Parallelizing Live Migration of Virtual Machines

Xiang Song † ‡ Jicheng Shi† ‡ Ran Liu † ‡ Yang Jian† ‡ Haibo Chen†

†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University ‡Software School, Fudan University

classicxsong@gmail.com rogershijicheng@gmail.com naruilone@gmail.com sheepx86@gmail.com

haibochen@sjtu.edu.cn

Abstract

Live VM migration is one of the major primitive operations to manage virtualized cloud platforms. Such operation is usually mission-critical and disruptive to the running services, and thus should be completed as fast as possible. Unfortunately, with the increasing amount of resources configured to a VM, such operations are becoming increasingly time-consuming. In this paper, we make a comprehensive analysis on the parallelization opportunities of live VM migration on two popular open-source VMMs (i.e., Xen and KVM). By leveraging abundant resources like CPU cores and NICs in contemporary server platforms, we design and implement a system called PMigrate that leverages data parallelism and pipeline parallelism to parallelize the operation. As the parallelization framework requires intensive mmap/munmap operations that tax the address space management system in an operating system, we further propose an abstraction called *range lock*, which improves scalability of concurrent mutation to the address space of an operating system (i.e., Linux) by selectively replacing the per-process address space lock inside kernel with dynamic and fine-grained *range locks* that exclude costly operations on the requesting address range from using the per-process lock. Evaluation with our working prototype on Xen and KVM shows that PMigrate accelerates the live VM migration ranging from 2.49X to 9.88X, and decreases the downtime ranging from 1.9X to 279.89X. Performance analysis shows that our integration of *range lock* to Linux significantly improves parallelism in mutating the address space in VM migration and thus boosts the performance ranging from 2.06X to 3.05X. We also show that PMigrate makes only small disruption to other co-hosted production VMs.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance

Keywords Parallelized VM Migration, Range Lock

1. Introduction

Live VM migraion [13, 22] has been key enabling techniques in maintaining virtualized data-centers, including load balancing, fault tolerance [20], and power management [21]. In many cases, such operation should be completed in a timely manner to avoid disruption and performance degradation of running services.

Usually, if the total downtime of live VM migration exceeds a certain number (usually several minutes in Giga-Ethernet), the clients will notice the unavailability of the service and the running service may be disrupted. Worse even, if a scheduled VM migration cannot be completed in time when proactively tolerating hardware/software faults, a possible server outage in the source machine might crash the VM and even a distributed service in a virtual cluster as a whole. Further, during live VM migration, the running service usually experiences a notably lower performance due to the overhead of tracking and processing the resources of the target VM.

On the other hand, the amount of resources allocated to a VM has been steadily increasing along with Moore's law. For example, the large and high-memory quadruple Extra Large VM instances in Amazon EC2 [4] are configured with 7.5 GByte and 68.4 GByte memory accordingly. It is also no surprise to see virtual machines with multiple dozens of Giga-bytes memory in some resource-intensive applications such as virtualized database servers (e.g., Oracle and Microsoft SQL Server). However, due to the necessity of touching a huge amount of resources, the execution time of live VM migration usually increases along with the amount of resources in a VM. For example, the total migration time and downtime for a modest-size Xen VM [9] with 16 GByte memory running a memcached server have increased to 1,586s and 251s accordingly (section 6). Such long migration time and downtime are apparently not satisfiable as the running service will be severely disrupted. Meanwhile, during the process of migration, the total throughput of the VM is only 74.5% of normal execution in our evaluation. Worse even, the migration may also degrade the performance of other co-located VMs.

In this paper, we conduct a comprehensive study on the parallelization opportunities of the live VM migration. Our analysis uncovers both data and pipeline parallelism underlying basic primitives of it, which motivates the design and implementation of PMigrate a system that aims at parallelizing live VM migation. PMigrate is inspired by that fact that the increasing amount of resources configured to a machine also opens opportunities to leverage such resources for parallelization. Actually, it is currently no surprise a server machine with several dozens of CPU cores, several hundreds of Giga-bytes memory and a dozen of NIC ports, which are not always being fully utilized.

The result of parallelization further uncovers a significant performance bottleneck to the parallelized live VM migration: concurrent mutation to an address space. This is because the migration operation requires frequent mapping/unmapping of memory pages owned by a guest VM to the address space of the management tool. However, our survey indicates that most commodity operating systems (e.g., Linux, Solaris and BSD) usually use a per-process lock to serialize mutation to an address space. To allow concurrent mu-

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

tation to an address space, we further propose an abstraction called *range lock*, a dynamic fine-grained lock abstraction that allows fine-grained protection of some costly operations, instead of a complete serialization of mutation to the address space. By integrating *range lock* with existing address space management code in Linux, the parallelism with concurrent address space mutation for PMigrate is significantly increased.

We have extended Xen 4.1.2 to support the parallelization of live VM migration, which add 1,860 lines of code to Xen tools and domain0[1] kernel. We also integrate the support of *range lock* to Linux to improve the scalability of concurrent mutation to address spaces. To further demonstrate the applicability of parallelizing VM management operations, we implement live VM migration based on kvm-qemu 0.14.0, with the support of migrating both memory and disk data. The implementation adds 2,270 lines of code to the kvm-qemu tools.

To measure the effectiveness of the parallelized live VM migration operation and the performance benefits of devoting more resources, we have conducted several evaluations using several widely-used applications. Performance results show that parallelized migration operation significantly outperforms the vanilla one. It accelerates live VM migration ranging from 2.49X up to 9.88X, with the downtime decreased ranging from 1.9X to 279.89X. We further show that the performance impact of parallelized live VM migration on the target VM is also very small compared to the vanilla live migration: the average throughput is reduced by 12.5% (with the maximum be 16%). We also show that our integration of *range lock* to Linux significantly improves parallelism in mutating the address space in virtual memory management operations, which boosts the performance ranging from 2.06X to 3.05X. The performance impact on the co-located VMs is also quite small with the control of resources dedicated to VM operations. We further show that PMigrate can be integrated with other optimization by evaluating it with live VM migration using compression. Our evaluation results show that PMigrate improves the performance by 3.98X compared to the vallina one with compression.

In summary, the contributions of this paper are:

- A case for parallelizing live VM migration.
- The *range lock* abstraction and its integration in Linux that increases the parallelism of concurrent mutation to an address space.
- The design, implementation and evaluation of PMigrate on Xen and KVM that confirm the effectiveness of parallelization and *range lock*.

The rest of the paper is organized as follows. The next section provides an overview of live VM migration and the underlying parallelism, which motivates our design on parallelizing them in section 3. Section 4 illustrates the concurrent address space mutation problem and describes the *range lock* abstraction as well as how it is integrated to Linux. Section 5 describes our implementation on Xen and KVM. The experimental results are shown in Section 6. We relate our work with previous work in Section 7 and conclude our work in Section 8.

2. An Overview of VM Migration

In this section, we briefly illustrate live VM migration based on Xen [9, 13] and KVM [16], and present a quantitative analysis on the primitive operations and the source of parallelism accordingly. Though there are several variants in live VM migration, we mainly describe the *Pre-copy* approach [24], which is the default strategy for most hypervisors such as Xen, KVM and VMWare. Parallelizing other migration strategies like *Post-copy* (which is rarely used due to its unreliability) is quite similar, which we omit here for brevity.

