Multi-hypervisor Virtual Machines: Enabling an Ecosystem of Hypervisor-level Services

Kartik Gopalan, Rohith Kugve, Hardik Bagdi, Yaohui Hu
Computer Science, Binghamton University

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

Abstract

Public cloud software marketplaces already offer users a wealth of choice in operating systems, database management systems, financial software, and virtual networking, all deployable and configurable at the click of a button. Unfortunately, this level of customization has not extended to emerging hypervisor-level services, partly because traditional virtual machines (VMs) are fully controlled by only one hypervisor at a time. Currently, a VM in a cloud platform cannot concurrently use hypervisor-level services from multiple third-parties in a compartmentalized manner. We propose the notion of a *multi-hypervisor VM*, which is an unmodified guest that can simultaneously use services from multiple coresident, but isolated, hypervisors. We present a new virtualization architecture, called *Span virtualization*, that leverages nesting to allow multiple hypervisors to concurrently control a guest's memory, virtual CPU, and I/O resources. Our prototype of Span virtualization on the KVM/QEMU platform enables a guest to use services such as introspection, network monitoring, guest mirroring, and hypervisor refresh, with performance comparable to traditional nested VMs.

1 Introduction

In recent years, a number of hypervisor-level services have been proposed such as rootkit detection [61], live patching [19], intrusion detection [27], high availability [24], and virtual machine (VM) introspection [30, 53, 26, 49, 42, 60]. By running inside the hypervisor instead of the guest, these services can operate on multiple guests, while remaining transparent to the guests. Recent years have also seen a rise in the number of specialized hypervisors that are tailored to provide VMs with specific services. For instance, McAfee Deep Defender [46] uses a micro-hypervisor called DeepSafe to improve guest security. SecVisor [56] provides code integrity for commodity guests. CloudVisor [68] guarantees guest privacy and integrity on untrusted clouds. RTS provides a Real-time Embedded Hypervisor [52] for real-time guests. These specialized hypervisors may not provide guests with the full slate of memory, virtual CPU (VCPU), and I/O management, but rely upon either another commodity hypervisor, or the guest itself, to fill in the missing services.

Currently there is no good way to expose multiple services to a guest. For a guest which needs *multiple* hypervisor-level services, the first option is for the single controlling hypervisor to bundle all services in its supervisor mode. Unfortunately, this approach leads to a "fat" feature-filled hypervisor that may no longer be trustworthy because it runs too many untrusted or mutually distrusting services. One could de-privilege some services to the hypervisor's user space as processes that control the guest indirectly via event interposition and system calls [63, 40]. However, public cloud providers would be reluctant to execute untrusted third-party services in the hypervisor's native user space due to a potentially large user-kernel interface.

The next option is to de-privilege the services further, running each in a *Service VM* with a full-fledged OS. For instance, rather than running a single Domain0 VM running Linux that bundles services for all guests, Xen [4] can use several disaggregated [23] service domains for resilience. Service domains, while currently trusted by Xen, could be adapted to run third-party untrusted services. A service VM has a less powerful interface to the hypervisor than a user space service. However, neither user space services nor Service VMs allow control over low-level guest resources, such as page mappings or VCPU scheduling, which require hypervisor privileges.

One could use *nested virtualization* [34, 10, 48, 29] to *vertically stack* hypervisor-level services, such that a trusted base hypervisor at layer-0 (L0) controls the physical hardware and runs a service hypervisor at layer-1 (L1), which fully or partially controls the guest at layer-2

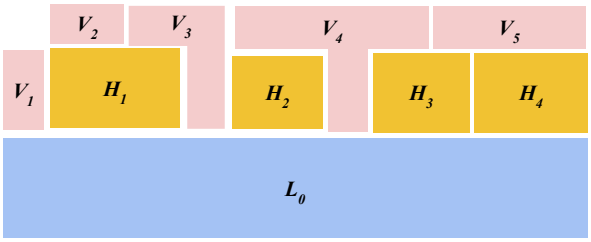


Figure 1: Span virtualization: V_1 is a traditional single-level VM. V_2 is a traditional nested VM. V_3, V_4 and V_5 are multi-hypervisor VMs.

(L2). Nested virtualization is experiencing considerable interest [25, 31, 64, 68, 55, 53, 56, 39, 7, 65, 45]. For example, one can use nesting [21] to run McAfee Deep Defender [46], which does not provide full system and I/O virtualization, as a guest on XenDesktop [20], a full commodity hypervisor, so that guests can use the services of both. Similarly, Bromium [15] uses nesting on a Xen-based hypervisor for security. Ravello [2] and XenBlanket [66, 57] use nesting on public clouds for cross-cloud portability. However, vertical stacking reduces the degree of guest control and visibility to lower layers compared to the layer directly controlling the guest. Also, the overhead of nested virtualization beyond two layers can become rather high [10].

Instead, we propose *Span virtualization*, which provides *horizontal* layering of multiple hypervisor-level services. A *Span VM*, or a multi-hypervisor VM, is an unmodified guest whose resources (virtual memory, CPU, and I/O) can be *simultaneously* controlled by multiple coresident, but isolated, hypervisors. A base hypervisor at L0 provides a core set of services and uses nested virtualization to run multiple deprivileged service hypervisors at L1. Each L1 augments L0's services by adding/replacing one or more services. Since the L0 no longer needs to implement every conceivable service, L0's footprint can be smaller than a feature-filled hypervisor. Henceforth, we use the following terms:

- **Guest** or **VM** refers to a top-level VM, with qualifiers *single-level*, *nested*, and *Span* as needed.
- **L1** refers to a service hypervisor at layer-1.
- **L0** refers to the base hypervisor at layer-0.
- **Hypervisor** refers to the role of either L0 or any L1 in managing guest resources.

Figure 1 illustrates possible Span VM configurations. One L0 hypervisor runs multiple L1 hypervisors (H_1, H_2, H_3, and H_4) and multiple guests (V_1, V_2, V_3, V_4 and V_5). V_1 is a traditional single-level (non-nested) guest that runs on L0. V_2 is a traditional nested guest that runs on only one hypervisor (H_1). The rest are multi-hypervisor VMs. V_3 runs on two hypervisors (L0 and H_1). V_4 runs on three hypervisors (L0, H_2, and H_3). V_5 is a fully nested Span VM that runs on two L1s (H_3 and H_4). This paper makes the following contributions:

- We examine the solution space for providing multiple services to a common guest and identify the relative merits of possible solutions.
- We present the design of Span virtualization which enables multiple L1s to concurrently control an unmodified guest's memory, VCPU, and I/O devices using a relatively simple interface with L0.
- We describe our implementation of Span virtualization by extending the nested virtualization support in KVM/QEMU [40] and show that Span virtualization can be implemented within existing hypervisors with moderate changes.
- We evaluate Span VMs running unmodified Linux and simultaneously using multiple L1 services including VM introspection, network monitoring, guest mirroring, and hypervisor refresh. We find that Span VMs perform comparably with nested VMs and within 0–20% of single-level VMs, across different configurations and benchmarks.

2 Solution Space

Table 1 compares possible solutions for providing multiple services to a guest. These are single-level virtualization, user space services, service VMs, nested virtualization, and Span.

First, like single-level and nested alternatives, Span virtualization provides L1s with control over the virtualized instruction set architecture (ISA) of the guest, which includes trapped instructions, memory mappings, VCPU scheduling, and I/O.

