VM Live Migration At Scale

Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, Tyler Sanderson

Google Inc

{ruprecht,jonesdl,dshiraev,gharm,mspivak,mk Krebs,miche,tysand}@google.com

Abstract

Uninterrupted uptime is a critical aspect of Virtual Machines (VMs) offered by cloud hosting providers. Google's VMs run on top of rapidly changing infrastructure: we regularly update hardware and host software, and we must quickly respond to failing hardware. Frequent change is critical to both development velocity—deploying new versions of services and infrastructure—and the ability to respond rapidly to defects, including critical security fixes. Typically these updates would be disruptive, resulting in VM termination or restart. In this paper we present how we use VM live migration at scale to eliminate this disruption with minimal impact to the guest, performing over 1,000,000¹ migrations monthly in our production fleet, with 50ms median blackout, 300ms 99th percentile blackout.

1 Introduction

Live migration of a Virtual Machine (VM) involves moving a running VM from one physical host to another, with minimal disruption to the guest [2]. At Google, it is a key technology behind the Google Compute Engine (GCE) [7] product.

GCE VMs run on top of Google's Borg [16] production infrastructure. Borg runs large-scale distributed services, maximizing uptime for a service rather than an individual task, as well as maximizing utilization across the production machine fleet.

The consequence of this is that tasks expect to be preempted (i.e. terminated) on a regular basis. This is not a problem for a typical distributed application; however, this is undesirable for a virtual machine. Users reasonably expect their VMs to be long lived (much like physical hardware).

¹The value of 1M/month is a lower bound intended to provide context for scale. It is *not* an estimate for the number of monthly migrations.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

VEE '18, March 25, 2018, Williamsburg, VA, USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5579-7/18/03.

<https://doi.org/10.1145/3186411.3186415>

Additionally, Google's fleet undergoes both rolling and large-scale hardware and software maintenance. Service owners (such as GCE) are expected to run systems that are resilient to hardware becoming unavailable for extended periods of time.

Finally, it is also important to continuously upgrade the implementation of the Virtual Machine software itself. Every external dependency and every bit of software on a host, with the exception of the guest itself, are routinely updated. Figure 1 shows some of these pieces. Our implementation of the Virtual Machine Monitor is as a userspace library that interacts with KVM.

The benefits of an up-to-date host OS and continuously upgraded hardware are numerous, but achieving those benefits posed a significant challenge for the GCE platform. We use VM live migration to overcome the volatility of Google's stack.

The implications of this setup are:

- VM live migrations are numerous. This means that even a rarely triggered bug has catastrophic consequences, and the operational characteristics of a live migration must be easy to manage. Failures occur regularly and need to be handled gracefully.
- The VMs in question are user VMs that exhibit a wide range of workloads. Live migrations cannot cause significant degradation of services running on the VMs. Performance is critical.

In this paper, we describe how we implemented and applied live migration at Google. We talk in some detail about the particular approach we took for live migration, the challenges we faced performing live migration at scale, and the consequences of working with a wide variety of user workloads. We describe our approach and how it is similar to other industry approaches. We discuss what it takes to keep a robust live migration infrastructure running at a large scale. We also detail the particular performance optimizations we have implemented so far and provide measures of their impact.

2 Applications at Google

When we began to work on live migration technology, a major challenge we faced was internal infrastructure maintenance. Google routinely brings down all hardware in each datacenter on a rolling basis over a relatively short period. This was unacceptable for GCE where virtual machine uptime is expected to match (or exceed) typical physical hardware. We

were able to solve this problem transparently to our users with live migration, but we were surprised to discover that there are many reasons for performing live migrations beyond just responding to rolling maintenance windows. Many migrations are due to hardware issues; many more are related to software health; and a surprising number are used to respond to operational issues in the fleet.

2.1 Hardware Reasons for Migration

- Datacenter maintenance. This was our initial motivation for implementing live migration. Regular maintenance on the power infrastructure in our data centers requires powering down subsets of machines for extended periods of time. We use live migration within a datacenter to drain machines before powering them down.
- Overheating power supplies. When live migration was well into test, our SRE (Site Reliability Engineers) team noticed there was a single host that had an overheating power supply. It had already overheated several of its neighbors and several of them had, in turn, overheated their neighbors. Our monitoring noticed this event; left unchecked, it would bring down a cascading set of neighboring hardware. The SRE team manually issued the commands to live migrate the VMs away from the affected hosts before bringing them down. This was our first unexpected use of live migration and our first view into the number of VM lives that could be saved.
- Flapping NICs, bad DIMMs [1] and hard drives [4], and other hardware mysteries. There is always failing hardware in the fleet, and we try to migrate the VMs off the host before bringing it down. When the rate of correctable ECC errors on a machine reaches a certain threshold, the machine is scheduled for repairs and VMs are migrated off the host (uncorrectable ECC errors are fatal).
- Hardware upgrades. We shut down physical hosts across a data center to install new hardware, such as new SSDs, upgrade memory (swapping DIMMs), and deploy new network fabrics. This allows us to launch new hardware features (local SSD VMs, high memory VMs) in existing data centers.

