Live Gang Migration of Virtual Machines

Umesh Deshpande, Xiaoshuang Wang, and, Kartik Gopalan
Computer Science, Binghamton University
{udeshpa1,xwang2,kartik}@binghamton.edu

ABSTRACT

This paper addresses the problem of simultaneously migrating a group of co-located and live virtual machines (VMs), i.e, VMs executing on the same physical machine. We refer to such a mass simultaneous migration of active VMs as *live gang migration*. Cluster administrators may often need to perform live gang migration for load balancing, system maintenance, or power savings. Application performance requirements may dictate that the total migration time, network traffic overhead, and service downtime, be kept minimal when migrating multiple VMs. State-of-the-art live migration techniques optimize the migration of a single VM. In this paper, we optimize the simultaneous live migration of multiple co-located VMs. We present the design, implementation, and evaluation of a de-duplication based approach to perform concurrent live migration of co-located VMs. Our approach transmits memory content that is identical across VMs only once during migration to significantly reduce both the total migration time and network traffic. Using the QEMU/KVM platform, we detail a proof-of-concept prototype implementation of two types of de-duplication strategies (at page level and sub-page level) and a differential compression approach to exploit content similarity across VMs. Evaluations over Gigabit Ethernet with various types of VM workloads demonstrate that our prototype for live gang migration can achieve significant reductions in both network traffic and total migration time.

Categories and Subject Descriptors

D.4 [**Operating Systems**]:

General Terms

Design, Experimentation, Performance

Keywords

Virtual Machines, Operating Systems, Live Migration

HPDC'11, June 8–11, 2011, San Jose, California, USA.

1. INTRODUCTION

Live migration refers to the act of migrating a virtual machine (VM) from one physical host to another even as the VM continues to execute. Live migration helps to decouple the applications executing within the VM from the hardware on which they execute. Consequently, live migration has become a key ingredient behind a number of cluster management activities such as load balancing, failure recovery, and system maintenance.

In a cluster, VMs that communicate with each other or complement each others' services may be *co-located* on the same physical host to reduce communication costs [27, 10, 12, 31] and improve consolidation [29]. Co-located VMs may need to be migrated simultaneously for a number of reasons, such as when the physical host needs to be shut down for maintenance, to save energy by consolidating VMs to fewer number of hosts, or to perform rapid load balancing to accommodate sudden spikes in load and meet Service Level Agreements (SLAs) [4, 25, 21]. We refer to the simultaneous migration of multiple active VMs from one physical host to another as *live gang migration*.

The cost of live gang migration should be minimized to reduce its performance impact on the applications within VMs as well as on the network interconnect. This cost can be quantified by metrics such as total migration time, VM downtime, network traffic overhead, application degradation, and CPU/memory overhead.

State-of-the-art VM migration techniques [5, 9, 8] optimize the migration of a single VM. However these techniques alone are insufficient when tens of VMs need to be migrated simultaneously, possibly at a moment's notice. For example, migrating even one 16GB active VM (in the QEMU/KVM [13] platform) over an uncongested Gigabit network can take more than 2 minutes without any optimizations, and up to half a minute with simple optimizations such as compressing pages that have uniform data before transmission. Needless to say that migrating multiple VMs simultaneously can take several minutes of critical time during which the network becomes overloaded and the applications within VMs suffer from degraded performance.

In this paper, we present the design, implementation, and evaluation of a new approach for live gang migration that uses de-duplication to reduce the overhead of concurrently migrating multiple co-located VMs. The key observation is that co-located VMs may often have significant common content, such as the same operating system, applications, libraries, etc, resulting in significant amount of identical memory content across VMs. Traditional techniques for single VM migration, such as the pre-copy approach [5], would independently transfer each copy of the identical content in each VM. In contrast, live gang migration pro-actively tracks identical content *across co-located VMs* and transmits such content only once. De-duplication is performed using content-hashing at the granularity of both page and sub-page levels. We also apply differential compression to exploit content similarity across *nearly*-identical pages.

We implement and compare various optimizations in live gang migration; these include offline and online de-duplication at the page and sub-page granularities, re-hashing of dirtied pages during memory transfer, differential compression for nearly identical memory content, and offline and online compression of uniform pages (online being the standard optimization in QEMU/KVM). We implemented live gang migration by modifying an existing implementation of the pre-copy single-VM migration [18, 5, 13] in the QEMU/KVM environment. Our evaluations show that the offline de-duplication mode at page granularity provides the best reduction in total migration time and network overhead, followed by incremental improvements due to other optimizations.

The rest of this paper is organized as follows. Section 2 presents the design of our approach for live gang migration of VMs. Section 3 presents implementation-specific details in the QEMU/KVM environment. Section 4 presents a detailed performance evaluation. Section 5 surveys existing work related to VM migration and memory de-duplication. Section 6 concludes the paper.

2. DESIGN

The key observation behind live gang migration is that VMs having the same operating system, applications, or libraries can contain a significant amount of identical memory content. Hence the basic idea is to identify and track all the identical memory pages across co-located VMs, and transfer only a single copy of such identical pages. By extension, we can also identify those pages having largely similar content and transfer only the delta difference in content among pages.

Figure 1 shows the high-level architecture of live gang migration. A migration controller at the source node identifies and tracks the identical or similar pages among co-located VMs. When it is time to migrate the VMs, the controller at the source node initiates the concurrent migration of all co-located VMs to the target host. Similarly, the controller at the target host prepares the host for the reception of all VMs.

Note that the existing implementation of single VM live migration in the QEMU/KVM platform [13] transmits a compressed version of pages that contain uniform data. We refer to such pages as *uniform pages*. For example, if a page contains all zeros, then a single zero byte is transmitted instead of the entire page to reduce the amount of data sent over the network. We found that most of these uniform pages have duplicate instances in a VM and also that most uniform pages contain all zeros. While the above simple compression can reduce network traffic, repeatedly identifying and compressing multiple instances of uniform pages can consume significant amount of CPU cycles during live migration. In contrast, our approach of de-duplicating identical content across VMs obviates the need for compression.

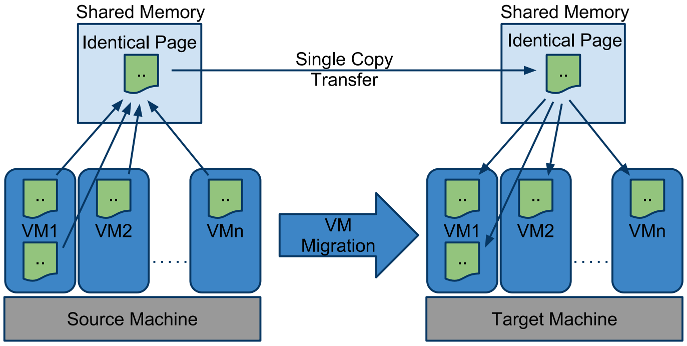

Figure 1: Architecture of live gang migration using offline de-duplication.

2.1 Performance Metrics

Total Migration Time is the time taken from the start of the first VM's migration at the source to the end of the last VM's migration at the destination. Note that live gang migration migrates all VMs concurrently, not one after another. Hence the total migration time of live gang migration may be smaller than the sum of individual VMs' total migration times.

