Fast and Live Hypervisor Replacement

Spoorti Doddamani
Binghamton University
New York, USA
sdoddam1@binghamton.edu

Piush Sinha
Binghamton University
New York, USA
psinha1@binghamton.edu

Hui Lu
Binghamton University
New York, USA
huilu@binghamton.edu

Tsu-Hsiang K. Cheng
Binghamton University
New York, USA
tcheng8@binghamton.edu

Hardik H. Bagdi
Binghamton University
New York, USA
hbagdi1@binghamton.edu

Kartik Gopalan
Binghamton University
New York, USA
kartik@binghamton.edu

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 HyperFresh, to transparently replace a 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. 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 a hyperplexor-based approach can used for sub-second relocation of containers for live OS replacement.

CCS Concepts • **Software and its engineering** → *Virtual machines*; *Operating systems.*

Keywords Hypervisor, Virtualization, Container, Live Migration

ACM Reference Format:
Spoorti Doddamani, Piush Sinha, Hui Lu, Tsu-Hsiang K. Cheng, Hardik H. Bagdi, and Kartik Gopalan. 2019. Fast and Live Hypervisor Replacement. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313821

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313821

1 Introduction

Virtualization-based server consolidation is a common practice in today's cloud data centers [2, 24, 43]. Hypervisors host multiple virtual machines (VMs), or guests, on a single physical host to improve resource utilization and achieve agility in resource provisioning for cloud applications [3, 5–7, 50]. Hypervisors must be often updated or replaced for various purposes, such as for applying security/bug fixes [23, 41] adding new features [15, 25], or simply for software rejuvenation [40] to reset the effects of any unknown memory leaks or other latent bugs.

Updating a hypervisor usually requires a system reboot, especially in the cases of system failures and software aging. Live patching [61] can be used to perform some of these updates without rebooting, but it relies greatly on the 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 VMs from the current host to another host that runs a clean and updated hypervisor. Though widely used, live migrating [18, 27] 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.

In this paper, we present **HyperFresh**, 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 impact on running VMs. Using nested virtualization [12], a lightweight shim layer, called *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 old hypervisor to the replacement hypervisor via intra-host live VM migration.

However, two major challenges must be tackled with this approach. First, existing live migration techniques [18, 27] 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. HyperFresh addresses these two challenges as follows.

First, instead of copying a VM's memory, the hyperplexor relocates the ownership of the VM's memory pages from the old hypervisor to the replacement hypervisor. The hyperplexor records the mappings of the 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 a very low hypervisor replacement time of around 10ms, irrespective of the size of the VMs being relocated. In contrast, traditional intra-host VM migration, involving memory copying, can take several seconds. For the same reason, HyperFresh also scales well when remapping multiple VMs to the replacement hypervisor.

HyperFresh addresses the second challenge of nesting overhead during normal execution 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 HyperFresh 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, HyperFresh uses direct device assignment (VT-d) for emulation-free I/O path to the hypervisor, dedicates physical CPUs to reduce scheduling overheads for the hypervisor, reduces CPU utilization on hyperplexor by disabling the polling of hypervisor VCPUs, and eliminates VM Exits due to external device interrupts.

Finally, as a lightweight alternative to VMs, containers [21, 52–54] can be used to consolidate multiple processes. We demonstrate how the hyperplexor-based approach can be used for live relocation of containers to support replacing the underlying OS. Specifically, we demonstrate sub-second live relocation of a container from an old OS to a replacement OS by combining hyperplexor-based memory remapping mechanism and a well-known process migration tool, CRIU [58]. In this case, the hyperplexor runs as a thin shim layer (hypervisor) beneath two low-overhead VMs which run the old OS and the replacement OS.

Our prior workshop paper [8] presented preliminary results for HyperFresh with hypervisor replacement time of around 100ms. This paper presents a comprehensive design and implementation of HyperFresh with lower replacement time of around 10ms, support for relocating multiple VMs,

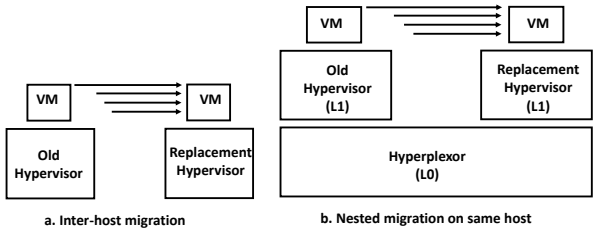


Figure 1. Hypervisor replacement in (a) Inter-host (non-nested) and (b) Intra-host (nested) setting.

optimizations to reduce nesting overheads, and live container relocation for OS replacement. In the rest of this paper, we first demonstrate the quantitative overheads of VM migration-based hypervisor replacement, followed by the HyperFresh design, implementation, and evaluation, 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.

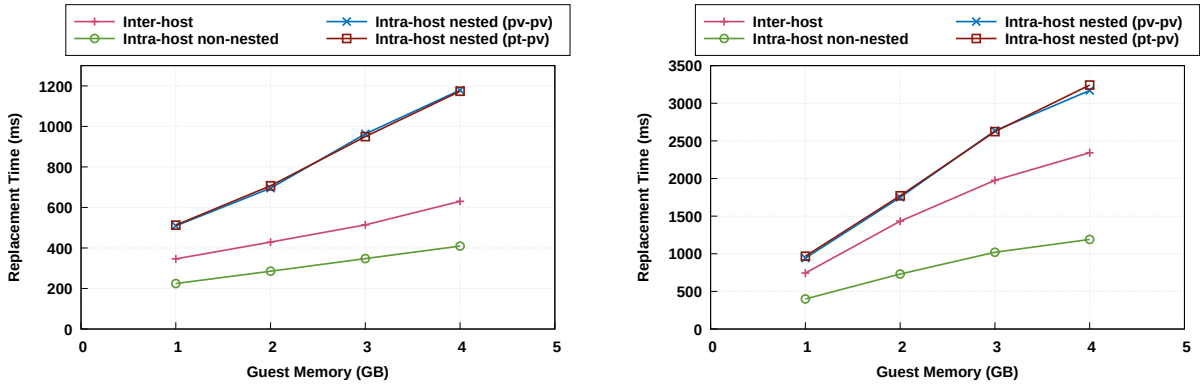
2.1 Using Pre-Copy For Hypervisor Replacement

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 a 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 Figure 1(b) shows, with nested virtualization, a base hyperplexor at layer-0 (L0) can run deprivileged hypervisors at layer-1 (L1), which control 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 old hypervisor to the replacement hypervisor.

