HyperFresh: Live Refresh of Hypervisors Using Nested Virtualization

Hardik Bagdi
Computer Science
Binghamton, NY, USA
hbagdi1@binghamton.edu

Rohith Kugve
Computer Science
Binghamton, NY, USA
rkugver1@binghamton.edu

Kartik Gopalan
Computer Science
Binghamton, NY, USA
kartik@binghamton.edu

ABSTRACT

Bugs in hypervisors are becoming common as hypervisors grow in size and complexity. Latent bugs, such as memory leaks, can lead to hypervisor failures resulting in complete loss of all its virtual machines (or guests). However, reliable operation of hypervisors, even in the presence of bugs, is critical in cloud platforms. A hypervisor can be regularly restarted, with or without updates, to reset its state, preempt unexpected failures, and extend its operational uptime. However, a hypervisor restart and update is highly disruptive to guests, which must be either migrated to another host, or shut down. We propose HyperFresh, a fast and guest-transparent approach to replace an old, and possibly unstable, hypervisor with a fresh one beneath live unmodified guests. Using nested virtualization, a thin hyperplexor layer runs the hypervisor and its guests. To prepare for refresh, all guest memory is co-mapped in advance to a fresh co-resident hypervisor. When the refresh operation is triggered, the hyperplexor simply switches control of the guest VCPUs and I/O state to the new hypervisor. Our HyperFresh prototype on the KVM/QEMU platform yields switching times of around 100ms with low performance impact on guest workload.

KEYWORDS

Hypervisor, Reliability, Virtual Machines, Virtualization

ACM Reference Format:
Hardik Bagdi, Rohith Kugve, and Kartik Gopalan. 2017. HyperFresh: Live Refresh of Hypervisors Using Nested Virtualization. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
https://doi.org/10.1145/3124680.3124734

1 INTRODUCTION

Modern cloud platforms [10, 18, 26] extensively use hypervisors to run system Virtual Machines (VM), or guests, to service customer workloads. Hypervisor software has become increasingly complex in response to an explosive growth of multi-tenant datacenters, much like the complexity of a monolithic operating system (OS) from the pre-cloud era. Unsurprisingly, increasing hypervisor complexity has led to a corresponding increase in bugs in various hypervisors [2, 15, 19, 30]. Building bug-free hypervisors is practically impossible given the complexity and heterogeneity within modern datacenters. When a hypervisor fails, all guests controlled by that hypervisor fail. Consequently, a hypervisor failure in a production setting has a larger blast radius than an just application or a guest OS failure.

Fixes for known hypervisor bugs are usually released soon after the bugs are discovered. Nonetheless, *latent* bugs can lurk around in operational hypervisors for a long time. Latent bugs are bugs that haven't yet been fixed in an operational hypervisor; they can be either undiscovered bugs, or known bugs whose fixes haven't yet been released, or applied, to the running hypervisor. Latent bugs can impact system correctness over time and trigger unpredictable failures at inopportune moments. For instance, hypervisor-level memory leaks, such as those reported in KVM [31] and Xen [35, 36], may accumulate over time before triggering a system crash or denial-of-service. Similarly, there may be a delay between assigning and dereferencing a `NULL` pointer, or between a buffer overflow and an access to the corresponding corrupted memory.

Software rejuvenation [17] refers to the practice of gratuitously resetting a long-running operational system, or its individual components, in order to reduce the cumulative effects of any latent bugs. Rejuvenation restores a software component to its pristine initial state and discards its old, possibly corrupt, internal state. For example, system administrators routinely upgrade and/or reboot their servers to preempt any latent bugs from crashing the system, as well as to recapture peak performance. While rejuvenation, on its own, cannot identify or much less fix latent bugs, it can temporarily relieve certain bug-induced conditions like memory pressure due to memory leaks, corrupted state due to race conditions, runaway threads, and undetected deadlocks. Such effects might escape detection during traditional testing and debugging cycle but manifest themselves in operational mode over several days, weeks, or longer.

At the same time, any upgrade or reset of a privileged system software, such as hypervisors and OS, is dreaded in production settings due to the possibility of significant service disruptions and downtime. Existing techniques for live patching of hypervisor [1, 6], while useful, rely on the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APSys '17, September 2, 2017, Mumbai, India
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5197-3/17/09. . . $15.00
https://doi.org/10.1145/3124680.3124734

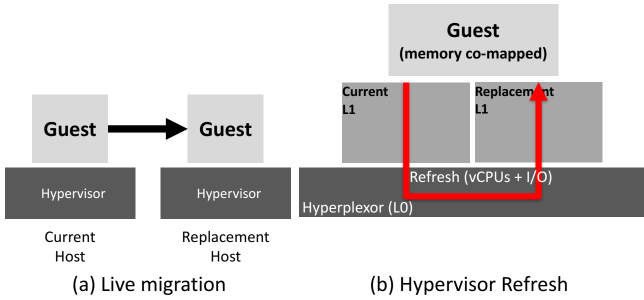

Figure 1: Replacing old hypervisor using (a) inter-host live migration and (b) HyperFresh.

very hypervisor being patched, which could be buggy and unstable.