Percent Network Traffic reduction is the amount of data transfer saved by either compressing or de-duplicating pages during migration.

Downtime is defined as the duration for which a VM's CPU execution is fully suspended during live migration. During downtime, the VM's CPU execution state and its pages in the Writable Working Set (WWS) are transferred to the target host.

Application Degradation is the extent to which the operation of live migration slows down the applications running within the migrating VMs.

2.2 Phases of Live Gang Migration

Live gang migration has two phases, namely *preparation* and *migration*.

Preparation: In this phase, the controller periodically scans the memory of co-located VMs to identify identical memory pages using content hashing [26]. A hash table is used to store a copy of each identical page discovered across VMs during scans. The first scan examines every memory page in each VM and the subsequent scans examine only the pages that are dirtied (or written to) since the last scan. For each page examined, the controller computes a hash value over the contents of the page. If the hash value matches that of a page already in the hash table, then the controller performs a byte-by-byte comparison to rule out the possibility of hash collisions.

Migration: During the migration phase all co-located VMs at the source are concurrently migrated to the destination over independent connections. The entire content of an identical page is sent the first time that the page is transmitted with any VM. For subsequent transfers of the identical page, only an identifier is transmitted to locate the page contents at the destination. Pages received at the des-

```
N is the total number of pages in a VM;
page[N] is the array of all pages in a VM;
identical[N] records if a page is identical to others;

Begin Migrate (VM)
   for i := 1 to N do
      send identifier of page[i];
      if identical[i] == 1 then
         if contents of page[i] are not yet sent then
            send page[i];
         endif
      else /* unique page */
         send page[i];
      endif
   endfor
End Migrate

Begin Receive (VM)
   for i := 1 to N do
      receive identifier for page[i];
      if identical[i] == 0 then /* unique page */
         receive content for page[i] from network;
         copy content into page[i];
      else /* identical page */
         if page[i] contents already received then
            retrieve the previously received content
               of page[i] from hash table using identifier;
            copy the content into page[i];
         else /*  contents of page[i] not yet received */
            receive the content for page[i] from network;
            copy the content into page[i];
            copy the content into hash table for
              de-duplicating subsequent identical pages;
         endif
      endif
   endfor
End Receive
```

Figure 2: Pseudo-code executed at the source and the target host for each VM.

tination, either entirely or via identifiers, are copied into the memory space of the corresponding VM. Figure 2 shows a simplified algorithm of actions taken at the source and the destination nodes. The source node executes the `Migrate` routine for each VM and the destination node executes the `Receive` routine for each VM. Transfer of each page consists of a page identifier optionally followed by the page content. The identifier could be either the page offset alone in the case of unique pages, or the page offset plus the content hash in the case of identical pages.

2.3 Re-hashing of Dirtied Pages

Once scanned, a page can be subsequently dirtied (or written to) by the VM. Dirtying changes the page contents and can either invalidate earlier identical matches or create new matching opportunities. As an example of the latter case, a memory scrubber might zero out de-allocated pages. The controller, with the help of the hypervisor, tracks all pages that are dirtied since the last scan of identical pages. Dirtied pages can be re-hashed at two different times – during the periodic memory scans and during the migration of a VM. We refer to the former as *periodic re-hashing* and the latter as *online re-hashing.* If no re-hashing mode is enabled then the dirty pages are transmitted in their entirety to the target during the migration phase.

HashSimilarity Detector	% Network Traffic Reduction	Total Migration Time (sec)
2, 1, 1	78.10	11.42
1, 1, 1	77.48	9.49

Table 1: Comparisons of different modes of HashSimilarityDetector.

2.4 Page Similarity Detection

Co-located VMs that run heterogeneous applications may have fewer pages with identical content. However, even unique pages from different VMs can contain partially similar data. We exploit this similarity to reduce the transfer of redundant data by calculating the difference between similar pages. To calculate similarity we use the HashSimilarityDetector(k, s, c)[7]. This algorithm calculates hashes from (k * s) randomly chosen 64-byte blocks of a page. These hashes are divided into k groups, each with s hashes. Each such page is indexed into a hash table at k locations. Each hash bucket can contain a maximum of c candidates for a reference page, against which the difference can be calculated. This difference is referred to as the *delta.* The reference page yielding a smaller delta is chosen for de-duplication. The hash table used for finding similar pages is distinct from the one used for tracking the identical pages, and we refer to it as the *similarity hash table.*

We empirically observed that the HashSimilarityDetector(2, 1, 1) provides the best de-duplication performance, but results in a significant increase in total migration time due to more expensive hash calculation. On the other hand, the HashSimilarityDetector(1, 1, 1) gives a moderate de-duplication performance at the cost of a slight increase in the total migration time. Table 1 shows the comparison of various modes of HashSimilarityDetector when accompanied with offline de-duplication and re-hashing for the migration of 4 VMs running mixed workloads.

To populate the similarity hash table, we choose the reference pages from the collection of identical pages contained in the hash table. During the migration of a VM, unique pages are compared against the reference pages from the similarity hash table to calculate the delta. At the target, deltas are applied to the reference page to reconstruct the original page.

Only the deltas of size less than 75% of a page are transferred to the destination, otherwise the entire page is transferred. This reduces the processing overhead incurred due to the decoding of the large deltas at the target.

3. IMPLEMENTATION

We implemented a prototype of offline de-duplication based gang migration in QEMU/KVM [13] environment, using Linux 2.6.32 and qemu-kvm-0.12.3 on source and target machines. The offline de-duplication implementation consists of around 2800 lines of code. It is completely transparent to the users of VMs and doesn't require any changes to the host kernel. Our implementation only modifies the QEMU/KVM code, which is a modified form of QEMU [3] to support KVM. QEMU/KVM runs as a combination of a per-VM user process and a kernel module on the host

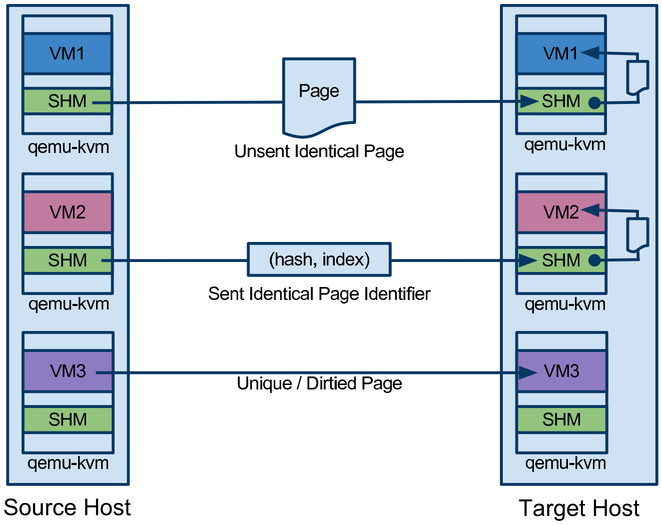

Figure 3: Types of pages transferred during live gang migration in offline de-duplication mode.