2.2 Improving the Default Pre-Copy in KVM/QEMU

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



(a) Idle guest

(b) Busy guest

Figure 2. Comparisons of hypervisor replacement time on the same or different hosts with nested or non-nested setups.

First, we observed that the total live migration time for a 1 GB 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 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 5,000 pages. The latter change yields a low downtime of less than 5 ms.

2.3 Pre-Copy Analysis 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 1a. The source machine runs the old 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 1b. The source L1 runs the old 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 [57].
- **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).

The experiments were conducted using machines having 10-core Intel-Xeon 2.2 GHz processors with 32GB memory and 40 Gbps Mellanox ConnectX-3 Pro network interface. Table 1 in Section 4.1 lists the memory and processor configurations in L0, L1, and L2 layers for these cases.

Figure 2a plots the total migration time for an idle VM when the VM's memory size is increased from 1 GB to 4 GB. 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 1 GB VM to 3,200 MB for a 4 GB 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 para-virtual *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. 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 0.4 seconds (for idle VM case) to more than 1.19 seconds (for busy VM case) to migrate a single 4 GB 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 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, we present our approach and show that our memory remapping-based technique can achieve sub-10ms live hypervisor replacement times.

3 HyperFresh

The key idea behind HyperFresh is as follows: Instead of copying memory of VMs from the old hypervisor to the co-located replacement hypervisor, HyperFresh transfers the ownership of VMs' physical memory pages via page table remapping. Such memory ownership relocation is fast and leads to sub-10 ms live hypervisor replacement. Further, HyperFresh includes optimizations to 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.

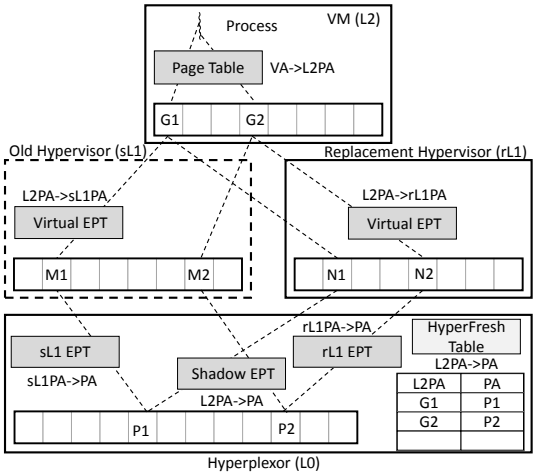


Figure 3. Memory translations for hypervisor replacement.

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) — the virtualized view of memory seen by the VM. The hypervisor uses another page table, the extended page table (EPT), to map GPA to its physical addresses (PA). As with traditional page tables, an EPT is constructed incrementally 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 old L1 hypervisor which in turn runs on the thin L0 hyperplexor. To replace the old 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 hyperplexor (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 old 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 old 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 switch over 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 old 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 old hypervisor. In other words, the shadow EPT reconstructed for the relocated VM ends up being identical to its erstwhile shadow EPT used under the old hypervisor, while the new intermediate mapping tables are correspondingly adjusted. We nest describe low-level details of HyperFresh implementation.

3.3 Implementation Details

We have developed a prototype of HyperFresh using the KVM/QEMU virtualization platforms with Linux kernel version 4.13.0 and QEMU version 2.9.0. The VM runs unmodified Linux kernel version 4.13.0 and uses para-virtual I/O devices. Our solution was implemented by modifying QEMU's live migration mechanism, besides modifications to KVM to implement page remappings described above.

KVM is a linux kernel module which is responsible for core functionalities such as memory management. QEMU is a user space process that emulates its I/O operations and manages VM control operations, including live migration and checkpointing. A part of QEMU's virtual address space is assigned to the virtual machine as guest address space. QEMU process sets up the VCPU threads and virtual I/O devices for the guest. QEMU interacts with KVM hypervisor through ioctl()s. The execution of any privileged instruction by the guest causes a trap to the KVM hypervisor. Traps due to EPT faults are handled by KVM itself whereas the traps due to I/O are forwarded to QEMU.

Different from QEMU's live migration, HyperFresh'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.

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

To construct the HyperFresh table, the hyperplexor needs a list of L2PA-to-L1PA page mappings of the VM from the old hypervisor, using the virtual EPT table. The old hypervisor invokes a `hypercall_put` hypercall multiple times to pass a complete list of L2PA-to-L1PA 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 L2PA-to-physical page mapping into the *HyperFresh table*.

To relocate the ownership of a L2 VM's memory to the replacement hypervisor, the hyperplexor needs the list of L2PA-to-L1PA 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 L2PA-to-L1PA page mapping, the hyperplexor (1) looks up the *HyperFresh table* to find the L2PA-to-PA mapping with the L2 page number as the key; and (2) installs the L1PA-to-PA page mapping in the replacement hypervisor's EPT table. More specifically, HyperFresh 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 the TLB before resuming the relocated VM.

3.4 Reducing 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. We apply several optimizations to reduce nesting overheads, provide the hypervisor with enough resources, and reduce the hyperplexor's footprint during normal execution. the These optimizations are optional, not essential, for the replacement mechanism described earlier.

