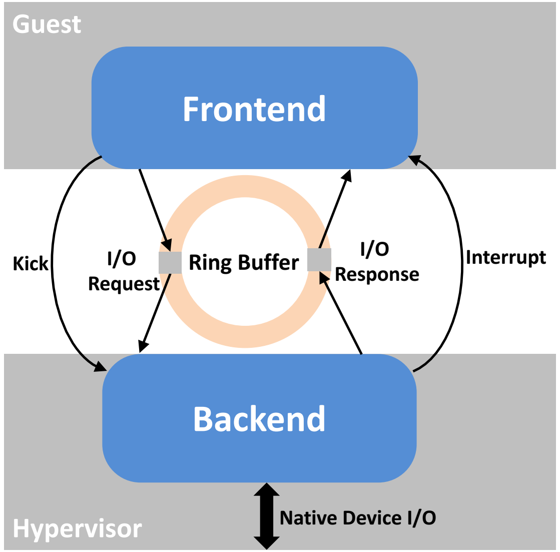
Synchronizing Guest Memory Maps

- Guest physical memory to Host physical memory translation should be the same regardless of the translation path.
- L0 syncs Shadow EPTs and EPT_{L1s}
  - Guest faults
  - Virtual EPT modifications by L1
  - When L1 directly accesses guest memory
- L1s subscribe to guest memory events via L0
  - E.g. to track write events for dirty page tracking



I/O Control

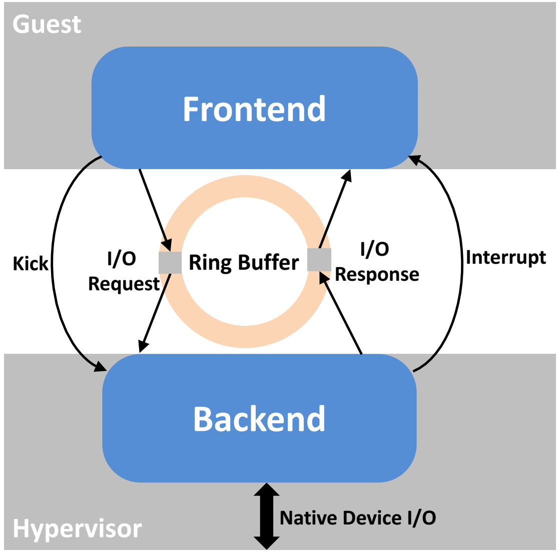
- We consider para-virtual I/O in this work



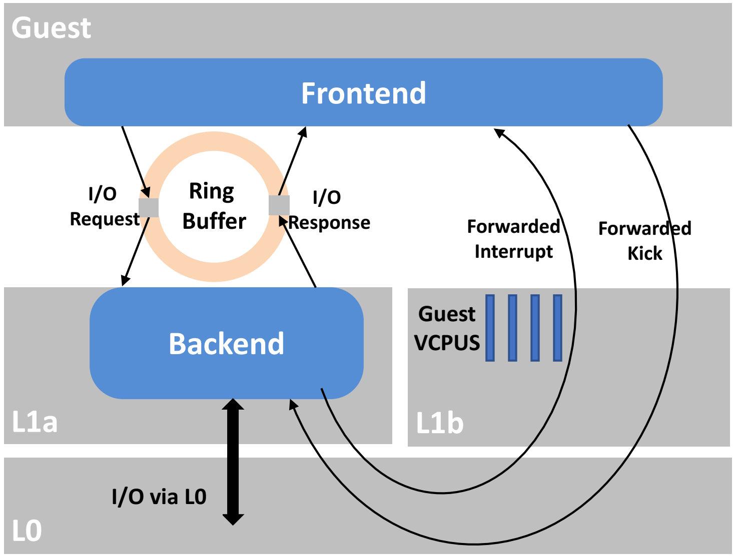
Traditional Para-virtual I/O

I/O Control

- We consider para-virtual I/O in this work
- A Span Guest's I/O device and VCPUs may be controlled by different L1s