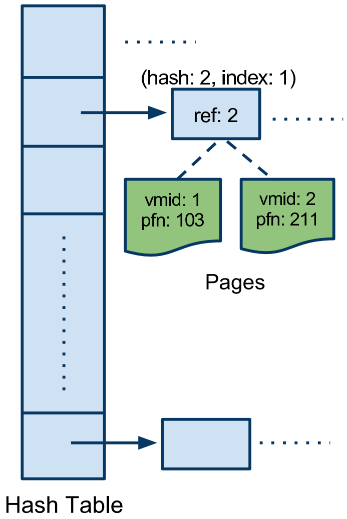

Figure 4: Hash table in the shared memory containing identical pages.

machine. The per-VM user process exports the part of its virtual memory to the VM, which the VM sees as its physical memory. As mentioned earlier, live migration in QEMU/KVM also performs a rudimentary compression of pages that have uniform content. Our implementation of the migration phase in live gang migration replaces this compression phase with the code to eliminate duplicate transfers. The following paragraphs discuss the implementation of the various components of offline de-duplication mode of live gang migration.

3.1 Controller

The controller coordinates the memory scan and the migration of co-located VMs. It is set up on the source and the target before the VMs are started. The controller sets up a shared memory region to track identical pages and commands the QEMU process associated with each VM to initiate the memory scans for the VM. A separate thread is initiated within each QEMU process to scan each VM's memory. In order to reduce the performance impact of memory scans on the VMs themselves, each thread runs with a lower priority than the regular processes in order to use idle CPU cycles.

3.2 Hash Table

We implement the hash table in the shared memory set up at the source and the target. Figure 3 shows the shared memory mapped in the virtual address space of the process qemu-kvm. We use SuperFastHash [1] to calculate a hash for every page during the VM's memory scan. This hash value is used as a key to insert the page into the hash table. A byte-by-byte comparison is performed between the scanned page and every other page in the same hash bucket. The reference count of the existing page is increased on a match, otherwise, a new entry containing the data from the scanned pages is created. The position of the page in the hash bucket is called as its *index*. $(hash, index)$ pairs identify unique pages in the hash table. Figure 4 shows a page in hash bucket number two representing two pages from two different VMs. The hash table contains 2 million buckets protected by a set of 200 semaphores to allow simultaneous insertion and lookup of pages from multiple processes without compromising its consistency.

3.3 Source End

At the source, each VM page is categorized into one of four types before being transmitted. This page category information page is sent to the target along with the page. (1) *Unique Pages*: These pages are not identical to any other page and their contents must be transferred during migration. (2) *Dirtied Identical Pages*: These pages were identified as identical to others during the preparation phase, but have been dirtied by the VM since then. The controller could choose to either re-hash the contents of the dirtied page or avoid the re-hashing overhead by treating all dirtied pages as unique and transfer the entire contents of such pages. We evaluate both options in Section 4. (3) *Unsent Identical Pages*: These are identical pages have not been dirtied since last scan and not yet been transferred to the target. Again, the entire page content is transferred and accompanied by the $(hash, index)$ identifier so that the page can be uniquely identified and stored by the controller at the target. (4) *Sent Identical Pages*: These are identical non-dirtied pages that have already been transferred to the target. For such pages we transfer only the $(hash, index)$ identifier to locate the corresponding page at the target and omit transferring the page content.

3.4 Target End

Upon initiating gang migration, the source host lets the target side know the size of the VM. The target side allocates a memory region of the same size, and copies the received pages to the allocated memory according to their address. However, for offline de-duplication based gang migration each received page is treated differently according to its page category described above. *Unique Pages* are received by the target and copied to the memory allocated to the migrated VM. Pages classified as *Unsent Identical Pages* on the source are also copied to the shared memory set up at the target, in addition to being copied to VM's memory. *Sent Identical Pages* are accompanied with an identifier, which is used to locate the corresponding page content from the hash table in the shared memory. Before resuming the VM, the target end ensures that all the VM memory pages are received and mapped into its address space.

3.5 Cache

Since the shared memory is simultaneously accessed by multiple VMs, its access is protected by semaphores to maintain the consistency of the hash table. However, the processes visiting the same bucket of the hash table must wait until the first process has released the semaphore. To avoid this resource contention at the bucket level, we implement a cache containing pointers to recently accessed pages. The qemu-kvm process corresponding to each VM accesses hash table for (1) marking a page as sent on the source side (2) adding a page to the hash table while scanning VMs' memory (3) removing unique pages having no other matching page (4) copying a page from the hash table to the VM's memory region at the target side. The cache for the hash table contains the data structures corresponding to recently accessed pages, consequently avoiding the need to visit the shared memory to copy a page, or to check if it has already been sent. However, a semaphore needs to be acquired to add, or remove the pages from the hash table. The Least Recently Used (LRU) cache replacement algorithm is used to replace entries when the cache is full. We observed that at any point in time, irrespective of VM workloads, a VM's memory contains significant number of zero pages. Hence we consider the zero page as a permanent member of the cache, and do not replace it by any other page even if the cache is full. The cache allows live gang migration to scale to tens of VMs without a significant performance penalty.

3.6 Delayed Pages

Since de-duplicated pages are transferred over independent concurrent connections, their sequence cannot be guaranteed. Therefore, an identifier indicating the location of the sent page can reach the target before the page itself is available. These pages are referred to as *delayed pages*. In such a scenario, a lookup for a page in the target hash table with its identifier can fail. To handle this problem, we implement a *waiting list* at the target to contain addresses and identifiers of pages not yet received by the target at the time of the hash table lookup. We traverse the waiting list periodically during a VMs' migration to locate the delayed pages from the target hash table and copy them into the VMs' memory. Since a VM cannot be resumed before locating all of its pages, waiting list traversal may add to the downtime of a VM in case the waiting list is non-empty at the end of pre-copy phase. However, out of sequence reception of an identifier and the corresponding page is a rare event. Consequently, the waiting list traversal overhead is observed to be not more than 1–5 milliseconds.

Finally, while traversing the waiting list, one cannot simply copy all delayed pages from the target hash table into the VM's address space. If a VM receives a new modified copy of the delayed page, then the corresponding page from the hash table becomes obsolete. Hence, upon reception of the latest copy of the delayed page, the corresponding entry in the waiting list pointing to the delayed page is deleted.

3.7 Re-hashing of Dirtied Pages

To perform re-hashing, dirtied pages need to be identified. The KVM hypervisor provides a facility for tracking dirtied pages through a dirty bitmap maintained in the host kernel, which is essentially one bit per page indicating whether the page was modified since the last scan. Before each rehashing round (either periodic or online) the VM's dirty bitmap is retrieved and the hypervisor is instructed restart logging the dirtied pages (for the next re-hashing round). For every dirtied page, we recalculate the hash value and re-inserts the page into the hash table. Online re-hashing is performed by the VM's main migration thread during VM's migration whereas periodic re-hashing is performed by the low-priority scanning thread. Tracking dirty pages could be expensive because the hypervisor essentially needs to first temporarily mark all the pages of VM as read-only, trap the first write access to each page, mark the page in the dirty bitmap, and finally reset the page access permissions to the original permissions. If the workload within the VM is write-intensive, then frequent re-hashing can result in application slowdown. Alternatively, one could choose not to re-hash dirtied pages and simply transmit them as unique pages during the migration phase, thus potentially foregoing any potential savings from de-duplicating dirtied pages.