In this paper, we propose a new technique called **HyperFresh**, or hypervisor refresh, that pro-actively replaces a long-running stale hypervisor with a fresh replacement hypervisor, all without disrupting running guests. The replacement hypervisor could either include bug fixes and updates, or could simply be a freshly initialized instance.

HyperFresh can be viewed as an optimized intra-host live VM migration mechanism between two co-resident hypervisors – the current stale hypervisor and its new replacement hypervisor. Figure 1 illustrates the analogy and differences between HyperFresh and traditional live VM migration [7, 16]. Figure 1(a) shows traditional live migration which transfers a running guest to another host having a clean and updated hypervisor. In traditional live migration, guest memory transfer over the network takes up the most time. Additionally, guest virtual CPUs (VCPUs), virtual I/O devices, and any residual memory pages are transfered during a downtime when the guest is paused; this downtime is generally small, except when using pre-copy [7] migration for guests running write-intensive workloads. Traditional live migration requires another physical machine to be available. It also generates significant network traffic, especially when simultaneously migrating multiple VMs [12, 13].

HyperFresh, shown in Figure 1(b), avoids these overheads when the goal is simply to update the hypervisor running beneath the guests. HyperFresh uses nested virtualization [4] to insert a thin *hyperplexor* layer beneath a traditional full-fledged hypervisor. Nested virtualization enables a guest to act as a hypervisor to another guest. This results in a base hyperplexor at layer-0 (L0) hosting a deprivileged hypervisor at layer-1 (L1) that controls the guest (L2). The hyperplexor's sole task is to periodically replace the currently running hypervisor with a co-resident replacement hypervisor. To speed up guest switch-over between the current and replacement hypervisors, the hyperplexor co-maps the running guest's memory to the replacement hypervisor's address space. Co-mapping bypasses the need to perform expensive copying of the entire guest memory, as required with traditional live migration. The refresh operation completes by transferring control of only guest virtual CPUs (VCPUs) and virtual I/O devices between the two hypervisors, and discarding the old hypervisor.

Guest observes only a small downtime of less than 100 milliseconds during the VCPU and I/O state transfer. Our preliminary results show that hypervisor refresh can be performed as often as every minute without a significant penalty on guest performance. Presently the hyperplexor performs periodic refresh but could use other event-based triggers such as high memory or CPU utilization.

The primary goal of HyperFresh is to shrink the vulnerability window during which a system is exposed to latent hypervisor-level bugs. Note that HyperFresh, as well as earlier techniques [1, 6, 20, 21], cannot fix latent bugs; they can only fix known bugs for which bug-fixes are available. They also cannot ensure the operational correctness of a bug-carrying hypervisor, nor guarantee that a bug won't crash the system before the hypervisor is reset or patched. They do not address bugs in the guest and cannot recover a failed guest. They cannot undo the effects of hypervisor-level bugs that corrupt guest memory since only the hypervisor is reset or patched. Post-failure recovery of a guest can be addressed by other reactive techniques [11, 25] which maintain a remote checkpoint or mirror of the guest, and can complement HyperFresh. HyperFresh is a proactive technique that delays the need to trigger an expensive post-failure recovery mechanism by reducing the likelihood of a fatal failure.

Nested virtualization is experiencing a surge of interest with support from public cloud platforms like Amazon EC2 [26]. Nesting has the potential to improve consolidation [24, 29], security [22, 27], privacy [39], and cross-cloud portability [28, 34]. At the same time, nesting also introduces the overhead of an additional virtualization layer. Although reducing nesting costs in general is beyond the scope of this paper, we note that recent processor support for VMCS Shadowing [33] and I/O support for direct device assignment [38] can be used to reduce overheads from VM exits and I/O virtualization. Furthermore, the hyperplexor has a narrow role of switching the hardware and guests between the old and new hypervisors. Hence, HyperFresh can potentially avoid much of the nesting overhead by assigning dedicated physical memory, CPU, and I/O resources to the hypervisor at L1, and bypass the need to perform any resource management in L0. Finally, solutions such as Microvisor [23] can be adapted to run the hypervisor natively during normal operations while deprivileging the hypervisor to L1 during the refresh operation.

2 DESIGN

HyperFresh reduces the latency of replacing a hypervisor by co-mapping the guest's memory between two co-located hypervisors, minimizing the amount of guest state that needs to be transferred during a refresh operation. A thin *hyperplexor* at L0 runs, and regularly replaces, a deprivileged *hypervisor* at L1. The L1 hypervisor executes guests, which remain unmodified. *Current* hypervisor refers to the L1 hypervisor that is currently hosting the guests whereas *replacement* hypervisor