2.1 Basic Steps in Live VM Migration

For live VM migration, CPU states, memory, persistent storage and other device states are required to be migrated to the destination node on the fly. The *migrate_send* function in Figure 1 shows the send-side algorithm of the pre-copy live migration: the source node iteratively transfers the memory and disk data to the destination node while keeping the source VM alive through multiple iterations, which is called the pre-copy step. At the very beginning, the initial memory and disk data are considered to be dirty and will be transferred as a whole. In the following iterations, the hypervisor will track the newly dirtied data generated by VM execution and add the dirtied pages to a page pool, which will be sent out in the following iteration. The iteration stops only when 1) the dirty data left is small enough or 2) its size cannot be further reduced by introducing more iterations or 3) too many resources are wasted on the pre-copy iterations (e.g., the total memory sent exceeds a threshold). After that, the migration enters into a *stop-and-copy* phase that transfers the remaining memory and disk data as well as CPU and device states.

In each iteration, the memory and disk data are processed on the source node and the destination node. For memory data, a *dirty bitmap* is usually maintained to keep track of dirty pages. The dirty memory pages are divided into multiple batches and are processed in turn. When processing each batch of memory pages, the migration tool first maps the guest VM memory pages into its own address space, and then handles specifically on unused pages and page table pages. Finally, these memory pages will be grouped together and sent to the destination. In the destination node (the *migrate_receive* function in Figure 1), the memory pages are first received and copied to the address space mapped from the target VM. The disk data is handled similarly and a dirty bitmap is maintained to keep track of dirty data.

Typically, the downtime of migration is directly affected by the rate of sending VM's data and the dirty rate of memory and disk data. If the migration rate is close or even less than the page dirty rate, the migration will be either very hard to converge (indicating a long migration time) or result in a huge amount of data to be transferred in the stop-and-copy phase (indicating a lengthy downtime).

2.2 An Analysis of Parallelism

Table 1 provides an overview of the associated parallelism, the estimated cost for the most important basic operations in a live VM migration operation and where they locate in Figure 1. There are mainly two types of parallelism: data parallelism and pipeline parallelism.

Data Parallelism: During migration, as there is no dependency among different portions of memory and disk data in the same iteration, several steps in migration have very good data parallelism, including resetting disk data into readonly, mapping guest VM memory pages into the address space of the management tool, processing unused pages and page table pages, transferring memory and disk data and restoring memory data. There are several cases where data parallelism is not appropriate. For example, getting the dirty bitmap and resetting memory into readonly are mainly done through invoking hypercalls [2]. As the time spent on these operations is not significant, it is not worthwhile parallelizing them.

Pipeline Parallelism: For some steps, if data parallelism is not appropriate, we can also leverage pipeline parallelism. For

[1] Domain0 is the management VM in Xen.

[2] Hypercalls are calls from a guest VM to the hypervisor.

Primitives	Parallelism		Send	Receive	Cost
	Data	Pipeline			
Get memory dirty bitmap	no	no	line 6	N/A	small
Reset memory readonly	no	no	line 7	N/A	small
Get disk dirty bitmap	no	no	line 6	N/A	small
Reset disk readonly	yes	yes	line 15	N/A	small
Check dirty bitmap	no	yes	line 8	N/A	small
Map guest VM memory	yes	yes	line 11	line 4	heavy
Handle Unused/PT Page	yes	yes	line 12	line 5	modest
Transfer memory data	yes	yes	line 16	line 2	heavy
Restore memory data	yes	yes	N/A	line 6	modest
Load/Save disk data	no	yes	line 14	line 8	heavy
Transfer disk data	yes	yes	line 16	line 2	heavy
Migrate CPU/Device States	no	no	line 17	line 10	small

Table 1. A summary of parallelism in different steps of live VM migration.

```
1  migrate_send(...)
2     while (!iteration_end)
3       if (judge_iteration_end())
4         pause_VM();
5         iteration_end = 1;
6       get_dirty_bitmap();  //memory and disk
7       reset_memory_readonly();
8       check_dirty_bitmap();
9       for (num_of_batches) //memory and disk
10        if (memory_data)
11          map_guest_vm_memory();
12          handle_unused_page();
            handle_pagetable_page();
13        if (disk_data)
14          load_disk_data();
15          reset_disk_block_readonly();
16        transfer_data();
17      transfer_cpudev_state();
```

```
1  migrate_receive(...)
2     while (receive_data())
3       if (memory_data)
4         map_guest_vm_memory();
5         handle_unusedpg_ptpage();
6         restore_memory_data();
7       if (disk_data)
8         restore_disk_data();
9       if (cpudev_state())
10        restore_cpudev_state();
11    resume_vm();
```

Figure 1. An overview of basic steps in live VM migration.

example, though it is not easy to apply data-parallelism to *check dirty bitmap*, we can partition it into a number of stages and feed the intermediate results to the next stage. Similarly, loading disk data can hardly be parallelized using data parallelism as the read from disk is constrained by the I/O system call. However, we can parallelize it with disk data transfer in a pipeline manner.

To give a sense on how to partition each operation into different threads and stages, we have conducted a quantitative evaluation to measure the cost in major primitive operations. The evaluation is done by migrating a modest-sized VM with 16 GByte memory, 4 virtual CPUs and 16 GByte disk. During migration, we run a memcached server on the guest VM with modest workload. Table 2 shows the primitive metrics of live VM migration. The migration takes about 1592s with a downtime of 257s. It takes about 10 iterations and then it is forced to be migrated. 58.8 GByte memory data was sent in total. In the example, about 29.9% of total time is spent on mapping memory of the guest VM, while the rest of time are spent mainly on data transferring through an SSH connection. The single thread migration process can only reach a network throughput of 37.7 MByte/s with an average CPU utilization of more than 95%. The high CPU utilization is due to the high cost

	Send	Receive
Total migration time	1592.0s	
Downtime	257.0s	
Get/Reset memory dirty bitmap	0.59s	N/A
Map guest VM memory	381.0s	571.1s
Handle Unused/PT Page (send)	44.8s	N/A
Transfer memory data	1,200.5s	979.1s
Handle/restore memory data (receive)	N/A	31.7s
Migrate CPU/Dev States	8.84ms	
# of iterations	10	
total memory sent	58.6 GByte	
Last iter memory size	9.3 GByte	
Avg. Network Cost	37.7 MByte/s	
Avg. CPU usage	95.4%	

Table 2. Costs of primitives in live migrating a modest-sized VM.

in mapping the guest VM memory, which includes acquiring a virtual address area for holding the guest memory, getting grants of accessing the guest pages and building up the page table through hypercalls as well as cleaning up the page table after processing the memory. It also includes high CPU consumption in encrypting and checksumming the data before transferring it. Thus, the key to parallelize live VM migration is to spawn multiple threads to leverage multiple cores to handle such tasks.

3. Parallelizing Live VM Migration

Based on our study on the sources of parallelism inside the live VM migration operation, this section describes how PMigrate leverages multiple cores and NICs to parallelize it.