2.2 Software Motivations for Migration

Figure 1 shows our virtualization stack and its dependencies, both on the host and externally. Except for the guest, we update every piece of the diagram continually and independently.

- Kernel upgrades. We upgrade the host kernel across the fleet regularly, which results in migrating every VM at least once. We optimize to minimize the number of VMs that need to migrate more than once.
- Upgrades to the Virtual Machine Monitor. We have historically used a technology internally called “in-place

Figure 1. Virtual machine monitor (VMM) and its immediate environment. Every system except for the guest is regularly and often independently updated.

upgrade” to upgrade and downgrade versions of the software that makes up the VMM. In this process, the old VMM serializes state to local RAM, exits, and then the new VMM starts and deserializes the state. This serialization path is also used by live migration. Our live migration implementation has now surpassed in-place upgrade in blackout performance, and, in spring 2016, we started replacing in-place upgrade with live migration.

The number of components involved leads to versioning challenges, which we discuss in Section 3.3.

2.3 Operational Opportunities

As we continued to learn about running a large virtual environment, we found an additional set of challenges that live migration solved.

- Bin packing to preserve space to create large instances. We live migrate enough smaller VMs to other hosts to ensure that there are large enough holes to accommodate our largest instances in every zone.
- License management. The license requirements for some underlying software used by our guests can implicitly determine the best location for a guest. For per-host licenses, it may make sense to move instances that require a common license to the same host.
- Load balancing. We move VMs to optimize performance load (for instance, the utilization of memory, CPU, or network bandwidth). We also move VMs to manage risk by balancing load across failure domains. We make these moves for overall fleet Service Level Objectives (SLOs), and for SLOs within single projects. Rack or power-circuit failures result in the loss of every VM on a set of hosts; we use live migration to ensure that no physical fault domain has an unreasonable number of VMs, either overall or for a single project.

2.4 Working Around Existing Bugs

Sometimes, bugs get through—that’s why we have a downgrade mechanism to rollback software changes. Occasionally, however, we encounter a bug that cannot be easily addressed with a rollback. This might happen because the bug took longer to manifest than the window in which rollback is possible. More catastrophic, despite all our testing, the downgrade software may have an issue that would result in VMs dying if the changes were rolled back. In these rare cases, we may still be able to live migrate the VM to a host that does not have the software issue and entirely avoid the problem. Real examples of this include:

- Making a stopgap fix for a kernel driver memory leak. We had a host kernel release that contained a slow memory leak in our local SSD component. We were alerted when several VMs crashed due to out of memory errors. We had to re-qualify a host kernel that contained the fix, a process that took several weeks. In the meantime, we used live migration to move the affected VMs to a fresh runtime environment (the leak was still present, but the migration bought us time). The rollout of the fixed kernel used live migration as well, as is normal for host kernel rollouts.
- Reducing risk of new hardware deployment. After new hardware is qualified and starts being deployed in production, if we detect a defect that was missed during qualification, we use live migration to move VMs from the affected hosts. This allows us to deploy new hardware platforms to users with less risk to user VMs.
- Responding to a bad software push that would otherwise crash VMs. We had a situation in 2015 where an upgrade caused out of memory (OOM) failures for some VMs due to a misconfiguration of their container limits. The normal in-place downgrade mechanism could not fix this. In fact, the act of downgrading risked OOM-killing the VM. We used live migration to move affected VMs into properly configured containers.

2.5 Causes of Migration

Our initial motivation for implementing live migration was to deal with scheduled hardware maintenance, which is very common at Google. We also knew that live migration could save VMs from certain categories of hardware failure. However, currently, our primary reason for live migrations is disruptive software updates. Figure 2 breaks down the reasons for our migrations between January and March 2017.

3 How Live Migration Works at Google

Executing a live migration goes well beyond the source and target tasks; Figure 1 shows the components required to run a VM in our environment. There are several coordination challenges among all of these components. Each component is independently versioned and multiple versions of the same

Figure 2. Causes of migration for all production migrations Jan 2017–March 2017.

component may coexist. The hardware environment is not homogeneous—we cannot rely on having the same capabilities available on the source and target hosts. In this section, we describe how we migrate the individual components as well as how we solve the overall coordination and versioning problems.

3.1 Live Migration & Borg

VMs in GCE run under the Borg cluster manager; each virtual machine runs in its own container as a single Borg task. Tasks running under Borg are typically designed to be restarted at any time. For example, Borg optimizes for utilization and evicts jobs for better bin packing [16]. Host software upgrades are also frequent and cause termination or restart of every task on the host. Borg focuses on the overall uptime of a service that is composed of many relatively stateless instances. This focus is not a perfect fit for virtual machines, which are highly stateful and individually important.

To bridge this gap, we built some additional pieces of infrastructure. All of the configuration information required to get a virtual machine running is handled by an instance manager, which is a standard Borg job (i.e. runs as a fault-tolerant, mostly stateless process). The instance manager also knows about the state of the VMs that are running and is capable of initiating migrations of those VMs. Borg was also extended to allow jobs to be marked as migratable; if a migratable task needs to be restarted or preempted, Borg performs a live migration instead [16]. Additionally, a separate Borg job, called the Safe Removal Service, rate-limits migrations according to a few dimensions: by host, by top-of-rack switch, and within the same customer project.

