Virtual Machine Preserving Host Updates for Zero Day Patching in Public Cloud

Mark Russinovich
Microsoft Corporation
markruss@microsoft.com

Naga Govindaraju
Microsoft Corporation
nagag@microsoft.com

Melur Raghuraman
Microsoft Corporation
mraghu@microsoft.com

David Hepkin
Microsoft Corporation
davidhep@microsoft.com

Jamie Schwartz
Microsoft Corporation
jamschw@microsoft.com

Arun Kishan
Microsoft Corporation
arunki@microsoft.com

Abstract

Host software updates are critical to ensure the security, reliability and compliance of public clouds. Many updates require a virtualization component restart or operating system reboot. Virtual machines (VMs) running on the updated servers must either be restarted or live migrated off. Reboots can result in downtime for the VMs on the order of ten minutes, and has further impact on the workloads running in the VMs because cached state is lost. Live migration (LM) is a technology that can avoid the need to shutdown VMs. However, LM requires turn space in the form of already-patched hosts, consumes network, CPU and other resources that scale with the amount of and level of activity of VM, and has variable impact on VM performance and availability, making it too expensive and disruptive for zero-day security updates that must be applied across an entire fleet on the order of hours. We present a novel update technology, virtual machine preserving host updates (VM-PHU), that does not require turn space, consumes no network and little CPU, preserves VM state, and causes minimal VM blackout time that does not scale with VM resource usage. VM-PHU persists the memory and device state of all running guest VMs, reboots the host and virtualization components into updated code, restores the state of the VMs, and then resumes them. VM-PHU makes use of several techniques to minimize VM blackout time. One is to use kernel soft reboot (KSR) to directly transition to an updated host operating system, bypassing firmware reset of the server and attached devices. To minimize resource consumption and VM disruption, VM-PHU leaves VM memory in physical memory pages and other state in persisted pages across the soft reboot, and VM-PHU implements a mechanism called fast close to enable a reboot to proceed without waiting for the completion of in-flight VM I/Os to remote storage devices. We have implemented VM-PHU in Microsoft Azure hosting millions of servers and show results of several zero-day updates that demonstrate VM blackout times on the order of seconds. VM-PHU provides significant benefits to both customers and public cloud vendors by minimizing application downtime while enabling fast and resource efficient updates, including zero-day patches.

CCS Concepts: • **Computer systems organization → Architectures**; • **Software and its engineering → Software organization and properties**.

Keywords: cloud computing, virtualization and security, data centers

ACM Reference Format:
Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. 2021. Virtual Machine Preserving Host Updates for Zero Day Patching in Public Cloud. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456232

1 Introduction

Major cloud computing service providers leverage machine virtualization so that they can implement secure and agile elastic, multi-tenant service offerings. The host infrastructure supporting the VMs, which includes the root (host) operating system and hypervisor must be regularly updated. For example, stringent compliance policies require security patch updates to prevent exploits and malicious activity that can compromise the data being processed by VMs. Cloud service providers also regularly update the host operating system (OS) and hypervisor components to deliver feature enhancements and reliability updates. Many updates cannot be done seamlessly, but require restarting the affected components.

If a component is providing data-plane functionality to VMs and the hosted VM must be shut down for the update to

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.
EuroSys '21, April 26–29, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456232

proceed, the downtime impact on customer workloads can be significant. In addition to the time the VM is unavailable while the host software is shutting down and restarting, applications in the VM are unavailable while they shut down and restart. The applications are also either unavailable or operating in degraded mode while they rebuild in-memory state.

Among these updates, zero-day vulnerabilities in the interfaces exposed to VMs by the infrastructure are the most challenging, since patches must be deployed in short-order across fleets of machines that number in the millions of servers to prevent compromise of customer data.

Several techniques have been developed to ensure high availability for VMs during updates that impact VM data plane. Failover and clustering techniques [8] allow VMs to achieve high availability by replicating VM state to a secondary instance. Because most infrastructure as a service (IaaS) applications run within a single VM instance, they cannot take advantage of clustering. Hotpatching [2, 24, 27] is another mechanism that can be applied to avoid VM downtime, and it is supported on many operating systems, including Linux and Windows Server 2003. With hotpatching, the addresses of buggy functions can be updated on a running system to point at patched versions. Many kernel vulnerabilities can be addressed with hotpatching, however, hotpatching is not universally applicable. Depending on the implementation, it cannot be used when an update requires changes to global state, data structures, dependency changes, and inline assembly code [19]. Almost by definition, any significant update to a software component violates one of the preconditions of hotpatching.

Live migration (LM) [7, 18] is another technique commonly used to avoid VM downtime in the face of reboot-requiring host updates. LM transparently migrates VM state, including memory and VM management metadata, from one host to another one before an update applies, and it is supported by all common hypervisors. A key requirement for enabling LM is sufficient turn space of hosts in order to update the data center. The time taken to update the data center is therefore based on turn space where the number of steps is the inverse of the percent of turnspace, the time it takes to migrate VM state, and the time it takes to update vacated hosts. Host memory sizes has increased at a pace that exceeds network bandwidth growth, and the introduction of host-local non-volatile memory [1] promises to increase the amount of state that must to be migrated by an order of magnitude.

In this paper, we focus on the specific scenario of zero-day updates in a public cloud environment that require a restart of software components implementing the VM dataplane, since the goal is to update the fleet quickly while causing minimal VM downtime. VM-PHU, the technique we introduce, is designed to perform updates on each node by preserving the state of the VMs in-place, and restart host components with patched versions within a few seconds. A straightforward approach to in-place updates is to persist state for VMs in host-local non-volatile storage before the restart to patched software components. However, a host may be running VMs with hundreds of gigabytes of virtual memory backed by RAM, so persisting and restoring to and from non-volatile storage can be time consuming, especially if the storage is spinning media. Moreover, if updating a component requires a full OS reboot and associated hardware reset, the VM downtime can be extended by tens of seconds while the OS is shut down, devices are discovered by firmware, RAM is zeroed, and the OS restarted.

VM-PHU uses four main techniques to perform fast updates:

- **Fast state preservation:** We avoid the overhead of storing state in non-volatile storage by marking the physical memory pages corresponding to the persisted state as in-use. The persisted state corresponds to the physical memory pages allocated to VMs, and physical pages corresponding to host drivers that are enlightened to preserve their state. Marking the persisted pages as in-use enables the memory manager to leave the pages intact and can be retrieved by the applications later on.
- **Fast reboot:** We eliminate the overhead of hardware reboot by skipping the firmware post and transitioning into the patched OS directly from the current one.
- **Fast close for storage devices:** We avoid the delay of straggler VMs with in-flight I/O to storage devices by allowing the hypervisor to perform fast close of virtual devices and restore their state post-update to resume the in-flight I/Os. Our fast close mechanism allows for deterministic, low blackout times, eliminating the inconsistencies in remote storage performance.
- **Hardware acceleration:** We support VM-PHU on VMs that enable hardware acceleration using discrete device assignment (DDA) and single root input output virtualization (SR-IOV) devices. VM-PHU handles VMs with SR-IOV devices using a transparent failover mechanism. For each SR-IOV device exposed to the VM, we create a shadow device in the host. We failover to the shadow device prior to VM-PHU. We fallback to the accelerated SR-IOV device post-update. For DDA devices, we ensure DDA device and input output memory management unit (IOMMU) is active during fast reboot. Furthermore, we preserve the page protections and domain mappings associated with the DDA device across fast reboot.

With the above techniques, our system can perform component updates that do not require a host reboot (rebootless) or updates that require a host reboot (rebootful) in a few seconds, and the VM experiences only a short network blip. Some latency sensitive applications may not tolerate the

network blips and therefore require a pre-notification of an impending update so that they can failover the application to another VM. Our system provides such a notification to VMs on a host-local endpoint a few minutes before a scheduled update, allowing applications to take desired actions.

We present our learnings over the last decade in deploying our techniques [37] to the Windows OS on Microsoft Azure hosting several million VM. We discuss several challenges that need to be addressed in designing a robust system that preserves memory state across reboots. One such challenge is host memory fragmentation that significantly increases the blackout times of VMs. High VM blackout can cause applications within VMs to timeout. Another challenge is memory range dependencies for components that may not be satisfied post-reboot due to memory state persistence in those regions. In these cases, the hosts may become unresponsive or crash. Furthermore, update failures can occur due to bugs in software that accumulate over time such as reference count mismatches in host drivers that prevent fast reboot, memory leaks that prevent host update processes, etc. Even a small failure percentage can cause significant customer pain. A primary contribution of our research are solutions that are designed to address such real world issues observed in production environments.