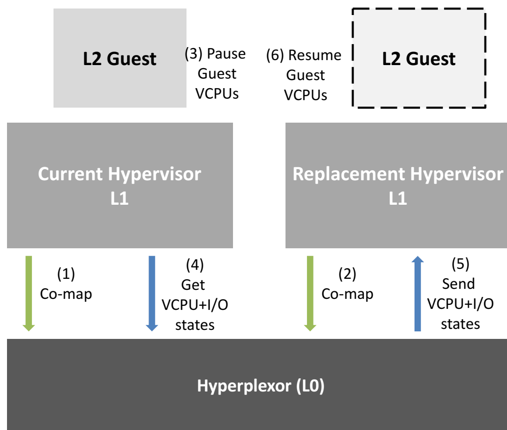

Figure 2: Operations performed during hypervisor refresh

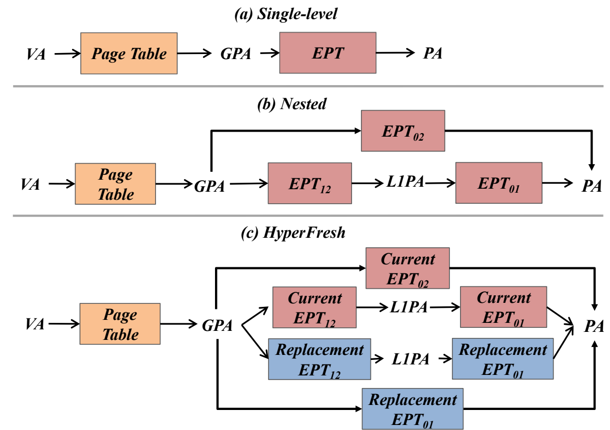

Figure 3: Illustration of memory translation mechanisms.

refers to the one which takes the control of guests after a refresh. Due to its narrow responsibilities, the hyperplexor can potentially omit traditional guest-facing services from its footprint, such as device drivers, complex memory management, file systems, and so forth. Figure 2 illustrates the following sequence of events during a refresh cycle. **Co-mapping**: The current hypervisor shares the memory mappings for its guests with the hyperplexor (Step 1). Later (Step 2), a replacement hypervisor requests the hyperplexor to co-map the memory of the guests running on current hypervisor into its own L1 physical address space (L1PA). The hyperplexor co-maps the guest's physical memory in both hypervisors so that both see a consistent view of each guest. Once all guests are co-mapped, the replacement hypervisor is ready to take control of guest's VCPUs and I/O devices.

Refresh: The refresh operation is initiated by the hyperplexor and is carried out as shown in Steps 3 to 6. The guest is paused on the current hypervisor (Step 3). The control of guest VCPUs and I/O devices is taken away from the current hypervisor (Step 4). Hyperplexor then forwards these states to the replacement hypervisor (Step 5), which finally resumes the guest (Step 6).

Unmapping: Once control of guests is transferred to the replacement hypervisor, the current hypervisor can safely unmap the guest's memory from its address space and safely be powered down without any effect on the guests. The replacement hypervisor becomes the current hypervisor for the next cycle and the process can be repeated.

2.1 Guest Co-mapping

The Memory Management Unit (MMU) in the hardware uses page tables to translate Virtual Addresses (VA) to Physical Addresses (PA). Figure 3 shows the virtual-to-physical address translation for single-level virtualization, nested virtualization, and during guest co-mapping in HyperFresh.

For single-level VMs, an additional level of translation is necessary. A process running in the guest uses virtual addresses which are translated to guest physical addresses (GPA) i.e. the address space that the guest sees as it's physical memory. The hypervisor constructs another page table – EPT for each guest, which translates a GPA to host physical address. Modern x86 processors provide hardware support for this second-level address translation, using Extended Page Table (EPT) by Intel [32] and NPT by AMD [3] to map guest-physical addresses to physical addresses. Initially, when a guest starts executing for the first time, the EPT of the guest is empty. When a guest tries to access its physical address range, guest page faults (or EPT violations) are generated, similar to page faults for a process. These faults are processed by the hypervisor by allocating pages for the faulting GPA.

For nested virtualization, three levels of translations are needed. A process virtual address is translated to GPA by guest-maintained traditional page tables. A GPA is translated to the L1 hypervisor's physical address (L1PA) using a virtual EPT for the guest maintained by L1, which we call EPT12. Finally, L1PA is translated to PA using the EPT for L1 maintained by the L0 hyperplexor, which we call EPT01. However, since x86 processors can translate only two levels of memory mappings in hardware, EPT01 and EPT12 are compressed by the hyperplexor at L0 to construct and maintain a shadow page table (or EPT02) for every L2 guest. Whenever an EPT01 or EPT12 entry is updated, the corresponding entry in EPT02 is updated. EPT02 is lazily constructed and updated as EPT01 and EPT12 updates take place via EPT violations mechanism. The MMU uses the process page table in the L2 guest and EPT02 to translate a process virtual address to its PA.