3.8 Sub-page Level De-duplication

The intuition behind the implementation of sub-page level de-duplication is that, even if an entire 4KB page doesn't have an identical counterpart, its sub-part can match with the sub-parts of other pages. Sub-page level de-duplication is implemented by dividing each page into a fixed number of parts and calculating a hash for each part separately. Subpages are treated in the same way as full pages during the migration. However, if a page is detected as dirty, then all of its sub-parts are re-sent to the destination, because the current dirty logging mechanism doesn't provide a facility of detecting dirty sub-parts of a page. Since the hash is calculated for every sub-part of the page, reduction in the size of the sub-page has negative impact on the time required to complete the preparation phase.

3.9 Delta Transfer

Before initiating the migration, we populate the similarity hash table by traversing the hash table containing identical pages. To minimize the storage overhead, instead of storing a complete page, we only store the hash identifier of a page into the similarity hash table. Each hash bucket contains an identifier of only one reference page. During the migration, every unique page is hashed using the SuperFastHash [1], and a page from the similarity hash table having the same hash fingerprint is chosen as its reference page. We use Xdelta [14], a tool for differential compression, to calculate the difference between the unique and the reference page. This delta is transferred to the target machine along with the hash identifier of its reference page. At the target, the decode routine of Xdelta reconstructs the original page by applying the delta to the reference page. Even though the delta is transferred to the target after the reference page, their reception sequence at the target cannot be guaranteed. Since the reference page and the delta can possibly be transferred on independent connections, target machine might receive the delta before its reference page arrives. In such a scenario, the reference page is treated as a *delayed page*, and its deltas are inserted into the waiting list setup at the target. During the traversal of the waiting list, all the deltas are applied to the reference pages to reconstruct the original pages.

3.10 Online De-duplication

This mode of gang migration de-duplicates pages while the VM migration is in progress. Therefore, its preparation phase is much shorter than the offline mode and just involves setting up the shared memory and the hash table. Before sending each page over the network, it is checked against the hash table to test if an identical page has already been sent. The rest of the migration takes place as explained in section 3.3. Since online de-duplication incurs the cost of calculating the hash and byte-by-byte comparison of pages, performance of VM migration with online de-duplication is unacceptably low. Therefore we do not consider it for further evaluation.

3.11 Scalability

Scalability of live gang migration can be defined as its ability to migrate increasing number of VMs without degrading its performance, while keeping the hash table memory space requirement as low as possible. Current implementation of gang migration allows it to scale to tens of VMs with minimal impact on its performance. The semaphore protected hash table becomes a point of contention when the number of VMs is increased. Our implementation of live gang migration obviates the need to access the hash table for most of the read operations by maintaining a per VM cache for the hash table data structure. As far as the hash table memory space requirement is concerned, currently gang migration requires memory size enough to contain a single copy of the pages having multiple instances across the VMs. For migrating 24, 1GB memory idle VMs, offline de-duplication uses approximately 256MB of memory.

3.12 Security

Our current implementation of gang migration assumes that the VMs are run in a data center environment, where the host environment on the physical machine is trusted and under the control of the cluster administrator. Since, the shared memory region containing the hash table is separate from the physical memory region of the guest operating system, the VM's kernel and applications cannot access it. The shared memory is only accessible from the protected QEMU mode of the host machine. Hence the possibility of one VM maliciously accessing/inferring another VM's memory contents does not exist in our prototype.

4. EVALUATION

Here we compare the performance of different modes of gang migration against the original QEMU/KVM live migration mechanism. The key performance metrics are total migration time (TMT), the amount of network traffic reduction, and downtime. Our experimental setup consists of two machines with dual quad core Intel Xeon processors and 70GB DRAM, running Ubuntu 9.10. We use the same OS distribution for the VMs.

We carry out the evaluation of live gang migration for the following four modes.

Without Optimizations: This configuration is the simple pre-copy migration of VMs without any type of compression or de-duplication of pages.

Online Compression: This is the default mode of VM migration currently used by QEMU/KVM. This mode compresses pages having uniform content, such as zero pages, during a VM's migration.

Offline Compression: This mode is a variant of online compression in which compression is carried out before initiating the VM migration. A bitmap is used to mark the compressed pages. The qemu-kvm process refers to this bitmap during a VMs' migration to decide if the page is already compressed or if it needs to send the entire page.

Offline De-duplication (DD): This mode de-duplicates the pages across co-located VMs before initiating their migration. For evaluation, we use online re-hashing in conjunction with offline de-duplication as an optimization. In the rest of this section, we simply refer to online re-hashing as re-hashing.

We now describe the various evaluations performed by migrating VMs between two hosts connected by a Gigabit Ethernet. We demonstrate through the migration of single and multiple VMs that offline de-duplication provides an additional 15% to 18% network traffic reduction over the online compression mode, which is the default mode used by QEMU/KVM. We show that the preparation phase imposes less than 6% overhead on the applications running inside the VMs. We also show that de-duplication also reduces the amount of data transferred during the downtime, effectively shortening its duration.

4.1 Network Bandwidth Utilization

Figure 5 shows the network bandwidth utilization of the four modes of gang migration. The graph demonstrates the network bandwidth measured with `tcpdump` [23] on the source side during the migration of a single 8GB idle VM over Ethernet interconnect. Note that, the line representing bandwidth for the "without optimization" mode has been truncated to save space. The VM migration with no optimizations continues for more than 70 seconds. The bandwidth drop during migration in the other three modes can be attributed to the following factors. In online compression, the compression and de-compression of zero pages is CPU-intensive. In offline compression, de-compression of zero pages, can be CPU intensive. In offline de-duplication, checking the sent flags of pages at the source and insertion and copying of pages at the destination introduces additional overhead. The following discussion further examines the implications of these overheads.

4.2 Single VM Migration

For a single VM migration, we migrate an idle VM between two physical hosts to measure the total migration time and the network traffic reduction as a function of the VM's memory footprint. Figure 6 compares the four modes of gang migration in terms of their total migration times.

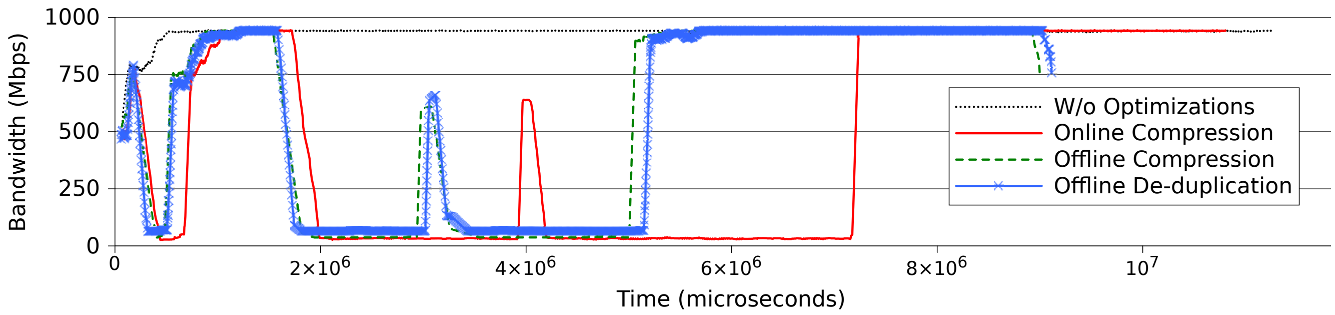

Figure 5: Network bandwidth utilization for the migration of a single VM.

