VFresh: Fast and Live Hypervisor Replacement

Abstract

Hypervisors are increasingly complex and must be often updated for applying security patches, bug fixes, and feature upgrades. However, in a virtualized cloud infrastructure, updates to an operational hypervisor can be highly disruptive. Before being updated, virtual machines (VMs) running on a hypervisor must be either migrated away or shut down, resulting in downtime, performance loss, and network overhead. We present a new technique, called VFresh, to transparently replace a running hypervisor with a new updated instance without disrupting any running VMs. A thin shim layer, called the hyperplexor, performs live hypervisor replacement by remapping guest memory to a new updated hypervisor on the same machine. The hyperplexor leverages nested virtualization for hypervisor replacement while minimizing nesting overheads during normal execution via direct device assignment and VM exit reductions. We present a prototype implementation of the hyperplexor on the KVM/QEMU platform that can perform live hypervisor replacement within 10ms. We also demonstrate how our hyperplexor-based approach can be used with CRIU-based process/container migration to perform sub-second replacement of the underlying operating system.

Keywords Hypervisor, Virtualization, Container, Live Migration

1 Introduction

Virtualization-based server consolidation is a common practice in today's cloud data centers [2, 19, 36]. Hypervisors host multiple virtual machines (VMs) on a single physical host to achieve benefits such as improved utilization (hence, better return-on-investment) and agility in resource provisioning required for modern cloud applications [3, 5–7, 40]. Hypervisors must be often updated or replaced for various purposes, such as for applying security/bug fixes [18, 33] adding new features [11, 20], or simply for software rejuvenation [32] to reset the effects of any unknown memory leaks or other latent bugs.

Conference'17, Washington, DC, USA

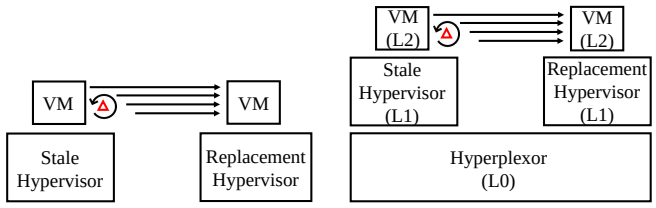
DOI: 10.1145/nnnnnnn.nnnnnnn

Updating a hypervisor or OS usually requires a system reboot, especially in the cases of system failures and software aging. Live patching [34] can be used to perform some of these updates without rebooting, but it relies greatly on old hypervisor being patched, which can be buggy and unstable. To eliminate the need for a system reboot and mitigate service disruption, another approach is to live migrate the virtual machines (VM) from the current host to another host that runs the clean and updated hypervisor. Though widely used, live migrating [14, 21] tens or hundreds of VMs from one physical host to another, i.e. *inter-host live migration*, can lead to significant service disruption, long total migration time, and large migration-triggered network traffic, which can also affect other unrelated VMs [35].

We propose a faster and less disruptive approach to live hypervisor replacement by leveraging nested virtualization and *intra-host live migration*. In this paper, we present **VFresh**, a faster and less disruptive approach to live hypervisor replacement which transparently and quickly replaces an old hypervisor with a new instance on the same host while minimizing runtime overheads on running VMs. Using nested virtualization [10], a lightweight shim layer, called the *hyperplexor*, runs beneath the traditional full-fledged hypervisor on which VMs run. The new replacement hypervisor is instantiated as a guest atop the hyperplexor. Next, the states of all VMs are transferred from the stale hypervisor to the replacement hypervisor, a co-located new hypervisor via intra-host live VM migration.

Unfortunately, two major challenges must be tackled with this approach. First, existing live migration techniques [14, 21] incur significant memory copying overhead, even for intra-host VM transfers. Secondly, nested virtualization can degrade a VM's performance during normal execution of VMs when no hypervisor replacement is being performed. VFresh addresses these two challenges as follows.

First, instead of copying the VM's memory, the hyperplexor relocates the ownership of VM's memory pages from the old hypervisor to the new hypervisor. The hyperplexor records the mappings from VM's guest-physical and host-physical address space from the old hypervisor and uses them to reconstruct the VM's memory mappings on the replacement hypervisor. Most of the remapping operations are performed out of the critical path of the VM state transfer, leading to very low hypervisor replacement times of around 10ms, irrespective of the size of the VMs being relocated. In contrast, traditional intra-host VM transfer, involving memory copying, can take several seconds – 0.4s for an idle 4GB VM and 1.19s for a busy write-intensive VM, increasing linearly for larger VM sizes. For the same reason, VFresh also



a. Inter-host migration b. Nested migration on same host

Figure 1. Replacing hypervisor using (a) Inter-host migration (b) Nested migration on same host

scales well when remapping multiple VMs to the replacement hypervisor.

VFresh addresses the second challenge of nesting overhead as follows. In comparison with the traditional single-level virtualization setup, where the hypervisor directly controls the hardware, nested virtualization introduces additional overheads, especially for I/O virtualization. Hence VFresh includes a number of optimizations to minimize nesting overheads, allowing the hypervisor and its VMs to execute mostly without hyperplexor intervention during normal operations. Specifically, VFresh uses direct device assignment (VT-d) for emulation-free I/O path to the hypervisor, dedicates physical CPUs for the hypervisor to reduce scheduling overheads, and eliminates common VM exits from the hypervisor to hyperplexor, such as for CPU idling and external device interrupts.

Finally, as a lightweight alternative to VMs, containers [17, 42–44] can be used to consolidate multiple processes. We demonstrate that the hyperplexor-based approach can also be applied for *live replacement of an operating system (OS)* beneath containers and processes. Specifically, we show how sub-second live OS replacement can be performed by using a combination of the hyperplexor-based memory remapping mechanism and a well-known process migration tool, CRIU [47]. In this case, the hyperplexor runs as a thin shim layer (hypervisor) beneath a traditional OS, which now runs in a low-overhead VM. For live OS replacement, the hyperplexor provisions a new VM, running the replacement OS, on the same physical host and transfers process/container memory mappings obtained via the CRIU tool.

In the rest of this paper, we first demonstrate the quantitative overheads of VM migration-based hypervisor replacement, followed by the design, implementation, and evaluation of VFresh for VMs and containers, and finally discussion of related work and conclusions.

2 Problem Demonstration

In this section, we examine the performance of traditional live migration for hypervisor replacement to motivate the need for a faster remapping-based mechanism.

2.1 Using pre-copy for hypervisor replacement

Inter-Host Live VM Migration: To refresh a hypervisor, a traditional approach is to live migrate VMs from their current host to another host (or hosts) having an updated hypervisor. As shown in Figure 1(a), we can leverage the state-of-the-art pre-copy live VM migration technique. Pre-copy VM live migration consists of three major phases: iterative memory pre-copy rounds, stop-and-copy, and resumption. During the memory pre-copy phase, the first iteration transfers all memory pages over the network to the destination, while the VM continues to execute concurrently. In the subsequent iterations, only the dirtied pages are transferred. After certain round of iterations, determined by a convergence criteria, the stop-and-copy phase is initiated, during which the VM is paused at the source and any remaining dirty pages, VCPUs, and I/O state are transferred to the destination VM. Finally, the VM is resumed at the destination.

Intra-Host Live VM Migration: As shown in Figure 1(b), using nested virtualization, a base hyperplexor at layer-0 (L0) can run deprivileged hypervisors at layer-1 (L1), which control the VMs running at layer-2 (L2). At hypervisor replacement time, the hyperplexor can boot a new replacement hypervisor at L1, and use intra-host live VM migration to transfer VMs from the stale hypervisor to the replacement hypervisor.

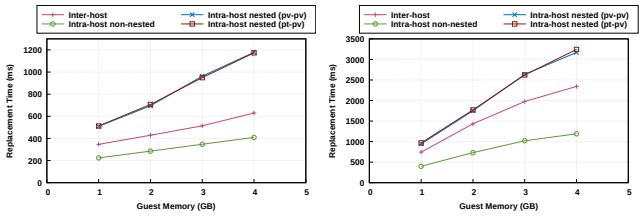
2.2 Improving the default pre-copy implementation in KVM/QEMU

We observed two performance problems when using KVM/QEMU's default pre-copy implementation. So that we can compare our remapping approach with the best case performance of pre-copy implementation, we modified KVM/QEMU's pre-copy implementation as described below.