Unlike all alternatives, Span L1s support both full and partial guest control. Span L1s can range from full hypervisors that control all guest resources to specialized hypervisors that control only some guest resources.

Next, consider the impact of service failures. In single-level virtualization, failure of a privileged service impacts the L0 hypervisor, all coresident services, and all guests. For all other cases, the L0 hypervisor is protected from service failures because services are deprivileged. Furthermore, failure of a deprivileged service impacts only those guests to which the service is attached.

Next, consider the failure impact on coresident services. User space services are isolated by process-level isolation and hence protected from each other's failure. However, process-level isolation is only as strong as the user-level privileges with which the services run. Nested virtualization provides only one deprivileged service compartment. Hence services for the same guest must reside together in an L1, either in its user space or kernel. A service failure in a nested L1's kernel impacts all coresident services whereas a failure in its user

	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

Table 1: Alternatives for providing multiple services to a common guest, assuming one service per user space process, service VM, or Span L1.

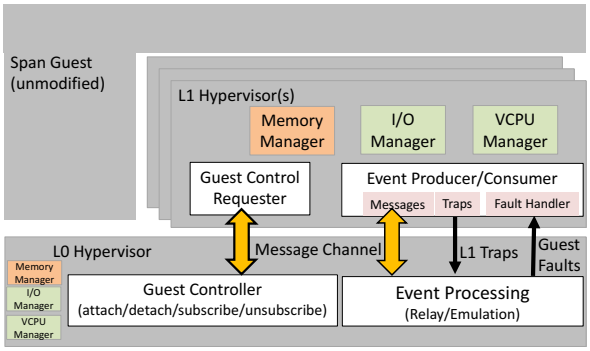


Figure 2: High-level architecture for Span virtualization.

space does not. Service VMs and Span virtualization isolate coresident services in individual VM-level compartments. Thus, failure of a service VM or Span L1 does not affect coresident services.

Finally, consider additional performance overhead over the single-level case. User space services introduce context switching overhead among processes. Service VMs introduce VM context switching overhead, which is more expensive. Nesting adds the overhead of emulating privileged guest operations in L1. Span virtualization uses nesting but supports partial guest control by L1s. Hence, nesting overhead applies only to the guest resources that an L1 controls.

3 Overview of Span Virtualization

The key design requirement for Span VMs is *transparency*. The guest OS and its applications should remain unmodified and oblivious to being simultaneously controlled by multiple hypervisors, which includes L0 and any attached L1s. Hence the guest sees a virtual resource abstraction that is indistinguishable from that of a traditional (single) hypervisor. For control of individual resources, we translate this requirement as follows.

- **Memory:** All hypervisors must have the same consistent view of the guest memory.
- **VCPUs:** All guest VCPUs must be controlled by one hypervisor at a given instant.
- **I/O Devices:** Different virtual I/O devices of the same guest may be controlled exclusively by different hypervisors at a given instant.
- **Control Transfer:** Control of guest VCPUs and/or virtual I/O devices can be transferred from one hypervisor to another, but only via L0.

Figure 2 shows the high-level architecture. A Span guest begins as a single-level VM on L0. One or more L1s can then attach to one or more guest resources and optionally subscribe with L0 for specific guest events.

Guest Control Operations: The Guest Controller in L0 supervises control over a guest by multiple L1s through the following operations.

- `[attach L1, Guest, Resource]`: Gives L1 control over the Resource in Guest. Resources include guest memory, VCPU, and I/O devices. Control over memory is shared among multiple attached L1s, whereas control over guest VCPUs and virtual I/O devices is exclusive to an attached L1. Attaching to guest VCPUs or I/O device resources requires attaching to the guest memory resource.
- `[detach L1, Guest, Resource]`: Releases L1's control over Resource in Guest. Detaching from the guest memory resource requires detaching from guest VCPUs and I/O devices.
- `[subscribe L1, Guest, Event, <GFN Range>]` Registers L1 with L0 to receive Event from Guest. The GFN Range option specifies the range of frames in the guest address space on which to track the memory event. Presently we support only memory event subscription. Other guest events of interest could include SYSENTER instructions, port-mapped I/O, etc.
- `[unsubscribe L1, Guest, Event, <GFN Range>]` Unsubscribes L1 Guest Event.

The Guest Controller also uses administrative policies to resolve apriori any potential conflicts over a guest control by multiple L1s. While this paper focuses on mechanisms rather than specific policies, we note that the problem of conflict resolution among services is not unique to Span. Alternative techniques also need ways to prevent conflicting services from controlling the same guest.

Isolation and Communication: Another design goal is to compartmentalize L1 services, from each other and from L0. First, L1s must have lower execution privilege compared to L0. Secondly, L1s must remain isolated from each other. These two goals are achieved by deprivileging L1s using nested virtualization and executing them as separate guests on L0. Finally, L1s must remain unaware of each other during execution. This goal is achieved by requiring L1s to receive any subscribed guest events that are generated on other L1s only via L0.

There are two ways that L0 communicates with L1s: implicitly via *traps* and explicitly via *messages*. Traps allow L0 to transparently intercept certain memory management operations by L1 on the guest. Explicit messages allow an L1 to directly request guest control from L0. An *Event Processing* module in L0 traps runtime updates to guest memory mappings by any L1 and synchronizes guest mappings across different L1s. The event processing module also relays guest memory faults that need to be handled by L1. A bidirectional *Message Channel* relays explicit messages between L0 and L1s including attach/detach requests, memory event subscription/notification, guest I/O requests, and virtual interrupts. Some explicit messages, such as guest I/O requests and virtual interrupts, could be replaced with implicit traps. Our choice of which to use is largely based on ease of implementation on a case-by-case basis.

Continuous vs. Transient Control: Span virtualization allows L1's control over guest resources to be either *continuous* or *transient*. Continuous control means that an L1 exerts uninterrupted control over one or more guest resources for an extended period of time. For example, an intrusion detection service in L1 that must monitor guest system calls, VM exits, or network traffic, would require continuous control of guest memory, VCPUs, and network device. Transient control means that an L1 acquires full control over guest resources for a brief duration, provides a short service to the guest, and releases guest control back to L0. For instance, an L1 that periodically checkpoints the guest would need transient control of guest memory, VCPUs, and I/O devices.

4 Memory Management

A Span VM has a single guest physical address space which is mapped into the address space of all attached L1s. Thus any memory write on a guest page is immediately visible to all hypervisors controlling the guest. Note that all L1s have the same visibility into the guest memory due to the horizontal layering of Span virtualization, unlike the vertical stacking of nested virtualization, which somewhat obscures the guest to lower layers.

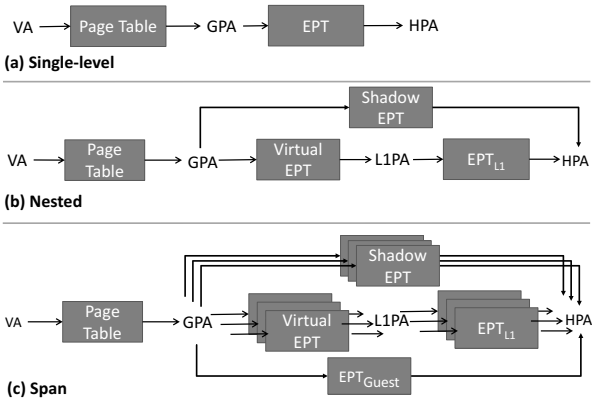


Figure 3: Memory translation for single-level, nested, and Span VMs. VA = Virtual Address; GPA = Guest Physical Address; L1PA = L1 Physical Address; HPA = Host Physical Address.