3.1 Parallelizing Live VM Migration

Parallelizing live VM migration includes applying data parallelism and pipeline parallelism to most primitive operations shown in table 1. Figure 2 shows an overview of how live VM migration is parallelized. In the figure, the blue boxes mean the memory tasks, the pink boxes mean the disk tasks and the squiggly lines shows how data is moved. The stages inside the square are data parallelized and the connecting square are pipeline parallized with each other. The migration data is divided and assigned to a number of tasks and the tasks are processed in parallel.

Source Node: The migration process spawns a memory task producer to process memory data, as well as a disk task producer to process disk data. Depending on the amount of available CPU cores and NICs, several consumer threads are spawned to handle the

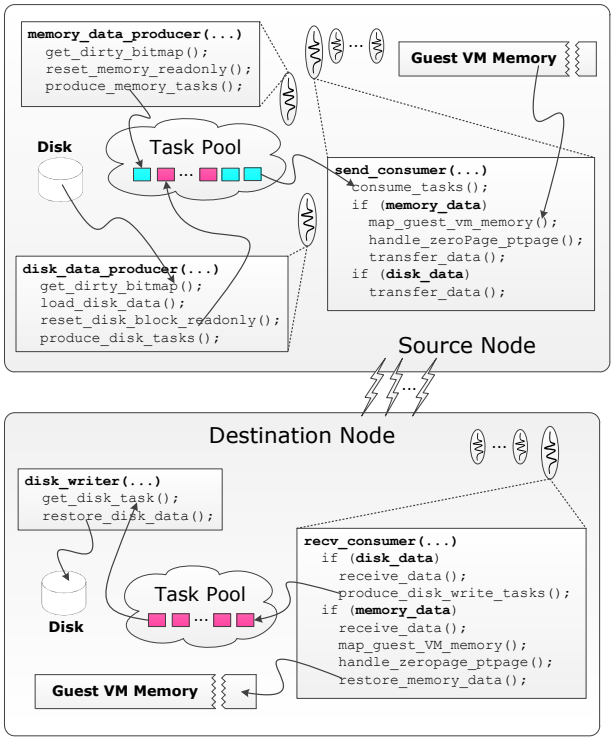

Figure 2. An overview of Parallel VM Migration.

tasks using data parallelism for the most time-consuming primitive operations. Handling a memory task includes mapping guest VM memory into the address space of the migration tool, handling unused pages and page table pages and sending out memory data. By contrast, handling a disk task just needs to simply send out the disk data.

We currently do not parallelize the disk producer thread as the parallelization may not gain enough benefit unless there are multiple virtual disks, as the disk I/O itself is serialized. Fortunately, we still can pipeline it with disk data sending.

Further, checking the *dirty bitmap* is done by the producers due to performance concerns. Other than the first iteration, memory and disk data to be sent in each iteration may not be contiguous, as a lot of pages and blocks may not be dirtied during the previous iteration. If the producers do not check the *dirty bitmap* before creating tasks, a lot of tasks will be created unnecessarily, resulting in sending a large amount of redundant data. As the cost of checking *dirty bitmap* is quite small, we place it in the producer threads and pipeline it with the consumer threads that send the memory and disk data.

There are also several primitive operations that are not appropriate for data or pipeline parallelism:

- *Get Dirty Bitmap:* Generating a separate *dirty bitmap* for each task is rather time consuming, as a task producer will usually generate thousands of tasks in live VM migration. It is more appropriate to do this at the beginning of each iteration.
- *Reset Memory to Readonly:* Resetting the access right of dirty pages into readonly for each memory batch will cause not only performance problem but also correctness problem. This is because repeatedly invoking the corresponding hypercall for each memory batch is not only very time consuming, but also may disturb the statistics in the *dirty bitmap*. This is because the access right for a memory page might have been changed before resetting its access right.
- *Handle CPU/Device States:* Loading and sending CPU and device states are done by a single thread as the basic cost is negligible. Parallelizing it may introduce cost instead of benefit.

Destination Node: The migration process spawns several consumer threads, each of which is responsible to handle the data batches sent from a corresponding consumer thread in the source node. Most of the work is done using data parallelism. Handling each memory data batch includes mapping guest VM memory pages; handling unused pages and page table pages; and restoring the data. By contrast, handling disk data only includes receiving disk data.

However, restoring data into disk image is not parallelized as it will ultimately be serialized by the disk itself. In this case, an additional disk thread is spawned to pipeline disk data restoring with disk data receiving. Finally, receiving and restoring CPU and device states and resuming the VM are done by the migration process itself.

3.2 Resource Usage Control

As the live VM migration tool needs to consume multiple resources that may also be needed by the production VMs, we try to control the amount of resources used by the tool to minimize disruption to other VMs, while use the spare resoures as much as possible.

Network Rate Control in Live Migration: In both the source node and the destination node, a network monitor daemon is spawned to collect the network usage of each NIC from the kernel. The destination daemon will periodically (1 second by default) negotiate with the source daemon to adjust the network bandwidth used by migration. In principle, the migration process will only consume the spare network bandwidth. The network bandwidth of each map consumer is adjusted by the daemon according to the network statistics of the corresponding NIC. To ensure a successful migration with certain requirement on migration time and downtime, a specific amount of bandwidth will be reserved by the migration process.

CPU Rate Control: There are lots of VMM scheduling strategies [12] to ensure the fairness and performance of VMs. The CPU usage of the migration process can be directly controlled by the VMM scheduler. By downgrading the execution priority of the migration process, we can limit its CPU consumption against the device backend daemons on a privileged VM. By upgrading its priority, we can devote more CPU resources into VM management tool. By default, its priority is set below the average.

Limiting Memory Consumption: As the time spent on different primitive operations varies and depends on the underlying execution environments (including software and hardware), there may be load imbalance between the producer and the consumer. Thus, the memory used to cache the data may become very large. To mitigate this problem, we maintain a memory pool for each pipeline stage. The memory buffers used to cache the "intermediate data" are directly allocated from this pool. The memory pool only contains a specific amount of memory (e.g., 128 MByte). If it is empty, the following allocation requests will be blocked. Thus, the total memory consumption can be limited.

4. Scaling Mutation to Kernel Address Space

After applying data and pipeline parallelism to the live VM migration, there is a serious kernel serialization: the frequent mmap/munmap operations from multiple threads require concurrent mutation to the address space of the management tool. This is unfortunately completely serialized by most existing OS kernel. For example, Linux and Solaris use a single read/write lock

and FreeBSD uses a single write lock, which serializes the address space mutation. Windows prior Windows 7 similarly use a system-wide PFN lock. Though Windows 7 does not publish its implementation details, our evaluation with a concurrent mmap microbenchmark on a 4-core machine shows that it may have a similar scalability issue[3].

This section first illustrates and analyzes the problem and then describes the *range lock* abstraction to mitigate it.

4.1 Serialized Address Space Mutation

The live VM migration operation needs to frequent memory mapping operations that map the guest VM memory into the address space of the VM management tool and unmap the memory after reading or writing the corresponding data. Figure 3 shows the main body of the VM migration algorithms. When being parallelized, each thread will : 1) first use the *mmap* syscall to allocate a new virtual memory area (VMA) (line 5) ; 2) then use an *ioctl* syscall to map the guest VM memory into the address space of the management tool (line 7-9); and 3) finally *munmap* the area when the data has been processed (line 13).