Our results in production environments confirm that fleet-wide updates can be performed within seconds per VM. Our update notification mechanism allows legacy incompatible applications to undergo updates 10× faster than updates that use a standard host reboot. Our techniques are generic to any OS with enhanced drivers, meet the goals of zero-day updates, do not scale with the amount of RAM or non-volatile memory used by VMs, do not require server firmware or network infrastructure changes, and scale to a high density of VMs and hosts in a public cloud. Moreover, our techniques to patch hosts quickly enables fast turn space generation and are therefore complimentary to LM.

The rest of the paper is organized as follows: In section 2, we briefly survey related work in the area of updating hosts in a data center. We present an overview of VM-PHU approach in section 3. We describe our KSR and VM state preservation algorithms in sections 4 and 5. We discuss the application of VM-PHU for host resilience in Section 6. We present the results of VM-PHU on updates with and without reboots in Section 7, and analyze the results of our algorithms on a public cloud and compare our approach with the state-of-the-art techniques. Finally, we conclude with a discussion of limitations and opportunities for future work.

2 Related Work

The problem of achieving high availability has been widely studied in many areas for decades. Several techniques are developed to improve the VM availability to address planned and unplanned failures. We broadly classify the techniques to improve availability of VMs into four categories.

Dynamic Software Updates: These techniques achieve high availability by live patching the process that needs to be updated. At a high level, a valid program transformation is required to ensure that the new code introduced to update the program does not run on an old state representation from the prior code. In order to perform dynamic updates, certain locations in the code are chosen for updates. Various systems [2, 6, 15, 16, 30, 33] are designed to handle these update locations to either use manually programmer specified locations, infer good locations or constrain to certain specific class of locations within a program avoiding program state transformations. As the programs are running while the updates are applied, the programs retain their current state and are not impacted by update application. For a detailed overview of dynamic software updating, refer to [17]. Advanced, complex process-level live updates that allow safe state transfer are also designed [4, 13, 14]. A primary issue with dynamic software updates is that the code needs to be written or compiled to support dynamic software updates. Only limited number of programs are written to handle such updates and there is a significant engineering cost to modify general purpose commodity operating systems where many legacy components are not designed for dynamic updates.

Hotpatching techniques to update the kernel leverage a specific dynamic software updating technique where only function level transformations are supported [2, 24, 27, 32]. It is widely supported to enable kernel patching as the data structures for a kernel are stable. However, many software updates such as program updates involving global state changes, addition of new functions cannot be applied using these techniques.

Failover Algorithms: Failover algorithms use redundant systems to achieve high availability to the processes. In these algorithms, secondary systems are maintained in sync with the primary using distributed consensus algorithms such as Paxos [5, 28], or the secondary systems periodically checkpoint the state with the primary at valid program locations. In the instance of a failure or an impending update to the primary, the secondary machine can take over as the primary. These techniques are widely employed in cloud software and infrastructure management solutions [8, 38–40] to perform automatic failure handling of VMs. Failover techniques are often complementary to techniques used to improve the availability of VMs and address the needs of customers maintaining additional redundant systems.

Live Migration: Live migration is an elegant technique to move a running VM from one physical host to another with minimum disruption to the VM. The migration process involves transfer of the VM's memory, processor and virtual device state [7]. Live migration is supported in most hypervisors including VMWare [35], Microsoft [20], Xen [3] and KVM [25]. Many of these hypervisors support pre-copy Live

Migration. In pre-copy live migration, we iteratively copy the VM's memory pages while maintaining a dirty set of memory pages for the VM on the source host. Based on the dirty page rate, available network bandwidth, target blackout time and amount of time for migrating the VM, the algorithm determines a maximum number of iterations to stop the VM on the source host. After the VM is stopped, the pre-copy LM algorithm synchronizes the dirty state with the target and resumes the VM on the target host. In order to achieve deterministic blackout times, a high bandwidth dedicated network is required. Most pre-copy algorithms may terminate early on memory or I/O intensive workloads and can impact VM availability [18].

To address this limitation, post-copy LM algorithms are designed to defer the memory transfer phase until the VM's CPU state is migrated and resumed on the target host [18]. Memory pages are faulted on by the VM at the target, and faults are minimized by pre-paging. Post-copy approaches have the benefit of sending the memory pages only once from the source to the target. However, both pre-copy and post-copy algorithms require the source host to be available for the entire duration of migration increasing the turn-space requirement for performing updates. Moreover, source host cannot be updated till the migration is complete. During pre-copy LM, the only fatal errors are ones that happen on the host. During post-copy LM, a failure of the source or destination are fatal, decreasing the reliability of the VM during the migration. Furthermore, the additional load imposed due to the transfer of VM state can degrade the performance of the applications for a significant duration.

Soft reboot: As updates to the hosts typically require reboots, and initializing devices during the reboot is the most time consuming portion on servers, techniques such as kexec are developed to optimize the Linux boot process bypassing the firmware post [34]. This enables the Linux systems to boot within a few seconds but the implementation does not support preserving memory pages across the reboot. Kashyap et al. [23] independently designed a technique kUP similar to ours [37] to preserve memory across a soft reboot and applied it to update memcache applications. While the technique extends kexec to preserve pages across a reboot, persisting state across reboots introduces reliability and performance issues. These issues include memory fragmentation that increases persist times, memory range dependencies that cause crashes when those ranges are unavailable due to persisted state, etc. kUP does not address these challenges that arise in real-world systems. Furthermore, kUP does not address persisting VM state and the challenges associated with state preservation of VMs such as handling SR-IOV and DDA devices attached to VMs. We discuss the reliability and performance challenges introduced by persisting state across reboots and our solutions in section 4.3. Orthus [42] is a real-world VMM live upgrade system that instantiates a new KVM instance while the current hypervisor is active and registers the VM state with the new hypervisor and transparently switches to the new hypervisor. HyperFresh [11] performs hypervisor updates by leveraging nested virtualization and using a thin hyperplexor layer. While hypervisor updates are critical in cloud environments, both Orthus and HyperFresh do not perform root partition updates and it is non-trivial to extend the techniques to load a new root partition while the current partition is active due to device conflicts across the partitions. VM-PHU handles both hypervisor and root partition updates by rebooting into the patched system.

Handling regressions: Updates can introduce regressions and impact availability of VMs in a large-scale cloud environment. As rolling updates are performed in batches for safety, latent issues such as crashes can take time and update methodologies need to incorporate resiliency primitives to ensure VM availability is not impacted. Otherworld [10] is a system designed to handle crashes by pre-loading a crash kernel and registering crash routines for applications. The crash routines are invoked to save and restore their state when a crash occurs. VM-PHU is similar to Otherworld in terms of handling crashes. In addition to crashes, VM-PHU leverages health monitoring of VMs in cloud to handle other issues in cloud systems such as system hangs and memory leaks.

3 Overview

In this section, we first present an overview of the challenges in performing host updates in a public cloud environment, then provide an overview of our system to meet the requirements of fast zero day patching.

Our technique VM-PHU combines the benefits of in-place updates with the state preservation benefits of LM to avoid VM reboots. Our technique follows three main phases – in phase one, we perform fast state preservation and fast close of virtual devices to quickly teardown the virtualization stack. We then follow it with a fast reboot into the patched OS. In the final phase, we restore the virtualization stack. These phases need to be designed such that they are done within few seconds in order to preserve the TCP/IP connection state of applications within the guest VMs.

Public cloud servers often provision dozens of VMs and have several hundreds of gigabytes of memory. Several challenges exist in preserving the state of the VMs in a few seconds across an OS reboot. First, the memory in the servers can get fragmented over time as VMs come and go, and preserving large amount of fragmented state in a few seconds is hard. Defragmentation of memory can help but physical pages can be associated with device DMA regions, so it may not be possible to defragment running VMs without rebooting and disrupting the VMs. Preservation of memory poses other challenges during operating system restart where drivers or devices make assumptions on availability of pages in

certain memory ranges that were previously available during the prior boot but may not be available during a subsequent boot. Another challenge in large scale production systems is the presence of straggler VMs that are unable to save their state as storage I/Os do not complete due to partial service interruptions. One of the primary contributions of this paper is it analyzes and addresses several real world problems that arise in a large scale production environment.