3.2 Live Migration Phases & Components

The virtualization stack has a number of components that need to be involved in live migration. They range from expensive-to-migrate local resources (memory, local storage) to remote resources that simply need to know when a migration occurs

Figure 3. Live migration, phases & components. Note: migration phase durations are not to scale. When the instance has no local SSD, networking and memory pre-copy start at the same time.

(persistent disk, networking). Figure 3 depicts the components that are involved in migration and how they progress through the three phases of a migration.

3.2.1 Phases

Our migration is defined in terms of three phases:

- **Source brownout**—the VM is still running on the source host while a new task starts on the target. Components may communicate with the target to pre-transfer state. For example, pre-copy memory migration will iteratively send state to the target.
- **Blackout**—the VM is paused. All remaining necessary state is transferred to the target and the VM resumes on the target.
- **Target brownout**—the VM has started running on the target host. Network routing is updated asynchronously, so any network packets that reach the migration source are forwarded to the target. Post-copy memory migration is completed, if necessary.

3.2.2 Memory Migration

Most live migration papers [2, 8, 9, 11, 15] focus on memory, and for good reason; it is difficult to migrate memory in a performant manner. A guest workload may rapidly mutate memory during migration, and our largest VMs have over

600GB of RAM. Our solution to memory migration is a hybrid of optimized pre-copy and post-copy, which we describe below.

Pre-Copy

Our initial solution focused on pre-copy [15]. We make multiple passes over memory, sending any changed pages from the source to the target at each pass. We use several criteria to determine when we should end the pre-copy step: dirty memory drops below a predetermined threshold, a predetermined number of passes have occurred, or the total amount of dirty memory increases between two passes. Once any of these conditions are met, we signal that the memory component is ready for blackout and continue making passes. When blackout begins we transmit any remaining memory.

This approach has been well studied in literature [2, 9, 11, 15] and we found it to be highly effective. In load testing, we observed that one consequence of the pre-copy algorithm is that if we kept a fixed working set size and increased the write rate, blackout times did not increase smoothly. As long as the write rate was lower than available network throughput, blackouts tended to stay near their minimum values. As write rate approached and exceeded available network throughput, blackouts rapidly degraded to the maximum time needed to transfer the entire working set size. Figure 4 demonstrates this behavior in a highly contended test environment, running a synthetic memory benchmark for a n1-standard-8 VM

Figure 4. Pre-copy blackout vs memory load using a synthetic memory load benchmark with a VM with 8 vCPUs and 30GB of RAM.

(8VCPU, 30GB). It is important to note that the blackout ceiling in this graph is entirely a function of working set size vs. available network bandwidth in our test environment.

Our environment with the Jupiter cluster network fabric [13] allows for very high bisection bandwidth. This means that we can support a very wide range of memory intensive workloads with minimal blackout using only the pre-copy technique. Figure 5 shows the distribution of blackouts for the same VM type over the course of a single month. The last bucket contains blackouts longer than the 99th percentile, with very few workloads having both a very large working set size and sufficiently high write rate. However, those few workloads were still user VMs that were paused for a sufficiently long time, enough to cause disruption. We implemented several optimizations to our pre-copy approach and added memory post-copy to our memory migration algorithm to support VMs with a large dirty set and high write rate.

Unpopulated Page Detection

When copying memory, the source may not have touched all of its available pages. Since these pages are not dirty, we do not need to copy them to the target. This information is available to the host kernel; however, we had to implement a system call (inspired by [3]) to make it available to our userspace migration logic. During the first memory pass in source brownout we use this syscall to determine whether a page is populated or not. We then refrain from sending any unpopulated pages to the target.

Unpopulated page detection provides two benefits. First, it improves performance for very large memory VMs. Second, it prevents the target VM from allocating physical memory for these untouched pages.

Post-Copy

The pre-copy memory migration approach provided favorable fleet-wide median blackouts. However, a small set of workloads had a sufficiently high write rate that resulted in

Figure 5. Distribution of blackouts for a VM with 8 vCPUs and 30GB of RAM (ms). Data from Nov 2015. Last bucket is worst 1% and contains blackouts longer than 512ms. This data is from a time period when our migrations did not use post-copy. This demonstrates the fraction of migrations that stood to benefit the most from memory post-copy. When we enabled post-copy in production, we found that we performed post-copy for approximately 3% of production migrations.

bad blackouts. Note in Figure 5 that the worst 1% of blackouts for a 30GB memory instance are represented by the 512+ms bucket. The distribution of migrations in the 512+ms bucket has an average of 1.2 seconds and also a significant tail with much longer duration outliers due to heavy memory workloads. We solved the long-tail blackout problem for high-memory workloads by implementing post-copy.

With post-copy [8], the VM may be resumed on the target before all dirty memory has been migrated. We use background fetching to continue migrating memory contents from the source. If a guest’s vCPU tries to access a page that has not yet been fetched, that page is requested from the source and the vCPU is blocked until it’s copied into the guest.

