VMScatter: Migrate Virtual Machines to Many Hosts

Lei Cui Jianxin Li Bo Li
Jinpeng Huai Chunming Hu Tianyu Wo
Beihang University, Beijing, China
{cuilei, lijx, libo, hucm, woty}@act.buaa.edu.cn,
huaijp@buaa.edu.cn

Hussain Al-Aqrabi
Lu Liu
University of Derby, Derby, United Kingdom
{H.Al-Aqrabi, L.Liu}@derby.ac.uk

Abstract

Live virtual machine migration is a technique often used to migrate an entire OS with running applications in a non-disruptive fashion. Prior works concerned with one-to-one live migration with many techniques have been proposed such as pre-copy, post-copy and log/replay. In contrast, we propose VMScatter, a one-to-many migration method to migrate virtual machines from one to many other hosts simultaneously. First, by merging the identical pages within or across virtual machines, VMScatter multicasts only a single copy of these pages to associated target hosts for avoiding redundant transmission. This is impactful practically when the same OS and similar applications running in the virtual machines where there are plenty of identical pages. Second, we introduce a novel grouping algorithm to decide the placement of virtual machines, distinguished from the previous schedule algorithms which focus on the workload for load balance or power saving, we also focus on network traffic, which is a critical metric in data-intensive data centers. Third, we schedule the multicast sequence of packets to reduce the network overhead introduced by joining or quitting the multicast groups of target hosts. Compared to traditional live migration technique in QEMU/KVM, VMScatter reduces 74.2% of the total transferred data, 69.1% of the total migration time and achieves the network traffic reduction from 50.1% to 70.3%.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Experimentation, Performance

Keywords Live Migration, Virtualization, De-duplication, Multicast, Placement

1. Introduction

Live migration is a key point of the current virtualization technologies; it allows the administrator to migrate one virtual machine (VM) from one host to another without dropped network connection or perceived downtime. Live migration offers a flexible and powerful fashion to balance system load, save power and tolerant fault [19] in data centers. VMware proposes vMotion [25], a live migration technology that leverages the complete virtualization of servers, storage, and networking to move an entire running VM instantaneously. Xen proposes XenMotion [10], a similar technology to vMotion but implemented on Xen platform. Other virtualized technologies such as KVM, Hyper-V, VirtualBox also provide the live migration. Although the implementation details are different owing to heteromorphic virtualization technologies, the state of a VM reserved during live migration is analogous, involving CPU state, network state, memory state and disk state.

The existing live migration schemes focus on migrating VMs from one host to another (one-to-one). The methods, such as pre-copy [10], post-copy [27], memory compression [15], trace and replay [19] and live gang migration [11] have been proposed with the chief concern on reducing the amount of transferred memory data during live migration. It is remarkable that in practical scenarios such as online maintenance, power saving or fault tolerance, migrating multiple VMs to one host will overload the target host, and eventually crashing it. Therefore, a live migration technology that migrates VMs to many target hosts (one-to-many) is urgent.

We consider two important issues on one-to-many migration: live migration technology and placement of VMs. There has been many works [6, 23, 24, 29] sharing a similar philosophy but there still exists some unsolved issues which should be considered further. Firstly, no optimization has been proposed on live one-to-many migration technology. Prior works simply leverage the traditional ways to carry out migration; as discussed previously, these techniques are only concerned with one-to-one migration. Secondly, the de-duplication technology [11] may reduce the transferred data by merging identical pages that target one host, but it is unable to merge identical pages that target two or more hosts. Thirdly, the placement of VMs is derived from the scheduling algorithm that focus on the workload for the purpose of power saving [23], load balance [29] and SLA requirements [6]. However, considering network traffic, which is practically critical in today's data centers where a large scale of data exist for processing and transferring, frequent live migration caused by load balance or power saving will introduce additional heavy network traffic.

Note that in works [11, 14] where state there are plenty of identical pages across VMs, multicast may be a natural approach to transfer identical pages of VMs to associated hosts. In this paper, we propose a multicast based approach named VMScatter to implement live one-to-many migration. VMScatter employs multicast to deliver identical pages to a group of destinations simultaneously in a single transmission from the source host. This avoids transferring each page individually, thus it not only reduces the transferred data but also reduces the network traffic. During migration, others including the unique pages and dirtied pages will be unicasted to associated target hosts. Figure 1 presents the overview of one-to-many migration approach for migrating two VMs, each of which targets a respective host. The multicast-based live migration will

VEE'13, March 16–17, 2013, Houston, Texas, USA.

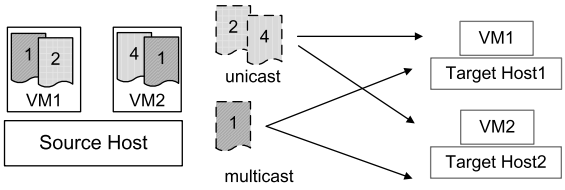


Figure 1. Overview of the live migration in VMScatter(Two pages with the same content 1 in VM1 and VM2 will be multicasted to Host1 and Host2 instead of individual transfer. Pages 2 and 4 that are unique will be unicasted to the associated host).

be valuable especially for the VMs having the same OS and applications, where result in plenty of identical pages.

Besides, two successive packets to be multicasted may target different hosts, thus some target hosts must join the multicast group for receiving expected packets while some quit to avoid receiving unneeded packets. This will introduce excessive network overhead owing to frequent *socket* related operations. We reduced this problem to be the Hamilton Cycle problem which is NP-complete, and leverage the existing algorithm to schedule an optimal permutation of packets for minimizing the overhead between multicast groups' switchovers.

Moreover, the network traffic of different placements are varied owing to the intricate association among pages, VMs and target hosts. To achieve better behavior, we introduce a grouping algorithm to specify the placement of VMs, with the aim of minimizing the network traffic while meeting the workload requirements meanwhile. We analyze the grouping impact on network traffic by a case study, give the problem formulation which is proved to be a bin-packing problem, and then present a greedy algorithm to find a preferable placement.

We implement VMScatter in QEMU/KVM, with User Datagram Protocol (UDP) to multicast the identical pages and Transmission Control Protocol (TCP) to unicast other pages. We design the protocol to guarantee consistency and integrity of the running state between the source and target hosts. Further, we implement various optimizations such as selective hashing for comparing pages, on-demand retransfer for transferring lost pages as well as compression and multithreading when sending packets. As we will see, the experiments conducted in a private data center under various workloads confirm the feasibility and efficiency of our multicast-based live one-to-many migration schema.

The remainder of this paper is organized as follows. Section II introduces the overview of live migration method via multicast, describes the phases of VMScatter, and proposes the greedy algorithm for finding a placement. Section III describes detailed implementation issues. Section IV presents the conducted experiments to evaluate the proposal. Section V surveys the related work to live migration, de-duplication, dynamic placement and multicast. Section VI concludes the paper and describes our future work.