Unfortunately, Linux, like other similar operating systems, protects the per-process address space using a single lock (called *mmap_sem* in Linux). All the three steps are treated as mutation to the address space by Linux, which should be mutually exclusive by acquiring the *mmap_sem* in write mode for all related operations. Worse even, mutating the process address space such as mapping the guest VM pages in *ioctl* and cleaning the mapping of the guest pages in *munmap* is pretty time-consuming, as it requires further calls to the hypervisor. Specifically, when mapping the guest VM pages using *ioctl*, the privileged VM has to build up the page table through hypercalls for each guest page. As the guest page does not belong to the privileged VM, such map operations (namely, the foreign page map) are rather costly. This significantly enlarges the critical section in the live VM migration operation.

To give a sense of how severe will the contention be, we profiled the execution time of the live VM migration operation and found that more than 8.91%, 22.94% and 16.08% of total execution time are spent within the *mmap_sem*. With a number of concurrent threads, the time spent on the critical section will be accumulated, which significantly degrade the performance.

One intuitive solution is to increase the memory chunk size processed in each thread, which may reduce the frequency of issuing the address space mutation operations. However, this will increase the size of the critical section for these operations, which, however, may even further exacerbate the contention on the critical section. Another approach to mitigating the contention would be using multiple processes instead of threads to parallelize the VM operations. Unfortunately, the relative complex task dispatching and synchronization make such an approach less appealing.

Read Protection of Guest VM Mapping: After a close inspection on the routine of mapping guest VM pages in the VM management tool, we find that holding the *mmap_sem* semaphore in write mode is too costly and largely unnecessary, as it only protects a private Xen-specific field in the VMA, which stores the grant mapping information. [4] Figure 4 shows how the guest VM memory is mapped. First, the *mmap_sem* is hold in write mode (line 3). Then the virtual memory area containing the requesting address is searched (line 4). After that, the required batch of guest

```
1  map_consumer(...)
2      ...
3      while (task = get_task()) {
4          ...
5          addr = mmap(NULL, batch*PAGE_SIZE,
                       prot, MAP_SHARED, fd, 0);
6          ioctlx.num = batch;
7          ioctlx.addr = addr;
8          ioctl(fd,
               IOCTL_PRIVCMD_MMAPBATCH_V2, &ioctlx);
9          ...
10         memory_process()
11         ...
12         munmap(addr, batch*PAGE_SIZE);
```

Figure 3. The main body of a parallel live VM migration operation.

```
1  privcmd_ioctl_mmap_batch(...)
2      ...
3      down_write(&mm->mmap_sem);
4      vma = find_vma(mm, m.addr);
5      ...
6      ret = traverse_pages(m.num,
                  sizeof(xen_pfn_t), &pagelist,
                  mmap_batch_fn, &state);
7      up_write(&mm->mmap_sem);
8      ...
```

Figure 4. The main body of mapping the guest VM memory.

VM memory is mapped by constructing the page table through the *mmap_batch_fn* call for each page (line 6). Finally, *mmap_sem* is released (line 7). As mapping guest VM memory (line 4 and 6) does not modify other virtual memory areas, the *mmap_sem* can be hold in read mode with a fine-grained lock to protect the field.

4.2 Range Lock

After changing the protection of mapping guest VM pages from write mode to read mode, there is still serious contention as the mutation to an address space is serialized. Actually, this is not a specific problem to PMigrate, but a general problem to most operating systems that use a per-process lock to serialize concurrent mutation to an address space. Though Clements et al. [14] have demonstrated the effectiveness in parallelizing the process of read accesses to address space (i.e., page faults) with write accesses (e.g., mmap) using a RCU balanced tree, the mutation to the address space like mmap/munmap still requires acquiring the *mmap_sem* in write mode. Hence, there is still only one mmap/munmap operation can proceed at a time.

To address this problem, an intuitive approach would be decomposing the per-process semaphore into a number of fine-grained locks that protect only the requesting ranges of an address space. However, the requesting range of a mmap/munmap system call is usually dynamic and unpredictable. Further, the requesting ranges from different requests may overlap. Hence, it is impossible to use a set of predefined fixed-size locks, which are also pretty costly in terms of space and execution time. To this end, we propose a dynamic lock-service to the address space, which is called *range lock*.

Range lock leverages a skip list [25] that dynamically maintains the address ranges that are currently locked. To acquire a range lock to a specific range, the *range_lock* function searches the skip list using the requesting address range ([start, start+len]). If there is already an existing/overlapping range in the skip list, another thread should be mutating the specific range and the requesting thread should wait and retry. Otherwise, a range will be added to the skip list, indicating that the range lock is granted. To release a range lock, the *range_unlock* function uses the requesting address to search the skip list and deletes the corresponding range in the list. The reason why we use a skip list is because both *range_lock* and

[3] In the microbenchmark, each thread mmaps a 4 MByte memory, touches the memory (thus causing page faults) and unmaps the memory. Our evaulation results in a 4-core machine with Windows 7 (x64_sp1) show that the execution time using 4 cores increases from 765ms in 1 core to 2,684ms in 4 core.

[4] Grant map is a mechanism used by Xen to share memory between VMs.

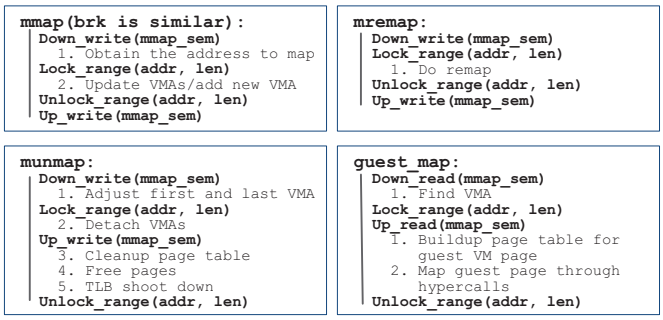

Figure 5. How range lock is integrated to Linux using hierarchical locking.

range_unlock requires intensive searching for existing/overlapping ranges and other data structures storing ranges such as *interval tree* and *segment tree* are a little bit too heavyweight for our purpose.

Ideally, it would be better to use a concurrent skip list to minimize the critical section. However, as typical concurrent skip list only supports either lookup or concurrent changes to one element in one operation, while *range lock* requires first looking up the range and then updates two elements (i.e., start, start + len) in the skip list. It would require non-trivial complexity and/or overhead to make the skip list concurrent. Hence, the skip list is currently protected by a spinlock. As the critical section of the range lock routines are very short, this spinlock does not become a new bottleneck in our evaluation.

Ideally, the entire address space can be protected in a fine-grained manner using *range lock*, where all accesses to different portions of the address space can be completely parallelized. In practice, there are a number of memory states that should be maintained consistently for an OS kernel. Further, current Linux kernel still uses a red/black tree like data structure to maintain both virtual memory areas and memory states, completely replacing it into a new data structure will be very resource-intensive due to a number of other correlated data structures such as *reverse map* and *mmap_cache* [14]. As a result, PMigrate currently only uses *range lock* to parallelize the time-consuming parts inside the mmap/munmap/foreign map calls by removing them from the critical sections protected by the *mmap_mem* and protecting them by the *Range Lock*s, while still leaving the original data structure intact. This makes the related changes to Linux relatively small, yet still brings notable performance improvement. For example, the current *range lock* implementation comprises of 290 SLOCs and the related change to Linux is only with 120 SLOCs. The implementation is stable enough that it passed Linux Test Project [6].