First, we observed that the total live migration time for a 1GB idle VM between two physical machines connected via a 40 Gbps network link was 12 seconds, which was way more than what we expected. Upon investigation, we found that, by default, QEMU limits the migration bandwidth to 256 Mbps. We modified QEMU to disable this rate limiting mechanism, so that the VM's memory can be transferred at full network bandwidth.

Secondly, we observed that when the VM is running a write-intensive workload then the pre-copy iterations never converge to the stop-and-copy phase. This was found to be due to an inaccurate convergence criteria based on the VM's page dirtying rate (the rate at which a VM writes to its memory pages), which not only breaks down for write-intensive VMs, but also misses opportunities for initiating an earlier stop-and-copy phase when the number of dirtied pages in the the last iteration could be low.

To ensure that the pre-copy rounds converge successfully, we made two additional changes to QEMU's pre-copy implementation. We placed a hard limit of 20 rounds on the



(a) Idle guest **(b)** Busy guest

Figure 2. Comparison of hypervisor replacement time between the same and different hosts with nested and non-nested setup

maximum number of pre-copy iterations, after which the stop-and-copy phase is initiated. We also simplified QEMU's default convergence criteria by triggering the stop-and-copy phase when the number of dirty pages from the prior round is less than 5000 pages. The latter change yields a low downtime of less than 5ms. [1].

2.3 Analysis of pre-copy performance for hypervisor replacement

With the above optimizations to QEMU's default pre-copy live migration, we analyzed the total migration time, which also represents the time taken for hypervisor replacement operation. We analyzed the following four configurations:

- **Inter-host:** where a VM is migrated between two physical machines, as in Figure 1(a). The source machine runs the stale hypervisor and the destination machine runs the new hypervisor.
- **Intra-host nested (pv-pv):** where the VM is migrated between two *co-located nested hypervisors*, as in Figure 1(b). The source L1 runs the stale hypervisor and the destination L1 runs the new hypervisor. The network interfaces of both L1 hypervisors and their L2 VMs are configured as para-virtual *vhost-net* devices [46].
- **Intra-host nested (pt-pv):** Same configuration as the above case, except that the both L1 hypervisors are configured to use pass-through network interfaces via virtual functions configured in the physical NIC.
- **Intra-host non-nested:** This is a baseline (and somewhat unrealistic) migration scenario where a single-level non-nested VM is migrated within the same host into another VM instance. This case helps us measure the intra-host live migration performance if there were no overheads due to nested virtualization and network interfaces (since source and destination QEMU processes communicate via the loopback device).

[1]We note that these modifications are meant specifically to improve pre-copy performance in our experiments and are not general recommendations for production settings

The experiments were conducted using machines having 10-core Intel-Xeon 2.2GHz processors with 32GB memory and 40Gbps Mellanox ConnectX-3 Pro network interface.

Figure 2a plots the total migration times for an idle VM when the VM's memory size is increased from 1GB to 4GB. Figure 2b plots the total migration time for a busy VM where the VM runs a write-intensive workload in the VM. Specifically, the write-intensive workload allocates a certain size of memory region and writes a few bytes to each allocated page at a controllable dirtying rate. The allocated memory size is 80% of the VMs memory, from 800 MB for a 1GB VM to 3200MB for a 4GB VM. The dirtying rate is set to 50,000 pages per second.

First we observe that in all the cases, as the VM size increases, the total migration time increases, because more memory pages must be transferred over a TCP connection for larger VMs. The migration times range from as low as several hundred milliseconds for idle VMs to more than 3 seconds for busy write-intensive VMs. Since pre-copy retransmits dirtied pages from previous iterations, write-intensive VMs take longer to migrate.

The second surprising observation was that the *Inter-host* configuration for migrating a non-nested VM was faster than the Intra-host configuration for migrating a nested VM. Replacing the paravirtual *vhost-net* interface for the L1 hypervisor in the *Intra-host nested (pv-pv)* configuration with a pass-through interface in the *Intra-host nested (pt-pv)* configuration did not produce any noticeable reduction in total migration time. (However we did observe a reduction in CPU utilization when pass-through I/O is used for the L1 hypervisor in the *pt-pv* case.) To verify that this worse performance was caused by nesting overheads, we carried out the *Intra-host non-nested* live migration described above. As expected, this fourth configuration performed better than the *Inter-host* configuration, confirming that nesting overheads indeed adversely affect intra-host VM migration.

Nevertheless, even in the ideal (though unrealistic) case represented by the *Intra-host non-nested* setting, traditional pre-copy migration takes between 400ms (for idle VM case) to more than 1 second (for busy VM case) to migrate a single 4GB busy VM. When multiple VMs must be relocated for hypervisor replacement, these times are bound to increase. We consider this performance unsatisfactory for simply relocating VMs within the same host.

In summary, using intra-host pre-copy live migration for hypervisor replacement is expensive in terms of total migration time, network overheads, and its affect on VM's performance. This motivates us to develop a much faster hypervisor replacement technique based on memory remapping that does not involve copying of VM's memory pages within the same host. In the following sections, we present our approach and show that our memory remapping-based technique can achieve sub-10ms live hypervisor replacement

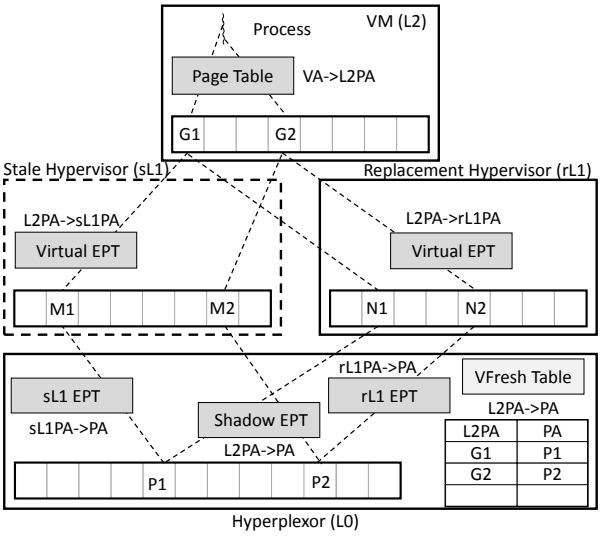


Figure 3. Memory mapping translations for hypervisor replacement using nested virtualization.

times for VMs and sub-second live OS replacement times for containers.

3 VFresh for Hypervisor Replacement

In this section, we introduce the design and implementation of VFresh for live hypervisor replacement, followed by live OS replacement in the next section. The key idea behind VFresh is as follows: Instead of copying the memory of VMs from the stale hypervisor to the co-located replacement hypervisor, VFresh transfers the ownership of VMs' physical memory pages via page table remapping. Such memory ownership relocation is fast and leads to sub-10ms live hypervisor replacement. In addition, VFresh includes optimizations to further mitigate nesting overhead during normal execution of VMs. We first introduce memory translation under nested virtualization, followed by hyperplexor-based hypervisor switching, and finally optimizations to mitigate nesting overheads.

3.1 Memory Translation for Nested Virtualization

In native execution mode (without virtualization), a page table stores the mappings between the virtual addresses (VA) of a process to its physical addresses (PA) where memory content actually resides. When a VA is accessed by a process, the hardware memory management unit (MMU) uses the page table to translate the VA to its PA.

In single-level virtualization, an additional level of address translation is added by the hypervisor for virtualizing the memory translations for VMs. The page table of a process running on a VM stores the mappings from the guest virtual addresses (GVA) of a process to the guest physical addresses (GPA), which is the virtualized view of memory seen by the VM. The hypervisor uses another page table, called the extended page table (EPT) to map GPA to its physical addresses (PA). As with traditional page tables, an EPT is constructed incremtally upon page faults. As a VM tries to access previously unallocated GPA, EPT violations are generated, like page faults for a process. These faults are processed by the hypervisor which allocates a new physical page for the faulting GPA.