4.1 Traditional Memory Translation

In modern x86 processors, hypervisors manage the physical memory that a guest can access using a virtualization feature called *Extended Page Tables* (EPT) [37], also called Nested Page Tables in AMD-V [5].

Single-level virtualization: Figure 3(a) shows that for single-level virtualization, the guest page tables map virtual addresses to guest physical addresses (VA to GPA in the figure). The hypervisor uses an EPT to map guest physical addresses to host physical addresses (GPA to HPA). Guest memory permissions are controlled by the combination of permissions in guest page table and EPT.

Whenever the guest attempts to access a page that is either not present or protected in the EPT, the hardware generates an *EPT fault* and traps into the hypervisor, which handles the fault by mapping a new page, emulating an instruction, or taking other actions. On the other hand, the hypervisor grants complete control over the traditional paging hardware to the guest. A guest OS is free to maintain the mappings between its virtual and guest physical address space and update them as it sees fit, without trapping into the hypervisor.

Nested virtualization: Figure 3(b) shows that for nested virtualization, the guest is similarly granted control over the traditional paging hardware to map virtual addresses to its guest physical address space. L1 maintains a *Virtual EPT* to map the guest pages to pages in L1's physical addresses space, or L1 pages. Finally, one more translation is required: L0 maintains EPT_{L1} to map L1 pages to physical pages. However, x86 processors can translate only two levels of addresses in hardware, from guest virtual to guest physical to host physical address. Hence the Virtual EPT maintained by L1 needs to be *shadowed* by L0, meaning that the Virtual EPT and EPT_{L1} must be compacted by L0 during runtime into a

Shadow EPT that directly maps guest pages to physical pages. To accomplish this, manipulations to the Virtual EPT by L1 trigger traps to L0. Whenever L1 loads a Virtual EPT, L0 receives a trap and activates the appropriate Shadow EPT. This style of nested page table management is also called *multi-dimensional paging* [10].

EPT faults on guest memory can be due to (a) the guest accessing its own pages that have invalid Shadow EPT entries, and (b) the L1 directly accessing guest pages that have invalid EPT_{L1} entries to perform tasks such as I/O processing and VM introspection (VMI). Both kinds of EPT faults are first intercepted by L0. L0 examines a Shadow EPT fault to further determine whether it is due to a invalid Virtual EPT entry; such faults are forwarded to L1 for processing. Otherwise, faults due to invalid EPT_{L1} entries are handled by L0.

Finally, an L1 may modify the Virtual EPT it maintains for a guest in the course of performing its own memory management. However, since the Virtual EPT is shadowed by L0, all Virtual EPT modifications cause traps to L0 for validation and a Shadow EPT update.

4.2 Memory Translation for Span VMs

In Span virtualization, L0 extends nested EPT management to guests that are controlled by multiple hypervisors. Figure 3(c) shows that a Span guest has multiple Virtual EPTs, one per L1 that is attached to the guest. When an L1 acquires control over a guest's VCPUs, the L0 shadows the guest's Virtual EPT in the L1 to construct the corresponding Shadow EPT, which is used for memory translations. In addition, an EPT_{Guest} is maintained by L0 for direct guest execution on L0. A guest's memory mappings in Shadow EPTs, the EPT_{Guest}, and the EPT_{L1} are kept synchronized by L0 upon page faults so that every attached hypervisor sees a consistent view of guest memory. Thus, a guest virtual address leads to the same host physical address irrespective of the Shadow EPT used for the translation.

4.3 Memory Attach and Detach

A Span VM is initially created directly on L0 as a single-level guest for which the L0 constructs a regular EPT. To attach to the guest memory, a new L1 requests L0, via a hypercall, to map guest pages into its address space.

Figure 4 illustrates that L1 reserves a range in the L1 physical address space for guest memory and then informs L0 of this range. Next, L1 constructs a Virtual EPT for the guest which is shadowed by L0, as in the nested case. Note that the reservation in L1 physical address space does not immediately allocate physical memory. Rather, physical memory is allocated lazily upon guest memory faults. L0 dynamically populates the reserved address range in L1 by adjusting the mappings in EPT_{L1}

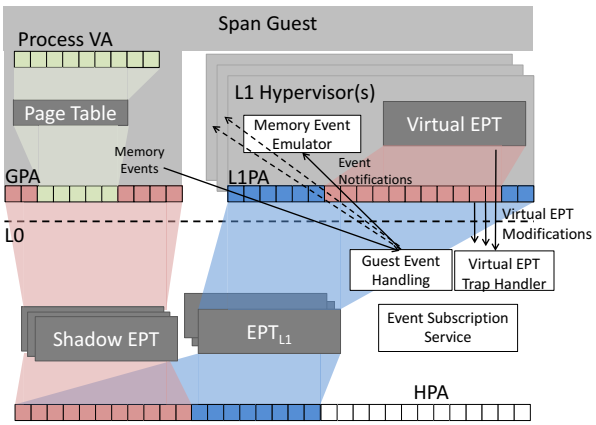


Figure 4: Span memory management overview.

and the Shadow EPT. A memory-detach operation correspondingly undoes the EPT_{L1} mappings for guest and releases the reserved L1 address range.

4.4 Synchronizing Guest Memory Maps

To enforce a consistent view of guest memory across all L1s, L0 synchronizes memory mappings upon two events: EPT faults and Virtual EPT modifications.

Fault handling for Span VMs extends the corresponding mechanism for nested VMs described earlier in Section 4.1. The key difference in the Span case is that L0 first checks if a host physical page has already been mapped to the faulting guest page. If so, the existing physical page mapping is used to resolve the fault, else a new physical page is allocated.

As with the nested case, modifications by an L1 to establish Virtual EPT mappings trap to a Virtual EPT trap handler in L0, shown in Figure 4. When the handler receives a trap due to a protection modification, it updates each corresponding EPT_{L1} with the new least-permissive combination of page protection. Our current prototype allows protection modifications but disallows changes to established GPA-to-L1PA mappings to avoid having to change mappings in multiple EPTs.

4.5 Memory Event Subscription

An L1 attached to a guest may wish to monitor and control certain memory-related events of the guest to provide a service. For instance, an L1 that provides live checkpointing or guest mirroring may need to perform *dirty page tracking* in which pages to which the guest writes are periodically recorded so they can be incrementally copied. As another example, an L1 performing intrusion detection using VM introspection might wish to monitor a guest's attempts to execute code from certain pages.

In Span virtualization, since multiple L1s can be attached to a guest, the L1 controlling the guest's VCPUs may differ from the L1s requiring the memory event notification. Hence L0 provides a *Memory Event Subscrip-*

tion interface to enable L1s to independently subscribe to guest memory events. An L1 subscribes with L0, via the message channel, requesting notifications when a specific type of event occurs on certain pages of a given guest. When the L0 intercepts the subscribed events, it notifies all L1 subscribers via the message channel. Upon receiving the event notification, a memory event emulator in each L1, shown in Figure 4, processes the event and responds back to L0, either allowing or disallowing the guest's memory access which triggered the event. The response from the L1 also specifies whether to maintain or discontinue the L1's event subscription on the guest page. For example, upon receiving a write event notification, an L1 that performs dirty page tracking will instruct L0 to allow the guest to write to the page, and cancel the subscription for future write events on the page, since the page has been recorded as being dirty. On the other hand, an intrusion detection service in L1 might disallow write events on guest pages containing kernel code and maintain future subscription. L0 concurrently delivers event notifications to all L1 subscribers. Guest memory access is allowed to proceed only if all subscribed L1s allow the event in their responses.