2. Design of Live One-to-Many Migration

In this section, we present the overview of the VMScatter along with some design building blocks.

2.1 Design Objectives

The live migration process must be transparent to the operating system and applications running on top of VMs, and the overhead on the source host and network must be considered. We do not address the issue of migrating disk state within this paper, yet we suggest that as part of our future work.

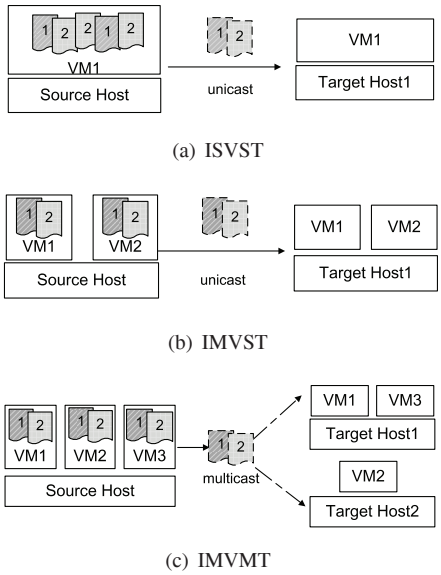


Figure 2. Three situations of identical pages transfer.

Total migration time: The time duration from the preparation at the source host to the end of the last VM's migration at the target host.

Total transferred data: The amount of data send from the source host to the target hosts to synchronize the VMs' state.

Network traffic: The network traffic is network topology-specific actually; within the paper we refer this metric to the total amount of data received by all target hosts.

Performance degradation: The influence on the performance of the applications running in the migrating VMs.

2.2 Situations for Pages Transfer

The unique pages should be unicasted, yet for identical pages, the situation is much more complicated due to the association among pages, VMs and target hosts: the identical pages may be self-identical which means existing in only one VM or inter-identical implying across many VMs, and the VMs may target one or more hosts. Figure 2 demonstrates three situations for identical pages transfer and the details are described as follows:

ISVST (Identical pages, Single VM Single Target): The identical pages exist in only one VM (self-identical); in this situation shown in Figure 2(a), transferring only one copy of identical pages to the target host is sufficient. Apparently, this reduces both total transferred data and network traffic.

IMVST (Identical pages, Multiple VMs Single Target): The identical pages exist across multiple VMs (inter-identical), and these VMs are migrated to one target host. This is similar to ISVST where only one copy of the identical pages is required to be transferred. This situation illustrated in Figure 2(b), also reduces both the transferred data and network traffic.

IMVMT (Identical pages, Multiple VMs Multiple Targets): The identical pages exist across multiple VMs (inter-identical), and these VMs are migrated to different target hosts. In this scene, the multicast mechanism is carried out to transfer a single copy of these identical pages to the multicast group where associated target hosts join to receive expected pages. Figure 2(c) illustrates such a case where VM1 and VM3 are placed to Host1 while VM2 targets Host2. This reduces the total transferred data from two aspects: one is the inter-identical pages that target the single host which is the

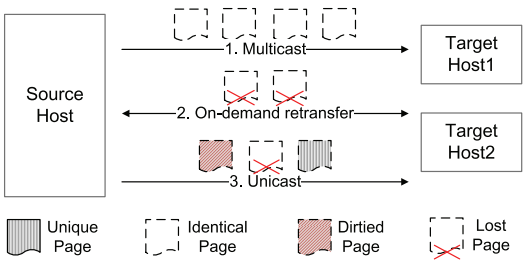


Figure 3. Three stages of page transfer.

same to IMVST, while another comes from multicasting the pages that target different target hosts. The network traffic reduction is due to the first aspect.

Since the identical pages will be transferred by single copy, the packet transferred via whether unicast or multicast must comprise three fields for referencing one page. The first is the *VM Id*, which tells the target host that which VM the page belongs to, and it can be the process id of VM, the MAC address or other unique marks. The second is the *Guest Physical Address (GPA)* of the page, which determines the memory position the page should be filled in. The last is the *Page Content*, which records the whole content of the page. The *VM Id* and *GPA* can be considered as a pair to reference one page exclusively. One packet would contain multiple *VM Id* and *GPA* pairs for the identical pages and a single copy of their *Page Content*.

2.3 Phases of Live One-to-Many Migration

VMScatter takes the similar approach to pre-copy[10], but the preparation phase contains collect stage and schedule stage, and page transfer phase consists of three stages: multicast, on-demand retransfer and unicast.

Preparation: The collect stage in preparation calculates the hash value of each page to distinguish pages having different content, and employs complete comparison to identify the identical pages. On the basis of these identical pages and their associated VMs, the schedule stage finds a preferable placement of VMs and specifies a permutation of packets to be multicasted.

Data transfer: The UDP transfers data without establishing a connection and consumes less resource, hence is suitable for multicast. Yet it is unreliable and can not guarantee successful transmission, and even results in the failure of running VM due to lost pages. As a result, TCP will be adopted as a supplementary of page transfer in a reliable manner.

As described in Figure 3, VMScatter transfers the identical pages first. For the IMVMT pages that target two or more target hosts, VMScatter packages them and multicasts the packets to associated targets via UDP. Because one lost packet may imply hundreds of lost pages(the packet loss rate is 0.3% in our private data center, and the same result can be seen in several work [8, 22]), which will bring the retransmission load for unicast, therefore we re-multicast the lost packets on-demand by the target host, which shares a similar philosophy to the post-copy method [27]. The packets in on-demand retransfer may be lost again, but the amount will be quite a few; for simplicity and robustness it is better to treat the twice lost pages as unique pages. These twice lost packets, with the unique pages among VMs, identical pages in ISVST and IMVST, and the dirtied pages during migration are unicasted to associated target host via reliable TCP.

2.4 Grouping Decision

Another key challenge in one-to-many migration is the placement of VMs, namely a grouping that describe the association between VMs and targets. In this section, we first illustrate the impact on network traffic of different groupings by a case study, then present the problem formulation which is proved to be an NP problem, and lastly propose a greedy algorithm to find a preferable grouping.

2.4.1 Grouping Impact Analysis

The grouping has few effects on the total transferred data, according to the three stages of page transfer: 1) The unique pages undoubtedly are unique regardless of the grouping, thus the amount is constant. 2) The identical pages whether multicast or unicast will be transferred by only one copy, thus the amount of transferred data is also constant. 3) The time cost is fixed in theory for transferring a constant amount of unique and identical pages, thus results in a constant amount of dirtied pages during live migration.