In nested virtualization, as shown in Figure 3, three levels of translations are needed. A process's virtual address is translated to the GPA of the layer-2 VM (labeled L2PA). An L2PA is translated to the guest physical address of the L1 hypervisor (L1PA) using a *virtual EPT* for the layer-2 VM maintained by the L1 hypervisor. Finally, the L1PA is translated to the physical address of the host (PA) using the L1 hypervisor's EPT maintained by the hyperplexor at L0. Since, the MMU can translate only two levels of memory mappings, the hyperplexor at L0 combines the virtual EPT at L1 and the EPT at L0 to construct a *Shadow EPT* for every L2 VM. The MMU uses the process page table in the L2 VM and the shadow EPT at L0 to translate a process VA to its PA. The shadow EPT is updated whenever the corresponding virtual EPT in L1 and/or the EPT in L0 are updated.

3.2 Hypervisor Replacement Overview

We now present an overview of the hypervisor replacement mechanism followed by low-level details. Consider a VM that initially runs on the stale L1 hypervisor which in turn runs on the thin L0 hyperplexor. To replace the stale hypervisor, the hyperplexor creates a new replacement L1 hypervisor (with pre-installed updates) and transfers the ownership of all L2 VMs' physical pages to this replacement hypervisor.

The page mappings from an L2 VM's guest physical address (L2PA) to the physical memory (PA) are available in the shadow EPT of the L2 VM. Ideally, the hyperplexor could accomplish the L2 VM's relocation simply by reusing the shadow EPT, albeit under the replacement hypervisor's control. However, since the shadow EPT is the result of combining a nested VM's virtual EPT in L1 and L1's own EPT in L0, these two intermediate mappings must be accurately reconstructed in the new address translation path via the replacement hypervisor.

To accomplish this reconstruction, the replacement hypervisor must coordinate with the hyperplexor. It does so by preparing a skeletal L2 VM to receive each incoming L2 VM and reserves unallocated L1 page frame numbers in its guest-physical address space (L1PA) where the incoming VM's pages will be mapped. The replacement hypervisor then communicates the identity of these reserved page frames in L1PA to the hyperpexor (via a hypercall), so that the hyperplexor can map these reserved page frames to the incoming L2 VM's existing physical pages. Note that the reserved L1 page frames in the replacement hypervisor may not be the same as that in stale hypervisor as shown in Figure 3.

The transfer of execution control is then performed as follows. The L2 VM's execution is paused at the stale hypervisor and its execution state (consisting of all VCPU and I/O states) are transferred to the replacement hypervisor's control, which then resumes execution of the L2 VM. This switchover operation can be accomplished quickly (sub-10ms in our prototype) since no memory content is copied during this step. Finally, once the control of the L2 VM is transferred to the replacement hypervisor, the stale hypervisor can unmap the L2 VM's memory from its address space and be safely deprovisioned.

As the relocated L2 VM begins execution on the replacement hypervisor, it generates page faults against its memory accesses. The entries in the new intermediate mapping tables (the virtual EPT and replacement hypervisor's L1 EPT) are populated on-demand to reflect the original physical pages used by the L2 VM under the stale hypervisor. In other words, the shadow EPT reconstructed for the relocated VM ends up being identical to its earstwhile shadow EPT used under the stale hypervior, while the new intermediate mapping tables are correspondingly adjusted. We nest describe low-level details of VFresh implementation.

3.3 Implementation Details

We developed a prototype of VFresh using the KVM/QEMU virtualization platforms with Linux kernel version 4.13.0 and QEMU version 2.9.0. KVM is a linux kernel module which is responsible for core hypervisor functionalities. QEMU is a user space process that manages VM control operations, including live migration, and emulates its I/O operations. The VM runs unmodified Linux kernel version 4.13.0 and uses para-virtual I/O devices. Our hypervisor replacement solution was implemented by modifying QEMU's live migration mechanism, besides modifications to KVM to implement page remappings described above.

Different from QEMU's live migration, VFresh's hypervisor replacement operation only involves the transfer of VCPU and I/O state of the VM. Since the relocation of L2 VM's memory pages is performed out of the critical path of hypervisor replacement, we modified QEMU's live migration to disable dirty page tracking, transfer of page contents via pre-copy iterations, and also the transfer of residual dirty pages during the stop-and-copy phase of live migration. The VCPUs are paused only during the stop-and-copy phase which results in very low hypervisor replacement time.

3.3.1 The VFresh Table

In KVM's current implementation, the L2 VM's shadow EPT cannot be easily transferred from the stale hypervisor to the replacement hypervisor, Hence our hyperplexor implementation constructs an parallel table, which we call the *VFresh table*, to store a copy of the L2-to-physical page mapping information contained in the shadow EPT. This VFresh table is used for reconstructing the same L2PA-to-PA page mappings for the relocated VM upon EPT violations.

To construct the VFresh table, the hyperplexor needs a list of L2-to-L1 page mappings of the VM from the stale hypervisor, using the virtual EPT table. The stale hypervisor invokes a `hypercall_put` hypercall multiple times to pass a complete list of L2-to-L1 page mappings of the VM to the hyperplexor. For each received L2-to-L1 page mapping, the hyperplexor translates the L1 page number to the physical page number using the EPT table at L0, and inserts the corresponding L2-to-physical page mapping into the *VFresh table*.

To relocate the ownership of a L2 VM's memory to the replacement hypervisor, the hyperplexor needs the list of L2-to-L1 page mappings of the newly created skeletal VM from the replacement hypervisor. The replacement hypervisor invokes another `hypercall_map` hypercall multiple times to pass these mappings to the hyperplexor. For each received L2-to-L1 page mapping, the hyperplexor (1) looks up the *VFresh table* to find the L2-to-PA mapping with the L2 page number as the key; and (2) installs the L1-to-PA page mapping in the replacement hypervisor's EPT table. More specifically, VFresh translates each L1 guest frame number to its host virtual address (HVA), installs the HVA-to-PA page mapping in the page table of the VM's QEMU process, and at last invalidates the corresponding entry in the EPT table, which is reconstructed later upon an EPT fault. Also, to clear any pre-existing GFN mappings of the replacement hypervisor, the hyperplexor flushes the entries in TLB and page table of QEMU, before resuming the relocated VM.

3.3.2 Mitigating Nesting Overheads

In comparison with the traditional single-level virtualization setup, where the hypervisor directly controls the hardware, nested virtualization can introduce additional emulation overheads, especially for I/O virtualization. To mitigate such overhead, VFresh uses the direct device assignment of network interface cards (NIC) to the hypervisor.

Existing virtualization techniques (e.g., KVM [27]) support the full virtualization mode through device emulation [45] and para-virtualization using virtio drivers [8, 38] in VMs. For example, QEMU [9] emulates I/O devices requiring no modifications to VMs. When VMs access the devices, the I/O instructions are trapped into hypervisor leading to a number of VM/host context switches, resulting in lower performance of VMs. The para-virtualized devices offer better performance compared to device emulation, as the modified drivers in VMs avoid excessive VM/host switches for the I/O workloads. With Intel's VT-d [1], direct device assignment offers improved performance, as VMs directly interact with the devices bypassing the hypervisor. Further, SR-IOV [16] enables a single network device to present itself as multiple virtual NICs to VMs through virtual functions (VFs). Using a VF, a VM directly interacts with the network device —

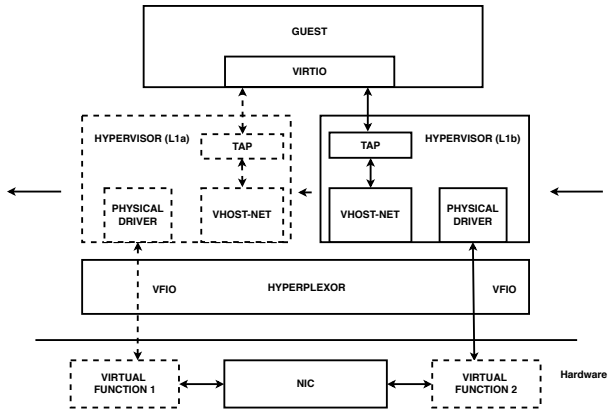


Figure 4. Architecture for hypervisor replacement with direct-device assignment and thin hyperplexor.

with Intel's hardware support, VMs can directly access device memory through IOMMU which converts VMs physical addresses to host physical addresses.