Figure 4 shows the guest co-mapping mechanism for HyperFresh. The hypervisor allocates an address range for the guest in its L1 physical address space (L1PA). Next, the hypervisor informs the hyperplexor (L0) about the range of pages to be used for guest memory. Specifically, the hypervisor sends a list of guest-to-L1 page mappings to the hyperplexor via a series of hypercalls. These hypercalls give hyperplexor the

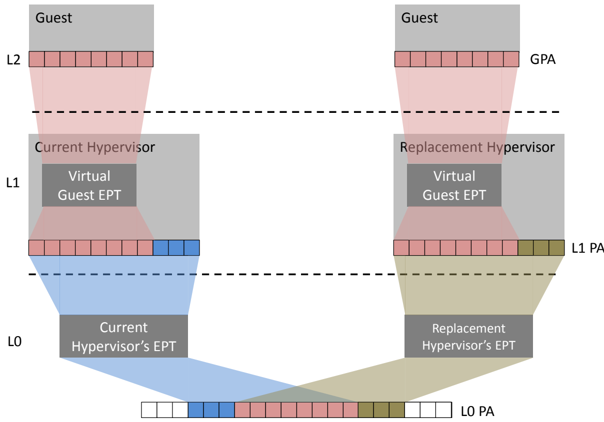

Figure 4: Guest memory co-mapping by hyperplexor

necessary pieces of information needed to co-map guest's memory between the two hypervisors. The hyperplexor then remembers these mappings (GPA to L1PA to PA) which are used to co-map guest memory when replacement hypervisor is prepared for the refresh operation.

The replacement hypervisor also first reserves guest pages in its own L1PA and invokes hypercalls to the hyperplexor. The hyperplexor uses the saved page mappings from the current hypervisor to populate the relevant EPT01 entries of the replacement hypervisor as well as to construct guest's shadow EPT (EPT02) entries.

The guest is live during the co-mapping operation and before the refresh operation is initiated. Hence, EPT mappings at all levels are populated lazily as the guest uses memory. Specifically, guest new page mappings may be added, or existing ones updated, while the guest runs on the current hypervisor. The hyperplexor tracks all guest EPT faults and updates the corresponding co-mapped entries in the replacement hypervisor's EPT01.

2.2 Refresh Operation

Once the new hypervisor has set up the necessary states, and co-mapped guest memory, the refresh operation can be performed by the hyperplexor. During a refresh operation, hyperplexor directs the current hypervisor to transfer control of all guest VCPU and I/O device states. First, the guest VCPUs running on the current hypervisor are paused. The VCPU and I/O states of the guest are then transferred over to the hyperplexor. The hyperplexor forwards these VCPU and I/O states to the replacement hypervisor, which now has all the necessary information needed to resume the VM. Once the VM resumes on the replacement hypervisor, the guest memory is unmapped from the current hypervisor, which is then shut down. To maintain isolation between the current and the replacement hypervisor, VCPU and I/O states are sent via the hyperplexor, rather than directly between the two hypervisors. During the refresh operation, the guests are paused only for a short duration (about 100ms) while the hyperplexor transfers guest VCPU and I/O device state from the current to the replacement hypervisor. The replacement hypervisor then becomes the current hypervisor and the entire process can be repeated over, as many times required. The refresh operation could be triggered periodically, manually, or based on a set of event-based triggers.

3 IMPLEMENTATION

We implemented a prototype of HyperFresh on the KVM/QEMU virtualization platform with Linux kernel version 3.14.2, kvm-kmod-3.14.2, and QEMU 1.2.0 in both the hyperplexor (L0) and hypervisor (L1). We configured both L0 and L1 Linux kernels with minimal required components, but did not otherwise attempt to shrink the hyperplexor footprint. This is an initial prototype to validate our ideas; hyperplexor footprint could be further reduced using one of many lightweight alternatives [5, 37]. Guests run unmodified Linux 4.4.2 and para-virtual I/O device drivers. Code modifications/additions are made to existing support for nested virtualization in KVM and live migration in QEMU. HyperFresh implementation required 800+ lines of code, the modifications being different for the hyperplexor (L0) and hypervisor (L1).

KVM/QEMU architecture separates out the responsibility of running various aspects of guests. QEMU runs as a user-space process in the hypervisor and is the point of control for the VM's user. KVM is a kernel module which takes care of managing memory allocation and translation, scheduling of VCPUs, handling traps and injecting faults. QEMU uses `/dev/kvm` device to communicate with the KVM kernel module. QEMU emulates I/O devices by processing the events received from KVM and is also responsible for management activities like pausing, checkpointing, or live migrating the guest.

During co-mapping and refresh operation, guest memory is mapped into both hypervisors whereas only the VCPU and I/O states of the guest are transferred from current to replacement hypervisor. VCPU and I/O states are transferred by modifying the existing live migration code in QEMU to skip guest memory copying, since guest memory is co-mapped before the refresh operation begins. The guest VCPUs on the current hypervisor are paused, and their current state is sent to the hyperplexor. The hyperplexor forwards these states to the replacement hypervisor via an intra-host TCP connection. The replacement hypervisor unpauses the VCPUs and the guest resumes normally.

4 EVALUATION

We evaluate HyperFresh on an Intel Xeon server with dual six-core 2.10 GHz CPUs, 128GB memory, and 1Gbps Ethernet network. Hypervisor is configured with 8GB memory with 4 VCPUs. The guest is configured with 2GB of memory along with one VCPU. Where applicable, each data value is averaged over five runs or more.