SR-IOV NICs: To mitigate I/O emulation overheads, HyperFresh uses direct device assignment of network interface cards (NIC) to the hypervisor. Existing virtualization techniques (e.g., KVM [34]) support the full virtualization mode through device emulation [55] and para-virtualization using virtio drivers [10, 49] in VMs. For example, QEMU [11] 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 [19] enables a single network device to present itself as multiple virtual NICs to VMs through virtual functions. Using a virtual function, a VM directly interacts with the network device — with Intel's hardware support, VMs can directly access device memory through IOMMU which converts VMs physical addresses to host physical addresses.

We use virtual functions to achieve network performance in nested guest to match the performance of a single-level guest and minimize the CPU Utilization in the hyperplexor. The VFIO [59] driver supports direct device assignment to a virtual machine. As shown in Figure 4 the old and the replacement hypervisors are each assigned one virtual function. The guest running on the hypervisor uses para-virtualized driver to run the I/O workloads.

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.

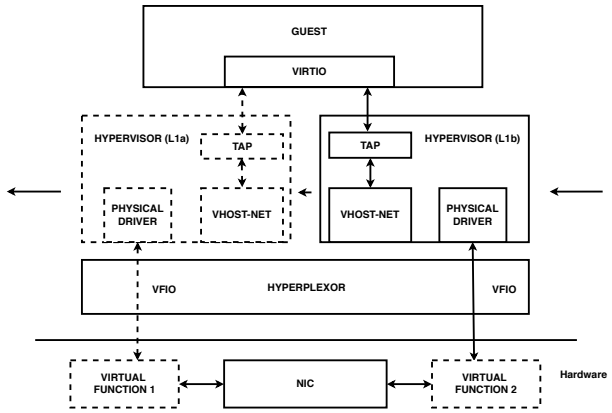


Figure 4. Reducing nesting overheads of hyperplexor using direct-device assignment to hypervisor.

We enable posted interrupt feature on the hyperplexor and deliver interrupts from the NIC directly to the hypervisor without causing exits.

Disabling HLT Exit Polling: 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.

Dedicated Cores: All system processes, including QEMU, in the hyperplexor are pinned to run on two CPU cores whereas the hypervisor's VCPUs run on dedicated cores. We enable the `isolcpus` feature in Linux to isolate and pin hypervisor VCPUs from hyperplexor's system processes.

Hyperplexor Memory Footprint: We configured the hyperplexor with only essential packages, device drivers and services to reduce its memory usage (without a VM) to around 90 MB. In comparison a default Ubuntu server takes around 154 MB memory. With more careful configuration tuning, the hyperplexor's footprint could possibly be reduced even further. The userspace processes and services that are not necessary to run the L1 hypervisor with direct-device assignment were identified and removed.

3.5 HyperFresh for OS Replacement

Using containers to host applications becomes a common usage scenario in today's cloud [21, 51–54]. Under this scenario, the act of OS replacement can be viewed as relocating

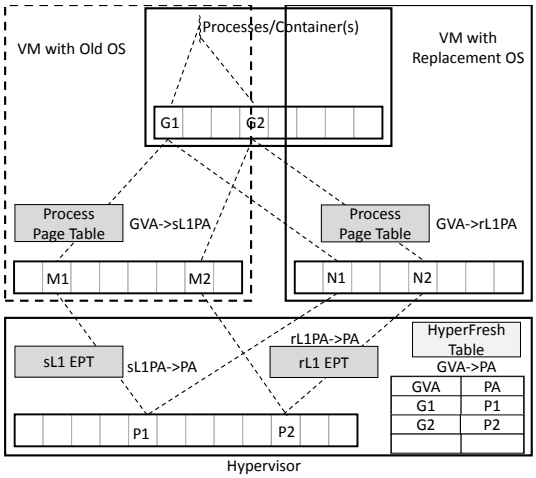


Figure 5. Memory translations for OS replacement.

the state of all containers from the old VM with the old OS to a new VM with the replacement OS. Again, we leverage intra-host live migration to avoid inter-host communication.

Overview: As shown in Figure 5, the page table of a process (belonging to 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). For OS replacement, the hypervisor creates a new VM with the replacement OS, and transfers the ownership of all the container's pages to this new VM. HyperFresh constructs a per-process table, similar to that described for hypervisor replacement, to store the virtual-to-physical memory mappings of the container from old OS. These mappings are used by the new OS for reconstructing the memory address space of the container during OS replacement. The replacement process on the new VM first reconstructs all the GVAs of the container. The new VM's OS then allocates corresponding GPAs and sets up the GVA-to-GPA page mappings in the page table of the container. Finally, the new VM's OS invokes a series of hypercalls to the hypervisor, which installs the GPA-to-PA page mappings in the new VM's EPT table using the HyperFresh table.

Implementation: We implemented a container relocation prototype for OS replacement based on CRIU [58] — a well-known checkpoint/restore tool implemented in the user space. CRIU consists of two stages: checkpointing (on the source VM) and restoration (on the destination VM). In the checkpointing phase, on the source VM (with the old OS) HyperFresh's collects all the state of a container and its processes (such as file descriptors, memory maps, registers, namespaces, cgroups, etc) except for the memory content and builds the HyperFresh table. To build the HyperFresh table for OS replacement, a guest kernel module invokes multiple `hypercall_put` hypercalls to pass a list of GVA-to-GPA page mappings to the hypervisor. For each received GVA-to-GPA mapping, the hypervisor translates GPA to PA (i.e., using the VM's EPT), and stores the corresponding GVA-to-PA page mapping in the HyperFresh table. In the restoration phase on the destination VM (with the replacement OS), HyperFresh restores the execution state, relocates the memory ownership, and resumes each process of the container, while the source VM unmaps the pages of each relocated process from its address space. This process is repeated until all processes of a container are moved to the destination VM. Restoring the address space of a container on the new VM requires reconstructing the GVAs, GPAs, and their mappings. Given the GVA-to-GPA page mappings of the container, the new VM's kernel invokes multiple `hypercall_map` hypercalls to pass a list of GVA-to-GPA mappings of the container to the hypervisor. For each received GVA-to-GPA mapping, the hypervisor looks up the HyperFresh table to find the GVA-to-PA mapping with GVA as the key and installs the GPA-to-PA mapping in the EPT page table of the new VM.