The tracking and sending of dirty memory is done on the source during pre-copy. During post-copy the target needs to know what the outstanding dirty pages are so that access by the guest to any of those pages results in a page fault. In order to reduce the risk of the metadata getting out of date, and also to simplify the implementation, we shift the onus of migrating memory completely to the target during blackout. The source stops sending memory as soon as blackout begins, and the target instead starts fetching memory as soon as it receives the list of dirty pages that are still outstanding. A more formal description of our approach is given in Algorithm 1.

Table 1 shows the 50th, 75th, 99th, and 99.9th percentiles for several post-copy metrics. The data was collected from the migrations that occurred during a continuous period of several weeks in 2016 and that performed post-copy during target brownout. These migrations make up 3% of all migrations during that time period; approximately 80% of our migrations go into blackout with outstanding dirty memory that will be

Table 1. Post-Copy Metrics (n=62650). Note these are for migrations of guests with the highest memory load - with the criteria used for entering blackout, only 3% of all migrations ever pause a vCPU in postcopy for the given sample period. Thus, P50 for post-copy migrations is really P98.5 for the fleet.

Fleet-wide	P98.5	P99.25	P99.97	P99.997
Post-copy subset (3%)	P50	P75	P99	P99.9
Duration (ms)	128	660	27970	53877
Normalized per-vCPU				
Total Wait Time (ms)	91	413	20476	39066
Median vCPU per-Fault Wait Time (ms)	0.49	0.83	3.76	16.00

transferred with our post-copy code, but almost all of them finish transferring that memory in the background during blackout. The first line in the table shows the total duration in milliseconds of the post-copy stage of migration. Next is the total wait time in milliseconds the vCPUs incurred on page faults, normalized by the number of vCPUs in the VM (total wait time / number of vCPUs). Lastly, we show the median vCPU page fault latencies in milliseconds.

3.2.3 Network & Persistent Disk

We also faced problems with high tail latencies in both networking and persistent disk.

During a migration, our network fabric must be configured to route packets for the VM’s IP addresses to the target instead of the source. To minimize downtime, we program the fabric with the new configuration during source brownout. During blackout we activate the new configuration. However, in some cases there may be a noticeable delay before the new configuration fully propagates across our network. We solved this by having the source host forward any packets that it receives during target brownout to the target.

Persistent Disk (PD) is our remote storage solution [6]. It does not need to move any guest data during a migration; it simply needs to disconnect from the source at the start of blackout and reconnect to the target before the end of blackout. However, the initial reconnect is occasionally slow. We solved this problem by sending additional disk state details to the target and beginning target-side initialization during source brownout.

3.2.4 Storage Migration: Local SSD

Most GCE VMs rely entirely on network attached Persistent Disk for storage, which is relatively inexpensive to migrate

Algorithm 1 Memory migration

```

1: procedure MIGRATEMEMORY
2:   EnableMemoryDirtyTracking()
3:   last_num_dirty  $\leftarrow$  max_int
4:   while not IsBlackout() do
5:     dirty_pages  $\leftarrow$  ReadMemoryDirtyLog()
6:     if dirty_pages.size > last_num_dirty or
       dirty_pages.size = 0 then
7:       SignalBlackoutReady()
8:       last_num_dirty  $\leftarrow$  dirty_pages.size
9:       for each page in dirty_pages do
10:        data  $\leftarrow$  RetrieveGuestMemory(page)
11:        SendUpdate(page, data)
12: RetrieveGuestMemory(page):
13:   d  $\leftarrow$  ReadFromGuest(page)
14:   checksums[page]  $\leftarrow$  Checksum(d)
15:   return d
16: OnBlackout():
17:   PauseGuest()
18:   dirty_pages  $\leftarrow$  ReadMemoryDirtyLog()
19:   StartPostCopyOnTarget(dirty_pages)
20: OnMigrationComplete():
21:   for each page in checksums do
22:     d  $\leftarrow$  ReadFromGuest(page)
23:     Assert(checksums[page] = Checksum(d))
24: OnTargetReceiveUpdate(page, data):
25:   WriteToGuest(page, data)
26: OnTargetStartPostCopy(dirty_pages):
27:   ResumeGuest()
28:   for each page in dirty_pages do
29:     data  $\leftarrow$  RetrieveSourceGuestMemory(page)
30:     WriteToGuest(page, data)
31: OnTargetGuestPageFault(page):
32:   data  $\leftarrow$  RetrieveSourceGuestMemory(page)
33:   WriteToGuest(page, data)

```

due to its remote nature. However, some also rely on high-performance, locally attached SSD storage [5]. The SSD storage is designed around high performance; a VM can issue up to 360,000 write IOPS (1.4 GB/s) or 680,000 read IOPS (2.7 GB/s) to up to 3TB of SSD. This high performance comes from the storage being locally attached and unreplicated. As a result, we must move its entire contents during a live migration. The size and speed of the SSDs, as well as the interaction with memory migration, both pose significant challenges.