In the following subsections, we provide an overview of the techniques in these phases.

Phase One: In the first phase, we preserve the state of the VMs and teardown the virtualization stack. Preserving the hundreds of gigabytes of VM memory state to the disk can take several minutes even on fast storage medium such as SSDs. A potential approach to preserve the physical memory pages intact is to mark the pages in a persisted pages database that the OS refers to during every boot to preserve, and modify the BIOS/EFI to bypass invalidating the memory pages. The two disadvantages with the approach are the lack of support from existing BIOS on hosts in the data centers, and the additional blackout time imposed on VMs by transitioning to BIOS. Our system bypasses the BIOS and transitions directly into the patched OS in Phase two. Persisted pages are preserved across the reboot by marking the pages in the patched kernel as in-use.

Rebooting the host OS in phase two requires a teardown of virtualization stack. However, virtualization stack teardown requires the storage devices to complete the in-flight I/O to the storage backend. Due to inconsistent remote storage device performance, the teardown can cause straggler VMs and increase VM blackout time. We achieve deterministic, reliable and fast update times by storing the in-flight I/O queue in memory during the tear down of the VM storage devices, and then replay the uncommitted in-flight I/O queue during VM restoration. Our approach eliminates the I/O performance variance and can also be used in LM to migrate the VMs to a different host.

Phase Two: After the VM state is persisted, we transition into the patched kernel. We developed a technique Kernel Soft Reboot to quickly transition from the current OS to a patched OS bypassing the firmware device reset. Operating system boot can take several minutes and is I/O intensive. We design our host OS to have minimal, core services consuming only few hundred megabytes of memory. Our minimal host OS reduces the number of zero day updates and enables fast reboot within few seconds.

Several challenges exist in quickly rebooting an operating system with persisted state. First, operating system boot is very read-intensive and depends upon the storage performance of the boot device. We address this issue by preloading the entire OS image, preserving the in-memory OS image state across reboot via phase one and then committing the modified memory pages to the disk after the update is completed. Second, drivers may assume memory range dependencies that may not be available post-reboot as the pages in those ranges are preserved across the reboot. We address these issues by partitioning the host memory range from the VM memory ranges. Finally, VM memory fragmentation can cause performance issues during the boot. We address the fragmentation issues by preallocating large memory pages in the order of hundreds of gigabytes for VM memory.

Phase Three: After rebooting into the patched kernel in phase two, we restore the VM state and complete the update. We restore the state of all VMs in parallel. Each VM in turn restores the state of its devices before unpausing the VM. As the device state is saved in memory in phase one and persisted across the KSR in phase two, the device restore operations are fast and the overall update period results in a short network blip of few seconds to the VM.

4 Kernel Soft Reboot

Kernel Soft Reboot is similar to the normal reboot. In the normal reboot, the boot manager loads the application parameters of the new OS from the boot configuration database (BCD) store. The application parameters include OS device, test signing options, etc. The boot manager then passes the control to boot loader to load the new kernel image. The boot loader then passes the boot related information in the loader block to the kernel. In the KSR boot, we developed a control application to perform the tasks of the boot manager and we refactored the boot loader to introduce a driver to load the new kernel image to boot into. Similar to boot loader in the normal reboot, the driver transitions to the new kernel image. The transition from the current OS directly into the new OS bypasses the BIOS and firmware initialization.

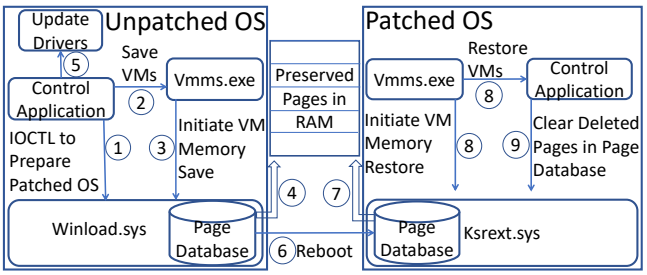

Figure 1. Simplified workflow for preserving pages across reboots.

4.1 Preserving Memory across Reboots

We extended the soft reboot capability to preserve physical pages across the reboots. To allow page preservation, we implemented a persisted pages database. The following steps are implemented to perform the page preservation across reboots.

1. The control application sends an input/output control (IOCTL) call to the KSR driver – winload.sys to initialize the new kernel image and its memory layout.
2. The control application notifies the hypervisor virtual machine management service (VMMS) to preserve the VM state. VMMS then sends notifications to all the VMs to preserve their state.
3. We preserve the run of memory pages for each VM with a unique GUID identifier.
4. Winload.sys marks the pages in the persistent page database for preservation and the hypervisor proceeds to save the rest of the VM state such as network state, disk state, etc.
5. Driver updates are performed.
6. The control application initiates the reboot into the new kernel image.
7. In the patched OS, the kernel detects the presence of the persistent page database and instructs a KSR extension driver ksrext.sys to retrieve the page database.
8. The control application instructs VMMS to restore the VMs. The hypervisor memory management driver checks whether the OS in booted from a soft reboot and requests VM's memory by passing the unique GUID identifier for the VM.
9. On successful restoration of VM's memory, the memory run is deleted from the persisted page database.

Fig. 1 illustrates these steps. As the virtualization stack is down after step 4, we can perform rebootless driver updates while skipping steps 6 and 7.

The performance of the algorithm is dependent on the number of pages that need to be preserved across reboots. Using large page allocations while minimizing fragmentation can significantly improve the performance of memory preservation.

4.2 Optimizing Memory Preservation

Each VM's physical memory representation can consist of several contiguous segments of physical pages on the host server. We refer to each contiguous segment of physical memory pages as a memory run. The memory runs of the VMs on the host server are preserved across the KSR by storing the physical page numbers in a persistent page database.

The persistent page database uses a linked list data structure sorted by the physical page number. A simple approach takes each memory page and inserts into the appropriate location within the persistent database. This approach works reasonably well when the number of runs is small but can have significant performance issues when memory is fragmented. A more sophisticated data structure such as AVL trees can help but each insertion and deletion will require locking the entire data structure.

To address the issue of memory fragmentation, we take advantage of bulk insertions and deletions of the memory runs for each VM. We first sort the memory runs for each VM based on its physical page numbers. The bulk insertion for the memory run corresponds to an insertion sort of the VM's sorted memory run with the persistent page database. The data structure is locked for every VM's memory run insertion. When the VMs are being restored, a similar issue occurs in deleting the memory runs that are retrieved by the hypervisor. We use a global VM memory map with memory page size granularity. We lazy mark the locations corresponding to the run of the restored VM as deleted. Once the memory is restored for all VMs, a single linear scan is initiated on the persistent page database to see if the pages are deleted and the persistent page database is correspondingly updated.

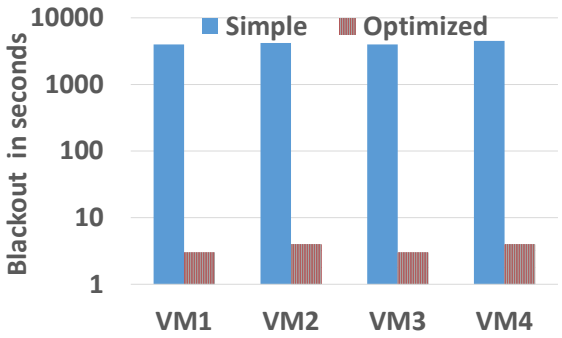

Figure 2. Performance comparison of simple and optimized version of preserving and claiming the memory runs on a highly fragmented system running four VMs. The blackout time for the four VMs is highlighted in seconds on a logarithmic scale. Our optimizations handle memory fragmentation issues and achieve low blackout time below four seconds for all VMs.

Fig. 2 highlights the performance comparison between the simple approach and the optimized persistence algorithm on a server with four VMs. The server has 32 GB of memory and each VM is configured to have 5 GB of RAM. To validate the performance on a fragmented system, we introduced fragmentation of the memory by allocating pages at random locations prior to starting the VMs. A total of 4.5 GB of fragmented pages are introduced into the system. The experiment is performed by first saving the VM's and then immediately restoring the VMs without undergoing a reboot. The blackout time is measured from the instrumentation in the hypervisor. Based on the results, our techniques are able to achieve a low blackout time within four seconds, even on highly fragmented systems.