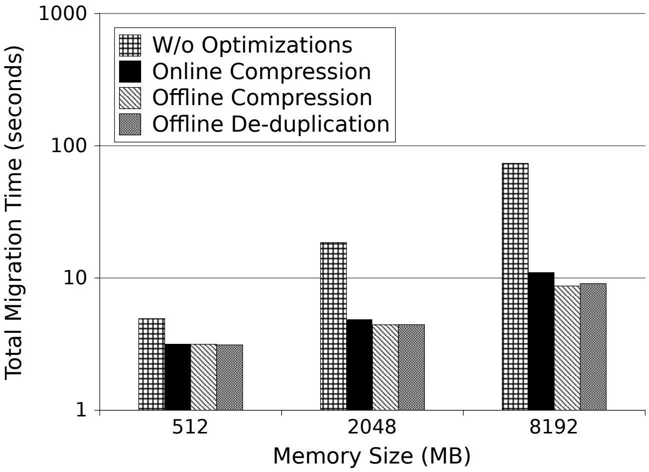

Figure 6: Total migration time vs. VM size.

Observe that any kind of compression or de-duplication significantly improves the performance of migration. In offline modes, compression is slightly faster than de-duplication due to the overhead of hash table lookup associated with de-duplication. In spite of its associated overhead, for an 8GB VM, 17.7% reduction in the total migration time is observed with offline de-duplication compared to the default migration mode, i.e. online compression. Furthermore, when migrating multiple VMs, we can amortize this small penalty of de-duplication and achieve a much higher reduction in the total migration time when compared to offline compression.

From Figure 7, the network traffic reduction achieved by compression and de-duplication is almost same for a single VM migration. This is because the VM's memory is mostly filled with zero pages, and for each zero page de-duplication or compression transmit small amounts of data over the network. For compression one byte is transferred to indicate the existence of a zero page, whereas for de-duplication an eight byte identifier pointing to the already transferred zero page is sent to the destination. For an idle VM, network traffic reduction due to de-duplication of non-zero pages is insignificant. The negative network traffic reduction with the non-optimized mode is due to the extra information sent along with the page, such as its address, other control messages, etc.

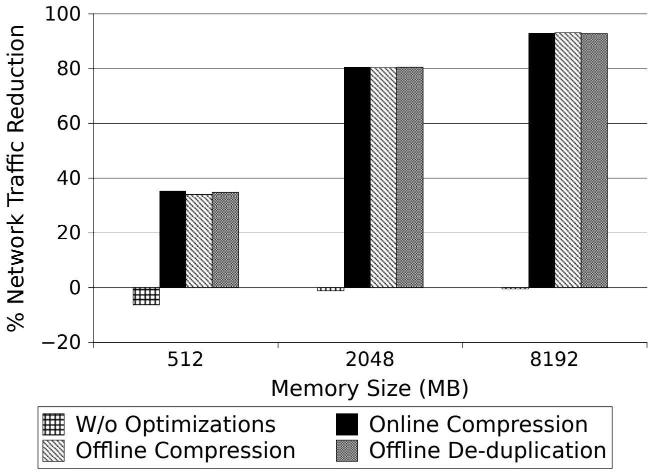

Figure 7: Percentage network traffic reduction for varying VM memory size. Baseline is the sum of memory allocated to all VMs being migrated.

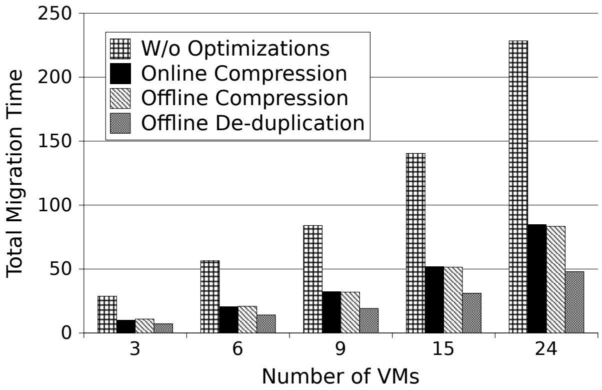

Figure 8: Total migration time for multiple VMs.

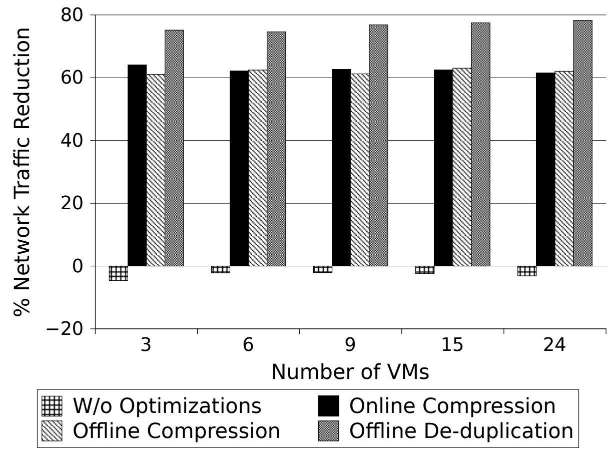

Figure 9: Percentage network traffic reduction for multiple VMs. Baseline is the sum of memory allocated to all VMs being migrated.

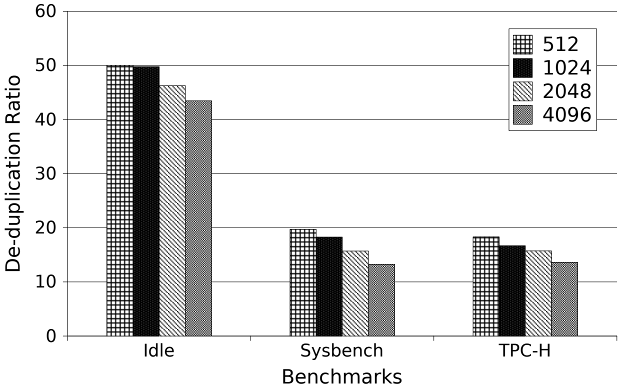

Figure 10: De-duplication ratio for different sizes of sub-page.

4.3 Live Gang Migration of VMs

In this section we evaluate the migration of multiple idle VMs between two hosts to measure the total migration time and the percent network traffic reduction. We keep the physical memory size of all VMs constant at 1GB, while increasing the number of VMs. Figure 8 compares the four modes of gang migration by measuring their total migration time. Compression of uniform pages doesn't prove as helpful as de-duplication to reduce total migration time of VMs. Offline de-duplication of pages shows an improvement of almost 45% for 24 VMs over online compression, which is the default mode used by QEMU/KVM. Online and offline mode of compression perform almost equally in terms of total migration time because the speed of migration for multiple VMs is limited by the bandwidth of the network, hence directly proportional to the amount of data sent.