Traditional Para-virtual I/O



Para-virtual I/O in Span Virtualization

VCPU control

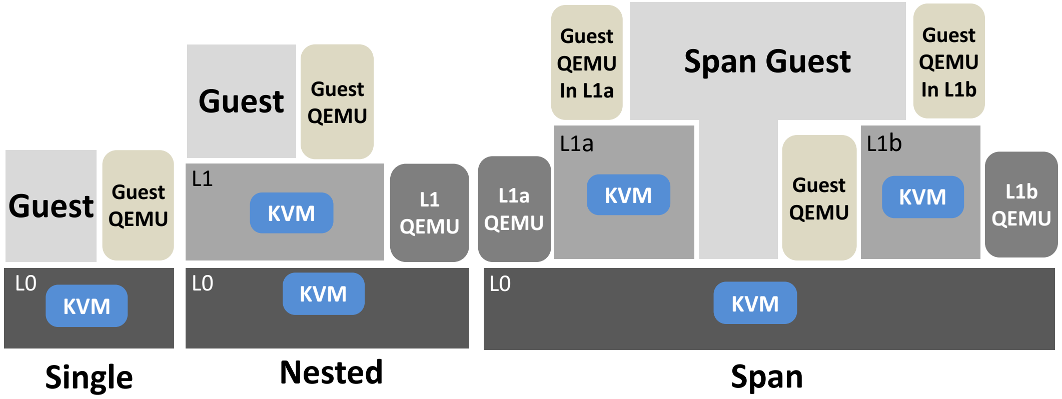
- Simple for now.

- All VCPUs controlled by one hypervisor
  - Either by L0 or one of the L1s

- Can we distribute VCPUs among L1s?
  - Possible, but no good reason why.
  - Requires expensive IPI forwarding across L1s
  - Complicates memory synchronization.

Implementation

- **<u>Guest</u>**: Unmodified Ubuntu 15.10, Linux 4.2



- **<u>L0 and L1</u>**
  - QEMU 1.2 and Linux 3.14.2
  - Modified nesting support in KVM/QEMU
  - L0 : 980+ lines in KVM and 500+ lines in QEMU
  - L1: 300+ lines in KVM and 380+ in QEMU

- **<u>Guest controller</u>**
  - User space QEMU process
  - Guest initialization, I/O emulation, Control Transfer, Migration, etc