Specifically, as shown in Figure 5, we still acquire the *mmap_sem* in *mmap* but additionally also acquires the corresponding *range lock*, as most operations in *mmap* just update the red/black tree for an address space, which is not time consuming. For *munmap*, we keep the *mmap_sem* in write mode for the portions of updating red/black tree. However, for the rest code in *munmap* such as clearing page table entries and free pages, freeing unused page table pages and shooting down local and remote TLBs, we release the *mmap_sem* but still keep the corresponding range lock. Similarly, in *guest_map*, we use both *mmap_sem* in read mode and the range lock to protect the VMA lookup (line 4 in Figure 4) but only use the range lock to protect the rest of execution. As the mutation and lookup of VMA tree usually only consists of a small portion of execution time, applying the range lock significantly increases the parallelism and boosts the performance of mutation-intensive workloads, as shown in section 6.

5. Implementation

To demonstrate the applicability of our parallelized scheme, we have implemented PMigrate based on Xen, a popular open-source VMM that has been used in many cloud platforms such as Amazon EC2, OpenNebula and OpenStack. To demonstrate the wide applicability of PMigrate, we further port PMigrate to KVM, a hosted-mode VMM that is also widely deployed in many cloud platforms.

Implementation on Xen: The current Xen VMM only supports live migration of memory and CPU states. We have parallelized live VM migration, namely PMigrate-Xen. The current PMigrate system is built by extending the vanilla Xen tools (version 4.1.2) by parallelizing most operations as shown in section 3 with the address space optimization mentioned in section 4. The overall implementation adds/changes 1,860 SLOCs to Xen tools and the privileged VM (i.e., Domain0) kernel.

Implementation on KVM: KVM supports live VM migration with both memory and persistent storage. The migration request is handled by the I/O thread which is also responsible to handle the I/O operations related to the guest VM. In vanilla KVM, the I/O thread uses an event-driven mode to handle both the migration process and guest I/O requests with time-slicing. In each migration time slice, namely one iteration, the I/O thread only processes a small chunk of data according to the rate limit. Then it will switch to handle the I/O requests from the guest VM and wait for the next migration time slice. Such iteration strategy spends too much time on generating *dirty bitmap* of memory and disk images and resetting the access right of dirty pages into readonly. It also results in notable performance disruption on the I/O intensive guest VMs as we will show in Section 6.2.2. Thus, we change the KVM's iteration strategy into image-oriented as the PMigrate-Xen, which sends the entire VM image in the first iteration and sends the dirtied data in the following iterations. Further, to keep the migration alive, in PMigrate-KVM, we let the I/O thread spawn a new migration thread to handle the migration task. Otherwise, I/O thread will be monopolized by the migration task as handling one image iteration is very time-consuming.

In the source node, the migration thread spawns a memory task producer and a disk task producer to prepare the memory and disk data. Several sender threads are spawned to handle the memory and disk tasks. One problem in preparing disk data is that the vanilla-KVM uses asynchronous I/O (AIO) operations to issue read request to disk data and let the I/O thread to handle the AIO completion notification. The I/O thread handles the notification when it is monitoring the guest VM I/O or when it is processing the disk data in migration. However, in PMigrate-KVM, the disk producer will process the disk data in migration and it executes with the I/O thread in parallel, resulting in a race condition in AIO processing. To handle this problem, we change the AIO operations into synchronized I/O operations. Using synchronized I/O will not block the migration process, as the disk task producer executes in parallel with the consumer threads.

In the destination node, the migration thread spawns several receiver threads to handle the memory data and receive the disk data. It also spawns a disk writer thread to restore disk data into the disk image.

The overall implementation takes about 2,270 SLOCs, which includes 830 SLOCs to change its iteration strategy into image-oriented.

6. Evaluation

This section evaluates the effectiveness of parallelizing live VM migration operation.

	Vanilla	PMigrate
Migration Time (sec)	422.8(3.6)	112.4(3.9)
Downtime (millisec)	310.0(11.5)	408.0(12.5)
# of Pre-copy Iterations	5.7(1.2)	4.3(0.6)
Total Memory Send (GByte)	16.2(0.0)	16.2 (0.0)
Downtime Send (MByte)	1.8(0.4)	1.7(0.2)
Avg. Network Cost (MByte/s)	39.3(0.3)	148.0(5.0)
Avg. CPU usage	89.6(0.5)%	364.9(14.0)%
Total CPU Cost (CPU-sec)	378.9(2.0)	409.9(1.9)

Table 3. Key metrics (with STDEV) of live migrating an idle VM on vanilla Xen and PMigrate-Xen.

	Vanilla	PMigrate
Migration Time (sec)	473.6(5.7)	115.8(10.8)
Downtime (millisec)	1,420.6(166.5)	746.6(74.2)
# of Pre-copy Iter.	29(0)	29(0)
Total Memory Send (GByte)	16.4(0.1)	16.8(0.03)
Downtime Send (MByte)	39.6(6.1)	28.4(9.0)
Avg. Network Cost (MByte/s)	35.4(0.31)	149.7(14.4)
Avg. CPU usage	82.8(2.1)%	359.7(46.5)%
Total CPU Cost (CPU-sec)	497.3	525.9
Avg. PostgreSQL Thr. (Trans/s)	497.3(3.6)	416.4(12.5)
Avg. Thr. Degrade	4.4%	20.31%

Table 4. Key metrics (with STDEV) of live migrating a PostgreSQL server VM on vanilla Xen and PMigrate-Xen.

6.1 Experimental Setup

All experiments were conducted on two Intel machines, each of which is with two 1.87 GHz Six-Core Intel Xeon E7 chips. Each core has a separate 32 KByte L1 data cache and 256 KByte L2 cache and each chip has a shared 18 MByte L3 cache. The size of physical memory is 32 GByte. Each machine is equipped with a quad-port Intel 82576 Gigabit Network Controller and a quad-port on-board Broadcom Gigabit Network Controller. We use another Intel machine as the NFS server to provide the shared global storage for guest VMs on Xen. We use an Intel machine as the client machine. It is with four 2.00 GHz Ten-Core Intel Xeon E7 chips and one quad-port Intel 82576 Gigabit NIC . All machines were connected to a subnet through a Gigabit switch. The maximum throughput of a single network connection is 117.6 MByte/s with the round trip time be 0.076ms, while the maximum throughput of an SSH connection is 48.6 MByte/s.

We evaluate the performance of PMigrate for both Xen and KVM. We use Debian GNU/Linux 6.0, Xen 4.1.2 and the management VM with Linux kernel version 3.2.6. The KVM version is kvm-qemu 0.14.0 with the host VM kernel version 3.2.6. The guest VM is launched using hardware-assisted virtualization technologies [1] with 16 GByte memory in total and a 16 GByte disk image running Debian GNU/Linux 6.0. All tests were conducted with five or six times with a very low variability and we report the average as well as the standard deviation.