Figure 9 compares the network traffic reduction of the four modes of gang migration. It can be observed that deduplication achieves almost 75% of traffic reduction against 60% to 65% with page compression. The difference between de-duplication and compression modes can be attributed to de-duplication of non-zero pages.

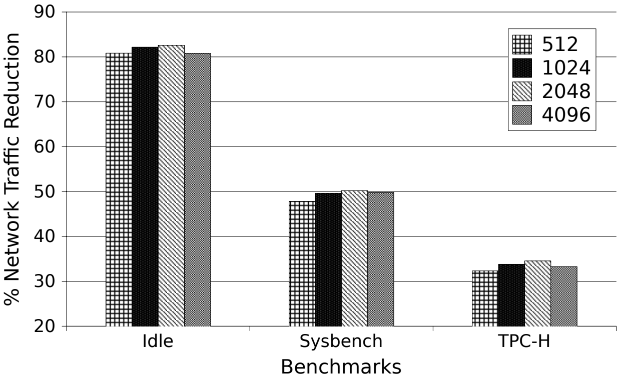

Figure 11: Percent network traffic reduction for different sizes of sub-page.

4.4 Sub-page Level De-duplication

In this section, we measure the effect of sub-page level de-duplication on the performance of live gang migration of VMs. We load the memories of 8 VMs, each of size 1.25GB, with the various application datasets and migrate them simultaneously in offline de-duplication mode while varying the size of the sub-pages.

Figure 10 compares the de-duplication ratio across the applications for the various sizes of sub-page. This is a ratio of the number of page instances de-duplicated to the number actual pages representing those instances. It can be seen that for decreasing sub-page size, at each step, we can get approximately 2 units of increase in the de-duplication ratio. However, same trend is not observed in the network traffic reduction. This is due to the fact that for every deduplicated sub-page, an 8 byte hash identifier and an 8 bytes address is sent to the target. From Figure 11, we can state that 2KB is the best size of a sub-page that delivers the highest network traffic reduction.

4.5 Application Evaluation

This section evaluates the performance of live gang migration on VMs executing either a database workload or mixed workloads.

4.5.1 Database Benchmarks

For the database benchmarks, we use the MySQL-Cluster [17] that allows us to generate a distributed database across multiple machines that are referred to as data nodes. We use 8 co-located VMs, each with a memory of size 1.25GB to serve as data nodes that store the in-memory database. Then, we query the database from a remote machine, in parallel to the migration of VMs containing the database, to measure the impact of the migration on the performance of the query. Such a test scenario can occur in data centers during off-hours when the VMs hosting the databases might be migrated to fewer physical machines.

The following two database benchmarks are used to evaluate the application degradation. (1) Sysbench [22] is an online transaction processing benchmark. We run it with the MySQL-Cluster set on top of co-located VMs. Sysbench database consists of a table containing 4 million entries. We perform 1000 transactions on the sysbench database during

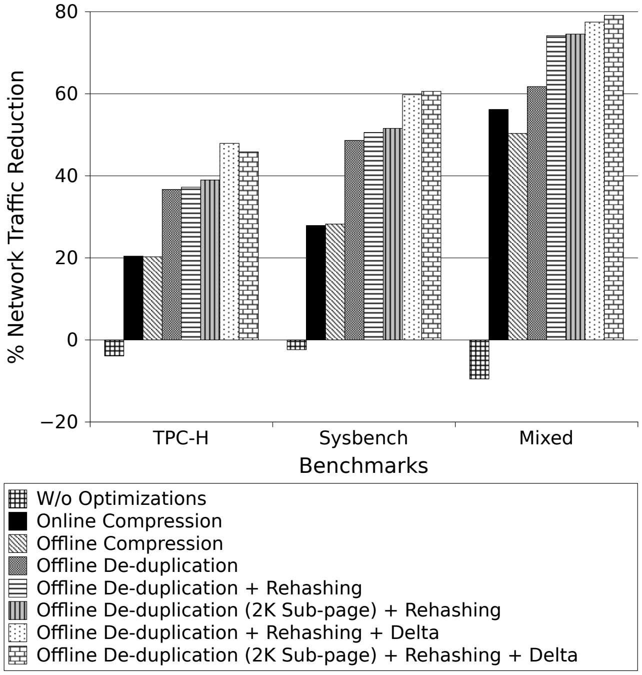

Figure 12: Network traffic reduction with applications workloads. Baseline is the sum of memory allocated to all the VMs being migrated.

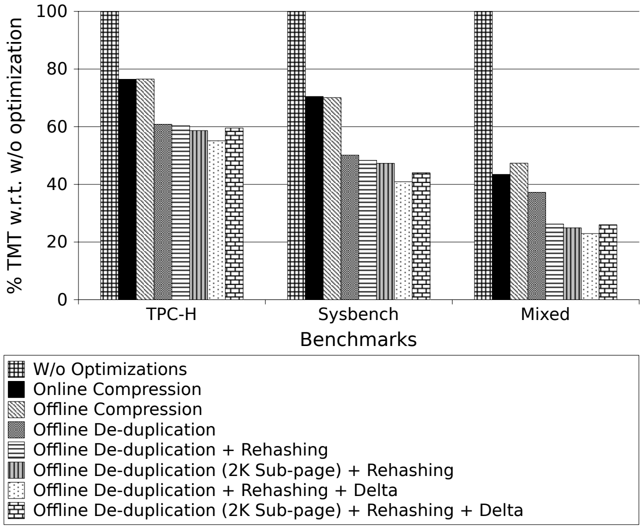

Figure 13: Total migration time with active workloads.

Applications	W/o Migration	W/o Optimi -zations	OC	OFC	OFD	OFD + Re-hashing	OFD + Re-hashing + Delta	OFD + Re-hashing + Delta + 2k-Subpage
Sysbench (trans/sec)	33.99	9	11.2	12.58	13.84	14.17	17.64	13.25
TPC-H (sec)	23.17	110.71	90.23	90.42	75.80	74.56	63.9	81.08
Sparse MM (sec)	27.25	31.52	28.27	28.47	29.6	27.89	27.61	29.52

OC = Online Compression, OFC = Offline Compression, OFD = Offline De-duplication

Table 2: Comparisons of different application workloads.

VMs' migration to measure the performance in terms of the number of transaction per second. (2) The TPC Benchmark H (TPC-H) [24] is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries. We set up 2GB TPC-H database with the help of MySQL-Cluster with VMs as data nodes. Then we run a query against the database during the migration of data nodes, to measure the time to complete the query.

4.5.2 Mixed Workload

To evaluate the gain from the differential compression technique in addition to the regular de-duplication optimization, we use the mixed workloads proposed in VMmark [15]. We migrate four 1GB memory VMs running the following four workloads to observe the total migration time and the network traffic reduction. (1) httperf running on a separate machine to request static pages from the VM running the Apache server. (2) Sysbench OLTP benchmark issuing 1000 request to sysbench database located in the VM. (3) A VM running Dbench [6] with 10 clients. (4) A VM carrying out sparse matrix multiplication of two matrices of size 5000 x 5000 to generate the output matrix in the VM's memory.