However for network traffic, different groupings result in significantly differences. This is because the network traffic reduction is mainly from the de-duplication of ISVST and IMVST pages that target the same host, and pages that should be transferred to the target is determined by the placement. For example, we assume the VM state is frozen and consider the state to be a set of pages; the assumption is reasonable because most pages would stay unchanged during live migration that last dozens of seconds. We suppose four VMs will be placed to two target hosts H_1 and H_2, the four VMs with their memory pages are: $V_1 = \{A, B, C\}$, $V_2 = \{A, B, D\}$, $V_3 = \{C, D, F\}$, $V_4 = \{A, C, E\}$. For one grouping in which V_1 and V_2 are placed on H_1, V_3 and V_4 target H_2, the memory pages transferred to H_1 are $\{A, B, C, D\}$, and are $\{A, C, D, E, F\}$ for H_2, the network traffic in this grouping is 9 pages. For another grouping that V_1 and V_3 target H_1 while V_2 and V_4 target H_2, the pages transferred to H_1 are $\{A, B, C, D, F\}$ and H_2 are $\{A, B, C, D, E\}$, this case generates 10 pages, which is one more page than the previous grouping. One thing to be noted is that the total transferred data of the two groupings are the same, i.e. $\{A, B, C, D, E, F\}$.

2.4.2 Problem Formulation

We consider a scenario where there are n VMs and m candidate target hosts, and we assume the VMs are frozen thus the memory pages in one VM are constant. We define the capacity of target host H_j is C_j, which refers to the maximum number of VMs that H_j can accept owing to the limited resource such as memory or workload specific factors. Each VM can be regarded as a set of memory pages and is denoted by V_i for VM i. We refer S_j to the set of VMs that are accepted by the target host H_j. The identical pages of VMs in S_j will be kept only one copy, therefore the network traffic related to target H_j can be regarded as the length of the union of memory pages owned by VMs in set S_j, we take L_j to denote the network traffic of the set S_j.

Problem definition. Given n VMs with their associated memory pages V, and m candidate target hosts with capacity C, we need to find a grouping that divides the n VMs to k target hosts $\{S_1, S_2, ..., S_k\}$, while minimizing the total network traffic L for these k target hosts.

$$L = \sum_{j=1}^{k} L_j = \sum_{j=1}^{k} |S_j| = \sum_{j=1}^{k} |\bigcup_{i=1}^{C_j} V_i|$$

This problem can be reduced from the bin-packing problem [20], and is proved to be NP-hard.

2.4.3 Greedy Algorithm

Because the global optimal solution is hardly acquired for NP-hard problem, we give the greedy algorithm for obtaining a preferable solution. Consider the purpose of grouping is placing the VMs to multiple target hosts, so the first key issue is to decide which target host should be selected prior. Note that the reduced network traffic is mainly from the de-duplication for ISVST and IMVST identical pages that target the same host (i.e., in one set S_j); intuitively, the network traffic may decrease greater if more VMs target one host in which more pages will become identical. So our greedy algorithm first fills the target host which has the maximum capacity, then fills the host with second largest capacity, and repeat until all the n VMs are filled into the targets. Moreover, this approximately minimizes the number of target hosts correspondingly.

Another key of the greedy algorithm is which VM should be selected prior to others. It is observed that for the set S_j with fixed number of VMs, the larger number of identical pages, the smaller length of the union of the set. Hence, we calculate the count of identical pages of every two VMs, and fill the target host with the VM which has the largest count. We use $N_{i,j}$ to denote the count between V_i and V_j, and define a V_i *relates* to a set if 1) the V_i has not been existed in any set, 2) there exits a V_j in the set, 3) the value of $N_{i,j}$ is nonzero. The VMs that *relate* to the set are candidates that can be added to the set. Similarly, we fill the set with the VM which has the largest $N_{i,j}$ among the candidates relate to the set.

The algorithm is described as follows: Firstly we select a target host with the maximum capacity, then we choose the two VMs which have the largest count in N, and place them into the host. Based on the two VMs, we select the VM that not only *relates* to the host but also has the largest $N_{i,j}$ among the rest N. This step will repeat until the capacity of this target host is reached. And the same procedure will be applied to other target hosts which have maximum capacity among the remaining hosts until all the n VMs are filled, thereafter we get the grouping. The algorithm 1 describes the procedure and some details are removed for clarity.

Algorithm 1 Greedy Algorithm

Require: $V = \{V_1, V_2, ..., V_n\}$; $S = \{S_1, S_2, ..., S_m\}$; $C = \{C_1, C_2, ..., C_m\}$;

```
Sort the hosts S in descending order by the capacity C;
Calculate N_{p,q} between each two VMs V_p and V_q;
N ← {N_{1,2}, ..., N_{1,n}, N_{2,3}, ..., N_{p,q}, ..., N_{n-1,n}};
Sort N in descending order;
j = 0;
for i from 1 to n do
    S_j ← {};
    while C_j ≠ 0 do
        if S_j = {} then
            Get the maximum N_{p,q};
            S_j ← S_j ∪ {V_p} ∪ {V_q};
            i ← i + 2, C_j ← C_j − 2;
        else
            Get the maximum N_{p,q} related to S_j;
            if V_p ∈ S_j then
                S_j ← S_j ∪ {V_q};
            else
                S_j ← S_j ∪ {V_p};
            end if
            i ← i + 1, C_j ← C_j − 1;
        end if
    end while
    Seek to the next host by j ← j + 1;
end for
```

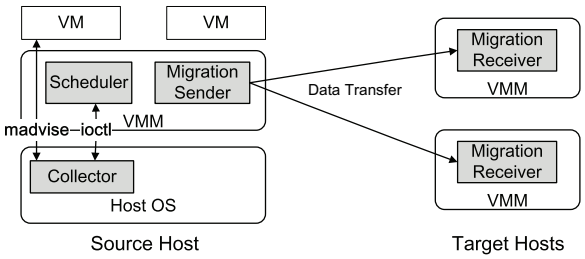


Figure 4. VMScatter architecture.

3. Implementation Issues

This section presents the implementation issues that we have made in VMScatter approach. We start by describing the overall architecture, then go on sub-level description of details and optimizations.

3.1 Architecture

We leverage the existing live migration mechanism in QEMU/KVM [16], and implement VMScatter using Linux 2.6.32 and qemu-kvm-0.12.5. We modify the QEMU code for support multicast in user mode, and implement a kernel module Collector to collect and organize the identical pages.

The VM, which is actually a process, uses a system call *madvise* to advise the Collector to handle the pages in the virtual memory address range, which represents the physical memory from the view of VM. Since the mapping from physical address of VM to virtual address of physical host is easily acquired, Collector only transfers the metadata including *Page Address* and *VM Id* of identical pages from kernel to QEMU via *ioctl*, and the Migration Sender accesses the page content in user space. According to the greedy algorithm, the Scheduler figures out a preferable grouping which determine the placement of VMs, and then the Migration Sender carries out the page transfer until all Migration Receivers in target hosts obtain the consistent state of associated VMs with that at the source host. Figure 4 illustrates the overall system architecture.