In this paper, VFresh uses VFs to achieve network performance in nested guest to match the performance of a single-level guest and minimize the CPU Utilization in hyperplexor. VFIO [48] driver supports direct device assginment to a virtual machine. As shown in Figure 4. every hypervisor running on hyperplexor is assigned one virtual function. The guest running on hypervisor uses para-virtualized driver to run the I/O workloads. VFresh further applies many optimizations to provide hypervisor with enough resources and reduce the CPU utilization on hyperplexor. The goal is to run the hypervisor with minimum hyperplexor intervention. We describe the optimization techniques in detail in the following sections.

Memory Footprint: As the hyperplexor involvement is minimal, it is configured with essential packages, device drivers and services reducing the memory consumption. The size of the hyperplexor without VM is 90 MB of the available memory compared to the Ubuntu server size 154 MB. The userspace processes and services that where not necessary to run the L1 hypervisor with direct-device assignment were identified and removed. The Linux kernel 4.13 is compiled with 32 kernel modules compared to 120 kernel modules in Ubuntu server.

HLT Exits: Although a directly assigned network device to a VM gives better performance, the CPU utilization on the host is high due to the idle polling of VCPUs. When the VM is idle, QEMU halts the idle VCPUs by executing a HLT instruction which triggers a VM exit. When the work is available for the VCPU, it has to be woken up to execute the new work. This transition from idle to ready state is costly as it involves context switch and hinders the performance of the VM. To avoid too many transitions, before executing the HLT instruction the VCPU polls for the specified amount of time to check if there is any additional work to be executed. This idle polling of VCPUs reduces the number of VM exits but increases CPU utilization on host. To reduce CPU utilization on host, we disable polling of VCPUs before executing HLT instruction. KVM provides a halt_poll_ns kernel module parameter to set the value of halt_poll_ns. To disable polling we set halt_poll_ns to zero. In this paper, as we assign the network device to the hypervisor, we disable the halt polling on the hypervisor to reduce the CPU Utilization in host.

Posted Interrupts: When an external interrupt arrives, the CPU switches from non-root mode to root mode (VM Exit) and transfers the control to the hypervisor. Increase in number of external interrupts causes increase in VM Exits. With Intel's VT-d posted interrupt support the external interrupts are delivered to guest without hypervisor intervention. With posted interrupts support the hypervisor can inject a virtual interrupt to guest by updating the fields in the VMCS in guest or non-root mode. Hypervisor notifies the arrival of posted interrupt to the guest by setting the fields Posted Interrupt Request (PIR), Notification Vector (NV) and Outstanding Notification (ON) in Posted Interrupt Descriptor structure. The PIR field is 256 bit long and provides storage for posting the interrupts. The NV field is used to notify the interrupt vector to the guest. The ON bit indicates the processing status of the posted interrupt. We enable posted interrupt feature on the hyperplexor and deliver external interrupts directly to hypervisor without causing exits to hyperplexor.

Dedicated Cores: All the system processes are run on two cores except the VCPUs which run on dedicated cores. The VCPUs are assigned dedicated cores to eliminate other processes from contending with VCPUs. As shown in Figure **??**. the QEMU process along with other hyperplexor processes are pinned to CPU0 and the hypervisor VCPUs run on the dedicated cores.

4 VFresh for OS Replacement

Using containers to host applications becomes a common usage scenario in today's cloud [17, 41–44]. Under this scenario, the act of OS replacement can be viewed as moving the state of all containers from the old VM with the stale OS to a new VM with the replacement OS. Again, we leverage intra-host live migration to avoid inter-host communication.

VFresh relocates the memory ownership of containers from the stale OS to the replacement OS using the VFresh table, whose purpose is similar to that described for hypervisor replacement. The per-process VFresh table stores the memory mappings from the pages of a container to the pages of the physical memory, constructed by the stale OS, and is leveraged by the new OS for reconstructing the memory address space of the process during OS replacement.

Essentially, a container consists of a hierarchy of processes as well as isolation mechanisms (e.g., `cgroups` and `namespaces` in Linux). If not specified, processes running upon an OS belong to a default container. Hence, moving a

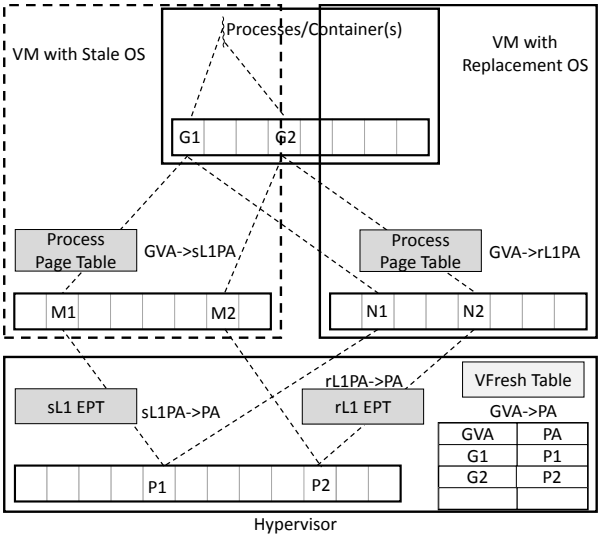


Figure 5. Memory mappings translations for OS replacement running containers.

container from one VM to another for OS replacement needs to copy the state of all processes in the container from one VM to another. Again, VFresh's OS replacement relocates the memory ownership of all processes instead of copying memory data. The design of VFresh's OS replacement is similar to the above hypervisor replacement, and we focus on the differences in the OS replacement.

4.1 Memory Management

The memory management under the container case belongs to the above-mentioned single-level virtualization. As shown in Figure 5, the page table of a process (in a container) running in the VM stores the mappings between the guest virtual addresses (GVA) of the process and the guest physical addresses (GPA) of the VM. The hypervisor uses the EPT to map GPA to physical addresses (PA).

4.2 VFresh Table

To build the VFresh table for OS replacement, the hypervisor needs a list of GVA-to-GPA page mappings of a process (IN a container) from the stale VM's kernel (i.e., using the process's page table). Similarly as the hypervisor replacement, VFresh provides a hypercall, `hypercall_set`, to the VM kernel for exposing such mappings, as listed in Table **??**.

The stale VM's kernel invokes this hypercall to pass a list of GVA-to-GPA page mappings to the hypervisor. For each received GVA-to-GPA page mapping, the hypervisor translates the page of GPA to the page of PA (i.e., using the VM's EPT), and puts the corresponding GVA-to-PA page mapping to the *VFresh table* (associated with the process's PID). The stale VM's kernel can invoke multiple `hypercall_put` hypercalls to pass all necessary GVA-to-GPA page mappings for setting up the VFresh table for the process. The same process is repeated for every process in the container.

Restoring the address space of a process (in a container) on the new VM (i.e., with the replacement OS) needs to first build GVAs, GPAs, and their mappings. Given the GVA-to-GPA page mappings of the process, the new VM's kernel invokes a hypercall, `hypercall_map`, to map the GVAs to the PAs, by passing a list of GVA-to-GPA page mappings of the process to the hypervisor. For each received GVA-to-GPA page mapping, the hypervisor: (1) looks up the *VFresh table* to find the GVA-to-PA page mapping with GVA as the key; and (2) installs the GPA-to-PA page mapping in the EPT page table of the new VM. Similarly, the new VM's kernel can invoke multiple `hypercall_map` hypercalls to pass all GVA-to-GPA mappings for fully relocating memory address space for the process on the new VM with the replacement OS.