For these applications, the major part of their memory is filled with non-zero data pages, containing either a database or a writable working set of the applications, and the VMs are migrated while modifications to the parts of their memory are being carried out. Figure 12 shows the network traffic reduction with the applications described above. Any type of de-duplication mode yields more network traffic reduction than compression modes due to the absence of uniform data pages. Further, re-hashing accompanied with offline de-duplication has a clear benefit when it comes to slightly write intensive applications, such as the ones included in the mixed workloads. Combining re-hashing with sub-page level de-duplication at size 2KB proves beneficial because, for minor modifications to a page, only the modified half of the page retransmitted. From Figure 13, the lowest total migration time is achieved with a combination of similarity de-duplication and re-hashing at the 4KB page size. Even though the highest network traffic reduction is possible at the 2KB sub-page level de-duplication, the overhead of delta calculation and reference page lookup results in relatively higher total migration time than for 4KB page size.

Modes of GM	Downtime (ms)		
	Min	Max	Avg.
W/o Optimization	240.05	3746.18	1820.02
OC	165.26	2413.04	1075.20
OFC	143.06	2160.50	916.25
OFD	107.38	2920.2	606.20
OFD + Re-hashing	43.82	987.84	307.80

OC = Online Compression, OFC = Offline Compression, OFD = Offline De-duplication

Table 3: Downtime for the migration of 24 idle VMs.

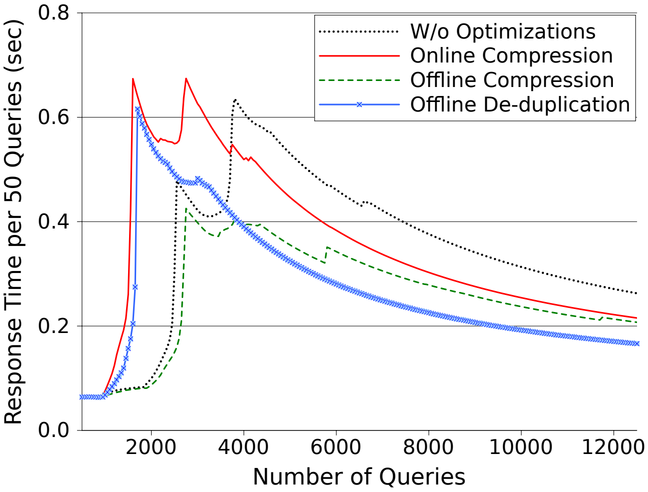

Figure 14: Sysbench query response time.

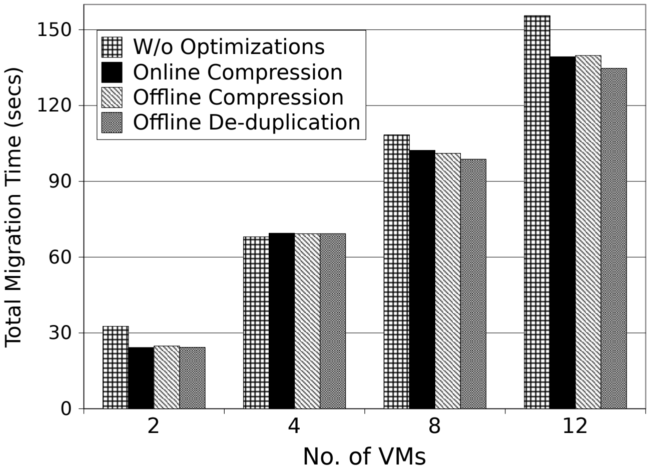

Figure 15: Evaluation of live gang migration with worst-case VM workload.

Modes of GM	% Overhead	
	CPU Intensive	Write Intensive
OFC	1	3.5
OFD	4.3	3.12
OFD + 2K Sub-page	3.5	5.7

OFC = Offline Compression, OFD = Offline De-duplication

Table 4: Overhead of the preparation phase of the various modes of gang migration on CPU and write-intensive VM workloads.

Table 2 shows the comparison of the various benchmarks for different modes of gang migration.

4.6 Downtime

Table 3 shows the comparison of downtime for the various modes of gang migration recorded for the migration of 24 1GB memory idle VMs. For fair comparison of all modes, we fix the number of pages sent during the downtime equal to the number of pages sent by the default migration mode, i.e. online compression. De-duplication of pages during the downtime allows offline de-duplication to migrate the VMs with lower downtime compared to all other modes of gang migration. High variation in the downtime of VMs is due to their parallel migration. As the first few VMs complete their migration, VMs completing towards the end of gang migration can use an additional share of network bandwidth, resulting in lower downtime for the latter.

4.7 Responsiveness of VMs during Migration

Figure 14 shows the cumulative moving average of the response time of queries against the progress of 1000 transactions on Sysbench database. Each transaction consists of multiple read/write queries. Migration is initiated after the completion of 500 queries. It can be seen that the Sysbench query response time suffers with both modes of compression. Higher response time is observed for longer duration of time. Even though VM migration in the offline compression mode doesn't involve CPU-bound operations at the source host, it decompresses a large number of pages in a short duration of time at the target. On the other hand, with offline de-duplication, degraded performance is observed for a shorter duration of time. Consequently, offline de-duplication provides the lowest average response time over the period of VM migration. This improvement can be attributed to the de-duplication of pages containing the Sysbench database.

4.8 Worst case VM workload evaluation

To evaluate the performance of the various modes of live gang migration for the worst-case VM workload, we use a write-intensive benchmark. This benchmark allocates 400MB of memory, and performs random write operations to fill the memory with random values. We vary the number of VMs to observe the effect of the workload on the total migration time. Such a write intensive application results in large number of dirty pages resulting in higher total migration time compared to the same number of idle VMs. From Figure 15, it can be noticed that for smaller number of VMs all the techniques except "W/o optimization" perform equally well. However, for a larger number of VMs, a slight improvement is observed with offline de-duplication over the compression techniques. This improvement can be attributed to the higher de-duplication ratio for the non-zero pages due to more number of VMs.

4.9 Evaluation of Preparation Phase

The preparation phase of each mode of gang migration incurs the following overheads compared to the normal operation of a VM. Online compression mode doesn't have a preparation phase as the compression is performed during the migration of VMs. In offline compression, uniform pages are compressed and dirty logging is enabled to track any page modifications. In offline de-duplication, hash is calculated for each page and inserted into the hash table. Dirty logging is enabled to track any page modifications. To measure the effect of the aforementioned overheads on the execution of VMs, we start eight 1GB VMs and run CPU and write-intensive benchmarks inside the VMs while the preparation phase is in progress.

Table 4 shows the comparison of overheads from the preparation phases of the various gang migration modes on CPU and write-intensive applications running inside the VMs compared to their normal operation. We use a Fibonacci series calculation as a CPU-intensive benchmark, whereas for the write-intensive workload we allocate 600MB of memory and write to it at random locations. We execute these benchmarks during the preparation phase of gang migration to measure the percent increase in the execution time when compared against their run without the preparation phase. Overall, we find that the overhead experienced by the two benchmarks is less than 6% in all cases. This is because the preparation phase is carried out by a low-priority thread using idle CPU cycles.