3.2 Collector Module and Selective Hash

We combine hash table and red-black tree to organize the identical pages in kernel: the pages in one hash bucket will be organized as a red-black tree, and the tree node represents a cluster of pages that have the same hash value. We adopt 32 bits hash value, the leftmost 20 bits are used to index the bucket in the hash table containing 1M (2^{20}) buckets, and the rightmost 12 bits are used to distinguish 4096 (2^{12}) tree nodes in each bucket. Therefore each node refers to unique hash value among all nodes in the hash table. For each page, we first calculate the hash value, then insert the page into the bucket indexed by the leftmost 20 bits of page hash value, and then organize the page into the red-black tree node indexed by the rightmost 12 bits of page hash value. Since hash collision may occur, resulting in the pages having the same hash value are still varied, the byte-by-byte content comparison of new inserted page to the pages have already in the node is carried out. Therefore, the different pages in the same tree node, which are hash value identical but have different content, will be distinguished.

Hashing the memory pages introduce time overhead, even the SuperFastHash [2] cost over 30s for twelve VMs with 1G memory in 2-way quad-core Xeon E6750 2GHz processors. To speedup the calculation, we just select disperse 200 bytes instead of the whole 4096 bytes to obtain the hash value of the page. Against the SuperFashHash method, the selective hash calculation for 12G memory can reduce the time cost from 37.3s to 1.8s, and the

number of hash collisions increase from about 100 to only 1000, minor compared to millions of pages.

3.3 Page Classification: Identical or Unique

Some identical pages may be short-lived due to the dirtied content during the live migration, causing the page that is unique to become identical later and vice versa. Figure 5 shows such a case in which the identical page in collect stage turns to be unique during page transfer then becomes identical when the migration is over after cycles of updates. One common approach is dynamic tracking: by setting the pages to be write-protect, we can track the content change of each page, and reorganize the dirtied page in the hash table, then decide whether the page is identical or not in real-time. Afterwards we notify the Migration Sender to multicast or unicast the page correspondingly when transferring.

However in our implementation, we consider the pages to be identical or unique according to the memory state when collect stage finishes, without dynamic tracking during later migration. This is due to three reasons: 1) The write-protect skill will cause page fault for each memory write operation, which introduces a heavy burden for memory-intensive applications, hence seriously decrease the performance of VM; 2) Some part of memory pages may be dirtied frequently in a certain time, resulting in lots of notifications. Most notifications will be useless because the new arriving notification may cover the older one ahead of transfer; 3) The experimental result says that majority of identical pages will keep content constant and identical during dozens of seconds running, and these long-lived pages suggest us adopting the page type in one certain epoch is feasible and simple, with bringing about minor unnecessary multicast pages.

3.4 Successful Page Transfer

To guarantee the successful migration of VMs, VMScatter should fall in two ways of what is needed: integrity and consistency of transferred pages.

Integrity: Integrity means that we should construct a complete memory space for each VM at the target host. The lost packets owing to unreliable UDP may cause plenty of missing pages, leading to system runtime error or even crash. We keep the integrity via the on-demand retransfer and reliable unicast. Each target host maintains an array recording *Packet Id* which is associated with packet for indexing the lost ones, and then requests the lost packets from the source host by *Packet Id*. After receiving the request, the source host will re-multicast the packets to all associated targets. The unicast stage will employ reliable TCP to retransfer the pages lost in on-demand retransfer. Thus with the combination of UDP and reliable TCP, we achieve transferring the pages in an integral mode.

Consistency: Consistency of memory pages between the source and target hosts preserves the newest state for VMs after migration; the main cause of inconsistency is the new dirtied memory pages during migration. We leverage the method proposed by Clark et al. [10], use bitmap to index the dirtied pages during transfer, transfer the dirtied pages iteratively until the amount of dirtied pages converges to below the threshold, then we stop the VM, transfer the left pages via TCP, and lastly boot the VM at the target.

3.5 Join and Quit the Multicast Group

The multicast group reflects a group of target hosts intended to receive the packet. A target host must join the multicast group to receive the expected packet, and quit to avoid receiving unneeded packets for reducing both the network traffic and overhead. Since each packet is related to one certain multicast group which consists of hosts targeted by the packet, the various packets may be transferred to different multicast groups, therefore cause the target hosts to join and quit the multicast group frequently. Thus the network overhead rise heavily for thousands of multicast groups' switchovers for various packets. To avoid this unfortunate behavior, we need to specify a permutation of packets.

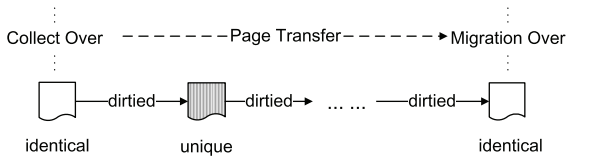


Figure 5. Page type change during migration.

Note that the multicast group can remain unchanged for packets targeting the same hosts. In addition, the number of multicast groups is far fewer than the amount of packets, because there exists at most $2^k - 1 - k$ combinations for k target hosts (k is not more than the number of VMs) but has millions of packets, implying many packets target the same multicast group. The two reasons encourage us to firstly group the packets that have the same multicast group, and then design a sequence of these various multicast groups to minimize the overhead between the multicast groups' switchovers. And this sequence represents the permutation of packets.

We consider n multicast groups, and refer G_i to one multicast group i. Providing the cost for one target joining or quitting the group are equal and constant, says c; and $W_{i,j}$ denotes the overhead of switchover from G_i to G_j, thus $W_{i,j}$ is the product of the count of target hosts' joining and quitting in the switchover and the cost c. For example, G_i contains a set of hosts $\{H_1, H_3, H_4\}$ while G_j contains $\{H_2, H_3\}$; in the switchover from G_i to G_j, H_1 and H_4 will quit the multicast group, meanwhile H_2 joins in. Thus the value $W_{i,j}$ is $3*c$ for two quitting and one joining. Our purpose is to find a permutation of multicast groups with the aim of minimizing the total overhead W:

$$W = \sum_{i=0}^{n} W_{i,i+1} \qquad G_0 = G_{n+1} = \{\}$$

This can be reduced to be the classic Minimum Hamiltonian Cycle problem which is NP-complete, we simply adopt the algorithm proposed by Bollobas et al. [7], which approximates the optimal value in polynomial time.

The procedure of multicasting the packets to groups would be simple once the permutation is known. First of all, the Migration Sender selects the first multicast group in the permutation, notifies the associated target hosts to join the group, and then multicasts the packets target this multicast group. After the packets target this multicast group being send over, the Migration Sender will notify associated target hosts to join or quit this multicast group to switch to the next multicast group, and then transfer the packets to the new multicast group. This procedure will repeat until all packets are transferred to associated multicast groups.

3.6 Compression and Multithreading

Compression is an efficient approach to reduce the size of transferred packets; the algorithms such as LZ77 [30], LZW [17] can bring about as many as 50% or more data saving, which is significant in the live migration scene where large amount of data exists. We take *zlib* which is an effective compression library to compress the packets.