The algorithm we use, dubbed “write-mirroring”, is one seen elsewhere in industry and academia [10]. During migration we perform a single read pass over the disk to read and send all content to the target, skipping unwritten blocks as a significant time-saving optimization. Meanwhile, we hook into the stream of guest I/O to the disk and make note

Table 2. SSD source brownout times in seconds

Capacity	Count	P25	Median	P75	P99
All	25498	353	522	821	2036
375 GB	13314	319	425	724	2013
750 GB	4385	486	688	934	2033
1500 GB	2016	664	767	951	2064
3000 GB	475	871	1505	1807	2523

of write and discard operations; operations addressed to regions that were already migrated are immediately mirrored to the target before the guest is notified of their completion. A rough overview of the algorithm is given in Algorithm 2; note methods beginning with “On” are event-driven and run asynchronously with the main migration loop.

For I/O devices this algorithm has advantages over the dirty-tracking algorithm we use for memory migration. It saves device-read bandwidth since the data to mirror is available in memory at mirroring time, whereas a dirty-tracking algorithm needs to reread the data from the device. The algorithm also guarantees forward progress and a near-zero blackout time because it implicitly provides backpressure on the guest when migration cannot keep up.

On the other hand, write mirroring has some downsides. If the guest writes to a few blocks very often then write mirroring requires forwarding all of these operations, whereas a dirty-tracking algorithm would only send each block once per pass. However, we expect the guest OS file cache to limit this scenario in practice. Write mirroring also couples the network performance and target-side device performance directly to the guest’s device I/O performance; if we cannot mirror the I/O then we cannot complete the I/O back to the guest. In the rare situations that this arises we detect the slow performance and cancel migration, resulting in the VM continuing to run on the source. In practice, we have found that mirrored writes, even in the worst cases, are not significantly delayed by the migration and that cancellation due to slow performance is usually caused by a bug or hardware problem.

Table 2 shows the time it takes to move SSDs. Data about SSD capacity was not available for all migrations, which is why N for “All” is larger than the sum of each individual capacity. The best performance comes from small SSDs with little data to send; the worst comes from large SSDs writing heavily. The data shows migration times two orders of magnitude longer than memory-only migrations.

3.2.5 Coordination

With many components migrating simultaneously (e.g. Memory, SSD, Persistent Disk, Network), coordinating the individual component migration algorithms is necessary for a short blackout and to minimize the overall amount of work performed. Each component implements a common migration interface. The component is notified as a migration reaches

Algorithm 2 SSD write-mirroring migration

```

1: procedure MIGRATE
2:   for each address  $i$  do
3:      $data, trimmed = \text{ReadFromDevice}(i)$ 
4:     if not  $trimmed$  then
5:        $\text{SendUpdate}(data, i)$ 
6:        $\text{SignalBlackoutReady}()$ 
7:    $\text{OnGuestWrite}(address, data)$ :
8:     if  $address \leq i$  then
9:        $\text{SendUpdate}(data, address)$ 
10:   $\text{OnTargetReceiveUpdate}(data, address)$ :
11:     $\text{WriteToDevice}(data, address)$ 

```

various milestones, and can request holds to prevent further overall progress until it completes all necessary work for a phase.

Each component provides continual progress reports to the main migration controller. These reports contain an estimate of source brownout work remaining and a bit indicating whether the component has performed as much pre-blackout work as possible.

Generally, the main migration controller waits for all components to report that they have completed as much pre-blackout work as possible before entering blackout. However, components in source brownout must be ready to enter blackout at any time. If there is an imminent hardware event, entering blackout early may keep the VM alive at the cost of a longer blackout.

VMs with local SSDs typically also have busy memory workloads. This means memory migration often has lots of data to send and uses large amounts of network bandwidth. Since SSD an order of magnitude more data to send and takes proportionally longer to migrate than other components such as memory, we achieve lower overhead and significantly shorter migration time by employing a simple heuristic we’ve named “staggered component start”. The main migration controller starts a component’s migration process when, based on all components’ progress reports, that component has the most work remaining. In this way, memory migration only begins when it has the same amount of data to send as the remaining SSD data, avoiding unnecessary network contention for as long as possible and ideally completing all component migrations simultaneously.

Figure 6 shows the effect staggered component start has on some selected production migrations. The rate of progress is dramatically lower after memory migration starts. This is particularly obvious for Migration C, but Migrations A and B also show noticeably slower progress after the start of memory migration. Table 3 shows this relationship explicitly by measuring the SSD migration progress rate before and after memory migrations starts and using the difference in progress rates to infer how much time was saved. At the 90th

Figure 6. Progress of SSD for 3 selected production migrations. The beginning of memory migration, as triggered by staggered component start, is noted for each migration. Note the slope change reflecting a lower rate of progress after memory migration starts.

Table 3. Staggered start’s effect on source brownout times for migrations with 375GB of SSD (n = 8871)

	Avg	P50	P75	P90	P99
Seconds saved	132	68	190	341	1468
Percent shorter	10%	14%	36%	52%	74%

percentile source brownout is shortened by over 2x. Besides shortening source brownout, delaying memory migration also relieves the guest of memory migration overhead (e.g. dirty-tracking page faults) for much longer.