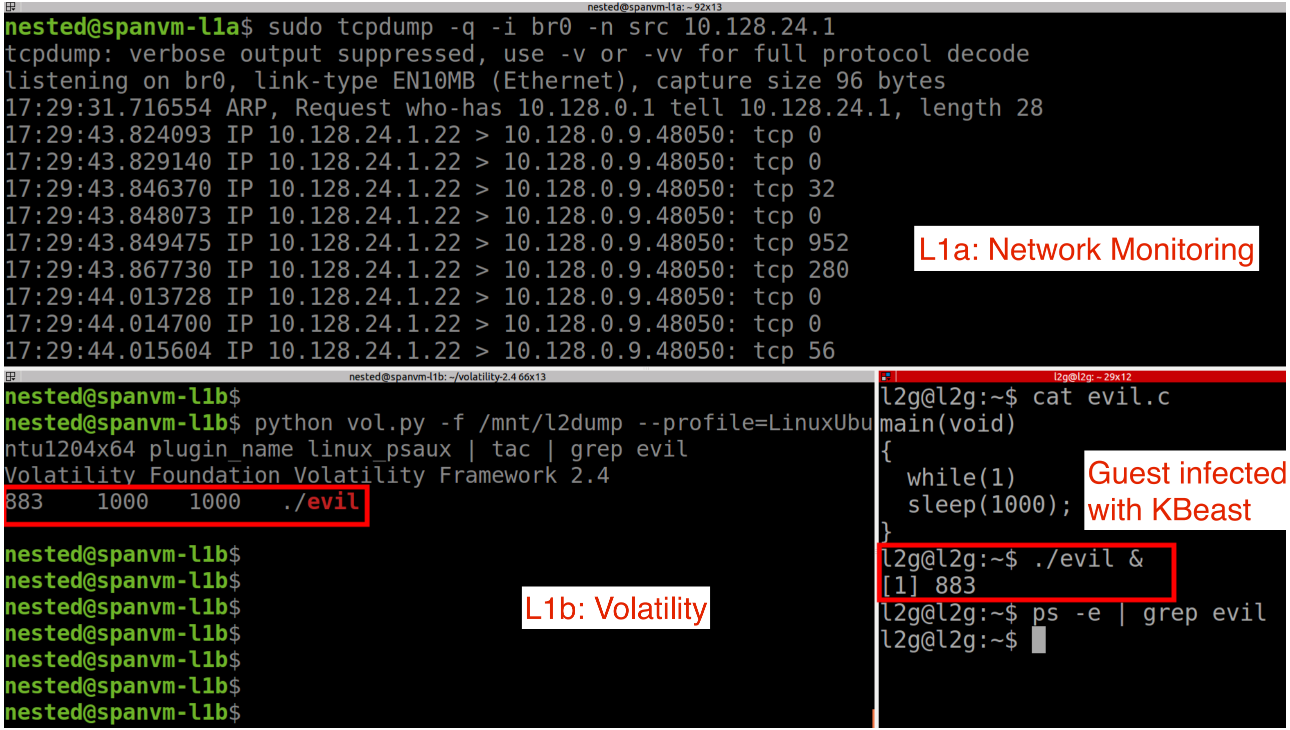
- **<u>I/O: virtio-over-virtio</u>**
  - Direct assignment: future work

- **<u>Message channel</u>**
  - For I/O kick and interrupt forwarding
  - Currently using UDP messages and hypercalls

- **<u>Control Transfer</u>**
  - Guest VCPUs and virtio devices can be transferred between L1s and L0
  - Using attach/detach operations

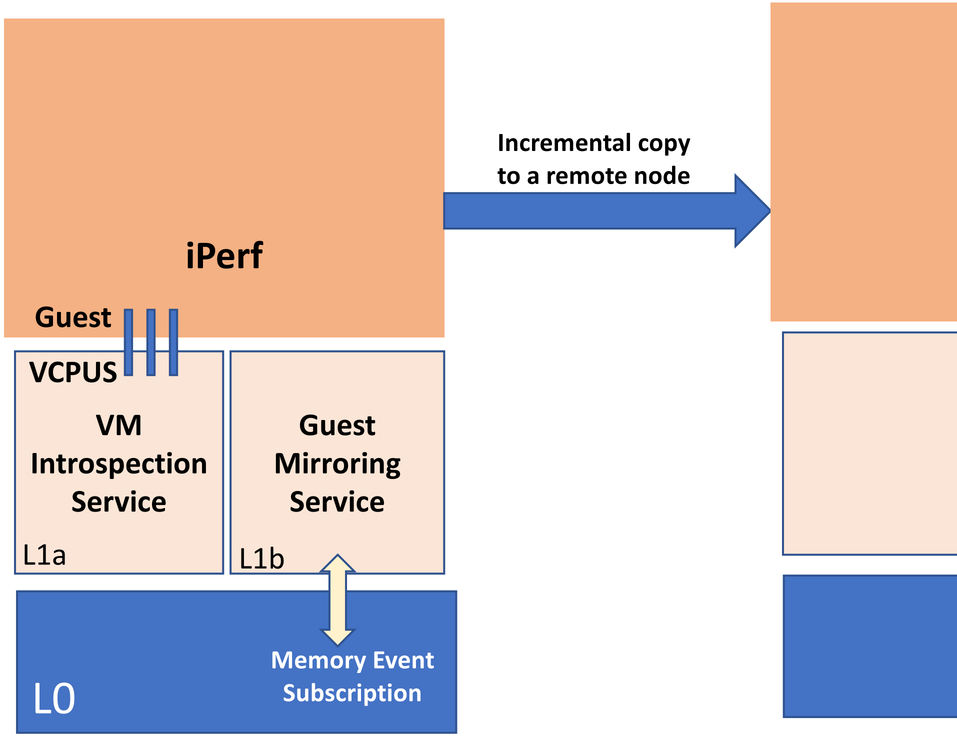
Example 1: Two L1s controlling one Guest