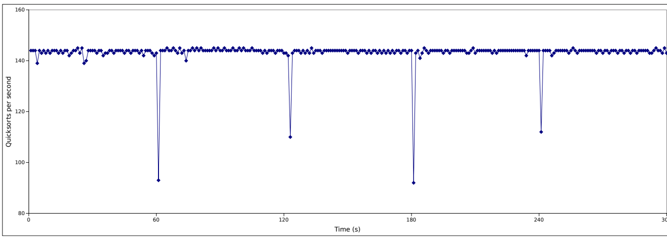

Figure 5: CPU performance across multiple refreshes.

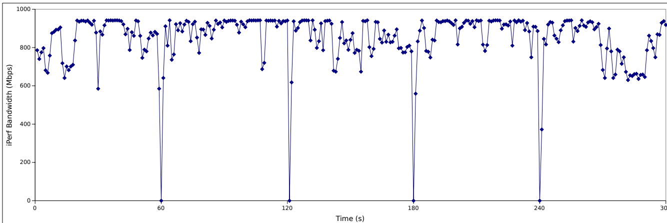

Figure 6: Network performance over multiple refreshes.

4.1 Performance Impact on Guest

First, we evaluate the performance of a guest while running on a system equipped with HyperFresh. Guest runs CPU and network intensive benchmarks. Refresh operation is invoked every minute by the hyperplexor to demonstrate that a high frequency refresh operation does not degrade guest performance significantly.

CPU-intensive workload: Quicksort is a CPU-intensive task and we measure the number of Quicksorts completed every second. The process begins by performing a `malloc()` operation for 1 GB of memory. Every Quicksort operation then, takes 200 KB (50 pages) from this pre-allocated memory, writes random integers into it and then performs Quicksort on it. Figure 5 shows the number of Quicksorts per second over time. At refresh boundaries, Quicksort performance reduces for a sub-second duration when transfer of VCPU and I/O control takes place, then returns back to normal.

Network-intensive Workload: iPerf [14] is a network bandwidth benchmarking tool. We run iPerf client inside the guest and iPerf server on a separate host and measure throughput every second. Figure 6 shows iPerf performance across multiple refresh operations. As with Quicksort, the refresh operation causes a sub-second dip in performance. Note that, in practice, refresh operation is expected to be much less frequent, and any resulting performance impact will be amortized over a longer time.

4.2 Co-mapping and Refresh Time

Figure 7 shows the time taken to map a guest's memory in the address space of the new hypervisor. Memory mapping is a two stage process – a hypervisor allocates L1 address space for the guest and then asks hyperplexor to map guest's pages into this allocated range. As the size of guest's memory increases, time to co-map guest memory also increases as more

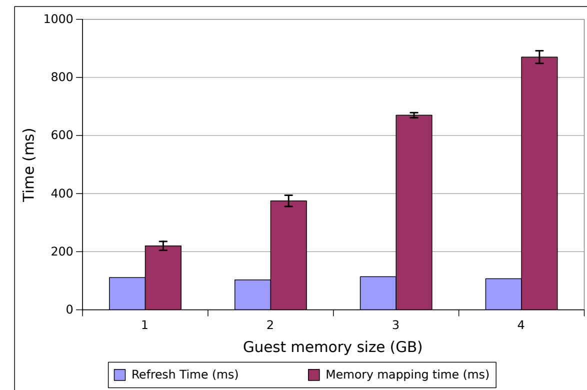

Figure 7: Time to co-map guest memory.

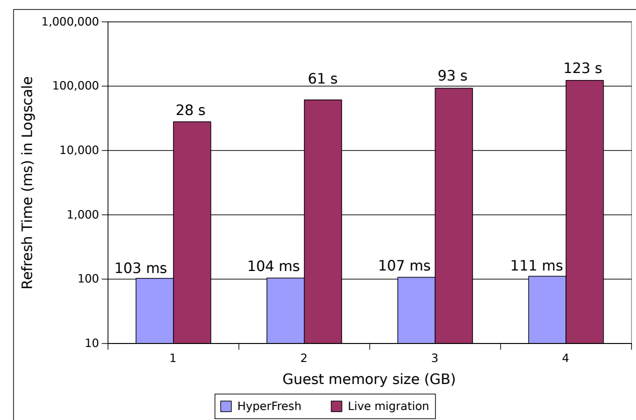

Figure 8: Comparison of refresh time for an idle guest with HyperFresh vs live migration. Y-axis is in log scale.

page-table entries need to be co-mapped, but remains below 900ms for a 4GB guest. Guest co-mapping can be performed in advance of the refresh operation or can be combined with the refresh operations. Figure 7 also shows that the refresh operation takes a constant time of around 100ms, since only the VCPU and I/O states are transferred from the current hypervisor to the replacement one. Thus, the combined co-mapping and refresh operations can be accomplished within a second even for a 4GB guest.

4.3 Comparison with Live Migration

For live migration, refresh time is the total migration time i.e. the time from starting the migration process, till the guest's control is transferred over to the receiving hypervisor. Downtime is the duration for which the guest's VCPUs are paused during a refresh process.