4.3 KSR Reliability and Performance

We discuss the performance and reliability issues in KSR due to persisting pages across a reboot. We first present the performance optimizations to improve the boot, then discuss the

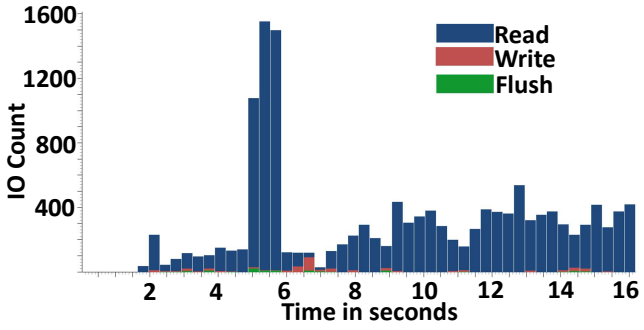

Figure 3. The IO count of the read, write, flush IO types during the KSR of a host running Windows Server 2016 using hard disks. This trace highlights that a typical server reads approximately 600 MB of data from the hard disk and is mostly read limited. Serving the I/Os from RAM can improve the boot performance by 3x.

reliability optimizations such as host device configuration preservation and host, guest memory partitioning.

Fast I/O Performance: KSR performance is largely dependent on the performance of the I/O subsystem hosting the OS partition. During a typical reboot of the OS, we read and write few hundred megabytes of data to the OS partition as illustrated in Fig. 3. As datacenters have hardware stock keeping units (SKUs) with hard disks hosting the OS, boot performance can be slow even with KSR. We alleviate the I/O latency overhead by serving all the reads and writes to the OS partition from RAM.

We use a filesystem filter driver to intercept the reads and writes to the OS partition. In order to simplify the filter driver, we use a preallocated virtual hard disk (VHD) to host the OS partition and intercept all accesses to the VHD. Before performing Phase One, we prefetch the entire VHD into physical pages and persist the pages to the KSR APIs. We configure our filter driver as a boot start driver. During a KSR, our driver first checks whether the VHD state is preserved and if so, reclaims the memory from KSR and continues to serve the reads and writes from the persisted memory. After the update is performed, we release the persisted memory to the host by synchronizing the blocks with the VHD.

It is possible that applications running on the server modify the files in the OS while we load the VHD into KSR memory. In order to guarantee data consistency, we synchronize the load of the VHD into KSR memory by decomposing the VHD into blocks and loading a block at a time. While the block is being loaded into KSR memory, accesses to each block is locked. As blocks are typically few megabytes in size, it introduces a few milliseconds of additional I/O latency to the OS partition for data accesses that conflict during the load or unload operation. The load or unload time is equal

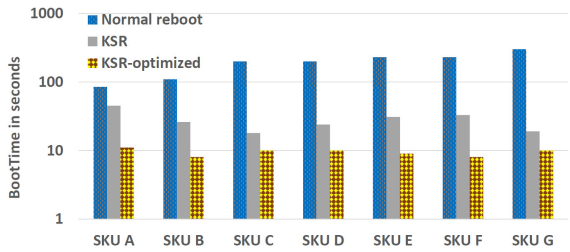

Figure 4. Performance comparison of normal boot, KSR and KSR-optimized on seven different SKUs deployed in Microsoft Azure in log-scale. KSR-optimized leverages the optimizations to serve all I/O from memory, and removed unused legacy devices and services from the OS. Overall, KSR-optimized boot is approximately 10–30× faster than normal boot.

to the time for sequentially reading the OS VHD from the hard disk. In order to further reduce the amount of time and memory required for loading or unloading the OS, we significantly reduce the size of the core host OS with minimal set of services. In addition to removal of unnecessary services in the host, we also disable unused devices such as USB devices and legacy PS2 devices that may be present in the motherboard and enabled in BIOS.

SKU Name	Memory	Boot Device
SKU A	32 GB	HDD
SKU B	32 GB	HDD
SKU C	128 GB	SSD
SKU D	128 GB	SSD
SKU E	192 GB	HDD
SKU F	192 GB	HDD
SKU G	512 GB	SSD

Table 1. Table highlighting the memory and boot device configuration of various SKUs in Microsoft Azure. SKUs can have diverse configurations with large memory, slow media type for boot devices. Real world production systems need to handle update of these diverse configurations within seconds irrespective of memory size or boot media.

Table 1 highlights the memory and boot device configurations for the hardware SKUs running in a production public cloud. Fig. 4 highlights the performance comparison on a log-scale between normal boot and KSR boot on these seven hardware SKUs, four with OS partition on HDDs and three with OS partition on SSDs. The figure illustrates that the optimized KSR boot is approximately 10–30× faster. It is important to note that prefetch has no direct impact on VMs or applications running on the host server as the host memory provisioning already accounts for the few gigabytes of memory required for prefetching the OS VHD.

Persisting Device Configuration: When an operating system boots from BIOS, the operating system's PCI bus driver enumerates all PCI devices and configures their device resources such as memory and I/O when starting their respective device drivers. In some cases, this operation requires rebalancing or reconfiguring resources across PCI bridges and root buses to ensure the requirements of all PCI devices are satisfied. This is required because a BIOS often only configures the devices necessary to boot the operating system leaving many PCI devices in a powered off state. This configuration step is not required during a KSR, since the previous OS would have already configured and started all PCI devices. To take advantage of the configuration from the previous operating system, during the shutdown phase of a KSR, PCI devices are not powered off and put into a D3 state as they would be during a normal shutdown or reboot. Additionally, device configuration (memory and I/O) as well as other PCI configuration state (e.g., decodes on/off) are passed through KSR preserved memory to the new operating system. The PCI bus driver in the target operating system leverages the preserved memory to initialize its internal state, skipping the potentially intensive device configuration and arbitration task. The preservation of resources allows the devices to function in a mode that is similar to rebootless updates to the drivers of devices within the same OS.

Table 2 highlights the results of soft reboot with and without persisting PCI configuration on several servers from different vendors that are deployed in Microsoft Azure. The removal of state transitions across the KSR, from a powered on D0 state in the source OS to a D3 state during shutdown and back to a D0 state in the target OS also improved reliability. On most SKUs, we observed host crashes. Our techniques to preserve the device boot configuration enabled such systems to function without issues, with no changes to the drivers or underlying device firmware.

SKU Name	without device config preservation	with device config preservation
SKU A	No issues	No issues
SKU B	12s boot regression	Boot optimized by 12 seconds
SKU C	Host crash	No issues
SKU D	No issues	No issues
SKU E	Host crash	No issues
SKU F	Host crash	No issues
SKU G	Host crash	No issues

Table 2. Table highlighting the soft reboot behavior on various SKUs in Microsoft Azure with and without preserving the device boot configuration across reboots. Most SKUs experienced host crashes on soft reboot without device configuration preservation due to address conflicts. One particular SKU (SKU B) did not experience a crash but exhibited a 12 second boot regression due to ACPI method being unable to retrieve the device state. Issues are resolved on all SKUs by preserving the device configuration.

Memory Partitioning: KSR with memory persistence can cause slow boot performance or crashes due to the following reasons:

- *Memory range dependencies:* Certain legacy drivers or early kernel initialization code can assume presence of memory within a low memory range. The memory range availability may be valid when the system boots with entire memory range being zeroed and available. However, the memory range may not be present if that memory range is freed and occupied by memory allocated to VMs and persisted across the reboot. In these cases, the system crashes.
- *Contiguous memory:* Similar to memory ranges, certain drivers can require contiguous memory, and the scan for a contiguous memory range on a system with large amount of memory can take a long time – this in turn has the side effect of large reboot time and subsequently, higher blackout time for VMs. Moreover, if no contiguous memory is available, the system may crash.
- *Memory fragmentation:* Memory fragmentation can occur when large number of physical page allocations and deallocations are triggered by the natural churn of VMs on the hosts. Memory fragmentation can cause performance issues to the applications within the VM and also increase the blackout time.