To intercept a subscribed memory event on a page, the L0 applies the event's mask to the corresponding EPT_{L1} entry of each L1 attached to the guest. Updating EPT_{L1} prompts L0 to update the guest's Shadow EPT entry with the mask, to capture guest-triggered memory events. Updating EPT_{L1} entries also captures the events resulting from direct accesses to guest memory by an L1 instead of the guest. For instance, to track write events on a guest page, the EPT entry could be marked read-only after saving the original permissions for later restoration.

5 I/O Control

In this work, guests use paravirtual devices [54, 6] which provide better performance than device emulation [59] and provide greater physical device sharing among guests than direct device assignment [11, 12, 50].

For single-level virtualization, the guest OS runs a set of paravirtual frontend drivers, one for each virtual device, including block and network devices. The hypervisor runs the corresponding backend driver. The frontend and the backend drivers communicate via a shared ring buffer to issue I/O requests and receive responses. The frontend places an I/O request in the ring buffer and notifies the backend through a *kick* event, which triggers a VM exit to the hypervisor. The backend removes the I/O request from the ring buffer, completes the request, places the I/O response in the ring buffer, and injects an I/O completion interrupt to the guest. The interrupt handler in the frontend then picks up the I/O response from the ring buffer for processing. For nested guests, paravirtual drivers are used at both levels.

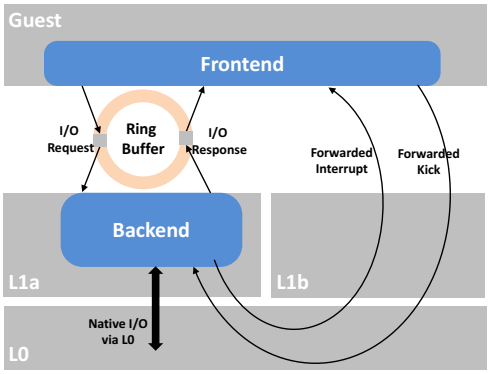


Figure 5: Paravirtual I/O for Span VMs. L1a controls the guest I/O device and L1b controls the VCPUs. Kicks from L1b and interrupts from L1a are forwarded via L0.

For Span guests, different L1s may control guest VCPUs and I/O devices. If the same L1 controls both guest VCPUs and the device backend then I/O processing proceeds as in the nested case. Figure 5 illustrates the other case, when different L1s control guest VCPUs and backends. L1a controls the backend and L1b controls the guest VCPUs. The frontend in the guest and backend in L1a exchange I/O requests and responses via the ring buffer. However, I/O kicks are generated by guest VCPUs controlled by L1b, which forward the kicks to L1a. Likewise, L1a forwards any virtual interrupts from the backend to L1b, which injects the interrupt to the guest VCPUs. Kicks from the frontend and virtual interrupts from the backend are forwarded between L1s via L0 using the message channel.

6 VCPU Control

In single-level virtualization, L0 controls the scheduling of guest VCPUs. In nested virtualization, L0 delegates guest VCPU scheduling to an L1. The L1 schedules guest VCPUs on its own VCPUs and L0 schedules the L1's VCPUs on PCPUs. This hierarchical scheduling provides the L1 some degree of control over customized scheduling for its guests.

Span virtualization can leverage either single-level or nested VCPU scheduling depending on whether the L0 or an L1 controls a guest's VCPUs. Our current design requires that all VCPUs of a guest be controlled by one of the hypervisors at any instant. However, control over guest VCPUs can be transferred between hypervisors if needed. When L0 initiates a Span VM, it initializes the all the VCPUs as it would for a single-level guest. After the guest boots up, the control of guest VCPUs can be transferred to/from an L1 using attach/detach operations.

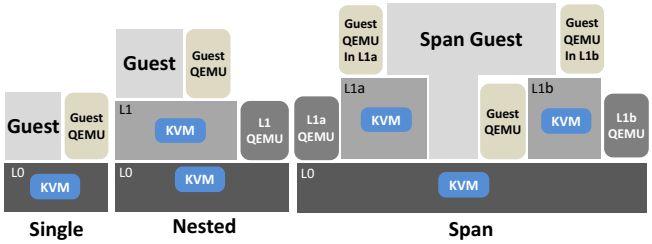


Figure 6: Roles of QEMU (Guest Controller) and KVM (hypervisor) for Single-level, Nested, and Span VMs.

7 Implementation Details

Platform and Modifications: Our prototype supports running an *unmodified* Linux guest as a Span VM in modes V_3, V_4, and V_5 from Figure 1. In our test setup, the guest runs Ubuntu 15.10 with Linux 4.2.0. The prototype for Span virtualization is implemented by modifying the KVM/QEMU nested virtualization support that is built into standard Linux distributions. Currently the implementation of L0 and all L1s uses modified KVM/QEMU hypervisors in Linux, specifically QEMU-1.2.0, kvm-kmod-3.14.2 and Linux 3.14.2. The modifications are different for the L0 and L1 layers. Ideally, we would prefer L1 to be unmodified to simplify its interface with L0. However, current hypervisors assume complete and exclusive guest control whereas Span allows L1s to exercise partial control over a subset of guest resources. Supporting partial guest control necessarily requires changes to L1 for attaching/detaching with a subset of guest resources and memory event subscription. In implementing L1 attach/detach operations on a guest, we tried, as much as possible, to reuse existing implementations of VM creation/termination operations.

Code size and memory footprint: Our implementation required about 2200 lines of code changes in KVM/QEMU, which is roughly 980+ lines in KVM and 500+ lines in QEMU for L0, 300+ in KVM and 200+ in QEMU for L1, and another 180+ in the virtio backend. We disabled unnecessary kernel components in both L0 and L1 implementations to reduce their footprint. When idle, L0 was observed to have 600MB usage at startup. When running an idle Span guest attached to an idle L1, L0's memory usage increased to 1756MB after excluding usage by the guest and the L1. The L1's initial memory usage, as measured from L0, was 1GB after excluding the guest footprint. This is an initial prototype to validate our ideas. The footprints of L0 and L1 implementations could be further reduced using one of many lightweight Linux distributions [14].

Guest Controller: A user-level control process, called the Guest Controller, runs on the hypervisor alongside each guest. In KVM/QEMU, the Guest Controller is a QEMU process which assists the KVM hypervisor with various control tasks on a guest, including guest initialization, I/O emulation, checkpointing, and migration. Figure 6 shows the position of the Guest Controller in different virtualization models. In both single-level and nested virtualization, there is only one Guest Controller per guest, since each guest is completely controlled by one hypervisor. Additionally, in the nested case, each L1 has its own Guest Controller that runs on L0. In Span virtualization, each guest is associated with multiple Guest Controllers, one per attached hypervisor. For instance, the Span Guest in Figure 6 is associated with three Guest Controllers, one each on L0, $L1_a$, and $L1_b$. During attach/detach operations, the Guest Controller in an L1 initiates the mapping/unmapping of guest memory into the L1's address space and, if needed, acquires/releases control over the guest's VCPU and virtual I/O devices.