Idle Guest: Figure 8 shows hypervisor refresh time for live migration and HyperFresh for a guest with memory size varying from 1GB to 4GB. An idle guest means that no benchmark is running and the guest is left to itself without

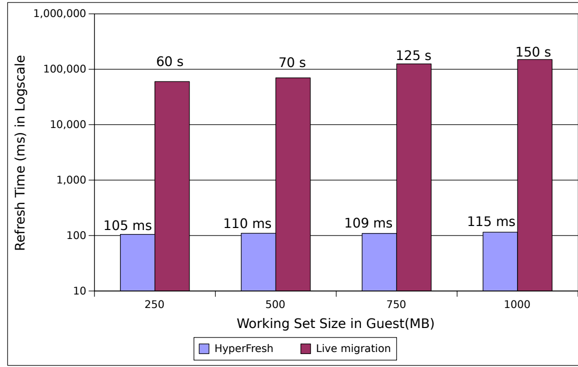

Figure 9: Comparison of refresh time for a busy guest with HyperFresh vs live migration. Y-axis is in log scale.

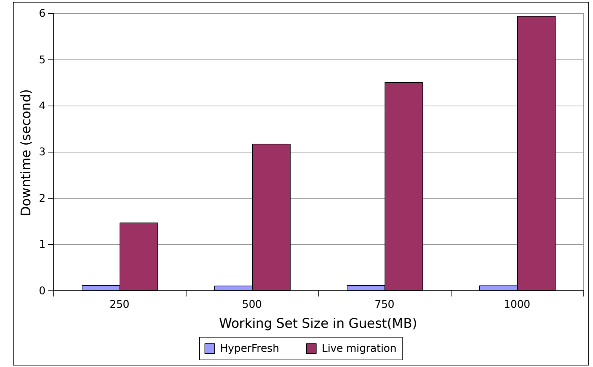

Figure 10: Comparison of downtime experienced by a guest running a write-intensive workload.

any external activity. To ensure that the measurement accounts for co-mapping the entire guest's memory between the two hypervisors, a process allocates large amounts of memory to fill up the guest and writes random values to the memory. HyperFresh takes constant time of around 100 ms to transfer control of the guest from the current to the replacement hypervisor, since the size of the state transferred for VCPU and I/O devices remains constant. In contrast, the refresh time with live migration is two orders of magnitude larger and worsens with increasing guest size because the entire guest memory needs to be explicitly copied from the current to the replacement hypervisor. Even if co-mapping time is combined together with the refresh operation, HyperFresh still completes the refresh operation well within 1 second, well below the time needed with live migration.

Busy Guest: We run a write-intensive application in the guest with a *Working Set Size (WSS)*, which is the amount of memory the process needs to execute a workload without triggering page faults. We vary the WSS for the workload as a process writes random bytes to each page of its allocated memory in an infinite loop. Figure 9 compares refresh time for HyperFresh vs live migration with varying WSS. Again, HyperFresh takes orders of magnitude less time than live migration. Live migration slows down as the amount of dirty pages being generated is much higher than the memory copying rate. Figure 10 compares downtime for HyperFresh with live migration with varying WSS. Live migration here does much worse due to the fact that the dirtying rate is very high. Hence, the guest VCPUs are paused for as long as 6 seconds to copy over the dirty pages and then the guest is resumed on the destination.

Memory and network overhead: Table 1 compares the memory overhead of HyperFresh during a refresh operation against two cases of live migration — between two physical hosts and between two co-located nested hypervisors. During host-to-host migration, the L0 hypervisor uses memory at both hosts, as does the guest. In addition, guest memory must be copied over the network. During co-located migration, although all memory usage is within a single host, one now needs two copies of the L1 hypervisor and the guest, besides

	L0 Hypervisor	L1 Hypervisor	Guest	Data Copied
Host-Host Migration	$2 \times L0_m$	N/A	$2 \times G_m$	$\sim G_m$ over network
Co-located Migration	$1 \times L0_m$	$2 \times L1_m$	$2 \times G_m$	$\sim G_m$ within host
HyperFresh	$1 \times L0_m$	$2 \times L1_m$	$1 \times G_m$	$\sim$0

Table 1: Memory usage during a refresh operation for different techniques. $L0_m$, $L1_m$ and G_m denote the bytes of memory used by the hyperplexor, hypervisor, and guest.

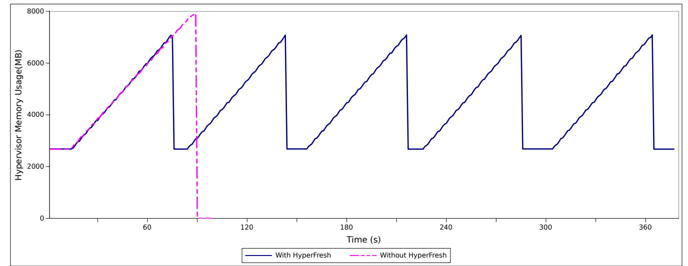

Figure 11: Memory usage timeline for hypervisor with a synthetic memory leak and using Hyperfresh.

locally copying guest memory over a TCP connection. During Hyperfresh, two copies of the L1 hypervisor are needed, but only one copy of the hyperplexor (L0) and the guest are needed, and there is negligible data copying overhead.