Table 1. Setup to compare hypervisor replacement time.

	L0	L1	L2
Inter-Host	1GB, 1–4CPUs	-	-
Intra-host Non-Nested	8GB, 4CPUs	-	1GB, 1–4VCPUs
Intra-host Nested	32GB, 10CPUs	8GB, 4VCPUs	1GB, 1–4VCPUs
HyperFresh	32GB, 10CPUs	8GB, 4VCPUs	1GB, 1–4VCPUs

4 Evaluation

This section evaluates our HyperFresh prototype in terms of replacement times, performance impact on applications before, during and after hypervisor replacement, and the performance of live container relocation. We run our experiments on machines equipped with a 10-core 2.2 GHz Intel Xeon Broadwell CPU, 32 GB memory and one 40 Gbps Mellanox ConnectX-3 Pro network interface. All experimental results are averaged over five or more runs; standard deviations range between 0 to 2% across all experiments.

4.1 Hypervisor Replacement Time

We compare the hypervisor replacement time under HyperFresh with the four cases based on the optimized pre-copy migration which were described earlier in Section 2.3. Table 1 lists the memory and processor configurations in L0, L1, and L2 layers for these cases.

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 and 0.6 seconds for 4 GB memory. 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, and 1.17 seconds for 4 GB memory. In contrast, the total hypervisor replacement time under HyperFresh is very low — 10 ms. Further, the replacement time remains constant as the VM's memory size increases. It is because, in HyperFresh, the memory is remapped between the L1 hypervisors and such operations are performed out of the critical

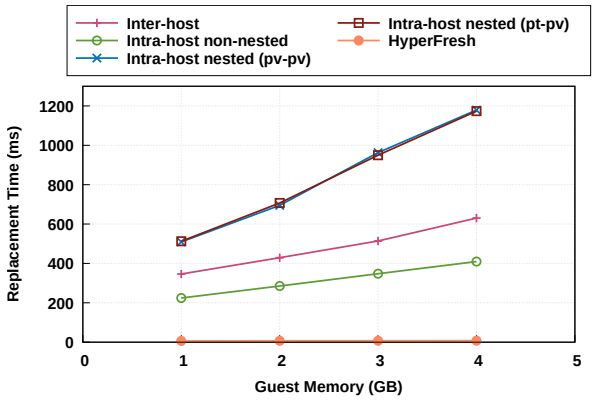


Figure 6. Hypervisor replacement time: one idle VM with varying memory sizes.

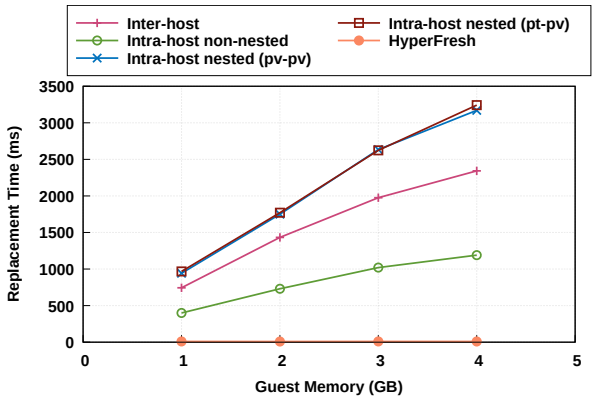


Figure 7. Hypervisor replacement time: one busy VM with varying memory sizes.

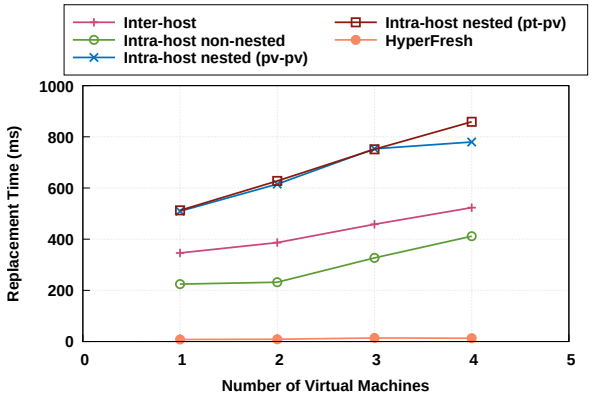


Figure 8. Hypervisor replacement time: multiple idle VMs.

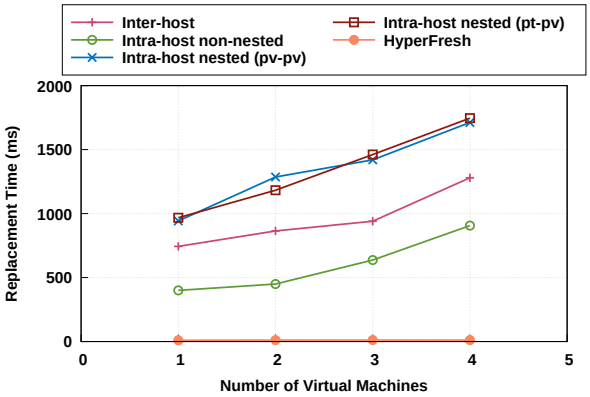


Figure 9. Hypervisor replacement time: multiple busy VMs.