Paravirtual I/O Architecture: The Guest Controller also performs I/O emulation of virtual I/O devices controlled by its corresponding hypervisor. The paravirtual device model described in Section 5 is called *virtio* in KVM/QEMU [54]. For nested guests, the virtio drivers are used at two levels: once between L0 and each L1 and again between an L1 and the guest. This design is also called *virtio-over-virtio*. A kick is implemented in virtio as a software trap from the frontend leading to a VM exit to KVM, which delivers the kick to the Guest Controller as a signal. Upon I/O completion, the Guest Controller requests KVM to inject a virtual interrupt into the guest. Kicks and interrupts are forwarded across hypervisors using the message channel. Redirected interrupts are received and injected into the guest by a modified version of KVM's virtual IOAPIC code.

VCPU Control: The Guest Controllers in different hypervisors communicate with the Guest Controller in L0 to acquire or relinquish guest VCPU control. The Guest Controller represents each guest VCPU as a user space thread. A newly attached L1 hypervisor does not initialize guest VCPU state from scratch. Rather, the Guest Controller in the L1 accepts a checkpointed guest VCPU state from its counterpart in L0 using a technique similar to that used for live VM migration between physical hosts. After guest VCPU states are transferred from L0 to L1, the L1 Guest Controller resumes the guest VCPU threads while the L0 Guest Controller pauses its VCPU threads. A VCPU detach operation similarly transfers a checkpoint of guest VCPU states from L1 to L0. Transfer of guest VCPU states from one L1 to another is presently accomplished through a combination of detaching the source L1 from the guest VCPUs followed by attaching to the destination L1 (although a direct transfer could be potentially more efficient).

Message Channel: The message channel between L0 and each L1 is implemented using a combination of hypercalls and UDP messages. Hypercalls from an L1 to L0 are used for attach/detach operations on guest

memory. UDP messages between an L1 and L0 are used for relaying I/O requests, device interrupts, memory subscription messages, and attach/detach operations on guest VCPU and I/O devices. UDP messages are presently used for ease of implementation and will be replaced by better alternatives such as hypercalls, callbacks, or shared buffers.

8 Evaluation

We first demonstrate unmodified Span VMs that can simultaneously use services from multiple L1s. Next we investigate how Span guests perform compared to traditional single-level and nested guests. Our setup consists of a server containing dual six-core Intel Xeon 2.10 GHz CPUs, 128GB memory and 1Gbps Ethernet. The software configurations for L0, L1s, and Span guests are as described earlier in Section 7. Each data point is a mean (average) over at least five or more runs.

8.1 Span VM Usage Examples

We present three examples in which a Span VM transparently utilizes services from multiple L1s. An unmodified guest is controlled by three coresident hypervisors, namely, L0, L1a, and L1b.

Use Case 1 – Network Monitoring and VM Introspection: In the first use case, the two L1s passively examine the guest state, while L0 supervises resource control. L1a controls the guest's virtual network device whereas L1b controls the guest VCPUs. L1a performs network traffic monitoring by running the `tcpdump` tool to capture packets on the guest's virtual network interface. Here we use `tcpdump` as a stand-in for other more complex packet filtering and analysis tools.

L1b performs VM introspection (VMI) using a tool called Volatility [3] which continuously inspects a guest's memory using a utility such as `pmemsave` to extract an accurate list of all processes running inside the guest. The guest OS is infected by a rootkit, Kernel Beast [38], which can hide malicious activity and present an inaccurate process list to the compromised guest. Volatility, running in L1b, can nevertheless extract an accurate guest process list using VM introspection.

Figure 7 shows a screenshot, where the top window shows the `tcpdump` output in L1a, specifically the SSH traffic from the guest. The bottom right window shows that the rootkit KBeast in the guest OS hides a process `evil`, i.e. it prevents the process `evil` from being listed using the `ps` command in the guest. The bottom left window shows that Volatility, running in L1b, successfully detects the process `evil` hidden by the KBeast rootkit in the guest.

This use case highlights several salient features of our design. First, an unmodified guest executes correctly

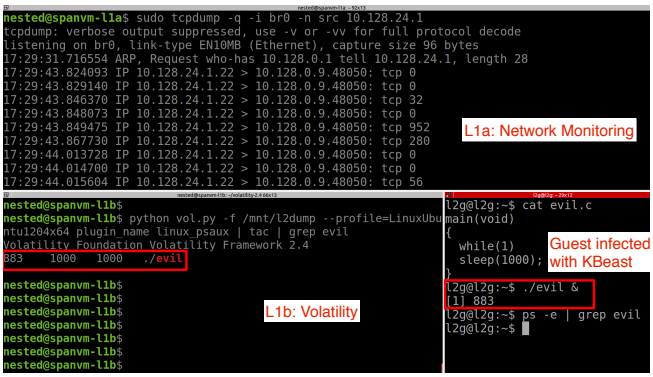
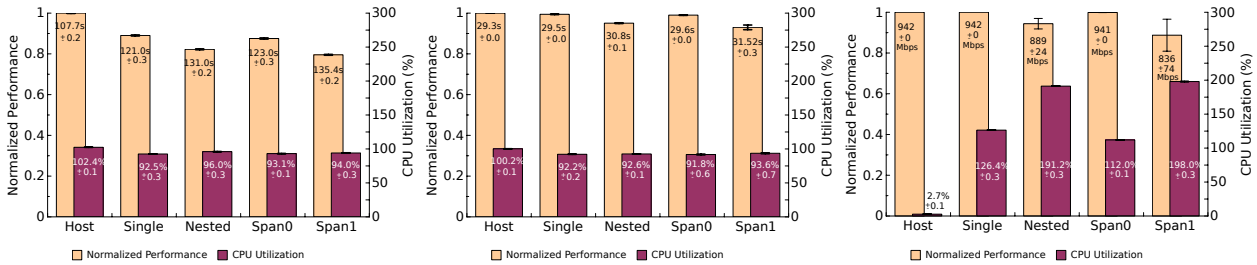


Figure 7: A screenshot of Span VM simultaneously using services from two L1s.

even though its resources are controlled by multiple hypervisors. Second, an L1 can transparently examine guest memory. Third, an L1 controlling a guest virtual device (here network interface) can examine all I/O requests specific to the device even if the I/O requests are initiated from guest VCPUs controlled by another hypervisor. Thus an I/O device can be delegated to an L1 that does not control the guest VCPUs.

Use Case 2 – Guest Mirroring and VM Introspection: In this use case, we demonstrate an L1 that subscribes to guest memory events from L0. Hypervisors can provide a high availability service that protects unmodified guests from a failure of the physical machine. Solutions, such as Remus [24], typically work by continually transferring live incremental checkpoints of the guest state to a remote backup server, an operation that we call *guest mirroring*. When the primary VM fails, its backup image is activated, and the VM continues running as if failure never happened. To checkpoint incrementally, hypervisors typically use a feature called dirty page tracking. The hypervisor maintains a dirty bitmap, i.e. the set of pages that were dirtied since the last checkpoint. The dirty bitmap is constructed by marking all guest pages read-only in the EPT and recording dirtied pages upon write traps. The pages listed in the dirty bitmap are incrementally copied to the backup server.

As a first approximation of guest mirroring, we modified the pre-copy live migration code in KVM/QEMU to periodically copy all dirtied guest pages to a backup server at a given frequency. In our setup, L1a mirrors a Span guest while L1b runs Volatility and controls guest VCPUs. L1a uses memory event subscription to track write events, construct the dirty bitmap, and periodically transfer any dirty pages to the backup server. We measured the average bandwidth reported by the iPerf [1] client benchmark running in the guest when L1a mirrors the guest memory at different frequencies. When guest mirroring happens every 12 seconds, iPerf delivers 800Mbps average bandwidth which is about the same as



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

Figure 8: No-op Mode: Normalized performance when no services run in host, L0, or L1s. The L0 controls the virtio block and network devices of the guest.

with a nested guest. When guest mirroring happens every second, the average bandwidth drops to 600Mbps, indicating a 25% performance impact of event subscription at very high mirroring frequencies.