- Guest: Infected with rootkit
- L1a: Monitoring network traffic
- L1b: Running VMI (Volatility)



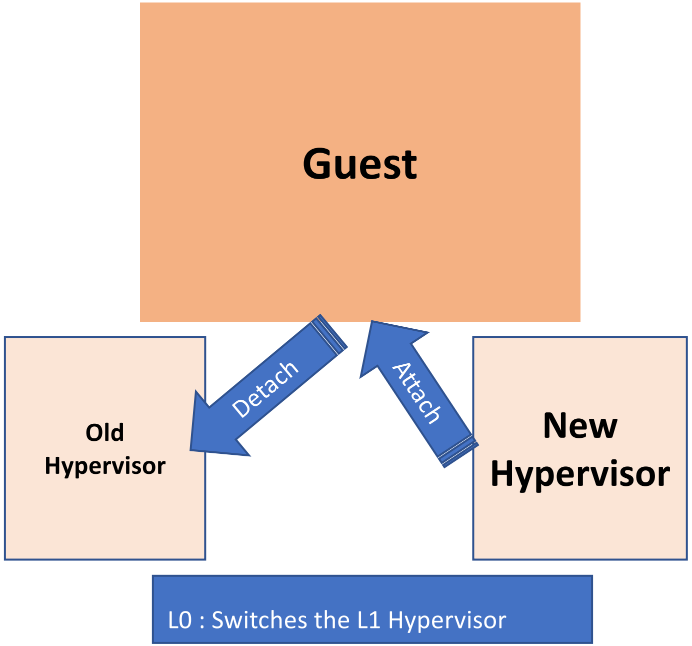
Example 2: Guest mirroring

- L1a runs Volatility
- L1b runs Guest Mirroring
  - Periodically copy dirty guest pages
  - Requires subscription on write events
- Guest runs iPerf
  - ~800Mbps when mirrored every 12 seconds. Same as standard nested.
  - ~600Mbps every 1 second.
  - 25% impact with high frequency dirty page tracking



Example 3: Live Hypervisor Replacement

- Replace hypervisor underneath a live Guest
  - L1 runs a full hypervisor
  - L0 acts as a thin switching layer
- Replacement operation
  - Attach new L1
  - Detach old L1
- 740ms replacement latency, including memory co-mapping
- 70ms guest downtime
  - During VCPU and I/O state transfer