3.3 Migration Versioning & Compatibility

In a complex stack, even serialization is not trivial. If the same code is running on both sides of a migration, serialization and deserialization are straightforward. In our stack, however, the source and target may be running substantially different versions of the code, or the device implementations on either side may be different.

To solve these issues, we use converters. Converters are independent objects used to change one version of serialized state into another. All serialized states are versioned; when a new version is created, it must include a converter from and a converter to the previous version. If the version of the saving and restoring objects do not match, the framework looks for a series of conversions that will generate a converted state for the desired version.

We want to be able to upgrade and downgrade our software without making any changes noticeable to the running guest. In some cases, a VMM must rollback to a version that is older than the one that started the guest. Introducing a new feature that affects serialized state is a two-step process. First,

a version of the code is introduced that can parse both the current version of state and the new version of state, without enabling the new feature yet. Once that code is completely deployed and can no longer be rolled back, the feature is deployed.

It is important to note that upgrades and rollbacks are a frequent part of our deployment process. The code that handles versions is exercised on a continuous basis and thus must be validated extensively. Figure 7 shows a timeline of deployment of our fleet over the year 2015, with the y axis being the percentage of the fleet that is running a particular version. Note that we frequently rollback a release. Stability of code that handles upgrades and downgrades is critical to our operational release cadence.

VMM deployments in Figure 7 were done using in-place upgrade, described in Section 2.2. The volume and performance of in-place upgrades are not included in the live migration numbers presented in this paper. The code used in managing version state is identical for both in-place upgrade and live migration.

4 Live Migration in a Production Environment

In this section we describe our system for verifying correctness as well as several performance improvements that reduce the impact of live migration on the guest. We outline the measured impact of these systems in Section 4.2.

4.1 Telemetry

One of the biggest challenges in live migration at scale is determining the impact, positive or negative, of changes. Tens or even hundreds of migrations in a validation environment may not suffice to show that the median post-copy time has declined, that every ten thousandth migration source brownout is excessively long, or that some set of user workloads lead to excessively long blackouts. We solved this by implementing an extensive telemetry stack for every migration in our production environment. We record events that provide detailed metrics for every migration performed. We have timestamps for every milestone during a migration along with milestone-specific statistics that allow further insights; for example, the “blackout completed” event records the length of time each component contributed to the blackout. This system allows us to track migration performance over time, quickly detect regressions in our validation environment during our release process, investigate regressions in releases, and study in detail any migrations related to user inquiries. One immediate payoff when we initially deployed this recording was that our telemetry exposed the long tail of memory blackout times, which led us to prioritize enabling post-copy migration.

Figure 7. VMM deployed versions (2015): Each color represents a different build of the VMM. The vertical axis is the fraction of the fleet on a given version. Places where a new color looks like a spike instead of a stripe are attempted rollouts that were later rolled back.

Figure 8. Median weekly blackout durations for several common instance types. Major improvements were postponing checksums in September 2015 (see Section 4.3) and code optimizations in October 2015, February and August 2016 (for example, we identified and eliminated excess copying of objects during state version checks for memory regions, which significantly improved blackout for large memory VMs).

4.2 Performance & Improvements

Our fleet-wide median blackout latencies are 50ms as of April 2017. They vary by machine shape—primarily memory size—as well as guest workload. Figure 8 shows blackouts for three sample machine shapes. Blackout times for other machine shapes are provided in an April 25, 2016 blog post on the Google Cloud Platform Blog [19].

Figure 8 shows the effects of a number of improvements that we have deployed. Three of the major ones were moving checksums out of the critical path, which is described in Section 4.3; optimizing some of the code used for tracking dirty memory; and allowing extra host CPU usage during migration, which is described in Section 4.4. Our median blackouts are now very similar across instances with vastly different memory sizes (4GB, 30GB, and 104GB), suggesting that we have eliminated most blackout contributions that scale with memory size.

Basic Blackout Interactions

Figure 9. Blackout profile, VM with 8 vCPUs and 30GB of RAM

We profile all migrations. Figure 9 shows the median time taken for each step of blackout in our production environment.

4.3 Memory Checksums

Correct memory migration relies on tracking changes to guest memory, which is a hard and subtle problem. Virtual devices failing to respect VM-pause, untracked hardware DMAs, untracked host-side changes to guest memory, and KVM dirty-tracking bugs could all result in incorrect tracking. At scale, and with the wide range of workloads we run, we can expect most corner cases in the dirty-tracking system to be exercised. A bug in tracking would result in guest memory corruption when the VM resumes after blackout. As protection, we implemented a memory checksum system as part of live migration.

The checksum system runs entirely on the source side. Our network layer ensures data is not corrupted in transfer, and, because the target is not yet running the guest, unexpected memory changes are very unlikely there. When the source sends a page, it copies the page into a buffer to ensure that the data checksummed corresponds with the data sent—with

no possibility of the guest modifying it. The buffer is then checksummed and sent, and the source stores the checksum for later. We verify correctness after the guest is paused for blackout so that no further changes to memory occur. The guest memory is re-checksummed and the new checksums are compared to the checksums computed earlier during the transfer. If any checksums differ, then the current contents of source memory do not match what we sent to the target.