Addressing the above issues is critical to ensure that the system boot performance is fast as well as improving the reliability of reboots. In order to address the above issues, we partition the system memory to isolate the memory required for VMs and host. We allocate higher memory ranges for the VMs and the lower memory ranges are allocated to the host. The allotment of lower memory ranges allows for legacy drivers and early boot initialization code relying on 32-bit address ranges to fulfill the allocation requests during the reboot. While partitioning the host memory to lower ranges improves the probability of success in allocations, allocations can still fail. To eliminate failures, we enlighten our operating system components to remove any contiguity or range dependencies. For third party drivers, we instrument the allocations during boot offline. Using the offline allocation traces, we preallocate the regions corresponding to specific driver pool tags to ensure that the memory manager can satisfy their range or contiguous allocations.

Fig 5 illustrates the impact of memory fragmentation on the memory persistence and reclamation times in our production environment with millions of servers. The figure illustrates that server memory can get severely fragmented

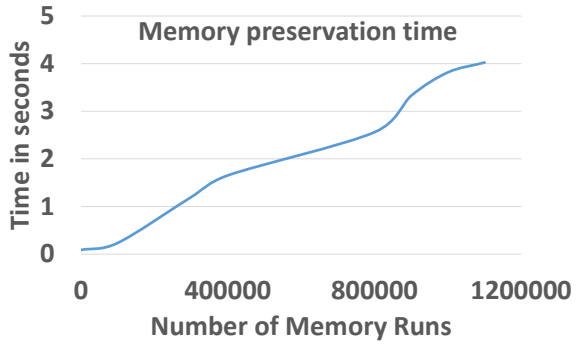

Figure 5. This figure highlights the memory fragmentation in our Azure production environment with millions of servers and VMs over the course of last three years. As the servers in the public cloud do not reboot, the host memory gets fragmented over time. We observe that the most fragmented nodes have over a million runs of memory pages and is plotted on the x-axis of the figure. Memory fragmentation increases the blackout time and the figure illustrates the linear time performance of our memory persistence and reclamation algorithms with the number of memory runs.

over time and there are over a million runs in the most fragmented nodes. Prior to our techniques such as memory partitioning applied, we typically observed approximately 5% of our servers failed to boot. Without our techniques applied, we have not observed server boot failures due to memory fragmentation highlighting that our techniques are effective in handling fragmentation.Furthermore, we observe that the memory fragmentation has little overhead on the VM blackout time of up to four seconds even on the most fragmented nodes due to the optimizations presented in Section 4.2.

5 Virtual Machine State Preservation

In a public cloud, each VM can be configured to have tens of virtual processors, hundreds of gigabytes of memory, tens of virtual hard disks attached to virtual disk controllers, and virtualized synthetic network devices with network flow state and isolation rules. Preserving a VM requires state preservation of the associated state of the virtual devices along with their configuration. In this section, we provide an overview of preserving VM state and discuss the optimizations incorporated in each of the phases.

Our VM state preservation workflow proceeds in two steps – *VM-PHU save* for saving the state of all the VMs, and *VM-PHU restore* for restoring the state of the saved VMs. These two steps correspond to *phase one* and *phase three* discussed in section 3. The blackout time of the VMs is therefore dependent on performance of saving and restoring the state of all VMs.

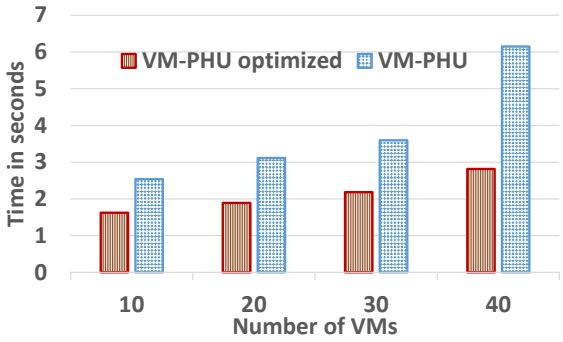

Figure 6. Blackout time comparison between VM-PHU and optimized VM-PHU for rebootless updates. In optimized VM-PHU, all virtual device state is persisted to memory as compared to SSDs.

5.1 VM-PHU Save

We present the stages involved in performing VM-PHU save and our optimizations to improve the performance while avoiding straggler VMs that may be experiencing transient faults. At a high level, the VM-PHU save operation initiates a state preservation operation on all VMs in parallel. The state preservation operation involves two stages as described below.

Save the VM memory and virtual devices: In this stage, we first pause the virtual processors to initiate the preparation for a save operation of the virtual devices attached to the VM. Once the processors are paused and the processor state is saved, the VM memory state is saved. After the memory is saved, we save the remaining virtual devices attached to the virtual motherboard for the VM. These virtual devices include video, IDE and SCSI devices, network switchport, etc.

In our approach, we persist the VM memory run via the KSR APIs discussed in Section 4. Moreover, as discussed in Section 4.3, the VM memory state is isolated into a separate partition and pre-allocated into a single large run eliminating memory fragmentation. VM-PHU state preservation operations are therefore fast and complete within few milliseconds.

We further improve the performance of VM-PHU by persisting all virtual device state via KSR APIs. Even though the rest of the persisted state of all virtual devices is only a few megabytes per VM and is stored on SSDs, the performance overhead of writes to SSD is still significant. We refer to our optimized approach to store all VM state in physical memory pages as VM-PHU optimized in Fig. 6. We observe that persisting all the device state to memory as opposed to SSDs improves the blackout time performance by 2×.

Power off the virtual devices: The next stage after the virtual devices are saved is to power off the virtual devices. While most virtual devices power off quickly, storage virtual devices can require significant amount of time to power

off. The high power off time is due to the storage devices draining the in-flight I/Os to the device. Moreover, remote storage virtualization devices can require considerable time to power off. The large overhead for remote storage devices is due to the load on the backend from concurrent updates across hundreds of hosts in the data center, and the high network latency to the storage backends. In order to close the devices quickly, we serialize the in-flight I/Os via memory persistence APIs and close the devices. After the pending I/O serialization, the storage devices can be powered off immediately. It is important to note that we do not return the I/O status of any write-through in-flight I/O to the guest storage stack during the storage device close operation. Similarly, we return the status of any in-flight flush operations to the guest storage stack only during the restore phase.

We refer to the capability to close storage devices quickly as fast close. Fast close also offers storage resiliency due to temporary storage and network failures [12, 22]. Moreover, the capability to power off instantaneously provides fault isolation and the blackout time of co-hosted VMs are isolated from stragglers with slow I/O performance or failures.

5.2 VM-PHU Restore

The VM-PHU restore operation is similar to the VM-PHU save operation. We initiate the virtualization stack to restore all VMs simultaneously. For each VM, we first check whether the memory is saved via KSR. If the memory is persisted, we first restore the memory. We then power on the remaining devices. Each device during power on checks whether there is any corresponding state to restore from the memory. When a device detects saved state in memory, we restore the device state. Storage devices restore the in-flight I/O queue from memory before starting any operations. Once all of the devices are restored, we start the virtual processors to complete the VM restoration.

5.3 Hardware Acceleration

Hardware acceleration by assigning devices to the VM significantly improves the performance of the workloads by eliminating the host paravirtualization overhead. In our public cloud, we support two types of devices – devices that are entirely assigned to the VM and devices that expose virtual functions that can be assigned to the VM. As the physical devices assigned to the VMs cannot be power-cycled, we handle these devices with additional enhancements to VM-PHU.

Single root input/output virtualization (SR-IOV) devices: SR-IOV allows a device to isolate the PCI Express resources and allows the device to expose the virtual functions to VMs on a physical host server. SR-IOV support in physical devices is similar to virtualization extension support in processors to support performance of virtual devices in VMs to be similar to physical devices in servers while maintaining isolation. In our public cloud, we configure the hardware devices on the hosts to support SR-IOV in the BIOS. We primarily focused on accelerating physical network virtualization though the concepts are applicable to any virtual device using SR-IOV.