6.2 Performance and Scalability of Parallelization

6.2.1 Parallelized Live VM migration in Xen

We use two widely-used applications, memcached [18] and PostgreSQL 9.1.2 [7], as well as an idle VM to evaluate the performance of memory-intensive, CPU-intensive and idle VMs for live VM migration. We spawn 8 consumer threads and set no limit on the maximum network bandwidth for each thread. The connections are with the SSH-style connection, which is the same as the vanilla Xen. The Intel NIC is used by migration, while the Broadcom NIC is used by the guest VM through an emulated network device. To minimize impact to the VM workloads, we separate the physical CPU set so that the migration process can be scheduled on a different core set from the production VMs. We let each two consumer threads share one port of the NIC.

Idle VM: Table 3 compares the basic metrics of live migrating an idle VM. It can be seen that the migration time is reduced by 3.76X, as the average data transferring throughput is increased by 3.76X. Although the average CPU utilization of PMigrate is increased by 4.07X, the total CPU cost is increased by only 8.2%.

PostgreSQL PostgreSQL [7] is a wide-used SQL server. In the evaluation, we use pgbench as the client to generate SQL requests using the workload from TPC-B [2]. The target database contains 5,000,000 accounts and the workload is generated using 32 concurrent connections through 8 threads. PostgreSQL is CPU-intensive and will generate moderate disk I/O workloads. However, the dirty rate of memory is low.

Table 4 compares the key metrics of live migrating a VM with PostgreSQL. It can be seen that the migration time is reduced by 4.09X and the total downtime is also reduced by 1.90X. The performance improvement is mainly due to the increased average migration network throughput (4.23X), as more CPU cores means more data will be available to be sent out. However, the accumulated CPU and network cost are not increased even if more resources are devoted into parallelized live migration. The performance degradation of PostgreSQL server on PMigrate-Xen is about 16% larger than that on vanilla Xen. It is mainly due to the side effect of the PMigrate threads on the NFS driver.

Memcached Memcached [18] caches multiple key/value pairs in memory. Each time the server receives a request containing a key, it will respond with the corresponding value. We use the memaslap testsuite from the libmemcached library [5] as the memcached client. The client first warms up the memcached server with multiple key/value pairs to fill the memory and then randomly issues get operations through 4 concurrent connections from 4 threads. The workload of memcached is both memory and network intensive.

Table 5 compares the key metrics of live migrating a memcached VM. It can be seen that the migration time is reduced by 9.88X. The tremendous performance improvement is due to two major reasons: 1) the average migration network throughput is increased by 3.8X; and 2) As the data processing and transferring speed is significantly increased, the execution time of each pre-copy iteration is also reduced, resulting in much less data being dirtied. As shown in the table, the total memory sent is reduced by 2.58X. Further, the total downtime is reduced by 279.89X. As the data migration speed on vanilla Xen is not faster than the memory dirty speed of the memcached server, the last migration iteration (which is offline) is mandated after several pre-copy iterations. As a result, a total of 9.2 GByte memory is transferred during the downtime. On the other side, as the data migration speed on PMigrate-Xen is much faster, the total amount of data to be sent during the downtime is greatly reduced to only 0.02 GByte after 29 pre-copy iterations. Thus, it is no surprise the total downtime on PMigrate-Xen is much shorter than that on vanilla Xen. Through PMigrate-Xen devotes more resources into parallelized live migration, the accumulated CPU and network cost is not increased, but instead reduced by 2.40X and 2.59X, respectively. The throughput of memcached on vanilla Xen is 17.4 MByte/s with a total of nonresponse time of 251.9s. While the throughput of it on PMigrate-Xen is 15.3 MByte/s with a total of non-responsive time less than 2.7s. Though the performance degradation of the server on PMigrate-Xen is about 9% larger than that on vanilla Xen, the overall negative performance impact is notably reduced as the migration time and downtime is significantly reduced.

	Vanilla	PMigrate
Migration Time (sec)	1,586.1(13.7)	160.5(2.2)
Downtime (sec)	251.9(12.1)	0.9(0.03)
Non-response Time (sec)	$\approx$ 253.0(11.4)	$<$2.7(0.6)
# of Pre-copy Iterations	9.7(0.6)	29(0)
Total Memory Send (GByte)	58.6(0.03)	22.7(0.5)
Downtime Send (GByte)	9.2(0.3)	0.04(0.0)
Avg. Network thr. (MByte/s)	38.0(0.3)	145.0(4.3)
Avg. CPU usage	95.5(1.5)%	392.6(18.3)%
Total CPU Cost (CPU-sec)	1,514.1	629.9
Avg. Memcached Thr. (MByte/s)	17.4(0.9)	15.3(0.5)
Avg. Thr. Degradation	25.5%	34.6%

Table 5. Key metrics (with STDEV) of live migrating a memcached server VM on vanilla Xen and PMigrate-Xen.

	Vanilla	PMigrate
Migration Time (sec)	203.9(8.6)	57.4(6.1)
Downtime (millisec)	630.7(59.5)	15.8(1.7)
# of Pre-copy Iterations	9,735.7(314.6)	1(0)
Downtime send (MByte)	10.1(5.7)	2.1(0.4)
Total Memory Send (MByte)	396.8(7.2)	395.6(6.5)
Total Disk Send (GByte)	16(0.0)	16(0.0)
Avg. Network Thr. (MByte/s)	82.4(3.6)	294.7(32.0)

Table 6. Key metrics (with STDEV) of live migrating an idle VM on vanilla KVM and PMigrate-KVM.

6.2.2 Parallelized Live Migration of KVM

We also use an idle VM and a VM running memcached [18] to evaluate the performance of live migrating an idle and memory-intensive VM in KVM. As KVM further supports disk data migration, we further use DBench [3], a well-known file-system benchmark, to evaluate the performance of live migrating a I/O intensive VM. We spawn 4 consumer threads, and set no limit on the maximum network bandwidth for each thread. The connections are through the Intel NIC with each thread using one port, where the Broadcom NIC is used by the guest VM through an emulated network device. In both PMigrate-KVM and vanilla KVM, each zero-page is compressed into 1 byte. We do not show the CPU usage because a guest VM execute as a process in the host OS, which makes the CPU statistic inaccurate.

Idle VM: Table 6 compares the basic metrics of live migrating an idle VM. It can be seen that the migration time is reduced by 3.55X as the average data transferring throughput is increased by 3.58X. The total number of iterations are greatly reduced, as PMigrate-KVM changes the iteration strategy into image-oriented[5], which reduces the accumulated cost on preparing each iteration.

Memcached: Table 7 compares the key metrics of live migrating a memcached VM. The migration time is reduced by 2.49X and the downtime is reduced by 4.81X. The performance improvement is mainly from the increased migration network throughput due to parallelization. Further, as the vanilla KVM uses the I/O thread to process both the network requests of live migration a guest VM, the network I/O of the guest VM is significantly disrupted. The average throughput is reduced by 86.8% during migration on vanilla KVM. In contrast, the throughput is reduced by only 5.05% on PMigrate-KVM. Further, on vanilla KVM, the throughput of the memcached server drops to only 0.22 MByte/s after the migration has been started for 185s and it lasts about 163.0s, which means that the server has nearly no response to the clients for a long period. As it takes nearly no workload during migration on vanilla KVM, it has less memory dirtied during memory precopy than that on PMigrate-KVM.