4.3 OS Switching

During OS replacement, the stale VM's kernel provides the GVA-to-GPA page mappings for `hypercall_set` hypercalls to build an VFresh table. Such mappings can be obtained by walking through the page table of the process in the stale VM's kernel. These mappings can also be obtained using a user-level replacement tool (e.g., CRIU [47]), which dump and transfer memory state from the user space (e.g., reading from `/proc/[PID]/pagemap`). To support such a user-level live OS replacement implementation, VFresh provides a new system call, `syscall_set`, which takes a list of GVA-to-GPA mappings from the user space. Thus, in this syscall, the VM kernel simply passes received GVA-to-GPA mappings to the hypervisor via `hypercall_set` hypercalls.

The replacement process on the new VM with the replacement OS first reconstructs all the GVAs (e.g., with the `mmap` system call). The VM kernel then allocates corresponding GPAs and sets up the GVA-to-GPA page mappings in the page table of the process (e.g., with the `set_pte_at` system call). Finally, the new VM's kernel invokes a series of `hypercall_map` hypercalls to the hypervisor, which installs the GPA-to-PA page mappings in the new VM's EPT table using the VFresh table as stated above.

4.4 Implementation Details

We have implemented an OS replacement prototype based on CRIU [47] — a checkpoint/restore tool implemented in the user space. CRIU consists of two stages: checkpointing (on the source VM) and restoration (on the destination VM). By automating these two parts, we implemented the live OS replacement for containers. VFresh's OS replacement operation involve the transfer of all the state of containers and their processes (e.g., file descriptors, memory maps, registers, namespaces, cgroups, etc.) except for memory content from the source VM (with the stale OS) to the destination VM (with the replacement OS).

In our implementation, when the OS replacement operation is invoked, one of the target processes is paused. During the checkpointing phase, we collect all state of the process

except for its memory content and build the VFresh table. During the restoration phase, we restore the process state on the destination VM and relocate the memory ownership. In the final stage, we resume running the process on the destination VM. The source VM unmaps the pages of the process from its address space. We repeat this process until all target processes are moved to the destination VM. Then the source VM can be gracefully shutdown.

Checkpointing: We modified CRIU's checkpointing code to replace the procedure of dumping memory content with the one of building the VFresh table by calling VFresh's provided system call, `syscall_set`. The function takes an array of GVAs and the their mapped GPAs. To obtain such GVA-to-GPA page mappings, we exploit CRIU's existing mechanism of reading the `/proc/[PID]/pagemap` file, which stores the GVA-to-GPA mappings. We further use the `memalign` function to allocate memory for the input arrays, which is page size aligned. The `syscall_set` system call translates the base address of the two arrays (containing GVAs and GPAs) into GPAs, and invokes the `hypercall_set` hypercall by transferring the GPAs of the arrays to the hyperplexor. As stated above, the `hypercall_set` hypercall locates (or creates) the VFresh table for the process (by PID), obtains the GVA-to-PA mappings, and insert them to the VFresh table.

Restoration: On the destination VM, we modify CRIU's restoration code to replace the procedure of loading memory content with the one of relocating address space via VFresh's new system call, `syscall_map`. Before invoking `syscall_map`, the restorer needs to restore GVAs (i.e., received from the source VM) using the `mmap` system call. It then invokes VFresh's `syscall_map` by passing an array of GVAs. The `syscall_map` system call in turn gets free VM memory pages, GPAs, and establish the GVA-to-GPA page mappings in the process's page table. With such GVA-to-GPA page mappings, the `syscall_set` system call invokes the `hypercall_mpa` hypercall by transferring such mappings to the hyperplexor. As stated above, the `hypercall_map` hypercall installs the GPA-to-PA page mappings in the destination VM's EPT table.

5 Evaluation

5.1 Hypervisor Replacement

We run our experiments on a server machine equipped with one Intel Xeon E5-2630 v4 CPU with 10 cores of 2.2 GHz and 32 GB total memory. All experimental results are averaged over five runs or more to increase the confidence in the results. Table 1 shows the detailed setup configuration.

5.1.1 Comparisons of Replacement Time

We measure the hypervisor replacement time under VFresh and compare it with the four cases based on the optimized pre-copy live migration as stated in Section 2, which are (a) inter-host, (b) intra-host nested (pv-pv), (c) intra-host nested (pt-pv), and (d) intra-host non-nested. The nested virtualization configurations are listed in detail in Table 1: The L2 VMs are configured with 1 VCPU (and varying memory sizes under different test cases); the L1 hypervisors are configured with 8 GB memory and 4 VCPUs; and the L0 hyperplexor is configured with 32 GB memory and 10 CPUs.

	Hyperplexor	Hypervisor	Guest
Host	1GB, 1CPU	-	-
Non-Nested	8GB, 4CPUs	-	1GB, 1VCPU
Nested	32GB, 10CPUs	8GB, 4VCPUs	1GB, 1VCPU
VFresh	32GB, 10CPUs	8GB, 4VCPUs	1GB, 1VCPU

Table 1. Experiment setup to measure the hypervisor replacement time for different setups.

Single VM. First, we use a single idle VM and vary its memory sizes from 1 GB to 4 GB. Figure 6 shows that the total replacement time under the inter-host case is around 0.35 seconds for 1 GB memory size and 0.6 seconds for 4 GB memory size. Under the intra-host nested cases (both pv-pv and pt-pv), the replacement time increases to around 0.5 seconds for 1 GB memory size, and 1.17 seconds for 4 GB memory. In contrast, the total hypervisor replacement time under VFresh is extremely low — 10 ms. The replacement time keeps constant as the VM's memory size increases. It is because, in VFresh, the memory is re-mapped between the L1 hypervisors and such operations are performed out of the critical path of the VM state transfer; the replacement time only comprises of the time of transferring the VCPU and I/O device state information, which is constant. However, under both the inter-host and intra-host cases, transferring the memory pages through the network accounts for higher total migration time and thus replacement time.

Next, we use a busy VM with varying memory sizes. The busy VM runs a "working set" benchmark which dirties the memory at a controllable page dirty rate — we fix it as 5,000 pages per second in all out experiments. In Figure 7, under all pre-copy live migration based cases the replacement time again increases as the memory size of the VM increases. The replacement time is higher than that under the idle VM, because of transferring dirtied pages in multiple rounds for the busy VM. With VFresh, the replacement time remains constant irrespective of the memory dirty rate — the replacement time remains around 10 ms for the cases with varying memory sizes.

Multiple VMs. Further, we measure the replacement time by running multiple VMs on the same stale hypervisor and later moving them to another same replacement hypervisor. We vary the VM number from 1 to 4: Each VM is configured with 1GB memory; all the VMs start the migration at the

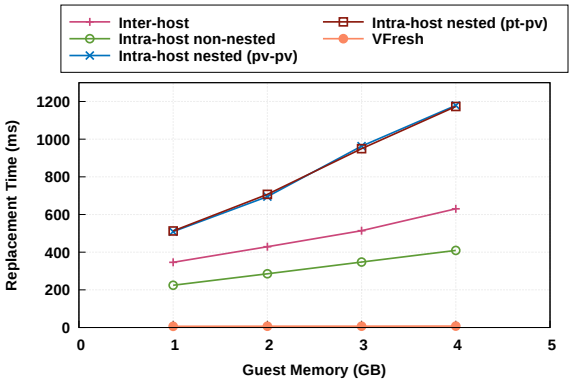


Figure 6. Comparison of hypervisor replacement time for an idle VM with varying memory sizes.

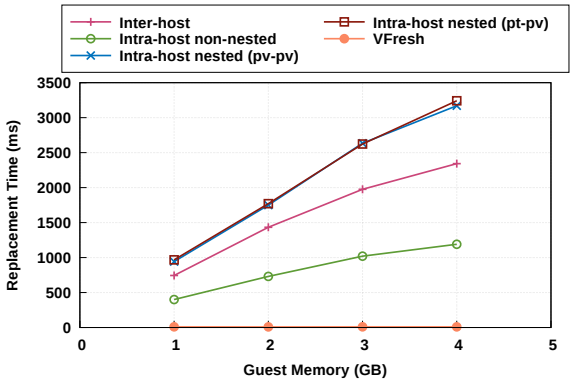


Figure 7. Comparison of hypervisor replacement time for a write-intensive VM with varying memory sizes.