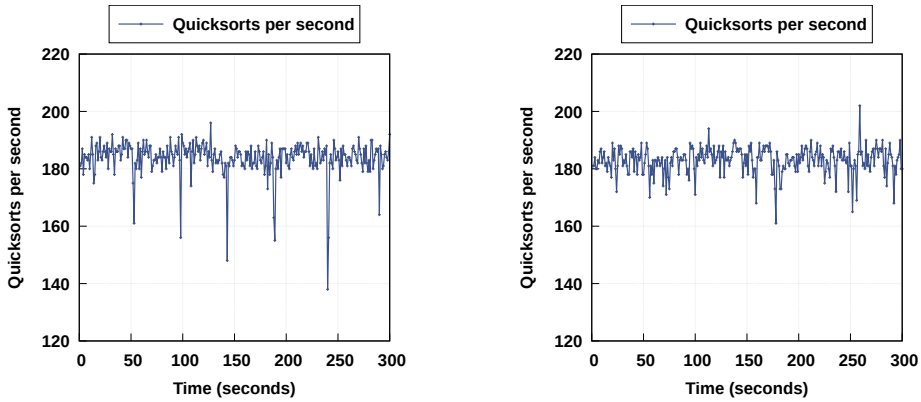
path of the VM state transfer; the replacement time only comprises of the time to transfer the VCPU and I/O device state information, which is fairly constant. However, under both the inter-host and intra-host cases, transferring the memory pages using TCP connections accounts for higher total migration time and thus higher replacement time.

Next, we use a busy VM with varying memory sizes. The busy VM runs a synthetic "writable working set" benchmark which writes to memory at a fixed rate of 5,000 pages per second. In Figure 7, under all pre-copy live migration cases the replacement time increases as the memory size of the VM increases. The replacement time is higher than that under the idle VM, because dirtied pages of the busy VM are transferred in multiple rounds. With HyperFresh, the replacement time remains constant around 10ms irrespective of the memory dirtying rate or guest memory sizes.

Multiple VMs: We also measure the hypervisor replacement time by relocating multiple VMs from an old hypervisor to a replacement hypervisor. We vary the VM number from 1 to 4. Each VM is configured with 1 GB memory; all the VMs start the migration at around the same time. We consider the following two cases: (1) with all VMs being idle and (2) with all VMs being busy dirtying 5,000 pages per second. In Figure 8, with all VMs being idle, 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 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 live migration (for both pv-pv and pt-pv setups), the migration time increases to 1 second for 1 VM and 1.75 seconds for 4 VMs. With HyperFresh, the time to replace the hypervisor remains around 10 ms with either idle or busy VMs. The replacement time does not increase as the number of VMs increases, because the VMs' memory is remapped and only the VCPU and I/O device states are transferred during hypervisor replacement.

4.2 Performance During Hypervisor Replacement

We measure the application performance during hypervisor replacement between HyperFresh and the intra-host nested (pt-pv) case. We use a CPU-intensive benchmark, Quicksort, which first allocates 1 GB memory. Then each it iteratively takes different 200 KB (50 page) regions from the



(a) Optimized pre-copy **(b)** HyperFresh.

Figure 10. Quicksort performance over multiple hypervisor replacements.

Table 2. Setup to compare nesting overhead reduction.

	Hyperplexor	Hypervisor	Guest
Host	8GB, 2–4CPUs	-	-
Non-Nested	16GB, 8CPUs	-	8GB, 2–4VCPUs
Nested	32GB, 10CPUs	16GB, 8VCPUs	8GB, 2–4VCPUs
HyperFresh	32GB, 10CPUs	16GB, 8VCPUs	8GB, 2–4VCPUs

Table 3. Benchmark Performance and CPU Utilization. BW: Bandwidth, CPU: CPU Utilization, Exec: Execution time

	iperf		Kernbench		Quicksort	
	BW (Gbps)	CPU (%)	Exec (s)	CPU (%)	Exec (s)	CPU (%)
Host	37.6	85.8	322.8	194.4	66.3	97.7
Non-nested	36.3	237.1	336.2	197.3	66.5	102.1
Nested	25.2	332	361.8	198.3	67.7	103
Hyperfresh	36.1	289.4	361.4	195.6	67.7	103.9

pre-allocated memory, writes random integers into it, and sorts the integers. In Figure 10, we conduct 6 hypervisor replacements over a 300-second time window for both HyperFresh and the intra-host nested case. We measure the number of QuickSort operations per second during both regular execution and replacement.

In Figure 10(a), in the intra-host nested (pt-pv) case, there is no significant performance degradation during pre-copy iterations. However, the sharp performance dip is observed during the stop-and-copy phase, during which the VM is paused. In Figure 10(b), with HyperFresh, the performance of QuickSort is not significantly affected during the stop-and-copy phase. It is because, unlike the intra-host live migration case, HyperFresh does not transfer any pages during the stop-and-copy phase, leading to much shorter downtime and hence less performance impact.

4.3 Nesting Overhead

To understand nesting overheads during normal execution and the impact of our optimizations, we measure the performance of various applications running on (a) the native host (i.e., without virtualization); (b) a VM under non-nested virtualization (with the para-virtualized driver); (c) an L2 VM under nested virtualization (with the pt-pv configuration), and (d) an L2 VM under HyperFresh with optimizations. The configurations are listed in Table 2. Particularly, to conduct fair comparisons among the above four cases, we ensure that the host or VMs running applications are configured with the same amount of resources — 2 VCPUs and 8 GB memory.

Iperf [29] is used to measure network performance. We run the iperf client on the native host or VM under test, and the iperf server on a separate host (with sufficient resources). The iperf client and server are connected via a 40 Gbps network. We measure the averaged TCP throughput over 100 seconds. **Kernbench** [35] uses multiple threads to repeatedly compile the Linux kernel source tree. In our experiments, the kernel sources are loaded into an in-memory `tmpfs` file system in the guest compiled for five iterations with two threads, equal to the number of guest VCPUs. **Quicksort** is a memory and CPU-intensive benchmark described earlier in Section 4.2.

In Table 3, we observe that under HyperFresh (with optimizations) the performance of iperf is higher than the default nested case without optimizations, and very close to the non-nested case (under single-level virtualization). Table 3 shows the total CPU utilization as measured in the L0 layer. In the default nested case, when the NIC card is assigned to the hypervisor directly (i.e., pass-through), the CPU utilization of the hyperplexor is expected to be small, as the network packets are directly delivered to the L1 hypervisor. However, Table 3 shows that the CPU utilization of the hyperplexor (i.e., the system mode) in the default nested case is very high.