Use Case 3 – Proactive Refresh: Hypervisor-level services may contain latent bugs, such as memory leaks, or other vulnerabilities that become worse over time, making a monolithic hypervisor unreliable for guests. Techniques like Microreboot[18] and ReHype[43] have been proposed to improve hypervisor availability, either proactively or post-failure. We have already seen how Span virtualization can compartmentalize unreliable hypervisor-level services in an isolated deprivileged L1. Here, we go one step further and proactively replace unreliable L1s with a fresh reliable instance while the guest and the base L0 hypervisor keep running. In our setup, an old L1 (L1a) was attached to a 3GB Span guest. To perform hypervisor refresh, we attached a new pre-booted replacement hypervisor (L1b) to the guest memory. Then L1a was detached from the guest by transferring guest VCPU and I/O devices to L1b via L0. In our implementation, the entire refresh operation from attaching L1b to detaching L1a completes on the average within 740ms. Of this, 670ms are spent in attaching L1b to guest memory while the guest is running. The remaining 70ms is the guest downtime due to the transfer of VCPU and I/O states. Thus Span virtualization achieves sub-second L1 refresh latency. If we attach the replacement L1b to guest memory well in advance, then the VCPU and I/O state transfer can be triggered on-demand by events, such as unusual memory pressure or CPU usage, yielding sub-100ms guest downtime and event response latency. In contrast, using pre-copy [22] to live migrate a guest from L1a to L1b can take several seconds depending on guest size and workload [65].

8.2 Macro Benchmarks

Here we compare the performance of macro benchmarks in Span VM against a native host (no hypervisor), single-level, and nested guests. Table 2 shows the memory and processor assignments at each layer for each case. The guest always has 3GB memory and one VCPU. L0 always has 128GB and 12 physical CPU cores. In the nested configuration, L1 has 16GB memory and 8 VCPUs. The guest VCPU in the Span0 configuration is controlled by L0, and in Span1 by an L1. Finally, in both Span0 and Span1, L1a and L1b each have 8GB of memory and 4VCPUs, so their sums match the L1 in the nested setting.

	L0		L1		L2	
	Mem	CPUs	Mem	VCPUs	Mem	VCPUs
Host	128GB	12	N/A	N/A	N/A	N/A
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

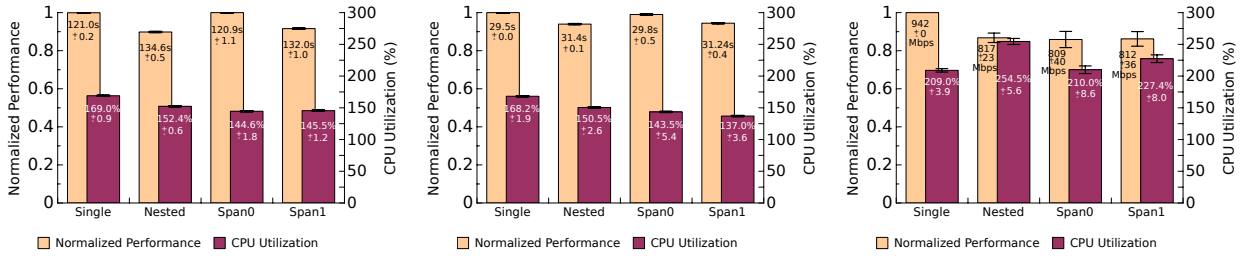
Table 2: Memory and CPU assignments for experiments.

The guest runs one of the following three benchmarks: (a) **Kernbench** [41] compiles the Linux kernel. (b) **Quicksort** sorts 400MB of data in memory. (c) **iPerf** [1] measures network bandwidth to another host.

The benchmarks run in two modes: *No-op Mode*, when no hypervisor-level services run, and *Service Mode*, when network monitoring and VM introspection services run at either L0 or L1s. The figures report each benchmark's normalized performance against the best case and system-wide average CPU utilization, which is measured in L0 using the `atop` command each second during experiments.

From Figures 8(a) and (b) and Figures 9(a) and (b), in both modes for Kernbench and Quicksort, Span0 performs comparably with the single-level setting and Span1 performs comparably with the nested setting, with similar CPU utilization.

For iPerf in No-op mode (Figure 8(c)), we observe that the Span1 guest experiences about 6% degradation over the nested guest with notable bandwidth fluctuation and 7% more CPU utilization. This is because the guest's VCPU in Span1 is controlled by L1a, but the guest's network device is controlled by L0. Hence, guest I/O requests (kicks) and responses are forwarded from L1a to L0 via the message channel. The message channel is currently implemented using UDP messages, which compete with guest's iPerf client traffic on the L1's vir-



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

Figure 9: Service Mode: Normalized performance with hypervisor-level services network monitoring and Volatility. For single-level, L0 runs both services. For nested, L1 runs both services. For Span0 and Span1, L1a runs network monitoring and controls the guest's network device; L1b runs Volatility; L0 controls guest's block device.

tio network interface with L0. We observed that if L1a controls the guest network device as well, then iPerf in the Span1 guest performs as well as in the nested guest.

For iPerf in service mode (Figure 9(c)), nested, Span0, and Span1 guests perform about 14–15% worse than the single-level guest, due to the combined effect of virtio-over-virtio overhead and `tcpdump` running in L1a. Further, for Span0, the guest VCPU is controlled by L0 whereas the network device is controlled by L1a. Thus forwarding of I/O kicks and interrupts between L0 and L1a via the UDP-based message channel balances out any gains from having guest VCPUs run on L0.

Figure 8(c) shows that the average CPU utilization increases significantly for iPerf in no-op mode – from 2.7% for the native host to 100+% for the single-level and Span0 configurations and 180+% for the nested and Span1 configurations. The increase appears to be due to the virtio network device implementation in QEMU, since we observed this higher CPU utilization even with newer versions of (unmodified) QEMU (v2.7) and Linux (v4.4.2). Figures 8(c) and 9(c) also show higher CPU utilization for the nested and Span1 cases compared to the single-level case. This is because guest VCPUs are controlled by L1s in the nested and Span1 cases, making nested VM exits more expensive.

8.3 Micro Benchmarks

Attach Operation: Figure 10 shows the time taken to attach an L1 to a guest's memory, VCPU, and I/O devices as the guest memory size is increased. The time taken to attach memory of a 1GB Span guest is about 220ms. Memory attach overhead increases with guest size because each page that L1 has allocated for Span needs to be remapped to the Span physical page in L0.

Attaching VCPUs to one of the L1s takes about 50ms. Attaching virtual I/O devices takes 135ms. When I/O control has to be transferred between hypervisors, the VCPUs need to be paused. The VCPUs could be running on any of the L1s and hence L0 needs to coordinate pausing and resuming the VCPUs during the transfer. The

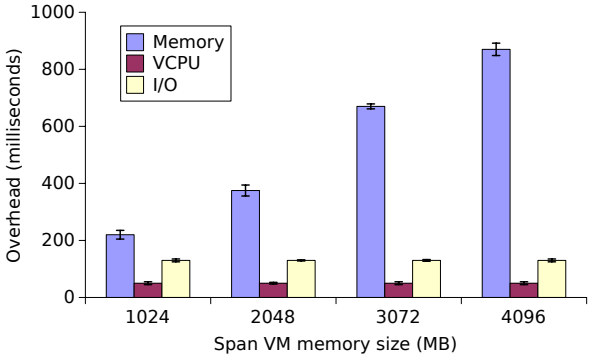


Figure 10: Overhead of attaching an L1 to a guest.