Macrobenchmarks

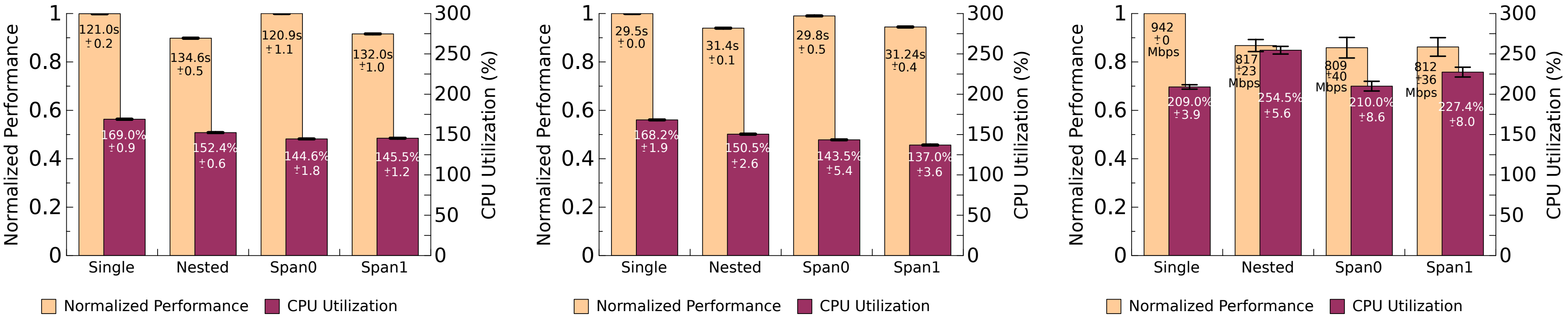
	L0		L1		L2	
	Mem	CPUs	Mem	VCPUs	Mem	VCPUs
Single	128GB	12	3GB	1	N/A	N/A
Nested	128GB	12	16GB	8	3GB	1
Span0	128GB	12	8GB	4	3GB	1 on L0
Span1	128GB	12	8GB	4	3GB	1 on L1

Guest Workloads

- Kernbench: repeatedly compiles the kernel
- Quicksort: repeatedly sorts 400MB data
- iPerf: Measures bandwidth to another host

Hypervisor-level Services

- Network monitoring (tcpdump)
- VMI (Volatility)



(a) Kernbench (b) Quicksort (c) iPerf

Macrobenchmarks

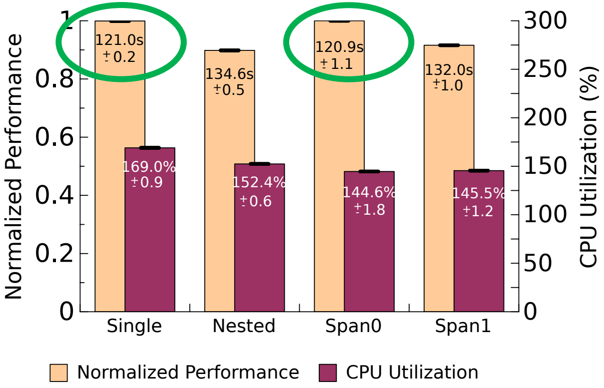
	L0		L1		L2	
	Mem	CPUs	Mem	VCPUs	Mem	VCPUs
Single	128GB	12	3GB	1	N/A	N/A
Nested	128GB	12	16GB	8	3GB	1
Span0	128GB	12	8GB	4	3GB	1 on L0
Span1	128GB	12	8GB	4	3GB	1 on L1

Guest Workloads

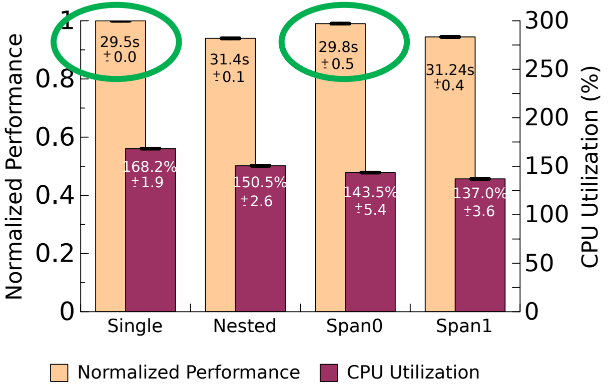
- Kernbench: repeatedly compiles the kernel
- Quicksort: repeatedly sorts 400MB data
- iPerf: Measures bandwidth to another host