4.4 Guest Survival on a Buggy Hypervisor

Figure 11 demonstrates a memory leak scenario where HyperFresh is used to extend the operational lifetime of the hypervisor while protecting guests from hypervisor failure. Although, memory leaks may happen over a much longer duration of time, we create an accelerated leak for the sake of demonstration. The refresh operation is trigger-based where, the amount of memory consumed by the kernel above a certain threshold triggers a refresh operation. Without a refresh

technique, such a system will either crash or terminate guests to make room for more memory. With HyperFresh, whenever the memory leak gets worse, the figure shows that the hypervisor is refreshed without interrupting guests.

5 RELATED WORK

There are two complementary approaches to protect a guest from hypervisor failures: *reactive* and *proactive*. The key difference is that reactive techniques are triggered upon a failure event whereas proactive ones reset the hypervisor regularly during normal operations to avoid or delay a failure.

Reactive Protection: Reactive protection aims to detect a guest failure event, and then recover the guest to a previously checkpointed consistent state. Remus [11] provides high availability for guests by incrementally checkpointing the guest image to a fail-over machine over the network. Upon any failure of the source host (hardware/hypervisor/guest), the guest execution switches over to the fail-over machine with low downtime. Such an incremental checkpointing to a remote host design incurs non-negligible overhead during normal guest execution such as additional network traffic, guest performance impact due to write-event trapping, and the need for a spare fail-over machine to hold the checkpointed guest state. VMWareFT[25] uses record and replay to maintain a backup guest replica, reducing the network overhead of incremental checkpointing techniques like Remus. ReHype [21] aims to protect a guest from hypervisor failures, but not from hardware failure or guest corruption, using case-by-case local recovery of the guest. While, ReHype does not require nesting, failure detection and recovery is built into an unstable hypervisor that has possibly failed, thus reducing the robustness of the recovery mechanism.

Proactive Protection: Proactive protection falls under the umbrella of software rejuvenation [17] wherein the hypervisor controlling a guest is pro-actively and periodically replaced. The simplest proactive approach for hypervisor replacement is to live migrate [7, 16] the guest from one physical machine to another or from one nested hypervisor to another co-located nested hypervisor. However, as shown in our evaluations, live migration takes much longer to complete and adds significant memory and network overheads during migration. Furthermore, live migration relies on the source hypervisor to transfer several gigabytes of guest memory at a time when the source hypervisor itself might be buggy and, possibly unstable from constant use and memory leaks. In contrast, HyperFresh reduces the dependence on the source hypervisor because the hyperplexor co-maps a running guest's memory into a new co-located hypervisor's address space. The source hypervisor in only relied on for VCPU and I/O state transfer, which is way smaller than the entire guest memory. VMBeam [20] speeds up guest migration between two co-located nested hypervisors on Xen by remapping a guest's memory from the source to destination hypervisors. However its switch-over time is as high as 10–16 seconds, possibly because the remapping operation is not live and requires guest to be paused during map relocation. In contrast HyperFresh achieves around 100ms guest switch-over times using live guest co-mapping.

Live patching allows a hypervisor [1, 6] or an OS [8, 9] to update itself while running. However, the feature has to be built into the hypervisor or OS, increasing dependence on the very system being patched and making it complex and unreliable. In addition, live patching only resolves known bugs for which bug fixes exist. Without a subsequent reboot, live patching cannot clean up the preexisting effects of old bugs, such as memory leaks or corrupted state, even if the corresponding bug fix was included in the patch.

HyperFresh is complementary to reactive approaches, and can be used in conjunction with techniques like Remus and ReHype. For instance, HyperFresh can provide the first line of defense against an unreliable hypervisor by resetting the hypervisor state before bugs are triggered thus greatly reducing the failure probability. ReHype could provide a second line of defense, that is activated upon a failure event via local recovery. Remus could provide the final line of defense against unrecoverable hardware/software failure of the host or hypervisor. Thus, HyperFresh makes it less likely that expensive reactive approaches would need to be activated.

6 CONCLUSION

Hypervisor failures due to latent bugs can have a widespread impact on multi-tenant cloud platforms. Existing approaches to refresh the hypervisor state either cause significant disruption to guest execution or depend heavily upon the unstable hypervisor being refreshed. In this paper, we proposed *HyperFresh*, a fast technique to replace old unstable hypervisors with new ones without disrupting guest operations. HyperFresh uses nested virtualization to co-map the memory of a live guest between the current and replacement hypervisors and transfers guest execution state with a switch over latency of around 100ms. HyperFresh has the potential to enable administrators to quickly deploy hypervisor upgrades and patches, maintain hypervisor availability for customers by delaying bugs from being triggered, clean up insecure/compromised hypervisors, and reduce the lifetime of sensitive guest data lying around in hypervisor memory. Future work includes extending HyperFresh to include reactive failure recovery upon sudden failure of the hypervisor and the use of pass-through I/O for the L1 hypervisor to reduce nesting overheads.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Timothy Wood, and all reviewers for their helpful feedback. This work was funded in part by the National Science Foundation through awards 1527338 and 1320689.