For each SR-IOV network interface card (NIC) attached to the virtual machine, we create a synthetic virtual NIC (vNIC) on the host server to shadow the SR-IOV NIC. The shadow vNIC allows the VM traffic to be redirected via the ordinary synthetic links in case there is a failover of the SR-IOV vNIC. The traffic failover is transparent to the application running within the VM. Our failover mechanism allows us to perform VM-PHU or LM or service the host device driver for a VM with SR-IOV devices. Prior to the start of VM-PHU, we detach the virtual functions from the VM allowing for the traffic to continue to work with ordinary vNIC. We then perform the VM-PHU update operations, and finally re-attach the VFs to the VM. The detach and attach operations are outside of the blackout window. Given the blackout window is small, the VM may experience some network performance degradation due to failover to ordinary vNIC only for few seconds. We regularly perform VM-PHU updates in production on VMs with accelerated networking. In practice, the failover mechanism works well and can be easily extended to other virtualization devices.

Discrete Device Assignment (DDA): DDA allows physical PCIe hardware to be direct accessible to a VM. The root operating system does not manage the DDA device and the DDA device is not shared across VMs. Moreover, the DDA device is managed using its driver in the VM and all of the DDA device capabilities are accessible to the VM. Unlike SR-IOV devices, DDA does not have a software fallback option that can be leveraged to power-off the device when the VM is being saved. As a result, many hypervisors do not support live migration or VM save operations for DDA VMs [21, 41].

Device operations including device I/Os are handled asynchronously from the CPU issuing the request. Since DDA devices are managed by guest VMs, the root operating system has no idea of outstanding operations when VMs are saved for a VM-PHU and does not have a driver running in the root during power off to flush outstanding I/Os. To handle this scenario, we must allow all outstanding device operations to succeed when VMs are paused and the root operating system is undergoing an update. The device operations that must be supported include device DMA and interrupts to signal I/O completion. Typically, DDA devices are mapped via the IOMMU to use the guest physical address (GPA) space of the VM. Addresses for DMA operations initiated by the device are treated as guest physical addresses and remapped to system physical addresses by the IOMMU. This ensures a device that is assigned to a VM is restricted to only being able to access physical memory that is assigned to the VM. To ensure that DMA operations can complete successfully during the KSR, the IOMMU is kept active for the duration of the KSR, and DMA remapping tables for a DDA device remain in-use during the KSR to ensure all DMA activity initiated by the device continues to flow as expected.

Similarly, for interrupts, interrupt remapping capabilities provided by the IOMMU are used to remap device interrupts for DDA devices. The interrupt remapping capabilities of the IOMMU also continue to remain active across a KSR. Thus, a device can continue to generate interrupts as usual while a KSR is occurring. However, the old and new hypervisors cannot handle processing device interrupt during the KSR transition, so interrupts for DDA devices are suppressed via the IOMMU during the KSR transition. This means that interrupts that occur during this period won't be seen by the hypervisor and thus could be lost. To mitigate this, after a KSR, we inject interrupts for each DDA device to trigger the device drivers for the DDA devices to execute in each VM. If the device did not actually interrupt during the KSR, the interrupt would be seen as spurious by the driver and discarded.

Maintaining the IOMMU across KSR required changes to our hypervisor and the hardware abstraction layer (HAL) of the root operating system to avoid reinitializing the hardware. Additionally, the hypervisor manages state for each VM which previously was reconstructed as part of the VM restore phase of VM-PHU. To support DDA, this state is constructed during the initialization of the new hypervisor to ensure contiguous protection and management. With these enhancements, VM-PHU supports updates to DDA devices such as GPUs and NVME devices.

5.4 Guest Update Notification

The goal of VM-PHU is to perform updates transparent to the guest VMs by minimizing the blackout time. In a public cloud environment, it is possible to have outliers for blackout time due to transient failures, bugs in the software stack, etc. While many applications are resilient to outliers, a small subset of applications may prefer restarting the application or the VM during the update period. For these applications, we provide a notification a few minutes before the update, and provide a post-notification after the update is complete. The guest can access the notification and take appropriate actions to failover or restart post-update.

6 Handling Regressions during Host OS Updates

Zero-day updates need to be deployed rapidly across the fleet and can introduce regressions. While typical regressions are surfaced as the updates roll through in stages from pre-production environments to production, the less frequent regressions can take a longer time to surface and are likely missed when the updates need to complete within a few days. Examples of these issues include memory leaks that can take several days to accumulate, driver hangs and server crashes that can take tens of days to surface and can cause significant customer disruptions. In order to safeguard the VMs from these rare regressions, we leverage VM-PHU as a critical safety mechanism during zero-day rollouts. As VM-PHU is able to reset the state of the host operating system while preserving the VMs, we can reset the host whenever regressions are identified. We specifically handle the following scenarios in our update methodology:

Memory leaks: Memory leak occurs when a driver allocates memory but does not release the memory after it is no longer in use. Such leaks can accumulate over time eventually causing other host drivers requiring memory to fail allocations and typically manifests as VM allocation failures. We detect these failures and trigger a VM-PHU with no updated bits. As the host operating system is reset, the leak is cleared. Furthermore, we detect regressions by analyzing the memory profiles of the components and rollback the update using VM-PHU to avoid the regression.

Driver hangs: A typical issue observed in the servers hosting VMs are reference count mismatches causing drivers to fail to reset. For instance, each paravirtualized device maintains a reference count initialized to zero. For each new IO that is being processed, we increment the counter and we decrement the counter after processing it. The device can be closed when there are no outstanding IOs in order to safely release all of the resources associated with the device and all devices assigned to a VM need to be closed when a VM is shutdown or powered off. However, if there are reference count mismatches, the driver cannot close the device and in turn result in VM hangs. These issues require restarting the server and will impact other VMs. We detect driver hangs and leverage VM-PHU to preserve all the remaining VMs except the hung VM. We then transition to the new OS clearing the state associated with the hung VM.

Driver crashes: A crash occurs when the host detects an incorrect behavior in the drivers. Few common examples include access violation issues due to buffer overflows, deferencing a bad pointer, etc. All of the VMs on the host experience an unexpected poweroff due to the crash. In these cases, we trigger VM-PHU and preserve the state of the VMs while clearing the state of the host OS by reinitializing the drivers in KSR.

In our production environment, we observe approximately a thousand server crashes per million servers over a period of a month. Among these, we identify 80% of the OS regressions via staged rollouts and fix them before the updates reach broad production. The remaining regressions are harder to detect via staged rollouts as the regressions take much longer time to surface and deployments may have reached majority of production – in these instances, we leverage VM-PHU to clear the host state while preserving the VMs and improve the overall VM availability.

7 Implementation and Analysis

We implemented our system on Windows Server 2016 operating system. We applied VM-PHU in Microsoft Azure

Driver	Description	# lines
PCI bus	Avoid powering off devices during KSR and remember resources (memory and I/O) for all devices	~ 1250
ACPI bus	Recall state from the cold boot (e.g., did BIOS boot in ACPI mode).	~ 500
Bitlocker	Provide encryption keys and internal meta-data	~ 750
NTFS	Persist in-memory filesystem meta-data, including the Bitmap file	~ 3000
NTOS Kernel	Trace buffers for performance analysis	~ 2500
Storage	Persists key meta-data in memory	~ 1500
Network	Persists the programmed set of rules.	~ 2000
Storage class	Persist controller information for IEEE 1667 devices to avoid reauthentication	~ 3000
Hypervisor	Hand off the IOMMUs from the old hypervisor to the new hypervisor while keeping them active	~ 45000

Table 3. Table highlighting the various components enlightened in the host and the lines of source code introduced for the enlightenment.

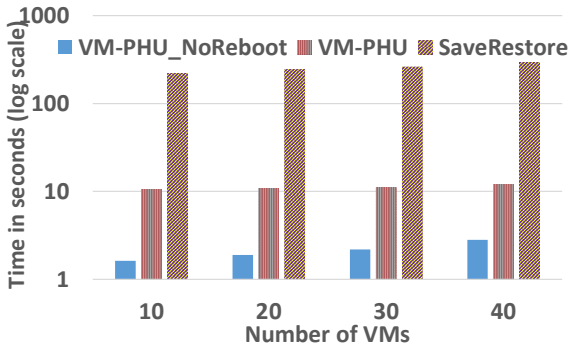

Figure 7. Blackout time comparison between our approach and traditional save and restore approach to apply updates in two scenarios – *rebootless updates* (VM-PHU_NoReboot) and *rebootful updates* (VM-PHU). Our approach has a low blackout time of few seconds even on high density hosts and is 30–100× faster than the traditional save restore approach.

cloud to millions of servers to perform two kinds of zero-day updates:

Rebootless driver updates: Rebootless updates require stopping of the dependent services on the host in order to eliminate any handles on the driver objects. The drivers can then be updated and the dependent services can be restarted. Examples of frequent updates in this category in a public cloud involve updates to the software defined virtualization (SDV) drivers on the host to enable new functionality or to fix critical bugs resulting in host crashes that impact VM availability. As the VM processes hold handles to the SDV drivers such as remote storage virtualization drivers, the VMs need to be suspended or shutdown to update the drivers, and the VMs are resumed post-update. We perform VM-PHU save operation to remove the handles on the drivers, update the driver and then perform VM-PHU restore operation to resume the VMs.