It is because, the hyperplexor burns CPU cycles when it executes HLT instruction as explained in Section 3.4. In contrast, HyperFresh reduces the CPU utilization of the hyperplexor by disabling event polling on HLT exits in L0, i.e. by disabling `halt_poll_ns`. This allows than the L1 hypervisor to use more CPU time to process network traffic than in the default nested case, leading to higher iperf throughput.

We also observe that, although HyperFresh configuration achieves comparable iperf throughout compared to the non-nested guest case in Table 3, the CPU utilization by the hyperplexor in HyperFresh is still significant, at around 90%. This overhead can be attributed to the cost of processing VM exits (with `halt_poll_ns` disabled in L0); frequent HLT instructions executed by the guest and hypervisor VCPUs must be trapped by the hyperplexor and either emulated (for hypervisor) or forwarded to the L1 hypervisor (for guest). We are investigating ways to eliminate this overhead possibly by disabling HLT exits from hypervisor's and/or the guest's VCPUs.

For in-memory Kernbench, nesting introduces a performance reduction of around 7.6% over non-nested case due to repeated VM Exits caused by guest invoking CPUID instruction, which must be trapped and emulated by L1 KVM. For Quicksort, nesting introduces only a minor performance reduction of 1.8% compared to non-nested case due to nested page fault processing. HyperFresh optimizations do not significantly reduce nesting overheads over the default nested case for these two in-memory CPU-intensive workloads, since our optimizations do not target VM Exits due to CPUID instructions and nested page faults. This points to another potential avenue for future optimization.

4.4 Performance After Hypervisor Replacement

After hypervisor replacement, the guest needs to update the shadow EPT, EPT of L1 and virtual EPT on page faults. In consequence, the performance of the applications in the guest could be impacted as the guest populates the page tables on demand. We run the experiments using in-memory Kernbench, SPEC CPU 2017 [13], Sysbench [36], and httperf [44] to measure this performance impact, as summarized in Table 4. We configure the guest with 18 GB memory and 2 VCPUs; the hypervisor with 20 GB memory and 5 VCPUs, and the hyperplexor with 32 GB memory and 10 CPUs. We measure the performance by running these benchmarks in the guest before and right after hypervisor replacement.

First, Kernbench compiles the kernel with two threads, equal to the number of the guest VCPUs. The kernel consumes 8% of the guest memory. The compilation time increases by 1.2% after hypervisor replacement. Further, the CPU-intensive benchmarks mcf_s and bwaves_s are run with 2 threads, cactuBSSN_r, omnetpp_r and xalancbmk_r with 1 thread from SPEC CPU 2017. The performance degrades by 3.05%, 0.81%, 2.71%, 1.23%, and 0.42% respectively at the new hypervisor after replacement, depending on the size of the memory used by the benchmark. Sysbench [36] generates an OLTP workload to stress a MYSQL database. We measure the read-only and read-write transactions per second by querying an in-memory 4 GB MYSQL database in guest over a 300 seconds. We observe that the read-only and read-write transaction rates decrease by 0.74% and 4% respectively. Lastly, httperf [44] is used to measure the performance of Apache (version 2.4.18) web server running in the guest. We measure the response latency of the requests sent from another client host machine at the rate of 5,000 requests per second. We observe that the difference in the absolute response latency before and after replacement is small – 0.5 ms vs. 0.6 ms.

However, this post-relocation slowdown is not unique to HyperFresh. Traditional pre-copy migration in KVM/QEMU also lazily populates guest EPT entries on EPT faults after migration. This could be addressed by pre-populating the relevant EPT entries prior to resuming the guest.

4.5 Container Relocation

In this section, we evaluate the performance of container relocation using HyperFresh to support live OS replacement. we run the experiments on a server machine equipped with two 6-core 2.1 GHz Intel Xeon CPUs and 128 GB memory. We use KVM/QEMU to run source and destination VMs on the same server, each with 4 VCPUs and 4 GB memory.

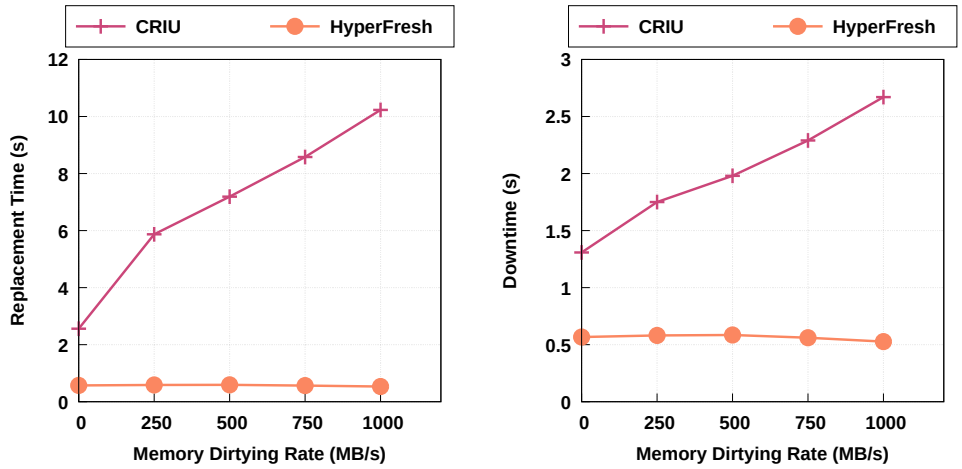
Replacement Time: We measure OS replacement time, as time to relocate a container, under HyperFresh and compare it with the pre-copy live migration approach implemented upon Phaul [46]. We run a container with one process which allocates 1 GB memory and then continuously performs writes to the memory pages at varying rates from 0 GB/second to 1 GB/second. We measure the total OS replacement time and the container downtime.