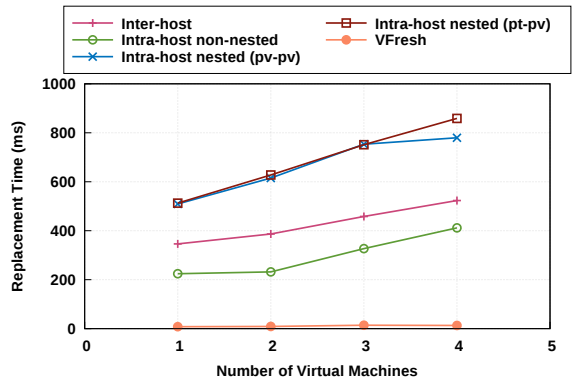


Figure 8. Comparison of hypervisor replacement time for multiple idle VMs with varying VM #.

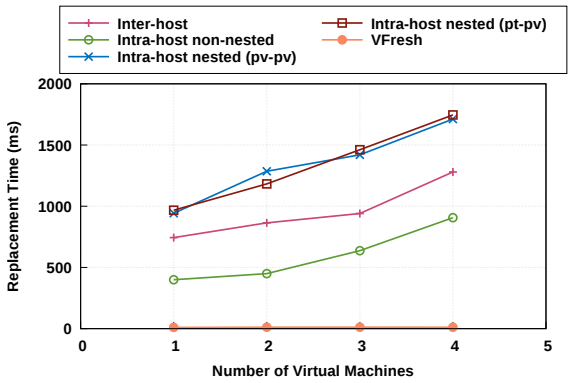
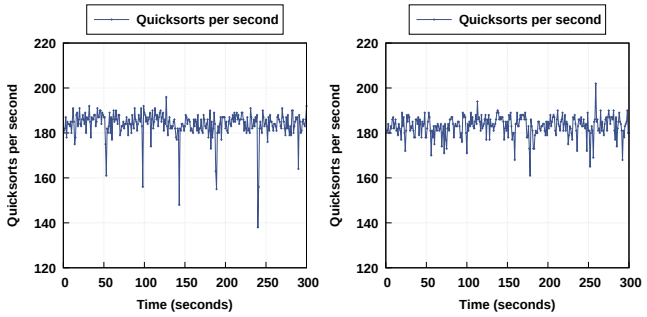


Figure 9. Comparison of hypervisor replacement time for mutiple write-intensive VMs with varying VM #.

same time. We consider the following two cases: (1) with all VMs being idle and (2) with all VMs being busy (with the same page dirty rate of 5,000 pages per second).

In Figure 8, with all idle VMs, it takes 0.3 seconds to migrate 1 VM and 0.5 seconds to migrate 4 VMs under inter-host live migration. The migration time increases to 0.5 seconds for migrating 1 VM and 0.8 seconds for 4 VMs under intra-host nested host live migration (for both pv-pv and pt-pv setups). In Figure 9, with all busy VMs, it takes 0.7 seconds to migrate 1 VM and 1.27 seconds for 4 VMs under inter-host live migration. Under intra-host nested host live migration (for both pv-pv and pt-pv setups), the migration time increases to 1 seconds for 1 VM and 1.75 seconds for 4 VMs. In contrast, the time to replace the hypervisor remains 10 ms with either idle or busy VMs. The replacement time does not increase as the number of VMs increases, because again the VMs' memory is remapped and only the VCPU and I/O device state is transferred during hypervisor replacement.

Performance of Quicksort. Last, we measure how application-level performance is affected during hypervisor replacement between VFresh and using a CPU-intensive benchmark, Quicksort. It first allocates 1 GB memory populated with random integers, and then performs quicksort on a 200 KB memory



(a) Hypervisor replacement using optimized pre-copy

(b) VFresh

Figure 10. Quicksort performance over multiple hyperfresh replacements

block on every iteration. We measure the number of quicksorts performed per second during regular operation and migration of nested guest.

Figure 11a. shows the performance of quicksort during optimized pre-copy migration. With optimized pre-copy, as the rate limit is disabled, a large number of dirty memory pages are transferred in less number of pre-copy iterations. As the number of pre-copy iterations decrease we do not observe

	Hyperplexor	Hypervisor	Guest
Hyperplexor	8GB, 2CPUs	-	-
Non-Nested	16GB, 8CPUs	-	8GB, 2VCPUs
Nested	32GB, 10CPUs	16GB, 8VCPUs	8GB, 2VCPUs
VFresh	32GB, 10CPUs	16GB, 8VCPUs	8GB, 2VCPUs

The guest is configured with 4VCPUs for iperf experiments. The host is also configured with 4CPUs when iperf is run in host.

Table 2. Experiment setup to measure the performance of Quicksort, Kernbench and iperf benchmarks in the guest.

variations in the performance of quicksort during pre-copy iterations. The only dip in the performance is during stop-and-copy phase. Figure 11a. shows the dip in performance of quicksort during multiple optimized pre-copy migrations. In Figure 11b., the performance of quicksort does not reduce during migration due to memory co-mapping and we do not observe dip during stop-and-copy phase we do not transfer any pages in the stop-and-copy phase.

5.1.2 Memory Mappings

The guest pages are pinned to the memory and populate the EPT and shadow EPT entries on boot up. The time taken to pin the pages in memory and populate the page table entries on the source host are 1.7 s, 3.3 s, 4.8 s, 6.3 s for a 1 GB, 2 GB, 3 GB and 4 GB memory respectively. The time taken on the destination host is 0.56 s, 1.2 s, 1.6 s, 2 s for 1 GB, 2 GB, 3 GB and 4GB respectively. The memory mapping time increases with increase in memory size of the guest as the number of pages to be mapped and hence the page table entries to be populated increase. The memory-mapping time measured represent the time taken to create the grant table used to transfer the memory mappings. The time taken to create the grant table increase with increase in guest memory as the number of hypercalls from hypervisor to hyperplexor increase along with grant table entries. However, this does not affect the total migration time as the grant table is created during the VM boot process.

5.1.3 Performance Impact on Guest VM

We measure the performance of host and guest with single-level and nested virtualization for different benchmarks and compare the performance with VFresh with optimizations. The configuration for the experiments is as shown in Table 2. The guest performance is measured by running one of the following benchmarks.

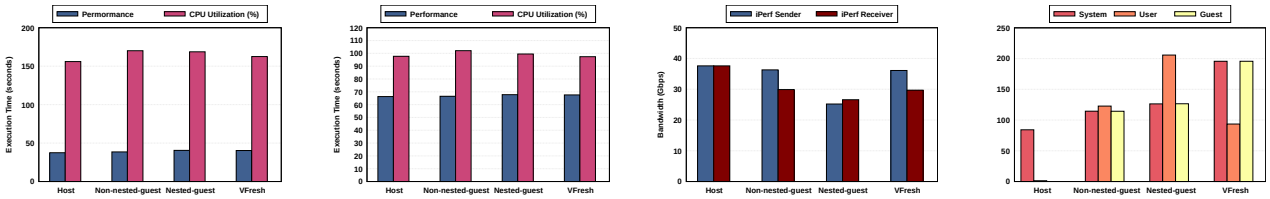
Iperf is a benchmark used to measure the performance of the system for I/O workload. iPerf client or server runs in the guest and the corresponding iperf server or client runs on a different host. We measure the average throughput of TCP network traffic over 100 s over a 40 Gbps NIC card. We also measure the CPU utilization in hyperplexor, hypervisor and guest due to the network workload in guest.

Kernbench [28] is memory, CPU and I/O intensive benchmark. It uses multiple threads to compile the kernel. The benchmark fetches kernel source code files from disk for compilation and writes the binary files back to disk after compilation. For experiments, the kernel is compiled for five iterations and the number of threads used to perform kernel compilation is set to number of VCPUs in guest.