However, compression will introduce additional CPU overhead and cost more migration time. Multithreading is a valuable assisting technology to parallelize the tasks by overlapping the processing time of threads and distributing the tasks to multiprocessors, thus

efficiently reduce the time for CPU intensive and IO intensive tasks running on multiple processors. In VMScatter, we exploit a thread pool containing six threads to reduce the thread creation and destruction overheads. Each thread will independently compose packets and then compresses the packets via *zlib*, lastly transfer the packets to targets during the thorough data transfer phase.

4. Experimental Evaluation

We test VMScatter on serval workloads, and give the detailed evaluation in this section. We begin by illustrating the results related to page content similarity of VMs acquired by Collector module, then compare the metrics including total migration time, total transferred data and network traffic between the QEMU/KVM live migration technology and VMScatter schema; furthermore, we present the results on network traffic of preferable grouping versus random groupings. Lastly we characterize the impacts on system performance both in a single VM and VM cluster.

4.1 Experimental Setup

We conduct our experiments on 14 physical servers, each with 2-way quad-core Xeon E6750 2GHz processors, 16GB DDR memory, and NetXtreme II BCM5708S Gigabit network interface card. The shared storage is configured with 1T disk, and connected to servers via switched Gigabit Ethernet. We configure 1GB memory for each VM unless specified otherwise; therefore the physical server can support as many as 16 VMs. The operating system on both physical and virtual machines is debian6.0, with Linux kernel version 2.6.32. All the servers share the storage so that the disk state does not need to be migrated. The workloads includes:

Idle workload means the VM does nothing except the tasks of OS self after boot-up.

Kernel Compilation represents a development workload involves memory used by the page cache. We compile the Linux 2.6.32 kernel along with all modules.

Sysbench [3] is a benchmark tool for evaluating OS parameters. We perform 5000 transactions on the database table containing 1 million entries.

TPC-W [4] is a transactional benchmark that simulates the activities of a business web server. We run TPC-W serving 600 simultaneous users accessing the site using the *shopping* workload that performs a mix of read/write operations.

4.2 Identical Rate Acquired by Collector

The identical rate, which is defined by the percentage of the identical pages on all the memory pages, is the key to this proposal because higher identical rate means more pages can be deduplicated and multicasted. The identical memory pages come from five sources: memory of kernel that loaded when booting up, the content of the loaded application data and code, some library codes related to the application, and content generated by application and zero pages. The zero pages will be dirtied after running a long time, in our experiment, the number of zero pages decreases from about 200,000 after boot-up to less than 5000 in one VM which has 262144 pages (1G memory). As a result we conduct the experiments after long time running to minimize the impact of zero pages. We illustrate the identical rate of the VMs with the same OS and applications, and then the identical rate of different VMs. Our experiment obtain similar results to many work [14, 21] which state about 50% to 90% of the pages have identical content with others for VMs having the same OS, providing a high degree of confidence that the VMScatter would be effective.

Same VMs. The same VMs have the same OS and the same applications. Figure 6 demonstrates the variation of identical rate with the increasing number of VMs. The rate is higher than 86%

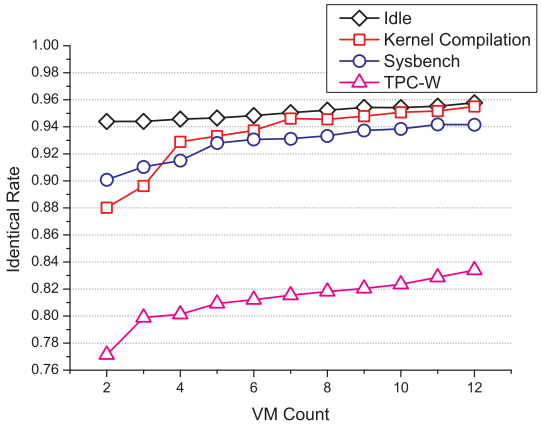


Figure 6. Identical rate of virtual machines.

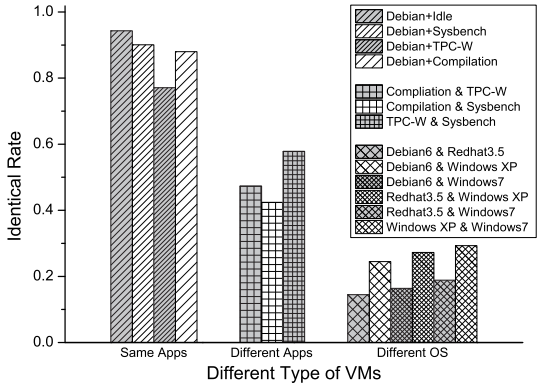
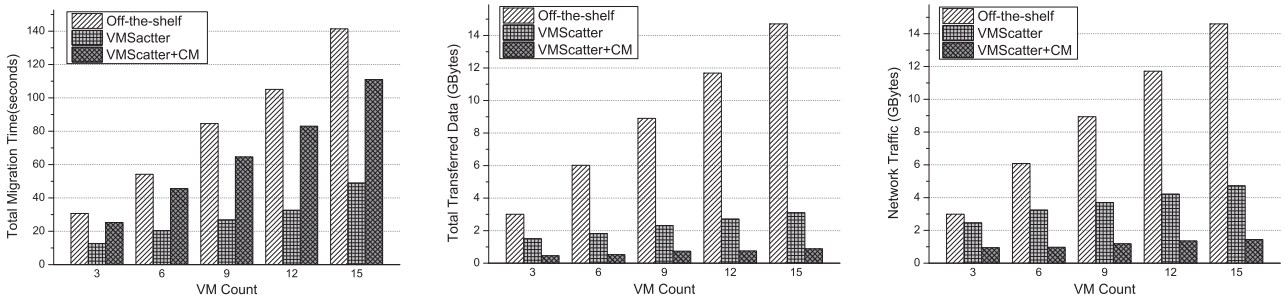


Figure 7. Identical rate of different VMs.

among the same VMs for Idle, Kernel Compilation and Sysbench workloads. Furthermore, we observe that the identical rate rises as the number of VMs increases, e.g. ranges from 88.03% to 95.3% with the Kernel Compilation as a test application. The rise is because the unique page may become identical to another page in the new added VM. This result is encouraging because more than 86% memory pages of 11 VMs may be eliminated via multicast, which will reduce the total transferred data by a lot.