Hypervisor-level Services

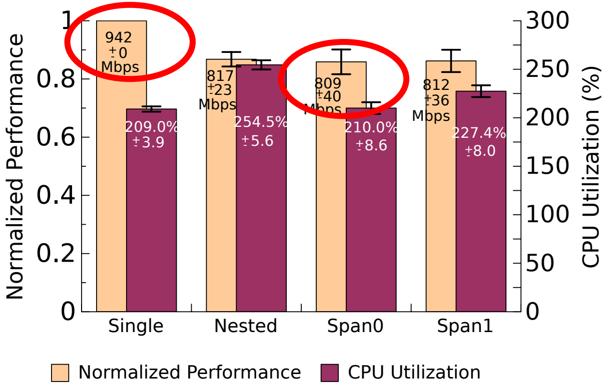
- Network monitoring (tcpdump)
- VMI (Volatility)



(a) Kernbench



(b) Quicksort



(c) iPerf

Macrobenchmarks

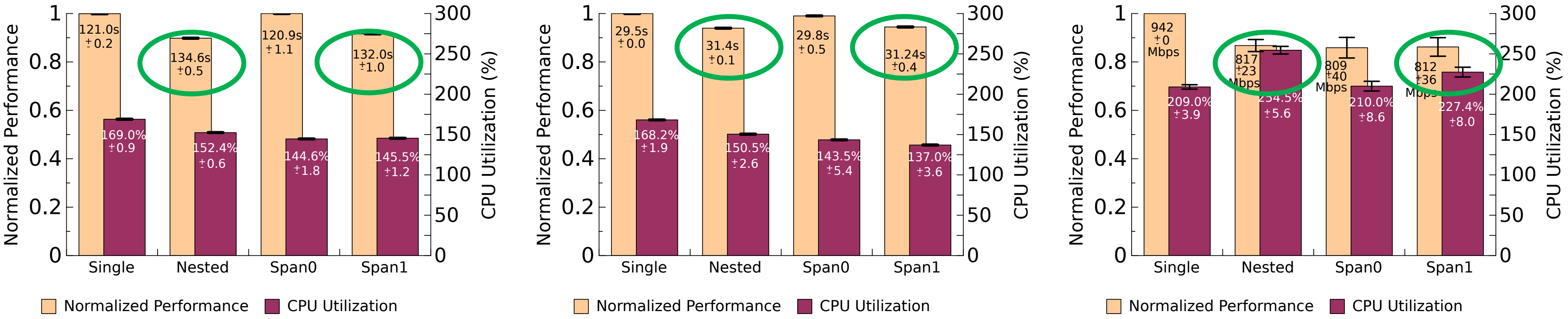
	L0		L1		L2	
	Mem	CPUs	Mem	VCPUs	Mem	VCPUs
Single	128GB	12	3GB	1	N/A	N/A
Nested	128GB	12	16GB	8	3GB	1
Span0	128GB	12	8GB	4	3GB	1 on L0
Span1	128GB	12	8GB	4	3GB	1 on L1

Guest Workloads

- Kernbench: repeatedly compiles the kernel
- Quicksort: repeatedly sorts 400MB data
- iPerf: Measures bandwidth to another host

Hypervisor-level Services

- Network monitoring (tcpdump)
- VMI (Volatility)



(a) Kernbench (b) Quicksort (c) iPerf

Microbenchmarks

	Single	Nested	Span
EPT Fault	2.4	2.8	3.3
Virtual EPT Fault	-	23.3	24.1
Shadow EPT Fault	-	3.7	4.1
Message Channel	-	-	53
Memory Event Notify	-	-	103.5