[5] For vanilla KVM, each pre-copy iteration only transfers a specific number memory or disk data (e.g., 2MByte), which results in a high pre-copy interation count.

	Vanilla	PMigrate
Migration Time (sec)	348.7(7.1)	140.2(6.5)
Downtime (millisec)	553.7(69.8)	115.1(65.2)
Non-response Time (sec)	$\approx$ 163.0 (3.6)	$<$1.0 (0.0)
# of Pre-copy Iterations	16,776.7(3.6)	5.3(0.6)
Downtime Send (MByte)	37.8(12.9)	32.5(20.9)
Total Memory Send (GByte)	19.1(4.9)	23.5(1.5)
Total Disk Send (GByte)	16.2(0.3)	16.0(0.01)
Avg. Network Cost (MByte/s)	90.7(2.4)	289.1(13.5)
Avg. Memcached Thr. (MByte/s)	2.3(0.3)	15.9(2.1)
Avg. Thr. Degradation	86.8%	8.36%

Table 7. Key metrics (with STDEV) of live migrating a memcached VM on vanilla KVM and PMigrate-KVM.

	Vanilla	PMigrate
Migration Time (sec)	256.1(20.0)	77.1(13.6)
Downtime (millisec)	455.7(62.2)	102.9(12.1)
# of Pre-copy Iterations	12,159.0(677.2)	3.7(1.2)
Downtime Send (MByte)	33.3(5.9)	37.4(12.8)
Total Memory Send (MByte)	603.8(45.3)	690.9(164.4)
Total Disk Send (GByte)	19.8(1.0)	17.2(0.4)
Avg. Network Cost (MByte/s)	81.3(2.5)	242.1(7.0)
Avg. DBench Thr. (MByte/s)	963.9(11.8)	974.3(38.9)
Avg. Thr. Degradation	6.05%	4.72%

Table 8. Key metrics (with STDEV) of live migrating a DBench VM on vanilla KVM and PMigrate-KVM.

DBench: DBench is a file-system benchmark that will generate modest workload on the file system, which will result in data write into the disk. Table 8 compares the key metrics of live migrating a DBench VM. The migration time is reduced by 3.32X. The downtime is reduced by 4.43X. The performance improvement is mainly due to the increased migration network throughput and the reduced amount of disk data (about 2.6 GByte) being transferred as less data are dirtied within a shorter pre-copy time. PMigrate-KVM reaches a lower network throughput (compared to the Idle VM case and the memcached VM case) due to disk throttling in the source node as the migration process will contend with DBench VM on disk accesses.

6.2.3 Sources of Speedup and Scalability

Scalability and contribution of *range lock*: We spawns 1, 2, 4 and 8 consumer threads to do migration and restrict maximum network bandwidth for each thread to be 40 MByte/s as this is the maximum network throughput the vanilla Xen migration process can achieve. As PMigrate-KVM scales well on our 12-core Intel machines, we omit its scalability evaluation in this section.

Figure 6 and Figure 7 show the scalability of migrating the idle VM and the memcached VM respectively, by using PMigrate without optimization, PMigrate with the read-lock optimization and PMigrate with the range lock optimization. The range lock optimization significantly boosts the performance, as it significantly reduces the size of the critical sections of the address space management operations protected by the *mmap_sem*. Table 9 shows the average cost of each single memory management operation used in live VM migration, which confirms the performance benefit from read lock and range lock. After introducing the read lock, the time spent on mmap and munmap can be reduced. After introducing the

range lock, the time spent on mmap and munmap are significantly reduced.

Performance of only Using Multiple Cores: To show whether PMigrate-Xen can work well with a single NIC port, we run PMigrate-Xen using an idle VM with 4 consumer threads sharing the same NIC port. The total migration time is 148.4s, which is quite close to that with four cores and four NIC ports (149.3s). this shows that PMigrate still performs reasonably well without an excessive number of NICs or bandwidth. Actually, the major benefit lies in using multiple cores to process and prepare the data to be sent. However, the average data transferring throughput of 112.1 MByte/s, which almost saturates the port. Here, adding additional NICs can further improve the performance if more cores are used for migration.

Performance of only using Multiple NICs: We also evaluate whether using multiple NICs along can provide reasonably good speedup. We assign four NIC ports to PMigrate-Xen and bind 4 threads to a single core. The total migration time is 377.28s, with an average throughput of 49.32 MB/s. The result indicates that using multiple NICs along can have very little performance benefit as there is not enough data prepared by CPUs to be sent through NICs.

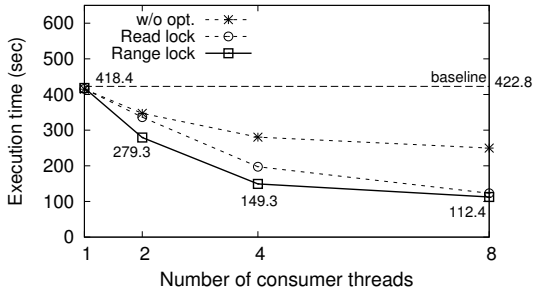

Figure 6. PMigrate Scalability of migrating idle VM on Xen using PMigrate without optimization, PMigrate with read lock and PMigrate with range lock.

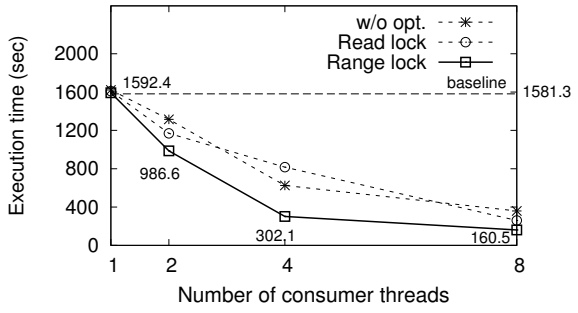

Figure 7. PMigrate Scalability of migrating memcached VM on Xen using PMigrate without optimization, PMigrate with read lock and PMigrate with range lock.

6.3 Impact on Other Co-located VMs

Ideally, if the resources of a virtual machine are abundant, the migration tool can be assigned with separated resources to the production VMs. In such cases, the performance impact of PMigrate on other VMs can be minimized. To evaluate the effectiveness of PMigrate in leveraging spare resources, we run a Xen VM with memcached serving requests from clients and measure its throughput during the parallel migration of an idle Xen VM.

Time Cost	Idle			Memcached		
	no-opt	rd-lock	range	no-opt	rd-lock	range
mmap	1.39	1.80	0.05	1.10	1.26	0.18
guest map	18.1	16.3	14.8	16.8	15.5	15.2
munmap	7.27	6.22	0.75	5.57	5.30	0.85

Table 9. Average cost of a single memory management operation in live VM migration on Xen using PMigrate without optimization, PMigrate with read lock and PMigrate with range lock in microsecond.

Our evaluation found that during migration, the average throughput degraded from 42 MByte/s to 34.6 MByte/s. By contrast, the original Xen tools cause a performance degradation from 42 MByte/s to 38 MByte/s. This performance gap is due to the fact that we leverage multiple cores and NICs to do scheduling and these threads may inference with each other. However, the parallelized migration significantly shortens the overall migration time. This indicates that the parallelization may even help to reduce the impact on other VMs as a whole.