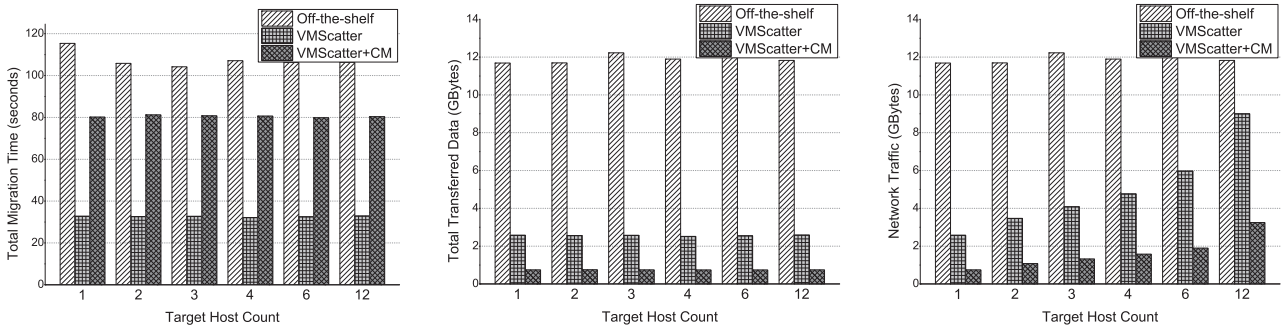
Different VMs. We test two kinds of different VMs: 1) VMs with the same OS but different applications, in this case, we test on three VMs with debian6.0, and initiate Kernel Compilation, Sysbench, and TPC-W separately in each VM. 2) VMs with different OS, four VMs are equipped with Debian6.0, Redhat5.3, Windows XP, and Windows 7, with variety of applications running inside such as web browser, video player, office, etc.

As we can see from Figure 7, the identical rate between each two VMs with different applications varies, 47.34% for Kernel compilation and TPC-W workloads, 42.43% for Kernel Compilation and Sysbench, 57.85% for TPC-W and Sysbench (higher than the other two pairs because they are transactional benchmarks related to MySql), and the identical rate across all the three VMs is 51.84%. This result indicates that about half of the memory pages are identical for VMs with the same OS, and the reduced identical pages compared to the same VMs are due to the different applications and their content. However for different VMs with different



(a) Total Migration Time (b) Total Transferred Data (c) Network Traffic

Figure 8. Comparison of three metrics in different modes, for three target hosts and varied number of VMs.



(a) Total Migration Time (b) Total Transferred Data (c) Network Traffic

Figure 9. Comparison of three metrics in different modes, for increasing number of target hosts and twelve VMs.

OS shown in the third set of bars, the identical rate is much lower, only at the average of 21.8% between two VMs. We suspect the identical pages mainly come from zero pages and universal libraries. These results imply VMScatter may still benefit for saving identical pages in transmission.

The above two results also suggest us that the identical rate is different in different number of VMs or various type of VMs. For fair comparison of all modes, we evaluate our method only on migrating equal numbers of VMs having the same OS and applications to each target host in the following experiments.

4.3 Live Migration via Multicast

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

Off-the-shelf migration: This method is the simple live migration method used in QEMU/KVM without optimizations except the compression of the page whose bytes are the same such as zero page.

VMScatter: This is our live one-to-many migration method in that the identical pages will be merged into one page in the packet and multicasted to different hosts.

Compression and multithreading (VMScatter+CM): This mode extends the VMScatter work, with threads each of which composes, compresses and then sends the packets.

We first migrate 3, 6, 9, 12 and 15 VMs separately to three target hosts, then migrate twelve VMs simultaneously while varying the number of hosts to evaluate the live one-to-many approach on three metrics: total migration time, total transferred data and network traffic. The results illustrated are the average of 20 trials with Kernel Compilation running inside the VMs.

Total migration time. Figure 8(a) and Figure 9(a) compare the total migration time of the three modes of live migration. It can be seen that the VMScatter mode gives the lowest total migration time, migrating the 12 VMs in 32.7 seconds, achieving about 69.1% reduction against the off-the-shelf mode which costs about 105 seconds. This is due to the fact that the pages that have identical content are transferred by only a single copy with reference information such as *Page Address* and *VM Id*, which reduces large amount of transferred data and IO overhead. The performance of VMScatter+CM mode, however, reduces only 25% of migration time. The reason is straight: packets compression is CPU intensive so that consumes the additional time compared to VMScatter. Yet as observed in the graph, this mode still consumes less time than the off-the-shelf mode due to the benefit of multicast and page deduplication.

Total transferred data. Figure 8(b) and Figure 9(b) plot the total transferred data of the three modes. One small anomaly is in the off-the-shelf mode where the total transferred data is less than 12G which should be the sum of 1G memory size for 12 VMs. This is because the compression of zero pages implemented in QEMU/KVM, which involves representing one page by only one byte instead of the 4096 bytes during transmission. As expected, the VMScatter method transfers far fewer data than off-the-shelf mode, and brings about 74.2% reduction attributable to unimplemented transmission of duplicate pages. Note that total transferred data and the total migration time show a similar trend which both introduce about 70% reduction, this is due to the limited network band-

width between the physical hosts. Although the VMScatter+CM mode consumes more time, it enhances the VMScatter further by 70.6%, and achieves a total of 92.4% reduction over the off-the-shelf method.

As Figure 8(b) shows, the increase of total transferred data in VMScatter is not proportional to the number of VMs. This is because the inter-identical pages are only transferred by multicasting a single copy, so the identical pages in the new added VM will not need to be transferred any more except extra page references such as *Page Address* and *VM Id* pairs. The increased amount are mainly from the unique pages in the added virtual machines as well as the additional dirtied pages caused by longer migration time.

It should be observed in the Figures 9(a) and 9(b): both the total migration time and total transferred data remain unchanged regardless of the number of target hosts. The reasons are explained as follows. 1) The definitive identical rate of 12 VMs as shown in Figure 6 implies that the amount of packets is almost fixed for both identical and unique pages, which further indicates the transfer time for these two types of packets is definitive in limited network bandwidth. As a result, the amount of dirtied pages can be regarded as fixed. In addition, the lost packets increase the transferred data but only a small number (only about 0.3%). Consequently, the amount of total transferred data consists of the above three types of packets and can be considered to be constant. 2) For the total migration time, the time spent on the preparation phase including collecting identical pages and scheduling groups is almost constant for fixed page numbers, therefore the total migration time is in line with the page transfer time, thus it is also constant.

Network traffic. Although there is no exact method to quantify network traffic during the live migration, we provide an approximate measure by the sum of packets received by target hosts. Figures 8(c) and 9(c) compare the network traffic with the increasing number of VMs and target hosts respectively. The network traffic is equal to the total transferred data when the number of targets is one, this is easily understood by the way we measure the network traffic. Another result to be observed is that when three (12) VMs are migrated to three (12) target hosts as shown in Figure 8(c) (9(c)), i.e., each target host receives only one VM, the VMScatter method still reduces the network traffic by 17.8% attributable to the self-identical pages within the VM. For other scenarios, the network traffic in VMScatter mode decreases significantly with a range between 50.1% to 70.3%.

Different from Figure 9(b) where the total transferred data are constant over various number of hosts, the network traffic increases as the number of target hosts increase as illustrated in Figure 9(c). This is because one additional copy of the packets needs to be forwarded by the switcher to the new added target host during multicast over the network. The VMScatter+CM mode also gain performance, reducing the multicast traffic further by about 69.7%.

We also evaluate the three metrics under Sysbench and TPC-W workloads. For the TPC-W which has lower identical rate, the VMScatter live migration method still performs nicely by reducing 63.3% of the total migration time, 67.4% of the total transferred data and 55.8% of the network traffic.