We performed this verification as the last step of blackout and resent corrupted memory if any was found. This system caught several bugs during initial testing of live migration and ensured that we never ran a guest with a corrupted memory state. However, the computation of the checksums inherently takes a long time, which directly resulted in long blackouts for large VMs. Additionally, this approach cannot fully cover post-copy, which transfers part of guest memory after blackout. In September 2015, after more than a year with no checksum failures, we postponed the verification pass until after the migration has resumed on the target and post-copy has completed. The source still performs the verification, but we now kill the VM if the check were to fail. This introduces some risk, but at this point we anticipate any new bugs will be caught in our testing or validation environment. As expected, postponing checksums improved blackout substantially for all instance types (Figure 8).

4.4 Guest Degradation

During brownout live migration adds CPU, RAM, and network costs to running the VM. In addition to minimizing the length of blackouts, we try to minimize any impact to guest performance during brownouts. We offset the brownout overheads in a couple of ways:

- We alleviate network contention at both the physical NIC on the hosts and at the top-of-rack switch by limiting the number of concurrent migrations per host and per switch.
- We offset CPU overheads by separating the threads doing migration work into their own container. If there is enough free CPU on the host, the migration container uses it and there is no problem. If there is contention, we use the containers to enforce a ratio of cycles which allows brownout progress while minimizing guest degradation.

4.5 Robustness & Testing

4.5.1 Fault Injection

A consequence of doing a very large number of migrations is that we can encounter even very unlikely error paths. We needed to ensure that we could also encounter them in our test environment. We built a migration component, called “Injecticus”, which can inject delays and faults at any point in a migration.

Injecticus is a special device, just for testing. It uses the notifications and holds described in Section 3.2.5 to inject faults and delays during specified phases of migrations. Injecticus accepts RPCs, which an outside test can send. These RPCs allow a test to introduce a fault from a list offered by Injecticus or to block a migration from continuing until the test responds to an RPC sent to the test by Injecticus.

These capabilities allow us to test timing-dependent scenarios consistently. Some examples include:

- Adding a persistent disk or local SSD to the virtual machine during source brownout or target brownout.
- Cancelling a migration at each possible phase of a migration.
- Rebooting a guest at each possible phase of a migration.
- Artificially inducing host networking errors during brownout or blackout to ensure that the guest workflow survives.
- Forcing the source VM to crash during target brownout.
- Forcing the target VM to crash during source brownout.

4.5.2 Qualification

Before we first enabled live migration for our users, we needed to validate its correctness and performance. Our goal was to catch failures that would occur as infrequently as one migration in one thousand; we did 50,000 migrations a week before our launch. We created VM workloads that stressed different subsystems and ran them while migrating in a loop.

Today, we perform about 30,000 migrations per week in our validation environment to qualify new releases. We look at the performance of a release candidate and validate the blackout and source brownout durations by looking at the median and 99th percentile values. Getting enough runs during validation is necessary to ensure that the tail latency has not regressed.

In our development environments, we perform 7,000 migrations per week. We have a suite of 60+ integration tests covering correctness and performance that we run continuously. Configurations vary by VM size, host platform, host kernel, and other test-specific parameters, making for over 1,500 configurations.

5 Related Work

We found that *Live Migration* by Clark *et al.* [2] did an excellent job at covering the fundamentals of VM live migration, including memory and some suggestions on local storage migration. We remark on how much of that work is relevant to our approach today. Certainly the biggest difference for us is that we do not use paravirtualization or expect any cooperation from the guest (our guests are customer workloads). It is also interesting that their blackout results (50-200ms across several workloads) are similar to our performance. We note that while our blackouts are similar, our VMs are much larger (compare our median blackout of 175ms for a 30GB guest vs their 200ms blackout for a 800MB guest). Some

of this is attributable to our cluster network having higher throughput [13].

Nelson *et al.* [12] anticipated many of the reasons a large environment would use live migration and the problems to be solved to effectively use it.

Ibrahim *et al.* [9] described improved heuristics for the completion of pre-copy memory migration. Hines *et al.* [8] described an implementation of post-copy memory migration. Mashtizadeh *et al.* [10] describe write mirroring for local storage migration.

Nathan *et al.* [11] provide a useful model for estimating pre-copy live migration of memory. Their assumptions do not match our environment in that our network throughput is not constant and our VMs may have sizable local SSDs that require migration.

Song *et al.* [14] detail numerous opportunities for optimization via parallelization. Our implementation uses many of these including pipelined and parallel reading, checksumming, and sending of memory regions while bounding the amount of memory overhead. Each SSD partition is migrated by a separate thread. Copying of CPU state during black-out remains single threaded. In addition, we utilize multiple network flows to maximize network bandwidth.

Commercial Live Migration—VMware has released a number of white papers describing what they have learned about live migration in practice, including for vSphere5 [17] and vSphere5.1 [18].

6 Conclusions & Future Work

Our novel contributions include performance results across an extremely wide range of real-world workflows and measurements of the impact of a number of performance optimizations described in the existing literature across this set of workflows. Additionally, we describe the details of testing strategies and safety measures needed to provide live migration in a production environment.