Driver updates requiring reboots: Updates to some of the core kernel, hypervisor, file system or TCP/IP drivers require restarting the host to apply the update. In these cases, the VMs can either be suspended or shutdown before the host is restarted and resumed post-restart. We perform VM-PHU save operation to persist and teardown the virtualization stack, KSR into patched operating system and apply VM-PHU restore operation to resume the VMs.

The blackout time for a VM is measured from the beginning of the fast save operation to the completion of fast restore operation for the VM. We applied our techniques to various densities of VMs in two main configurations in Fig. 7 – VM-PHU_NoReboot for rebootless updates, VM-PHU for performing the same core kernel update requiring a host reboot. We also compare the performance of our approach with a traditional save restore approach requiring reboot of the physical server. Fig. 7 highlights the blackout time in seconds on a log-scale. Our results indicate that rebootless updates can be performed in 3 seconds and rebootful updates can be performed within 11 seconds.

We enlightened many host components to improve the reliability and performance of VM-PHU. Table 3 summarizes the changes done in various host components along with the number of lines of code added to enlighten the drivers. We started the code development in 2012 and table 3 represents a development effort of two years. We deployed our changes to production in 2015. The enlightenments described in table 3 are primarily related to driver startup and shutdown phases. Two of the drivers – storage and network virtualization drivers, can be updated without requiring a reboot and we leveraged our rebootless driver update mechanism to deploy the updates to these drivers. Remaining drivers in table 3 are not designed to be unloaded on a running operating system and require a host OS reboot for updates. The host reboot will in turn result in disruptive VM reboots. As VM reboots are not an option, our deployment strategy to enable VM-PHU had to account for a mechanism to update several hundreds of thousands of servers already hosting VMs in production, via VM-PHU. For the drivers in table 3, we hotpatched the driver shutdown routines. These drivers also need to be enlightened during the startup phase. To enlighten the startup phase, we cold patched the updated driver on disk and loaded it into the new OS prior to KSR. We therefore leveraged rebootful VM-PHU to avoid VM reboots to deploy VM-PHU. As VM-PHU enablement is not a

zero-day issue, we rolled to production over three months using our safe deployment practices [36].

7.1 Zero-Day Use Cases

In this section, we highlight some of the real-world cases that required a zero-day rollout in Microsoft Azure during the last five years. We used VM-PHU to apply the patch in all of these cases:

- Spectre [26] and Meltdown [29] are modern CPU vulnerabilities that exploit speculative execution mechanisms to access information across trust boundaries. The mitigation for these class of attacks required changes to the core kernel and hypervisor. We applied these updates across production rapidly using VM-PHU.
- A vulnerability in the CPU instructions can trigger a host crash or hang from the guest by running a malicious program [31]. We designed an update that intercepts the specific instructions in the usermode virtual machine worker process. Our update is used to crash the guest as opposed to the host. The vulnerability was discovered externally and required patching the entire data center within few days.
- A vulnerability is discovered in the Hyper-V host storage stack where a malicious guest can send a specially crafted request to the host and can crash the host [9]. The update can be applied rebootlessly via VM-PHU and has been applied to all hosts in a few days.
- We observed crashes in the VM-PHU update as the core kernel had assumptions in the presence of contiguous memory for the page database. As the specific issue impacts the zero-day update path itself, we rolled out an update to patch the kernel via VM-PHU. The updated kernel is activated during KSR and is used to successfully address the crash.

Overall, we are able to rollout updates quickly and safely to production in a short order of few days. Moreover, we are able to perform updates to the VM-PHU mechanism itself by leveraging VM-PHU.

7.2 Analysis

Our zero-day patching results indicate that majority of the VMs observe a blackout time within TCP/IP retransmit timeout. Furthermore, the applications running within the VMs were resilient to the network latencies similar to the expected network resilience behavior of most internet applications. Fig. 8 illustrates the blackout times observed for rebootful and rebootless zero-day updates in our production environment with millions of servers. The blackout times are dependent on both the density of the VMs on a host, the kernel soft reboot times, and the number of memory runs that are persisted. In general, the blackout times are linear to the number of VMs and the number of runs. At higher percentiles, fragmentation introduces an overhead of upto

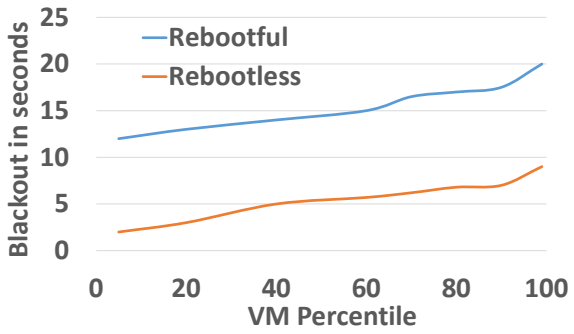

Figure 8. This figure highlights the blackout times of VMs in our production environment. Our production environment hosts millions of servers and tens of millions of VMs. The blackout times for rebootful updates range from 12 – 20 seconds and our rebootless updates range from 2 – 9 seconds.

four seconds as illustrated in Fig. 5 and higher VM density introduces a few seconds of overhead as shown in Fig. 6.

Some applications were sensitive to the clock drift experienced after the VM is restored. The clock drift is experienced due to the time synchronization of the guest VM with the host. This small subset of incompatible applications can opt-out of VM-PHU using our guest notification mechanism. Moreover, many latency sensitive applications are designed to use failover configurations. These applications can further tolerate the network delays during VM-PHU.

Due to fast patching time via soft reboot, even the opt-out VMs observed higher availability. Furthermore, as we improve the blackout time parallelizing VM-PHU Save and VM-PHU restore operations with host shutdown and restart operations, more applications can leverage VM-PHU to achieve significantly higher availability.

We also compared our techniques with live migration. Specifically, we used the live migration implementation in Microsoft Hyper-V in Microsoft Azure. LM requires all of the VMs on the server to migrate to a new server for host OS updates. Once the server does not host VMs, it can be updated. As discussed in Section 5.3, LM does not work on DDA VMs. Due to the LM applicability limitation, even if a server has a single DDA VM, it cannot be updated via LM and the probability of inapplicable hosts increases with the increased density of VMs on a host. We observed that over 50% of the hosts in our environments cannot be updated via LM due to its limitation to migrate all of the VMs. On the hosts where the VMs are emptied via live migration, the blackout due to LM is observed to be 13 seconds at 99^{th} percentile. Unlike VM-PHU which is near constant blackout per server, LM blackout is dependent on the workloads and the amount of memory pages that are dirty. We expect LM blackout to be higher at the larger percentiles due to the memory intensive applications in the cloud and the blackout due to LM is comparable to VM-PHU. Moreover, for updates that

do not require reboots, VM-PHU has 2 – 3× lower blackout compared to LM. Furthermore, VM-PHU is applicable to all VMs including DDA.

Our analysis in Microsoft Azure indicates that transient network and storage outages can occur intermittently due to maintenance, unplanned network switch reloads, etc. These sporadic failures can impact around one percent of the VMs. Using the fast close technique, VM-PHU isolates and tolerates network failures in the order of few minutes. Moreover, our technique to fast save VMs can handle unplanned outages by detecting high storage or network latencies and preserving the VM state during the outage period.

While it is hard to predict the zero-day occurrences, we implement several security best practices to reduce the surface area of the Azure host operating system and further go through considerable security reviews and code scanning techniques to fix issues proactively. On an average, we perform about 2–3 rebootless updates and at most one rebootful update per year. During our updates, we observed that approximately 0.1% of VMs undergo failures during the update due to unrelated quality bugs exposed during reboots. As reboots are relatively rare in the data centers, we expect failures when the updates are performed at scale. We use our resiliency techniques to minimize the failure impact. Furthermore, if the failure rate is high, we stop the ongoing rollout and incorporate the required hotfixes to address the failures and resume the rollout.