Figure 11 shows that, with the pre-copy live migration approach the total OS replacement time and downtime increase as the memory dirty rate increases. For example, the replacement time is around 2.5 seconds when the dirty rate is 0, and increases to 10 seconds when the dirty rate is 1 GB/second. The downtime is around 1.7 seconds when the dirty rate is 0, and increases to 2.7 seconds when the dirty rate is 1 GB/second. In contrast, with HyperFresh, the total replacement time and downtime remain constant, around 0.5 seconds (i.e., the total replacement time equals to the downtime in HyperFresh). It is because, HyperFresh relocates the memory ownership instead of copying pages leading to much shorter OS replacement and downtime.

Notice that, the OS replacement time (e.g., 0.5 seconds) is higher than the hypervisor replacement time (e.g., 10 ms). The main reason is that, HyperFresh's OS replacement builds on a user space checkpointing/restoration tool (i.e., CRIU), which incurs high overhead in collecting state of processes during which processes are paused. A low-overhead, kernel-level solution is a subject of our ongoing investigations.

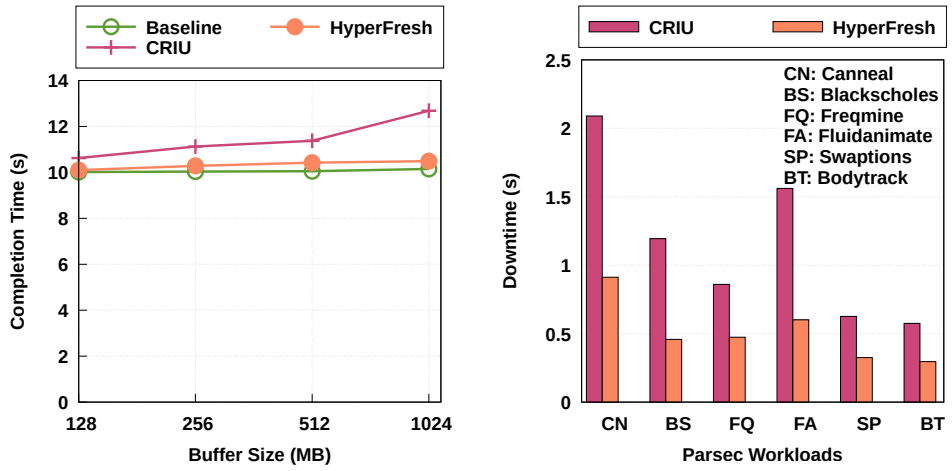
Table 4. Performance slowdown after hypervisor replacement due to on-demand population of page table entries

Benchmarks		Workload (GB)	Before Replacement Warm VM Performance	After Replacement Cold VM Performance	Slowdown (%)
Kernbench		1.4	361.4s	365.8s	1.2
SPEC CPU	mcf_s	6.5	794.8s	819s	3.04
	cactuBSSN_r	13.5	304.6s	313s	2.8
	omnetpp_r	3.6	549.3s	556s	1.23
	xalancbmk_r	3.2	412.3s	414s	0.41
	bwaves_s	3.5	4581s	4618s	0.81
Sysbench	read-only	4	1358.8 trans/s	1348.6 trans/s	0.75
	read-write	4	646.6 trans/s	625.1 trans/s	3.3
httperf		12KB/5000	0.5ms	0.6ms	20



(a) Replacement time. **(b)** Downtime.

Figure 11. Container relocation with memory-intensive workloads.



(a) Sysbench. **(b)** Parsec.

Figure 12. Comparison of application-level performance during container relocation.

Performance Impact: We evaluate two benchmarks, Sysbench [37] and Parsec [16], when they run in a container during live relocation. Sysbench provides multi-threaded memory testing by reading from or writing to preallocated memory. We vary the buffer sizes from 128 MB to 1 GB, and test write operations to this buffer. We compare three cases: (a) the baseline without container relocation; (b) pre-copy approach using CRIU during container relocation; and (c) HyperFresh during container relocation. In Figure 12(a), the completion time under HyperFresh is almost the same as

the baseline. In contrast, the pre-copy based approach takes longer time to complete for each run and its completion time increases as the buffer size increases; the larger the buffer size, the more memory is dirtied and copied using pre-copy.

Parsec is a shared-memory multi-threaded benchmark. We run a variety of sub-workloads of Parsec during container relocation and compare the total downtime during the OS replacement between HyperFresh and the pre-copy. Figure 12(b) shows that the total downtime for workloads with higher memory usage is longer than those with lower memory usage. For example, *canneal* and *fluidanimate* have very high memory usage, whereas *swaptions* and *bodytrack* use only a small amount of memory. Hence, *canneal* and *fluidanimate* have longer downtime during the OS replacement than swaptions and bodytrack. We observe that, the total downtime of all workloads with HyperFresh is almost half that with the pre-copy based approach.

5 Related Work

Software aging [45] is the phenomenon where software errors, such as memory leaks, accumulate over time, leading to performance degradation or complete system failure. Software Rejuvenation [28] is a proactive technique where the system is periodically restarted to a clean internal state to counter software aging. In cloud environments, software aging of hypervisors and OS is of utmost concern, where the impact of their failures could be widespread.

In cold rejuvenation of hypervisors, all VMs must be restarted whereas in warm rejuvenation the VMs' memory state is preserved in persistent memory and restored after hypervisor rejuvenation, avoiding costly read/writes to persistent storage. Roothammer [38] proposed warm reboot to rejuvenate a Xen hypervisor and avoids cache-misses by preserving the VM images in the main memory.

A simpler approach is to live migrate VMs to a different host. Live migration [18, 27] reduces the downtime by letting the virtual machines run when the memory is copied in multiple rounds to another host. However, live migration also incurs network overhead due to large memory transfers. In this paper, we eliminate the memory copies for intra-host migration by page relocation on the same host. ReHype [40] performs a micro-reboot [17] of the hypervisor by preserving the state of the VMs. The state of the rebooted hypervisor is then re-integrated with the state of the preserved VMs. Since this reintegration depends heavily on the hypervisor's state, the success rate is low. Kourai et. al proposed VMBeam [39] 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 approach, VMBeam takes 16.5 seconds to migrate a 4 GB VM with zero memory copy, potentially due to non-live or sub-optimal transfer of memory maps. Use of a hyperplexor for guest memory co-mapping has also been proposed to support 'hypervisor-as-a-service' model [26, 60] for cloud platforms. HyperFresh's hyperplexor uses guest memory remapping to enable fast hypervisor replacement. Additionally, HyperFresh also addresses reduction of nesting overheads during normal execution and live container relocation.