REFERENCES

[1] Xen Live Patching, https://wiki.xenproject.org/wiki/LivePatch.
[2] Xen Security Advisories. http://xenbits.xen.org/xsa/.
[3] AMD. AMD Virtualization (AMD-V) http://www.amd.com/us/solutions/servers/virtualization.
[4] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman,

and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of Operating Systems Design and Implementation*. Vancouver, BC, Canada.
[5] Swapnil Bhartiya. Best Lightweight Linux Distros for 2017 https://www.linux.com/news/best-lightweight-linux-distros-2017.
[6] Franz Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. 2014. Swap and play: Live updating hypervisors and its application to xen. In *Proc. of the 6th edition of the ACM Workshop on Cloud Computing Security*. ACM, 33–44.
[7] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. 2005. Live Migration of Virtual Machines. In *Proc. of Network System Design and Implementation*.
[8] Jonathan Corbet. A rough patch for live patching https://lwn.net/Articles/634649/.
[9] Jonathan Corbet. Topics in live kernel patching https://lwn.net/Articles/706327/.
[10] Microsoft Corporation. Window Azure: Microsoft's Cloud Platform https://azure.microsoft.com/en-us/.
[11] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proc. of Networked Systems Design and Implementation*.
[12] Umesh Deshpande, Brandon Schlinker, Eitan Adler, and Kartik Gopalan. 2013. Gang Migration of Virtual Machines using Cluster-wide Deduplication. In *Proc. of the 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*.
[13] U. Deshpande, X. Wang, and K. Gopalan. 2010. Live gang migration of virtual machines. In *Proc. of High Performance Distributed Computing(HPDC)*.
[14] ESNet/LBNL. iPerf: The Network Bandwidth Measurement Tool, http://iperf.fr.
[15] Dan Goodin. Oct 29 2015. Xen patches 7-year-old bug that shattered hypervisor security. In *Ars Technica* *http://arstechnica.com/security/2015/10/xen-patches-7-year-old-bug-that-shattered-hypervisor-security/*.
[16] M. Hines, U. Deshpande, and K. Gopalan. 2009. Post-Copy Live Migration of Virtual Machines. *In SIGOPS Operating Systems Review* (July 2009).
[17] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. 1995. Software rejuvenation: analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*. 381–390.
[18] Google Inc. Google Compute Engine, https://cloud.google.com/compute/.
[19] Google Inc. Jan 2017. Google Infrastructure Security Design Overview, https://cloud.google.com/security/security-design/resources/google_infrastructure_whitepaper_fa.pdf.
[20] Kenichi Kourai and Hiroki Ooba. 2015. Zero-copy Migration for Lightweight Software Rejuvenation of Virtualized Systems. In *Proc. of the 6th Asia-Pacific(APSys) Workshop on Systems*.
[21] Michael Le and Yuval Tamir. 2011. ReHype: enabling VM survival across hypervisor failures. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 63–74.
[22] McAfee LLC. Root Out Rootkits: An Inside Look at McAfee Deep Defender https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/mcafee-deep-defender-deepsafe-rootkit-protection-paper.pdf.
[23] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. 2004. Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance. *SIGARCH Comput. Archit. News* 32, 5 (Oct. 2004), 211–223.
[24] Ravello Systems (Oracle). 2013. Nested Virtualization: Achieving Up to 2x better AWS performance! https://www.ravellosystems.com/blog/nested-virtualization-achieving-up-to-2x-better-aws-performance/.
[25] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. 2010. The Design of a Practical System for Fault-tolerant Virtual Machines. *SIGOPS Oper. Syst. Rev.* 44, 4 (Dec. 2010), 30–39.
[26] Amazon Web Services. *Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2*.
[27] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Operating Systems Review*, Vol. 41(6). 335–350.
[28] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Ben Rainero, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2016. Follow the Sun Through the Clouds: Application Migration for Geographically Shifting Workloads. In *Proc. of the Seventh ACM Symposium on Cloud Computing*. 141–154.
[29] Ravello Systems. https://www.ravellosystems.com/.
[30] Linux Bug Tracker. https://bugzilla.kernel.org/buglist.cgi?quicksearch=kvm.
[31] QEMU-KVM Bug tracker. Huge memory leak (qemu-kvm 0.12.3) https://sourceforge.net/p/kvm/bugs/539/.
[32] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.
[33] Orit Wasserman. 2013. Nested Virtualization: Shadow Turtles. In *KVM Forum, Edinburgh, Spain*.
[34] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. of EuroSys, Bern, Switzerland*.
[35] Xen Security Advisory . CVE-2015-7969: Leak of main per-domain VCPU pointer array, https://xenbits.xen.org/xsa/advisory-149.html .
[36] Xen Security Advisory . CVE-2015-8341: Libxl leak of PV kernel and initrd on error, https://xenbits.xen.org/xsa/advisory-160.html .
[37] XVisor. http://xhypervisor.org.
[38] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. 2008. *Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines*. Technical Report. IBM Research.
[39] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 203–216.