Learnings: Below are few learnings from our operational experience:

- Continuous exercise of the VM-PHU scenario in pre-production environments is necessary to ensure there is sufficient coverage and identify regressions early. We configured our pre-production environments to exercise the VM-PHU scenario on a weekly basis. We trigger the update steps without requiring a new host OS to exercise VM-PHU.
- Reboots can expose issues unrelated to host OS updates or the update mechanism when applied at scale. Resiliency is key to ensure that updates at scale do not cause impact to customers.
- Multiple update mechanisms are required to manage the fleet including hotpatching, VM-PHU and LM. Fixes to a mechanism can often leverage other mechanisms. For instance, enablement of VM-PHU in production required hotpatching support.
- Updates need to be cumulative and should leverage the least impact approach accounting for potential retakes. For instance, if a rebootful update (v_1) is partly rolled out on the fleet and we perform a retake on a driver (d) that can be rebootlessly serviced, the new update (v_2) will need to be cumulative. On the nodes that received v_1, we only service d and the update will be rebootless. On the remaining nodes that did not receive v_1, the update will be rebootful leveraging KSR.

Limitations: Our techniques have some limitations. We mainly focused our work on addressing the scenario of quickly updating the hosts in a data center. Our techniques do not address other issues in a data center such as hardware decommissioning. Such scenarios may be better handled via techniques such as live migration. Also, our techniques do not allow for dynamic load balancing opportunities in a data center that are possible with live migration.

8 Conclusions and Future Work

In this paper, we presented techniques to perform host updates quickly to address zero day updates in Microsoft Azure. We presented a novel approach to preserve the VM state across a soft reboot. We improved the performance and reliability of reboot bypassing the firmware post while preserving the physical device configuration and memory across reboots. We also designed optimizations to improve the memory preservation time even in situations with severe host memory fragmentation. On the virtualization stack, we presented optimizations to fast close the devices eliminating dependencies on network and remote storage. We also presented a transparent failover mechanism for handling SR-IOV devices by leveraging shadow virtual devices in the host and leveraged the failover mechanism to extend VM-PHU updates to VMs with SR-IOV devices. We discussed the preservation of page protections and domain mappings associated with DDA devices across KSR to extend our techniques for DDA VMs.

We compared our approach to normal reboots and observed our soft reboot technique to be 10–30× faster. We highlighted the results of our system on several zero day scenarios over the last decade on one of the largest cloud environments. Our results demonstrate that VM-PHU is an efficient technique for handling all host OS updates including time-critical zero day scenarios.

Acknowledgements

We would like to thank the several team members of the VM-PHU project for their contributions and support: Mukhtar Ahmed, Brandon Allsop, Craig Barkhouse, Deepak Bansal, Alex Bendetov, Vivek Bhanu, Kevin Broas, Omar Cardona, Jim Cavalaris, Terence Chan, Harish Kumar Chandrappa, Alireza Dabagh, Aacer Daken, Julien Daouphars, Suhas Deshmukh, Dustin Douglas, Daniel Firestone, Milos Kralj, Sai Ganesh Ramachandran, Vijaybalaji Madhanagopal, Attilio Mainetti, Dmitry Malloy, Karan Mehra, Tom Ootjers, Avinash Patil, Suraj Puri, Vaishak Salin, Ilya Sacson, Andrey Shedel, Max Yanchenko and Jie Zhou. This work would not be possible without the strong support from the members of the Microsoft leadership team: Girish Bablani, Albert Greenberg, Scott Guthrie, Satya Nadella, and Jason Zander.

References

[1] 3D XPoint 2015. Analyzing Intel-Micron 3D XPoint: The Next Generation Non-Volatile Memory. http://www.anandtech.com/show/9470/intel-and-micron-announce-3d-xpoint-nonvolatile-memory-technology-1000x-higher-performance-endurance-than-nand.

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

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.

[4] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. 2005. Providing Dynamic Update in an Operating System. In *USENIX Annual Technical Conference, General Track*. 279–291.

[5] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 398–407.

[6] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. Polus: A powerful live updating system. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 271–281.

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

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

[9] CVE-2017-0184 2017. CVE-2017-0184 | Hyper-V Denial of Service Vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2017-0184.

[10] Alex Depoutovitch and Michael Stumm. 2010. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/1755913.1755933

[11] 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*. 45–58.

[12] Fast Close 2015. Virtual Machine Storage Resiliency in Windows Server 2016. https://techcommunity.microsoft.com/t5/virtualization/virtual-machine-storage-resiliency-in-windows-server-2016/ba-p/382250.

[13] Cristiano Giuffrida, Călin Iorgulescu, and Andrew S Tanenbaum. 2014. Mutable checkpoint-restart: automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*. ACM, 133–144.

[14] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2013. Safe and automatic live update for operating systems. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 279–292.

[15] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks, and Jeffrey S Foster. 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 249–264.

[16] Christopher M Hayden, Edward K Smith, Michael Hicks, and Jeffrey S Foster. 2011. State transfer for clear and efficient runtime updates. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*. IEEE, 179–184.

[17] Michael Hicks, Jonathan T Moore, and Scott Nettles. 2001. *Dynamic software updating*. Vol. 36. ACM.

[18] Michael R Hines and Kartik Gopalan. 2009. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 51–60.

[19] Hotpatch Limitations 2003. Using Hotpatching Technology to Reduce Servicing Reboots. https://technet.microsoft.com/en-us/library/cc778993(v=ws.10).aspx.

[20] Hyper-V 2009. Microsoft Hyper-V Server 2008 R2: Using Live Migration. https://technet.microsoft.com/en-us/library/ee815293(v=ws.10).aspx.

[21] Hyper-V Discrete Device Assignment Limitations 2019. Virtual Machine Limitations. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/plan/plan-for-deploying-devices-using-discrete-device-assignment#virtual-machine-limitations.

[22] Hyper-V Storage Resiliency 2016. Windows Server 2016: Hyper-V Aspects. https://social.technet.microsoft.com/wiki/contents/articles/33635.windows-server-2016-hyper-v-aspects.aspx.

[23] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. 2016. Instant OS updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 605–619.

[24] kgraft 2015. SUSE Releases kGraft for Live Patching of Linux Kernel. https://www.suse.com/company/press/2014/3/suse-releases-kgraft-for-live-patching-of-linux-kernel.html.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Vol. 1. 225–230.

[26] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[27] kpatch 2014. Introducing kpatch: Dynamic Kernel Patching. http://rhelblog.redhat.com/2014/02/26/kpatch/.

[28] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[30] Kristis Makris and Rida A Bazzi. 2009. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX Annual Technical Conference*, Vol. 2009.

[31] Microsoft Advisory 2015. Update for Windows Hyper-V to Address CPU Weakness. https://technet.microsoft.com/library/security/3108638.

[32] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. PatchDroid: scalable third-party security patches for Android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 259–268.

[33] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/1133981.1133991

[34] Hariprasad Nellitheertha. 2004. Reboot Linux faster using kexec. *developerWorks technical library* (2004).

[35] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. 2005. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference, General Track*. 391–394.

[36] Mark Russinovich and Cristina del Amo Casado. 2020. Advancing safe deployment practices. https://azure.microsoft.com/en-us/blog/advancing-safe-deployment-practices/.

[37] Mark Eugene Russinovich and Melur K Raghuraman. 2015. Virtual machine-preserving host updates. US Patent 9,110,762. Filed December 4, 2012.

[38] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshi Moriai. 2008. Kemari: Virtual machine synchronization for fault tolerance. In *Proc. USENIX Annu. Tech. Conf.(Poster Session)*.

[39] Timothy Wood, Emmanuel Cecchet, KK Ramakrishnan, Prashant Shenoy, Jacobus Van Der Merwe, and Arun Venkataramani. 2010. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*. 1–7.

[40] Timothy Wood, H Andrés Lagar-Cavilla, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. 2011. PipeCloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 17.

[41] xen pci limitation 2011. Xen PCI passthru limitations. https://wiki.xen.org/wiki/Xen_PCI_Passthrough#Xen_PCI_passthru_limitations.

[42] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 93–105.