From our experience running large numbers of VMs for multiple years, we have determined that:

- In a cloud environment, live migration is a key technology for keeping the entire software stack updated, beyond just addressing hardware issues or load balancing.
- For the vast majority of workflows, optimized pre-copy provides reasonable blackout times and minimal disruption during brownout. However, post-copy is vital to reduce tail latencies for large VMs.
- Staggered start of source brownout is key for avoiding unnecessary work, network traffic, and guest degradation when migrating multiple components with very different migration durations.
- Well-chosen validation, telemetry, and data collection during the migration process are extremely important

elements of operating live migration smoothly for our users.

Our focus with VM live migration has been iterating on performance without compromising correctness and stability. Any potential new performance improvement must take into account engineering time, system complexity, and the chance to introduce bugs. There are more optimizations we want to investigate and likely implement, such as page skips [11]. Our initial implementation benefited greatly from our existing RPC system, which provided us with fast and efficient compression, authentication and encryption.

We continue to gain benefits from deeper integration with the cluster management stack [16], including the ability to more intelligently decide which VMs should be migrated and in which order. We expect VM placement for load balancing to be a rich area for improvements.

Quantifying live migration performance remains an open area for improvement. The metric most frequently used, black-out (or downtime), is well understood and we have been able to optimize for it. Other metrics, such as total migration duration, are also well understood, but so far have been of secondary importance. However, impact on the guest is harder to define and measure generically in our environment. We would like a set of metrics that capture degradation of CPU, memory access, and IO (including persistent disk, local SSD, and network), such that when a user has questions or suspects a workload experiences degradation, we can answer authoritatively using our metrics. This is currently an active area for development for us. Note that we cannot rely upon any in-guest metrics directly, since in general the virtualization layer has no special visibility into the guest; for privacy reasons, we will never look at guest state unless the user has explicitly given us permission.

7 Acknowledgments

Many teams at Google contributed to the implementation, launch and improvements of VM Live Migration including Virtual Machine and Environment, Instance Manager, Kernel, Compute Engine and Site Reliability Engineering (SRE), Borgmaster, Persistent Disk and Andromeda.

We would like to thank the anonymous VEE reviewers for their feedback.

We would like to thank our Google reviewers and approvers: Bill Thiede, Brent Welch, Eric Brewer, Jeremy Sugerman, Mike Dahlin and Scott Van Woudenberg.

We also received guidance and feedback on this paper from Jim Miller, John Wilkes, Rebecca Isaacs and Steve Hand.

References

- [1] W.-D. W. Bianca Schroeder, Eduardo Pinheiro. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS/Performance*, SIGMETRICS/Performance'09, Seattle, WA, USA, 2009. ACM.
- [2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc*

- of NSDI, 2005.
- [3] J. Corbet. `fincore()`. <https://lwn.net/Articles/371538/>, 2010.
 - [4] W.-D. W. Eduardo Pinheiro and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST'07, 2007.
 - [5] Google. Adding SSDs - Compute Engine - Google Cloud Platform. <https://cloud.google.com/compute/docs/disks/local-ssd>, 2016.
 - [6] Google. Storage Options - Compute Engine - Google Cloud Platform. <https://cloud.google.com/compute/docs/disks/#pdspecs>, 2016.
 - [7] Google. What is Google Compute Engine? - Compute Engine - Google Cloud Platform. <https://cloud.google.com/compute/docs/>, 2016.
 - [8] M. R. Hines, U. Deshpande, and K. Gopalan. Post-Copy Live Migration of Virtual Machines. volume 43, pages 14–26, New York, NY, USA, 2009. ACM.
 - [9] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized Pre-copy Live Migration for Memory Intensive Applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 40:1–40:11, New York, NY, USA, 2011. ACM.
 - [10] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The Design and Evolution of Live Storage Migration in VMware ESX. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.
 - [11] S. Nathan, U. Bellur, and P. Kulkarni. Towards a Comprehensive Performance Model of Virtual Machine Live Migration. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 288–301, New York, NY, USA, 2015. ACM.
 - [12] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.
 - [13] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hlzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network. In *Sigcomm '15*, 2015.
 - [14] Song, Shi, Liu, Yang, and Chen. Parallelizing Live Migration of Virtual Machines. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '13, pages 85–96, New York, NY, USA, 2013. ACM.
 - [15] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 2–12, New York, NY, USA, 1985. ACM.
 - [16] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
 - [17] VMware. vMotion Architecture, Performance, and Best Practices in VMware vSphere 5. <https://www.vmware.com/files/pdf/vmotion-perf-vsphere5.pdf>, 2011.
 - [18] VMware. VMware vSphere 5.1 vMotion Architecture, Performance, and Best Practices. <https://www.vmware.com/files/pdf/techpaper/VMware-vSphere51-vMotion-Perf.pdf>, 2012.
 - [19] S. V. Woudenberg. Lessons learned from a year of using live migration in production on Google Cloud. <https://cloudplatform.googleblog.com/2016/04/lessons-learned-from-a-year-of-using-live-migration-in-production-on-Google-Cloud.html>, 2016.