	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

Table 3: Low-level latencies(μs) in Span virtualization.

detach operation for VCPUs and I/O devices has similar overhead.

Page Fault Servicing: Table 3 shows the latency of page fault handling and message channel. We measured the average service times for EPT faults in Span at both levels of nesting. It takes on the average 3.3μs to resolve a fault caused against EPT_{L1} and on the average 24.1μs to resolve a fault against the Virtual EPT. In contrast, the corresponding values measured for the nested case are 2.8μs and 23.3μs. For the single-level case, EPT-fault processing takes 2.4μs. The difference is due to the extra synchronization work in the EPT-fault handler in L0.

Message Channel and Memory Events: The message channel is used in Span virtualization to exchange events and requests between L0 and L1s. It takes on the average 53μs to send a message between L0 and an L1. We also measured the overhead of notifying L1 subscribers from L0 for write events on a guest page. Without any subscribers, the write-fault processing takes on the average 3.5μs in L0. Notifying the write event over

the message channel from L0 to an L1 subscriber adds around 100μs, including a response from L1.

9 Related Work

Here, we review prior work on user space services, service VMs, and nested virtualization. We build upon earlier discussion of their relative merits in Section 2.

User space Services: Microkernels and library operating systems have a long history [44, 13, 28, 35] of providing OS services in user space. μDenali [63] allows programmers to use event interposition to extend the hypervisor with new user-level services such as disk and network I/O. In the KVM/QEMU [40] platform, each guest is associated with a dedicated user space management process, namely QEMU. A single QEMU process bundles multiple services for its guest such as VM launch/exit/pause, paravirtual I/O, migration, and checkpointing. One can associate different variants of QEMU with different guests, allowing some degree of service customization. However, QEMU's interface with the KVM hypervisor is large, consisting of system calls, signals, and shared buffers with the kernel, which increases the KVM hypervisor's exposure to potentially untrusted services. Also, while user space services can map guest memory and control paravirtual I/O, they lack direct control over low-level guest resources such as EPT mappings and VCPU scheduling, unlike nesting and Span.

Service VMs: Another option is to provide guest services via specialized Service VMs that run alongside the guest. For instance, the Xen [4] platform runs a trusted service VM called Dom0 which runs paravirtualized Linux, controls all guests via hypercalls to the Xen hypervisor, and provides guests with services related to lifecycle management and I/O. To avoid a single point of failure or vulnerability, the Xoar [47, 23] project proposed decomposing Dom0 into smaller service domains, one per service, that can be replaced or restarted. Possible support for third-party service domains has been discussed [16], but its status is unclear. Nova [58] minimizes the size of the hypervisor by implementing the VMM, device drivers, and special-purpose applications in user space. Self-service clouds [17] allows users to customize control over services used by their VMs on untrusted clouds. Services, such as storage and security, can be customized by privileged service domains, whereas the hypervisor controls all low-level guest resources, such as VCPUs and EPT mappings.

Nested Virtualization: Nested virtualization was originally proposed and refined in the 1970s [32, 33, 51, 8, 9, 48] and has experienced renewed interest in recent years [29, 34, 10]. Recent support [25], such as VMCS shadowing [62] and direct device assignment [67] aim to reduce nesting overheads related to VM exits and I/O.

Nesting enables vertical stacking of two layers of hypervisor-level services. Third parties such as Ravello [2] and XenBlanket [66, 57] leverage nesting to offer hypervisor-level services (in an L1) over public cloud platforms (L0) such as EC2 and Azure, often pitching their service as a way to avoid lock-in with a cloud provider. However, this model also leads to a different level of lock-in, where a guest is unable use services from more than one third party. Further, these third-party services are not fully trusted by the base hypervisor (L0) of the cloud provider, necessitating the use of nesting, rather than user space services. Span virtualization prevents guest lock-in at all levels by adding support for multiple third-party L1s to concurrently service a guest, while maintaining the isolation afforded by nesting.

Ephemeral virtualization [65] combines nesting and optimized live migration [22, 36] to enable transient control over guest by L1s. L1s and L0 take turns exchanging full control over the guest by co-mapping its memory. In contrast, Span allows multiple L1s to concurrently exercise either full or partial control over a guest, in either continuous or transient modes.

10 Conclusions

A rich set of hypervisor-level services have been proposed in recent years, such as VM introspection, high availability, live patching, and migration. However, in modern cloud platforms, a sole controlling hypervisor continues to host all such services. Specifically, support for third-parties to offer hypervisor-level services to guests is lacking. We presented a new approach, called *Span virtualization*, which leverages nesting to enable multiple coresident, but isolated, hypervisors to control and service a common guest. Our prototype of Span virtualization on the KVM/QEMU platform can support unmodified guests which simultaneously use multiple services that augment the base hypervisor's functionality. Span guests achieve performance comparable to traditional nested guests. Looking ahead, we believe that the opportunity for a cloud-based ecosystem of hypervisor-level services is large, including security services, cross-cloud portability, custom schedulers, virtual devices, and high availability.

11 Acknowledgement

This work was funded in part by the National Science Foundation via awards 1527338 and 1320689. We thank our shepherd, Nadav Amit, and all reviewers for insightful feedback; Umesh Deshpande, Spoorti Doddamani, Michael Hines, Hani Jamjoom, Siddhesh Phadke, and Piush Sinha, for discussions, implementation, and evaluation; and the Turtles Project [10] authors for inspiration.

References

[1] iPerf: The Network Bandwidth Measurement Tool. http://iperf.fr/.

[2] Ravello Systems. https://www.ravellosystems.com/.

[3] Volatility Framework. http://www.volatilityfoundation.org/.

[4] Xen Hypervisor. http://www.xen.org/.

[5] AMD. AMD Virtualization (AMD-V). http://www.amd.com/en-us/solutions/servers/virtualization.

[6] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. of SOSP* (Bolton Landing, NY, USA, 2003), pp. 164–177.

[7] BEHAM, M., VLAD, M., AND REISER, H. Intrusion detection and honeypots in nested virtualization environments. In *Proc. of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)* (Budapest, Hungary, June 2013).

[8] BELPAIRE, G., AND HSU, N.-T. Formal properties of recursive virtual machine architectures. In *Proc. of SOSP, Austin, Texas, USA* (1975), pp. 89–96.

[9] BELPAIRE, G., AND HSU, N.-T. Hardware architecture for recursive virtual machines. In *Proc. of Annual ACM Conference* (1975), pp. 14–18.

[10] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The Turtles project: Design and implementation of nested virtualization. In *Proc. of Operating Systems Design and Implementation* (2010).

[11] BEN-YEHUDA, M., MASON, J., XENIDIS, J., KRIEGER, O., VAN DOORN, L., NAKAJIMA, J., MALLICK, A., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symposium* (July 2006).

[12] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. The price of safety: Evaluating IOMMU performance. In *Proc. of Ottawa Linux Symposium* (July 2007).

[13] BERSHAD, B. N., CHAMBERS, C., EGGERS, S., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. SPIN : An extensible microkernel for application-specific operating system services. *Proc. of ACM SIGOPS Operating Systems Review 29*, 1 (1995), 74–77.