Overall, these results confirm the effectiveness of VMScatter. Although the compression and multithreading method produces longer total migration time, it reduces numerous transferred data and network traffic further by about 70% on the basis of VMScatter mode.

4.4 Downtime

Downtime is another important metric of live migration. It consists of the time to suspend the VM at the source host, transfer the dirtied pages, and activate the migrated VM at the target host. The downtime is inevitable because the dirtied pages generated during continuous data transfer will lead to the inconsistency of VM state between the source and target host.

Table 1 shows the comparison in terms of downtime for the three modes for migrating 12 VMs to three targets evenly. The variation in the downtime is due to the parallel migration. The VMScatter mode performs better than the off-the-shelf method, and this could because this mode generates less dirtied data in a shorter migration time, thus consumes less time in the final data transfer after suspending the VM. Consider the VMScatter+CM mode, the overhead of compression at the source and decompression at the target cause the minimum value to be larger than the other two modes, and the average is less than off-the-shelf due to lesser time to transfer the reduced packet size.

4.5 Grouping Benefit

The two most significant results we have seen so far are in Figures 8 and 9 where the total transferred data and network traffic are reduced. We then conduct experiments to evaluate our grouping method which aims to reduce the network traffic further by deciding a preferable placement.

Figure 9(c) demonstrates the variation of network traffic with different groupings when the number of target hosts varies. Furthermore, we fix the number of targets and construct a group that distributes twelve VMs evenly to each target, we distribute evenly for fair comparison since the difference in number may result in volatile identical rate which affects the results. For each fixed number of targets, we simulate 60 different groupings, migrate the 12 VMs to the associated target hosts decided by each grouping and then count the network traffic. Besides, we obtain the preferable grouping by the greedy algorithm. We set the capacity of target hosts as identical, which means the hosts will accept the same number of VMs.

Intuitively, there is only one grouping method when there is one host, where all the VMs would target one host; the same is true for 12 targets where each VM targets a respective host. Thus, the two scenarios are not our concern. Table 2 illustrates the results of the network traffic on different groupings under various workloads, including the maximum value, minimum value and average value of the network traffic on 60 groupings; along with the network traffic of our preferable grouping. As the table shows, the maximum traffic is 4.07G while the minimum value is 3.47G for targeting three hosts when Kernel Compilation running inside the VMs, and the difference between the two groupings is about 17.3%. Our preferable placement of VMs decreases the network traffic to 3.31G, a 13.4% reduction compared to the average value. Generally, the preferable grouping achieves 10% to 15% reduction of the network traffic against the average of random groupings, thus it proves the improvement of grouping algorithm. The 10% to 15% reduction of network traffic is particularly valuable in the data-intensive data centers.

4.6 Performance Impact

In this section, we quantify the side effects of migration on a couple of sample applications. We evaluate the performance impact on both single VM and VM cluster with migrating 12 VMs to

Modes	**Max**	**Min**	**Avg.**
Off-the-shelf	2351	192	1518
VMScatter	1573	184	863
VMScatter+CM	1483	576	1132

Table 1. Comparison of downtime (ms).

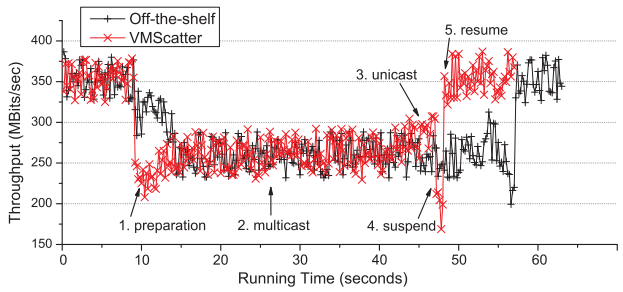


Figure 10. Throughput during live migration (result of off-the-shelf mode has been truncated to save space).

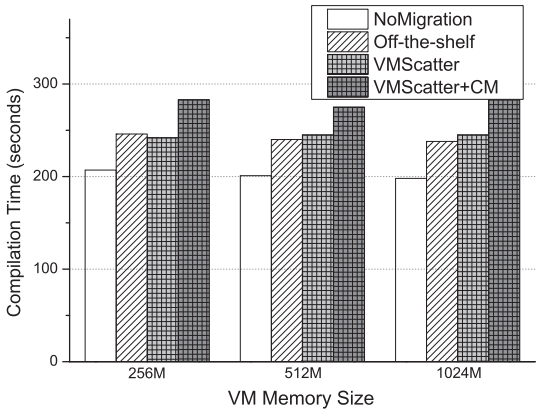


Figure 11. Compilation time on migration.

three targets, and illustrate the results of off-the-shelf mode versus VMScatter mode.

Impact on Single VM. We start first by measuring the performance on a single VM in terms of throughput per second by examining the live migration of a Apache web server serving static content at a high rate. The web server served 1000 files of 512 K-Bytes, all of which were cached on memory. In this experiment, 10 httperf processes in a remote client host sent requests to the server in parallel. Figure 10 illustrates the throughput achieved when continuously serving concurrent clients. At the start of the trace, the normal running of the VM can serve about 354Mbits/sec. After the live migration starts at the 9th second, the throughput of VMScatter decreases to about 214Mbits/sec which is lower than 309Mbits/sec of off-the-shelf. This is because the collecting and scheduling stages consume more CPU resource than off-the-shelf mode which only set flags. Then the page transfer phase serves 272Mbits/sec for about 24 seconds. There is no obvious decrease compared to off-the-shelf mode, thus implying the optimized permutation of packets takes effect. The throughput in transferring dirtied pages keeps about 305Mbits/sec which is higher than multicast. This may be because the amount of dirtied pages in unicast is less than the amount of pages during multicast, thus reserving more CPU and network resource for applications running insides VMs. One sudden decrease is the result of VM suspending. After the VM is resumed at the target host, the throughput returns to normal.

Impact on VM Cluster. We evaluate the performance of VMScatter on VM Cluster via *distcc* [1] to build a kernel compilation cluster to distribute the compilation tasks across the 12 VMs, and migrate back and forth repeatedly between the source and three target hosts. Figure 11 compares the completion time for various memory size of VM under three live migration modes, the result without migration is also given for comparison. The VMScatter mode consumes almost the same compilation time as the off-the-shelf method, and both increase by less than 20% compared to NoMigration, owing to the similar CPU and network utilization. The VMScatter+VM mode cost more time because the CPU overhead of compression at the source host and decompression at the target hosts.