5. RELATED WORK

In this section, we review related research under three categories: (1) content-independent VM migration, (2) content-dependent VM migration, and (3) page-sharing and de-duplication approaches.

Content Independent VM Migration: These approaches reduce the number of pages sent during VM's migration without peeking into their contents. Post-copy [9] improves upon the pre-copy [18, 5] VM migration technique to transfer every page only once to the destination host. It also uses Dynamic Self Ballooning (DSB) to reduce the memory footprint of a VM before its migration. Live gang migration operates without modifying the core VM migration algorithm. Hence, it can be used in conjunction with pre-copy, post-copy, or any optimization of VM migration.

Content Dependent VM Migration: These approaches reduce the amount of data transferred during migration by either compressing pages or transferring a single copy of pages having identical content. These techniques optimize either live or non-live migration of a single VM and do not address the simultaneous live migration of multiple VMs. Sapuntzakis [20] et. al. propose the use of content hashing to avoid transfer of the data blocks already present at the target host as an optimization for non-live migration of a single VM and its disk image over a wide-area low bandwidth network. Similarly, Shrinker [19] optimizes VM migration over a WAN by avoiding the transmission of pages already present at the target machine. MDD [30] and CloudNet [28] also optimize the live migration of a single VM through the use of de-duplication and the transfer of differences between modified and original pages (as opposed to the transfer of the entire dirtied page). MECOM [11] compresses VM's pages during its migration. It varies the compression algorithm for every page according its cost of compression.

Page Sharing for Consolidation: Consolidation refers to maximizing the number of VMs on a physical host by reducing their resource requirements. Page sharing has been studied in the context of reducing the memory footprint of VMs to achieve consolidation [26, 7, 16, 2]. Difference Engine [7] discusses a way of sharing the memory across the VMs residing on the same machine. It also achieves memory saving by compressing and patching the pages. Satori [16] implements memory sharing mechanisms in the Xen environment by detecting opportunities for page sharing while reading data from a block device. Kernel Samepage Merging (KSM) [2] also allows Linux kernel to compare the memory of two already running programs. If any memory regions are exactly identical, the kernel can merge the two regions into one. KSM can be used in conjunction with QEMU/KVM to share identical regions of memory between multiple co-located VMs. Our proposal for live gang migration is the first to exploit page-sharing to improve the simultaneous migration of multiple VMs. For ease of evaluation, we implemented our own de-duplication mechanism, which enables us to selectively combine the de-duplication and compression optimizations for specific VMs. However, live gang migration can potentially be implemented to use any of the above mentioned de-duplication mechanisms.

6. CONCLUSIONS

We presented the design, implementation, and evaluation of live gang migration of virtual machines on the QEMU/KVM Linux platform. Our approach relies on de-duplication of identical memory content and differential compression of nearly identical content across VMs during migration. Offline de-duplication yields the maximum reduction in total migration time and network traffic overhead. Re-hashing further improves the performance by processing the pages that are dirtied during live migration. Sub-page level de-duplication further improves performance by exploiting the identical sub-parts of pages. We also track nearly-identical pages across co-located VMs and transfer only the difference between them during the migration. Detailed evaluations show that our techniques for live gang migration significantly reduce the total migration time required for simultaneous migration of co-located VMs and the amount of data transferred during their migration. The memory de-duplication techniques proposed in this paper can also be adapted to address simultaneous inter-rack migration of VMs for evacuating an entire rack of machines. This would require mechanisms to exchange the content hash of pages across the physical machines located on the same rack to keep track of identical pages.

Acknowledgement

We would like to thank our shepherd, Kenneth Yocum, for his invaluable suggestions that helped improve this paper. This work is supported in part by the National Science Foundation through grants CNS-0845832 and CNS-0855204.

7. REFERENCES

[1] Superfasthash www.azillionmonkeys.com/qed/hash.html.

[2] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proc. of Linux Symposium*, July 2009.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conference*, April 2005.

[4] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proc. of Integrated Network Management*, page 119–128, May 2007.

[5] 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 Network System Design and Implementation*, May 2005.

[6] Dbench. http://samba.org/ftp/tridge/dbench.

[7] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C Snoeren, G. Varghese, G. M Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. of Operating Systems Design and Implementation*, December 2010.

[8] J.G. Hansen and E. Jul. Self-migration of operating systems. In *Proc. of ACM SIGOPS European Workshop*, September 2004.

[9] M. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *SIGOPS Operating Syst. Review*, 43(3):14–26, July 2009.

[10] W. Huang, M. Koop, Q. Gao, , and D.K. Panda. Virtual machine aware communication libraries for high performance computing. In *Proc. of SuperComputing*, November 2007.

[11] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *Proc. of Cluster Computing and Workshops*, August 2009.

[12] K. Kim, C. Kim, S-I. Jung, H Shin, and J-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proc. of Virtual Execution Environments*, March 2008.

[13] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: The linux virtual machine monitor. In *Proc. of Linux Symposium*, June 2007.

[14] J. MacDonald. Xdelta http://www.xdelta.org/.

[15] V. Makhija, B. Herndon, P. Smith, L. Roderick, E. Zamost, and J. Anderson. VMmark: A scalable benchmark for virtualized systems. *Technical Report 2006-002*, 2002.

[16] G. Milos, D.G. Murray, S. Hand, and M.A. Fetterman. Satori: Enlightened page sharing. In *Proc. of USENIX Annual Technical Conference*, June 2009.

[17] MySQL. MySQL Cluster http://www.mysql.com/products/database/cluster.

[18] M. Nelson, B. H Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proc. of USENIX Annual Technical Conference*, April 2005.

[19] P. Riteau, C. Morin, and T. Priol. Shrinker: Efficient wide area live virtual machine migration using distributed content-based addressing. In *http://hal.inria.fr/inria-00454727/en/*, February 2009.

[20] C. P Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. of Operating Systems Design and Implementation*, December 2002.

[21] VMWare Distributed Resource Scheduler. http://www.vmware.com/products/vi/vc/drs.html.

[22] Sysbench. Sysbench benchmark http://sysbench.sourceforge.net/index.html.

[23] TCPDUMP. http://www.tcpdump.org.

[24] TPC-H Benchmark. http://www.tpc.org/tpch/.

[25] A. Verma, P. Ahuja, and A. Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *International Conference on Middleware*, December 2008.

[26] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proc. of Operating Systems Design and Implementation*, December 2002.

[27] J. Wang, K. L Wright, and K. Gopalan. XenLoop: a transparent high performance inter-vm network loopback. In *Proc. of High performance distributed computing*, June 2008.

[28] T. Wood, K. Ramakrishnan, J. van der Merwe, and P. Shenoy. CloudNet: a platform for optimized WAN migration of virtual machines. *University of Massachusetts Technical Report TR-2010*, 2, 2010.

[29] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *The International Journal of Computer and Telecommunications Networking*, 53(17), December 2009.

[30] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proc. of International Conference on Cluster Computing*, September 2010.

[31] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. Xensocket: a high-throughput interdomain transport for virtual machines. In *Proc. of International Conference on Middleware*, November 2007.