Low-level latencies in Span virtualization

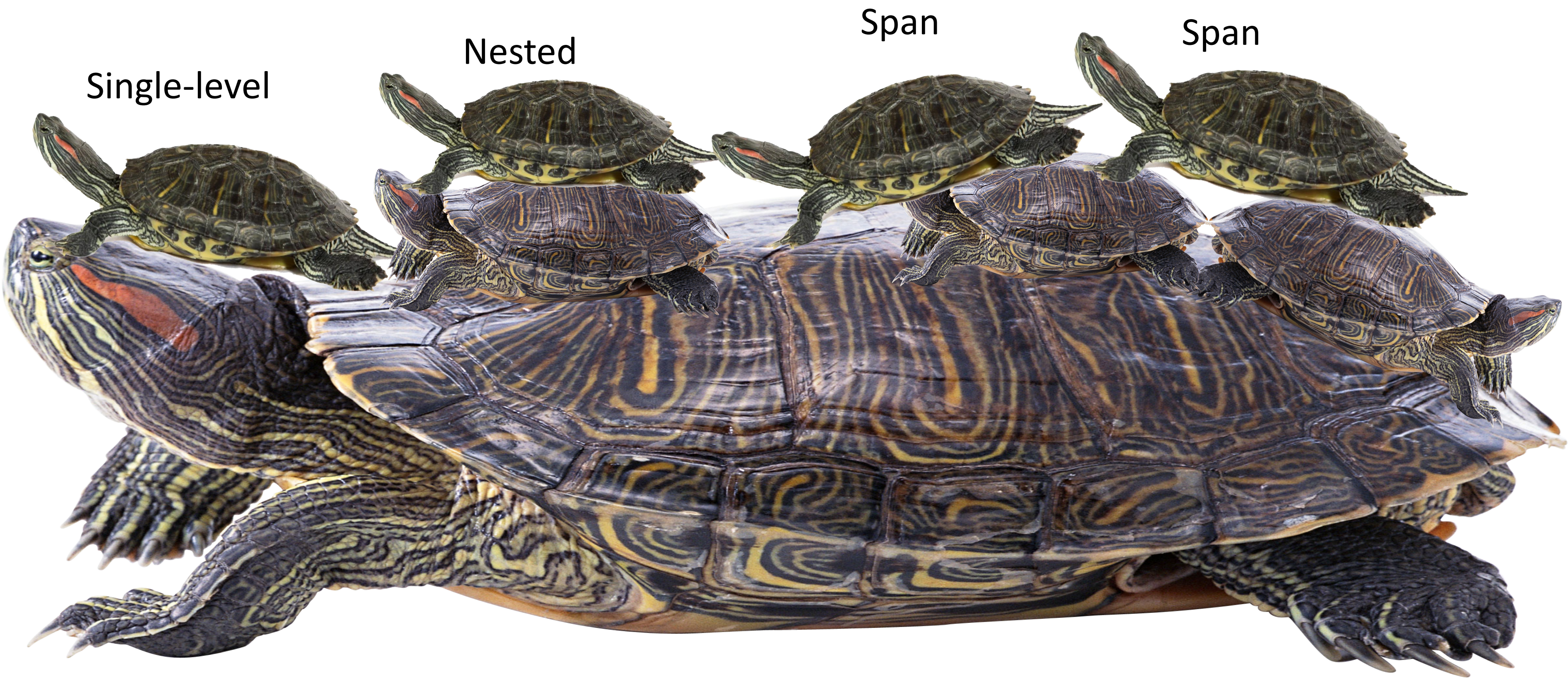
Related Work

- User space Services
    - Microkernels, library OS, uDenali, KVM/QEMU, NOVA
- Service VMs
    - Dom0 in Xen, Xoar, Self-Service Cloud
- Nested virtualization
    - Belpaire & Hsu, Ford et. al, Graf & Roedel, Turtles
    - Ravello, XenBlanket, Bromium, DeepDefender, Dichotomy
- Span virtualization is the first to address <u>multiple</u> <u>third-party</u> hypervisor-level services to a <u>common</u> guest

Summary: Span Virtualization

- We introduced the concept of a *multi-hypervisor virtual machine*
  - that can be concurrently controlled by multiple coresident hypervisors
- Another tool in a cloud provider's toolbox
  - to offer compartmentalized guest-facing third-party services
- Future work
  - Faster event notification and processing
  - Direct device assignment to L1s or Guest
  - Possible to support unmodified L1s?
    - Requires L1s to support partial guest control. Current L1s assume full control.
- Code to be released after porting to newer KVM/QEMU

Questions?