5. Related Work

Live Migration. Clark et al. [10] first propose the pre-copy live migration based on Xen platform, they transfer the page iteratively, and boot the VM when the consistent state are reserved in target host. However pre-copy migration may fail in harsh scenes such as low network bandwidth and memory-intensive workload where the amount of dirtied pages cannot converge. Hines et al. [27] propose post-copy method, they first boot the VM on target host and then copy the pages on demand, thus the memory pages will be transferred only once which both solves the problem of pre-copy and reduces the total transferred data. Liu et al. [19] adopt the methods of ReVirt [12], achieve live migration by transferring the log which records the execution of VM and replaying them at target host. Deshpande et al. [11] consider migrating multiple machines from one host to another, and propose live gang migration by page sharing and delta transfer to reduce the amount of transmission. In contrast, our concern is with the one-to-many migration and implementing VMScatter by multicasting a single copy of the identical pages instead of individual transfer.

Multicast. Multicast has been used to transfer images or snapshots for deploying multiple identical VMs in the IaaS platform [18, 26]. VMScatter employs the multicast method to transfer the identical pages by single copy, combined with unicast to transfer the unique pages and dirtied page. Besides, we specify a permutation of packets with the solution of Hamilton Cycle problem to reduce the network overhead occurred in multicast groups' switchovers.

Page Sharing and De-duplication. Page sharing saves memory consumption of VMs by merging the identical pages into one physical page. Bugnion et al. propose Disco [9], a tool that uses transparent page sharing to de-duplicate the redundant copies across VMs. Waldspurger et al. [28] improve Disco further by content-based page sharing. Milos et al. [21] use sharing-aware block devices for detecting duplicate pages on Xen virtual machine monitor. Gupta et al. [14] improve the page sharing rate among VMs by dividing

Target Count	2				3				4				6			
Benchmarks	**Max**	**Min**	**Avg.**	**Prefer.**	**Max**	**Min**	**Avg.**	**Prefer.**	**Max**	**Min**	**Avg.**	**Prefer.**	**Max**	**Min**	**Avg.**	**Prefer.**
Compilation	3.56	3.12	3.37	3.05	4.07	3.47	3.82	3.31	4.86	4.11	4.65	4.03	5.97	5.12	5.73	5.18
Sysbench	3.82	3.33	3.67	3.32	4.22	3.69	4.07	3.52	5.08	4.33	4.86	4.30	6.23	5.34	6.02	5.45
TPC-W	4.84	4.37	4.60	4.34	5.33	4.71	5.15	4.89	6.17	5.41	5.98	5.49	7.29	6.86	7.15	6.89

Table 2. Comparison of network traffic(GBytes) for groupings, the target host count is 2, 3, 4, 6.

the page to sub-pages. Arcangeli et al. proposes KSM [5], a kernel module in Linux that uses an unstable red-black tree to improve the efficiency. We share a similar philosophy to page sharing, but against the motivation of page sharing that focus on less physical memory consumption, VMScatter is interested in de-duplicating the identical pages in the packet to be multicasted. Furthermore, our approach combines the selective hash with the red-black tree, and achieves an order of magnitude speedup over the original hash method on organizing millions of memory pages.

Placement of VMs. Many works have adopted live migration technology to achieve power saving [23], load balance [29], SLA [6], quality of service (QoS)[24], etc. In this paper, we consider the network traffic metric and propose a grouping algorithm with the aim of minimizing network traffic by selecting a preferable placement of VMs.

6. Conclusions

We implemented VMScatter to migrate VMs to multiple hosts. Our design and implementation addressed the issues involved in live one-to-many migration, placement of VMs and multicast specific options. By merging the identical pages into one page, VMScatter multicasts the single page to many targets instead of transferring these pages individually. The novel grouping method guides the VM's destination with respect to the network traffic over the network. And we explore a further benefit allowed by compression and multithreading. Through detailed evaluation, we show that the performance is sufficient to make VMScatter a practical tool in data centers even for VMs running interactive loads. In the future, we plan to investigate providing disk state migration, perhaps using existing techniques to improve VMScatter for hosts connected to independent storage, and evaluate VMScatter in complex network topologies such as BCube [13].

Acknowledgments

We acknowledge Yang Cao for his contributions to the algorithm of this work and Bin Shi, Kun Liu for the experimental setup. We also thank the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302602, National High Technology Research 863 Program of China under Grant No. 2011AA01A202, National Nature Science Foundation of China under Grant No. 61272165, No. 60903149 and No. 91118008, and New Century Excellent Talents in University 2010 and Beijing New-Star R&D Program under Grant No. 2010B010.

References

[1] Distcc. `http://code.google.com/p/distcc/`.

[2] Superfasthash. `http://www.azillionmonkeys.com/qed/hash.html`.

[3] Sysbench. `http://sysbench.sourceforge.net/`.

[4] Tpc-w. `http://www.tpc.org/tpcw/`.

[5] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[6] N. Boboroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.

[7] B. Bollobas, T. I. Fenner, and A. M. Frieze. An algorithm for finding hamilton paths and cycles in random graphs. *Combinatorica*, 7(4): 327–341, 1987.

[8] M. S. Borella, D. Swider, U. S, and B. G.B. Internet packet loss: Measurement and implications for end-to-end qos. In *Proceedings of ICPP Workshps*, pages 3–12, 1998.

[9] E. Bugnion, S. Devine, Kinshuk, Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of NSDI*, pages 273–286, 2005.

[11] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proceedings of HPDC*, pages 135–146, 2011.

[12] G. W. Dunlap, S. T. Kin, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of OSDI*, pages 211–224, 2002.

[13] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74, 2009.

[14] 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. *Communications of the ACM*, 53(10): 85–93, 2010.

[15] H. Jin, L. Deng, and S. Wu. Live virtual machine migration with adaptive memory compression. In *Proceedings of CLUSTER*, pages 1–10, 2009.

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[17] M. J. Knieser, F. G. Wolff, C. A. Papachristou, D. J. Weyer, and D. R. McIntyre. A technique for high ratio lzw compression. In *Design, Automation & Test in Europe*, pages 10–16, 2003.

[18] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannel, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of EuroSys*, pages 1–12, 2009.

[19] H. Liu, H. Jin, and X. Liao. Live migration of virtual machine based on full system trace and replay. In *Proceedings of HPDC*, pages 101–110, 2009.

[20] R. E. Miller and J. W. Thatcher, editors. *Complexity of Computer Computations*. Plenum Press., New York, 1972.

[21] G. Milos, D. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference*, pages 1–14, 2009.

[22] S. B. Moon, J. Kurose, P. Skelly, and D. Towsley. Correlation of packet delay and loss in the internet. Technical report, 1998.

[23] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *ACM Symposium on Operating Systems Principles*, pages 265–278, 2007.

[24] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of EuroSys*, pages 237–250, 2010.

[25] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX '05 Technical Program*, 2005.

[26] B. Nicolae, J. Bresnahan, and K. Keahey. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *Proceedings of HPDC*, pages 147–158, 2011.

[27] M. R, Hines, and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of VEE*, pages 51–60, 2009.

[28] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of OSDI*, pages 181–194, 2002.

[29] Y. Zhao and W. Huang. Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In *Fifth International Joint Conference on INC, IMS and IDC*, pages 170–175, 2009.

[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1997.