In Figure 11c. for non-nested guest, to measure the iperf performance the guest is configured with para-virtualized driver. The performance of iperf is 36.3 Gbps for a 40 Gbps NIC card when iperf client is run in guest. However, the performance reduces to 29.9 Gbps when iperf server is run in the guest. The loss in performance is due to the processing of huge number of incoming packets. When iperf is run in nested guest the performance reduces to 25.2 Gbps. In our solution, as the halt_poll_ns is disabled in the hyperplexor, the CPU scheduler gets invoked and schedules the nested guest VCPUs sooner to handle the interrupts. As a result the performance of iperf is now comparable with the performance in the single-level guest.

In Figure **??** we measure the CPU utilization in hyperplexor while running iperf benchmark in guest. We measure the CPU consumed in user mode, system mode and by the guest separately. The CPU utilization in user mode also includes the CPU consumed by the guest. For a non-nested guest using para-virtual driver, more time is spent in the system to handle the network packets. As a result, the CPU utilization in system mode is high compared to user mode. The CPU utilization of guest is almost equal to the user CPU utilization indicating that the guest is utilizing the CPU for the rest of the time. When NIC card is assigned to the hypervisor, the CPU utilization in the hyperplexor is expected to be less as the network packets are directly delivered to the hypervisor. But the CPU utilization in hyperplexor is high as the hypervisor VCPUs burn CPU cycles when they execute HLT instruction. The nested guest uses para-virtual driver to handle the network traffic. The CPU utilization in the system mode in the hypervisor is higher than the CPU utilization in guest or user mode. In VFresh, we reduce the CPU utilization in system mode in hyperplexor by disabling halt_poll_ns. As seen from the results, the hypervisor now utilizes the CPU more in guest mode compared to the system mode in hyperplexor.

The performance of kernbench and quicksort introduces 2.8% and 0.3% overhead respectively in non-nested guest compared to host, whereas the nested guest introduces 8.1% and 2.23% overhead due to the additional layer of virtualization. With our proposed solution the performance of kernbench and quicksort benchmarks causes overhead of 7.6% and 1.9% for kernbench and quicksort respectively, which do not show



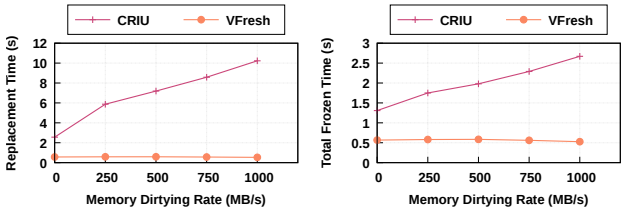
(a) Kernbench: Comparison of execution time and CPU utilization on hyperplexor with VFresh and different setups.

(b) Quicksort: Comparison of execution time and CPU utilization on hyperplexor with VFresh and different setups.

(c) iperf: Comparison of Bandwidth with VFresh and different setups.

(d) Breakdown of CPU utilization in system, user modes and guest for iperf benchmark.

Figure 11. Applications performance comparison.



(a) Comparison of OS replacement time with CRIU and VFresh

(b) Comparison of total frozen time with CRIU and VFresh

Figure 12. Memory Dirtying Rate Variations

Hyperplexor	VM (stale OS)	VM (replacement OS)
128GB, 12 CPUs	4GB, 4 VCPUs	4GB, 4 VCPUs

Table 3. Setup for OS replacement experiments

significant improvement. However, the performance does not degrade due to the optimizations applied.

5.2 OS Replacement

The experiment setup consists of 12-core Intel-Xeon 2.10GHz processors with 128GB memory. We use KVM-QEMU for virtualization to run source and destination VMs on the same host. On VMs, we used CRIU version 3.5 for OS replacement experiments. Table 3. shows the configuration of the hyperplexor and the VMs.

5.2.1 Comparisons of Replacement Time

Figure 12 shows the effect of container's memory usage on the OS replacement time. We start a container with memory intensive processes which allocate 1 GB memory and perform a continuous write operation on it at a varying rate ranging from 0MB/second to 1GB/second. We compare the OS replacement time of VFresh with CRIU. To complete the OS replacement, CRIU takes multiple iterations to send all the memory state. With increase in memory writing rate, memory getting dirtied during each iteration increases. Hence, the replacement time and the downtime increases as well. Additionally, if the iterations are not stopped manually, then the replacement operation will never complete, reason being the size of the dirtied memory during the iterations is not converging. However, OS replacement and downtime using VFresh stay not only very small compared to CRIU, but also constant with variation in memory writing rate. Since VFresh is not iterative and the total memory usage is contant, so the replacement time doesn't get affected by the change in memory writing rate.

5.2.2 Performance Impact on Container

We used following three macro-benchmarks to show our application performance:

Quicksort [49] benchmark fills random data in 1024 MB of allocated memory and sort them using quicksort sorting algorithm. We use this benchmark to show the total number of quicksorts per second.

Sysbench [29] provides multi-threaded memory test by allocating memory and then reading from or writing to it. We use this benchmark to show the total completion time of a write-intensive experiment. We allocate a buffer of varying size using `--memory-block-size` option and perform write operation on the buffer till its completion.

Parsec [12] benchmark mimics large scale commercial applications. We perform migration by running different workloads of parsec benchmark and compare the total frozen time during the migration.

In Figure 13a, we compare the sysbench completion time while changing the buffer size from 128MB to 1024MB. The baseline data represents the completion time of sysbench a VM. To compare with this baseline data, we migrate a running sysbench application and observe its completion time after migration. Performance of sysbench with migration using VFresh is almost similar to baseline. However, sysbench takes much longer time to complete when migrated using CRIU. Furthermore, its completion time using CRIU increases more rapidly with the increment in buffer size. In VFresh the change in completion time is not significant.

Figure 13b compares the total frozen time of different workloads of parsec benchmark while being migrated using VFresh and CRIU. We have used workloads with varying memory usage. We observe that the total frozen time for

workloads with high memory usage is higher compared to workloads with lower memory usage. For example, canneal and fluidanimate workload have very high memory usage whereas swaptions and bodytrack use only a small amount of memory. Hence, canneal and fluidanimate workloads are frozen longer during migration compared to swaptions and bodytrack. However, the total frozen time of the workloads using VFresh is almost half compared to CRIU.

Figure 13.c shows the total number of quicksorts per second during OS replacement reduces to zero for a period using CRIU. However, we see atleast 200-400 quicksorts per second during the replacement using VFresh.

6 Related Work

Some errors such as memory leaks accumulate over time and degrade the performance of the system or may lead to a complete failure of the system. This phenomenon is termed as software aging [37]. Software Rejuvenation [22] is a proactive technique where the system is periodically stopped and restored to a clean internal state as a countermeasure to software aging. Software aging of VMMs in a cloud environment are of utmost concern. VMM rejuvenation avoids the failure of the system by frequent reboots but all the virtual machines have to be restarted. The problem escalates when the server has thousands of VMs running on it.