Kernel patching is a complex process of replacing original outdated functions or data structures with new ones. Ksplice [4], Kpatch [31, 32], Kgraft [33] 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 is retried 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 the new functions while the other processes use the updated functions. Live patching [30] is a combination of Kpatch and Kgraft kernel patching methods. Kernel patching technique can be useful for applying simple fixes to delay system failure until the next maintenance period, but it still requires an OS reboot at some stage. Major changes to the kernel also require immediate reboot to take effect. Further these techniques cannot patch asm, vdso, or functions that cannot be traced. In contrast, HyperFresh bypasses these problems by relocating the memory of containers to a fresh co-located instance of the kernel. Process and container live migration techniques have also been extensively studied [9, 14, 20, 22, 42, 47, 48, 56, 58, 62], but all require the transfer of the memory contents via network. HyperFresh combines memory remapping with intra-host process migration used by CRIU [58] to eliminate memory copying, making it suitable for live OS replacement.

6 Conclusion

We have presented HyperFresh, a faster and less disruptive live hypervisor replacement approach. Leveraging nested virtualization, a thin hyperplexor transparently relocates the ownership of VMs' memory pages from an old hypervisor to a new replacement hypervisor on the same host, without memory copying overheads. HyperFresh also includes a number of optimizations to mitigate nesting overheads during normal execution of VMs. We also demonstrated how hyperplexor-based remapping approach can be applied for live relocation of containers to replace the underlying OS. Evaluations of HyperFresh show around 10ms hypervisor replacement and sub-second container relocation times.

The source code for the prototype described in this paper is available at https://github.com/osnetsvn/vfresh.git

Acknowledgment

We would like to thank our shepherd Tim Merrifield and anonymous reviewers for their thoughtful feedback. This work is supported in part by the US National Science Foundation through award CNS-1527338.

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] Hardik Bagdi, Rohith Kugve, and Kartik Gopalan. Hyperfresh: Live refresh of hypervisors using nested virtualization. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 18. ACM, 2017.

[9] Barak, A. and Shiloh, A. A Distributed Load-Balancing Policy for a Multicomputer. In *Software-Practice and Experience*, volume 15, pages 901–913, 1985.

[10] 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.

[11] Fabrice Bellard. QEMU: A fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[12] 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.

[13] SPEC CPU 2017 benchmark suite. https://www.spec.org/cpu2017/.

[14] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczinski, M., Becker, D., Chambers, C., and Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, 1995.

[15] 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.

[16] 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/.

[17] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[18] 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.

[19] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *First Workshop on I/O Virtualization, San Diego, CA*, 2008.

[20] Douglis, F. and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. In *Software-Practice and Experience*, volume 21, pages 757–785, 1991.

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

[22] Engler, D. R., Kaashoek, M. F., and O'Toole, J. J. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 26–284, 1995.

[23] 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.

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

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

[26] Kartik Gopalan, Rohit Kugve, Hardik Bagdi, Yaohui Hu, Daniel Williams, and Nilton Bila. Multi-hypervisor virtual machines: Enabling an ecosystem of hypervisor-level services. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 235–249. USENIX Association, 2017.

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

[28] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *IEEE International Symposium on Fault-Tolerant Computing*, 1995.

[29] iperf: The network bandwidth measurement tool. https://iperf.fr/.

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

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

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

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

[34] 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.

[35] C. Kolivas. Kernbench. http://ck.kolivas.org/apps/kernbench/.

[36] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

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

[38] Kenichi Kourai and Shigeru Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *Proc. of Dependable Systems and Networks (DSN)*, pages 245–255, 2007.

[39] 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.

[40] Michael Le and Yuval Tamir. ReHype: Enabling VM survival across hypervisor failures. *ACM SIGPLAN Notices*, 46(7):63–74, 2011.

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

[42] Litzkow, M., Livny, M., and Mutka, M. Condor: A Hunter of Idle Workstation. In *Proc. of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 104 – 111, 1988.

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

[44] David Mosberger and Tai Jin. httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[45] David Lorge Parnas. Software aging. In *Proc. of the 16th international conference on Software engineering*, pages 279–287, 1994.

[46] P.Haul. https://criu.org/P.Haul.

[47] Platform ComputIng. LSF User's and Administrator's Guides. In *Platform Computing Corporation*.

[48] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization. In *Proc. of the 15th Symposium on Operating Systems Principles (SOSP)*, pages 314–324, 1995.

[49] Rusty Russell. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

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

[51] Amazon Elastic Container Service. https://aws.amazon.com/ecs/.
[52] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/.
[53] IBM Cloud Kubernetes Service. https://www.ibm.com/cloud/container-service.
[54] VMware Pivotal Container Service. https://cloud.vmware.com/vmware-pks.
[55] 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*, pages 1–14, 2001.
[56] Theimer, M., Lantz, K., and Cheriton, D. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on OS Principles*, pages 2–12, 1985.
[57] Michael S. Tsirkin. vhost-net: A kernel-level virtio server, 2009. https://lwn.net/Articles/346267/.
[58] Checkpoint/Restore In Userspace. https://criu.org/Main_Page.
[59] VFIO Driver. https://www.kernel.org/doc/Documentation/vfio.txt.
[60] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. *ACM SIGPLAN Notices*, 51:79–92, 2016.
[61] Xen Project. Live Patching of Xen. https://wiki.xenproject.org/wiki/LivePatch.
[62] Zayas, E. Attacking the Process Migration Bottleneck. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 13–24, 1987.