[14] BHARTIYA, S. Best lightweight linux distros for 2017. https://www.linux.com/news/best-lightweight-linux-distros-2017.

[15] BROMIUM. https://www.bromium.com.

[16] BULPIN, J. Whatever happened to XenServer's Windsor architecture? https://xenserver.org/blog/entry/whatever-happened-to-xenserver-s-windsor-architecture.html.

[17] BUTT, S., LAGAR-CAVILLA, H. A., SRIVASTAVA, A., AND GANAPATHY, V. Self-service cloud computing. In *Proc. of ACM Conference on Computer and Communications Security(CCS)* (Raleigh, NC, USA, 2012).

[18] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot-a technique for cheap recovery. In *Proc. of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, USA, 2004), vol. 4, pp. 31–44.

[19] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P. Live updating operating systems using virtualization. In *Proc. of ACM International Conference on Virtual Execution Environments (VEE)* (Ottawa, Canada, June 2006).

[20] CITRIX. *XenDesktop*. https://www.citrix.com/products/xenapp-xendesktop/.

[21] CITRIX. *XenDesktop and The Evolution of Hardware-Assisted Server Technologies*. https://www.citrix.com/content/dam/citrix/en_us/documents/go/2015-edition-hosted-desktop.pdf.

[22] CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proc. of Network System Design and Implementation* (2005).

[23] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. of SOSP* (2011), pp. 189–202.

[24] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. In *Proc. of Networked Systems Design and Implementation, San Francisco, CA, USA* (2008).

[25] DAS, B., ZHANG, Y. Y., AND KISZKA, J. Nested virtualization: State of the art and future directions. In *KVM Forum* (2014).

[26] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proc. of 15th ACM conference on Computer and communications security (CCS)* (Alexandria, VA, USA, 2008), pp. 51–62.

[27] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002).

[28] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29(5), pp. 251–266.

[29] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels Meet Recursive Virtual Machines. In *Proc. OSDI, Seattle, Washington, USA* (1996), pp. 137–151.

[30] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network & Distributed Systems Security Symposium* (San Diego, CA USA, 2003).

[31] GEBHARDT, C., AND DALTON, C. LaLa: A Late Launch Application. In *Workshop on Scalable Trusted Computing, Chicago, Illinois, USA* (2009), pp. 1–8.

[32] GOLDBERG, R. P. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems, Cambridge, MA, USA* (1973), pp. 74–112.

[33] GOLDBERG, R. P. Survey of Virtual Machine Research. *Computer 7*, 6 (1974), 34–45.

[34] GRAF, A., AND ROEDEL, J. Nesting the virtualized world. In *Linux Plumbers Conference* (Sept. 2009).

[35] HAND, S., WARFIELD, A., FRASER, K., KOTSOVINOS, E., AND MAGENHEIMER, D. J. Are virtual machine monitors microkernels done right? In *Proc. of HotOS* (2005).

[36] HINES, M., DESHPANDE, U., AND GOPALAN, K. Post-copy live migration of virtual machines. *In SIGOPS Operating Systems Review* (July 2009), 14–26.

[37] INTEL CORP. *Intel 64 and IA-32 Architecture Software Developers Manual, Volume 3, System Programming Guide. Order number 325384.* April 2016.

[38] IPSECS. Kernel Beast. `http://core.ipsecs.com/rootkit/kernel-rootkit/`.

[39] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. *ACM Trans. Information Systems Security 13*, 2 (Mar. 2010), 1–28.

[40] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Proc. of Linux Symposium* (June 2007).

[41] KOLIVAS, C. Kernbench. `http://ck.kolivas.org/apps/kernbench/`.

[42] KOURAI, K., AND CHIBA, S. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *ACM/USENIX International Conference on Virtual Execution Environments* (2005), pp. 197 – 207.

[43] LE, M., AND TAMIR, Y. ReHype: enabling VM survival across hypervisor failures. In *Proc. of ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 63–74.

[44] LIEDTKE, J. On micro-kernel construction. *Proc. of ACM SIGOPS Operating Systems Review 29*, 5 (1995), 237–250.

[45] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. *Proc. of SIGARCH Comput. Archit. News 32*, 5 (Oct. 2004), 211–223.

[46] MCAFEE. Root Out Rootkits: An Inside Look at McAfee Deep Defender. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/mcafee-deep-defender-deepsafe-rootkit-protection-paper.pdf.

[47] MURRAY, D. G., MILOS, G., AND HAND, S. Improving Xen Security Through Disaggregation. In *Proc. of Virtual Execution Environments* (2008), pp. 151–160.

[48] OSISEK, D. L., JACKSON, K. M., AND GUM, P. H. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal 30*, 1 (Feb. 1991), 34–51.

[49] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. of IEEE Symposium on Security and Privacy* (2008), pp. 233 – 247.

[50] PCI SIG. Single Root I/O Virtualization and Sharing. http://www.pcisig.com/specifications/iov/single_root/.

[51] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Proc. of Communications of ACM 17*, 7 (July 1974), 412–421.

[52] REAL TIME SYSTEMS GMBH. *RTS Real-Time Hypervisor*. http://www.real-time-systems.com/products/index.php.

[53] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proc. of Recent Advances in Intrusion Detection* (2008), pp. 1–20.

[54] RUSSELL, R. Virtio: Towards a de-facto standard for virtual I/O devices. *Proc. of SIGOPS Operating Systems Review 42*, 5 (July 2008), 95–103.

[55] RUTKOWSKA, J. Subverting vista kernel for fun and profit. In *Blackhat* (Las Vegas, USA, Aug. 2006).

[56] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of ACM SIGOPS Operating Systems Review* (2007), vol. 41(6), pp. 335–350.

[57] SHEN, Z., JIA, Q., SELA, G.-E., RAINERO, B., SONG, W., VAN RENESSE, R., AND WEATHERSPOON, H. Follow the Sun Through the Clouds: Application Migration for Geographically Shifting Workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), pp. 141–154.

[58] STEINBERG, U., AND KAUER, B. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proc. of EuroSys*, pp. 209–222.

[59] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of USENIX Annual Technical Conference, Monterey, CA, USA* (2002).

[60] SUNEJA, S., ISCI, C., BALA, V., DE LARA, E., AND MUMMERT, T. Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud. In *Proc. of SIGMETRICS'14, Austin, TX, USA* (2014).

[61] TOLDINAS, J., RUDZIKA, D., ŠTUIKYS, V., AND ZIBERKAS, G. Rootkit Detection Experiment within a Virtual Environment. *Electronics and Electrical Engineering–Kaunas: Technologija*, 8 (2009), 104.

[62] WASSERMAN, O. Nested Virtualization: Shadow Turtles. In *KVM Forum, Edinburgh, Spain* (October 2013).

[63] WHITAKER, A., COX, R., AND SHAW, M. Constructing services with interposable virtual hardware. In *Proc. of First USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California, 2004).

[64] WIKIPEDIA. Phoenix Hyperspace. https://en.wikipedia.org/wiki/HyperSpace_(software).

[65] WILLIAMS, D., HU, Y., DESHPANDE, U., SINHA, P. K., BILA, N., GOPALAN, K., AND JAMJOOM, H. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. In *Proc. of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2016).

[66] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The Xen-Blanket: Virtualize once, run everywhere. In *EuroSys, Bern, Switzerland* (2012).

[67] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Tech. rep., IBM Research, 2008.

[68] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 203–216.