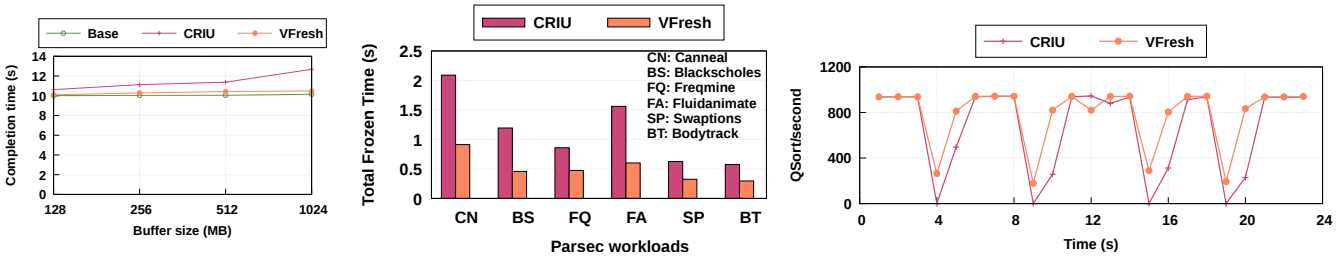
In cold-VM rejuvenation the virtual machines have to be restarted whereas in warm-VM rejuvenation the virtual machine memory state is preserved in persistent memory and is restored after VMM rejuvenation, avoiding costly read/writes to persistent storage. Roothammer [30] proposed warm-VM reboot to rejuvenate a Xen hypervisor which avoids cache-misses by preserving the VM images in main memory reducing the VM reboot time. A more simpler approach is to migrate the VMs to a different host. Migration [14, 21] of virtual machines reduces the downtime by letting the virtual machines run when the memory is copied in multiple rounds to another host. This technique incurs network overhead as huge memory is transferred. In this paper, we eliminate the memory copy rounds by memory co-mapping on the same host hence reducing the network overhead. We also eliminate multiple copies of VM images as the hypervisor replacement happens on the same machine. Kourai et. al proposed VMBeam[31] where a clean virtualized system is started and all the virtual machines are migrated from old virtualized system to new one with zero-memory copy. Unlike our proposed mechanism, VMBeam takes 16.5seconds to migrate a VM of size 4GB with zero memory copy, potentially due to non-live transfer of memory maps, and does not address reduction of nesting overheads during normal execution.

Remus [15] is a reactive technique that achieves high availability by periodically checkpointing the VM image and incrementally storing the changes in the memory of backup host. On failure of current host, the VM can be immediately started on the backup host using the checkpointed memory. The VM images along with network and disk state are checkpointed as frequently as 25ms. To transfer the incremental changes in the VMfis memory, it requires higher network bandwidth. The traffic generated over the network results in performance degradation of the applications running in the virtual machines. VMWareFT [39] uses deterministic replay to keep both primary and backup VMs in sync. Again it requires high quality network connections. In our technique, we transfer only the processor and the I/O state during hypervisor replacement through the network. As shown in the evaluation section, it does not affect the performance of the applications running in the VM. Upon encountering an error in the hypervisor, Rehype [32] performs a microreboot [13] of hypervisor by preserving the state of the VMs. The state of the booted hypervisor is then re-integrated with the state of the preserved VM. The reintegration of state depends heavily on the corrupted VMM state. As the recovery mechanism depends on the state of failed VMM, the success rate is very low.

Applying updates to kernel comes with high cost of downtime due to system reboot. Kernel patching is a complex process of replacing original outdated functions or data structures with new ones. Ksplice [4], Kpatch [24, 25], Kgraft [26] follow the mechanism of replacing old vulnerable functions with new functions using ftrace mechanism. Kpatch and Ksplice stop the machine to check if any of the threads is executing in the function to be patched. If any of the processes is executing or is sleeping in the function to be patched, the kernel patching will be tried later or called off after several attempts. Kgraft keeps a copy of both old and new functions. If the kernel code is active during patching, it runs till completion before switching to new functions while the other processes use the updated functions. Live patching [23] is a combination of Kpatch and Kgraft kernel patching methods. Kernel patching technique can be used to delay the failure of a system until the maintenance period. Although kernel patching maximizes uptime it still requires a reboot at some stage. Live kernel patching can be used for simple fixes and requires reboot if kernel requires major change. The mentioned techniques do not fully support patching as they cannot patch kernel data structures, asm, vdso, or functions that cannot be traced.

7 Conclusion

Hypervisors are tend to software aging and require frequent reboots. We show that pre-copy migration takes significant time to migrate VMs from one host to another and affect the performance of the applications running in the guest during migration. We propose a technique, VFresh to replace the hypervisor without causing service disruption in the guest using nested virtualization to switch the old hypervisor by a healthy one. We co-map the memory of guest to avoid pre-copy memory iteration rounds and transfer only the



(a) VFresh Sysbench performance **(b)** VFresh Parsec performance **(c)** VFresh Quicksort performance

Figure 13. Application's performance comparison with VFresh based migration.

VCPU and I/O state during migration and show that the hypervisor replacement time is within 10ms. The healthy hypervisor can be a patched hypervisor or a hypervisor with clean state. VFresh achieves network performance in nested guest comparable to single-level guest by applying optimizations to hypervisor.

References

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3), 2006.

[2] Amazon EC2. https://aws.amazon.com/ec2/.

[3] Amazon Lambda Programming Model. https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html.

[4] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[5] Autoscaling groups of instances. https://cloud.google.com/compute/docs/autoscaler/.

[6] Autoscaling with Heat. https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html.

[7] Azure Autoscale. https://azure.microsoft.com/en-us/features/autoscale/.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[10] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: Design and implementation of nested virtualization. In *Proc. of Operating Systems Design and Implementation*, 2010.

[11] Franz Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. Swap and play: Live updating hypervisors and its application to xen. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security*, pages 33–44. ACM, 2014.

[12] C. BIENIA, S. KUMAR, J. P. SINGH, AND K. LI. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. http://parsec.cs.princeton.edu/.

[13] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot–a technique for cheap recovery. *arXiv preprint cs/0406005*, 2004.

[14] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[15] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[16] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, volume 2, 2008.

[17] Kubernetes Engine. https://cloud.google.com/kubernetes-engine/.

[18] DAN GOODIN. Xen patches 7-year-old bug that shattered hypervisor security. In *https://arstechnica.com/information-technology/2015/10/xen-patches-7-year-old-bug-that-shattered-hypervisor-security/*, 2015.

[19] Google Cloud platform. https://cloud.google.com/.

[20] Google Infrastructure Security Design Overview, 2017. https://cloud.google.com/security/infrastructure/design/.

[21] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 2009.

[22] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *ftcs*, page 0381. IEEE, 1995.

[23] Kernel live patching. https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt.

[24] Kernel live patching - Kpatch. https://lwn.net/Articles/596854/.

[25] Kernel live patching - Kpatch2. https://lwn.net/Articles/706327/.

[26] kGraft: Live Kernel Patching. https://www.suse.com/c/kgraft-live-kernel-patching/.

[27] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[28] C. KOLIVAS. Kernbench. http://ck.kolivas.org/apps/kernbench/.

[29] KOPYTOV, A. Sysbench manual. 2009. http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[30] Kenichi Kourai and Shigeru Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 245–255. IEEE, 2007.

[31] Kenichi Kourai and Hiroki Ooba. Zero-copy migration for lightweight software rejuvenation of virtualized systems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 7. ACM, 2015.

[32] Michael Le and Yuval Tamir. Rehype: Enabling vm survival across hypervisor failures. In *ACM SIGPLAN Notices*, volume 46, pages 63–74. ACM, 2011.

[33] Linux Bug Tracker. https://bugzilla.kernel.org/buglist.cgi?quicksearch=kvm.

[34] Live patching of xen. https://wiki.xenproject.org/wiki/LivePatch.

[35] Vijay Mann, Akanksha Gupta, Partha Dutta, Anilkumar Vishnoi, Parantapa Bhattacharya, Rishabh Poddar, and Aakash Iyer. Remedy: Network-aware steady state vm management for data centers. In *International Conference on Research in Networking*, pages 190–204. Springer, 2012.

[36] Microsoft azure. https://azure.microsoft.com/en-us/.

[37] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[38] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[39] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[40] Serverless. https://cloud.google.com/serverless/.

[41] Amazon Elastic Container Service. https://aws.amazon.com/ecs/.

[42] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/.

[43] IBM Cloud Kubernetes Service. https://www.ibm.com/cloud/container-service.

[44] VMware Pivotal Container Service. https://cloud.vmware.com/vmware-pks.

[45] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[46] Michael S. Tsirkin. vhost-net: A kernel-level virtio server. https://lwn.net/Articles/346267/.

[47] Checkpoint/Restore In Userspace. https://criu.org/Main_Page.

[48] VFIO Driver. https://www.kernel.org/doc/Documentation/vfio.txt.

[49] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. Enabling efficient hypervisor-as-a-service clouds with eemeral virtualization. *ACM SIGPLAN Notices*, 51:79–92, 2016.