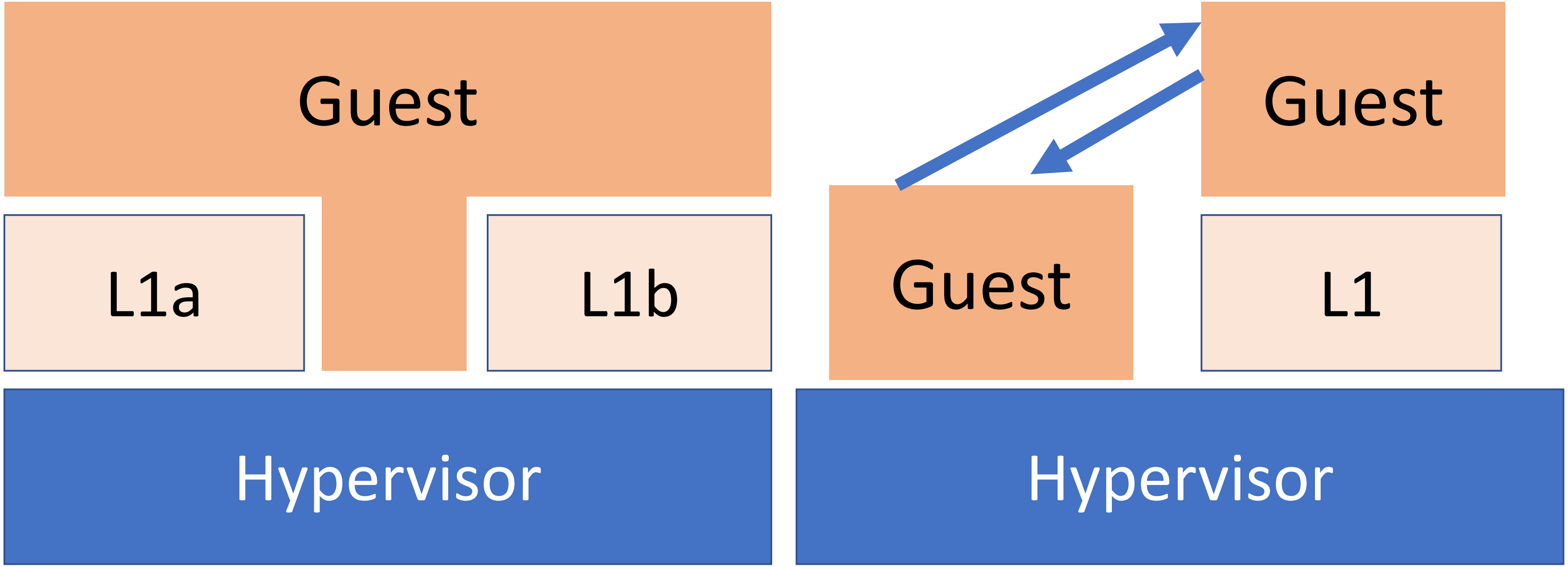


Backup slides

Comparison

	Level of Guest Control		Impact of Service Failure			Additional Performance Overheads
	Virtualized ISA	Partial or Full	L0	Coresident Services	Guests	
Single-level	Yes	Full	Fails	Fail	All	None
User space	No	Partial	Protected	Protected	Attached	Process switching
Service VM	No	Partial	Protected	Protected	Attached	VM switching
Nested	Yes	Full	Protected	Protected in L1 user space	Attached	L1 switching + nesting
Span	Yes	Both	Protected	Protected	Attached	L1 switching + nesting

Continuous and Transient Control



<u>Continuous Control</u>
L1s always attached to guest

<u>Transient Control</u>
L1 attaches/detaches from guest as needed