However, if the resources in a virtualized platform are limited, the migration tool may cause performance degradation on other co-located VMs. To evaluate the effectiveness of our resource rate control, we illustrate the effectiveness of our network rate control mechanism and our CPU rate control mechanism.

Network Rate Control To evaluate the effectiveness of network rate control mechanism of PMigrate, we run a VM with Apache web server serving requests from clients and measure its throughput during the parallel migration of an idle VM. The migration process uses two NICs, one of which is shared with the apache web server. Before the migration starts, the throughput of the web server is 101.7 MByte/s. While during the migration, its throughput only drops to 91.1 MByte/s. The average network bandwidth of the shared NIC consumed by the parallel migration process is 17.6 MByte/s, while the bandwidth of another NIC consumed by it is 57.2 MByte/s. The migration process does limit its network consumption of the busy NIC shared with other VMs.

CPU Rate Control To evaluate the effectiveness of CPU rate control of PMigrate, we run a VM with a memcached server serving requests from clients and measure its throughput during the parallel migration of an idle VM. The VM running memcached with four virtual CPUs (VCPU) scheduling on 4 physical CPUs. The throughput of the memcached server is 48.4 MByte/s before migratoin and the CPU usage is above 100%. During parallel migration, we spawns 4 consumer threads. We compare the throughput of the memcached server VM under two conditions: 1) The vanilla Xen migration process shares a physical CPU of the memcached server VM during migration (for example, CPU2); and 2) The PMigrate-Xen process shares only three physical CPUs of the memcached server VM (for example, CPU2, CPU3 and CPU4). As the result, the total migration time is reduced from 131s to 41s. While the throughput of memcached during migration drops to 48.1% from the origin on vanilla Xen and 34.9% on PMigrate-Xen. It can be seen that PMigrate-Xen does not introduce much further performance impact to VMs sharing CPU compared to vanilla Xen. However, the parallelized migration significantly shortens the overall migration time, which can help to reduce the overall impact on other VMs.

6.4 Combining PMigrate with Compression

There have been several techniques [15, 23] on improving the performance of live VM migration, such as data compression. Here we show that techniques on improving the serial performance of live VM migration can also be applied to PMigrate by taking data compression as an example. We modify the vanilla KVM and

	Vanilla	PMigrate
Migration Time (sec)	114.7(0.1)	28.8(0.5)
Downtime (millisec)	45.3(0.6)	54.3(15.4)
# of Pre-copy Iterations	906.3(0.6)	1(0)
Total Memory Send (MByte)	122.8(1.2)	120.3(0.06)
Memory Compress Rate	28.0(0.0)%	28.1(0.0)%
Total Disk Send (MByte)	1692.0(2.9)	1693.8(0.6)
Disk Compress Rate	10.0(0.0)%	10.0(0.0)%
Total Compress Time (sec)	41.5(0.06)	10.8(0.06)

Table 10. Key metrics (with STDEV) of live migration with an idle VM on vanilla KVM and PMigrate-KVM with data compression.

PMigrate-KVM migration process to compress the memory data and disk data during migration using the quicklz [8] compression library. Figure 10 compares the basic metrics of live migrating an idle VM with 4 GByte memory and 16 GByte disk on vanilla KVM and PMigrate-KVM with 4 consumer threads. It can be seen that the migration time on both cases is reduced (compared to that shown in Table 6) due to the reduced amount of memory and disk data being transferred. The data compression rates on both cases are similar. The total compression time is significantly reduced on PMigrate-KVM as it takes more threads to process the data.

7. Related Work

Live VM Migration: Currently, most hypervisors have provided the support for live VM migration, such as VMWare [19, 22], Xen [13] and KVM [16]. Among these operations, VM migration has been extensively studied. Other than VM migration with only memory and within LAN, Bradford et al. [11] further extend Xen with the support of live VM migration with persistent states across wide-area network.

The increasing importance of live VM migration also stimulates interests in optimization. For example, though most VMMs uses pre-copy [26] VM migration by default for the sake of reliability, Hines et al. [15] propose a post-copy based migration methodology to reduce repetitive memory transfers under memory-intensive workloads. To save bandwidth, Svärd et al. [23] design and implement a memory delta compression technique to reduce memory copy during live VM migration. The storage VM migration solution in industry has also been evolved dramatically [19]. The SnowFlock [17] uses a set of technique to support fast application-aware VM clone, which leverages a post-copy clone policy. Using a post-copy policy may have the advantage of quickly creating a replicate VM and makes it alive. However, this suffers from lengthy post-copy migration time and a break in network collection may corrupt the target VM. Further, live VM clone usually requires specific OS support for state consistency.

However, none of the prior solutions have considered the parallelizing the process of VM migration. We believe the work of PMigrate can be integrated with most prior optimization, which may result in further performance improvements.

Scaling Operating Systems: There have been a number of studies on the scalability of commodity operating systems [10, 14]. Among them, the concurrent address space using RCU balanced tree [14] is the closest one. However, the RCU balance tree focuses on parallelizing the process of read accesses to an address space (e.g., page faults) and mutation to the address space (e.g., mmaps), while the *Range Lock* abstraction parallelizes the mutation to an address space. We believe these two abstractions can be combined together to further improve the scalability of address space, which will be our future work. *Range lock* shares some similarity with byte-range locking in distributed file systems. However, *Range lock* targets at the address space management and is much simpler than file locking. Further, we are the first to demonstrate that *Range Lock* can be easily integrated into existing Linux kernel with very little programming effort.

8. Conclusion

The increasing amount of resources configured to both physical and virtual machines created both challenges and opportunities. This paper made an attempt to parallelize the live VM migration. Based on a comprehensive analysis on the underlying parallelism, this paper leveraged both data and pipeline parallelism to parallelize live VM migration. To mitigate intensive contention on concurrent mutation to an address space, this paper further proposed a new abstraction in operation system, called *Range Lock*, which provided more fine-grained protection to concurrent mutation of an address space. Performance evaluation with two popular open-source VMMs showed that the parallelized version significantly boosted the performance of the live VM migration, yet with small disruption to running services.

9. Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported by an NetApp Faculty Fellowship, China National Natural Science Foundation under grant numbered 61003002, a grant from Shanghai Science and Technology Development Funds (No. 12QA1401700), a Foundation for the Author of National Excellent Doctoral Dissertation of PR China and Fundamental Research Funds for the Central Universities in China.

References

[1] Intel virtualization technology. http://www.intel.com/technology/virtualization/.

[2] TPC-B. http://www.tpc.org/tpcb/default.asp.

[3] Dbench. http://dbench.samba.org/.

[4] Instance Types of Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/#instance.

[5] LibMemcached. http://libmemcached.org/.

[6] Linux test project. http://ltp.sourceforge.net/.

[7] PostgreSQL. http://www.postgresql.org/.

[8] Quicklz. www.quicklz.com/.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, 2003.

[10] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, N. Zeldovich, et al. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[11] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual Execution Environments*, pages 169–179, 2007.

[12] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in Xen. *Performance Evaluation Review*, 35(2):42, 2007.

[13] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, 2005.

[14] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2012.

[15] M. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, pages 51–60, 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] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, 2009.

[18] R. LERNER. Memcached integration in rails. *Linux Journal*, 2009.

[19] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The design and evolution of live storage migration in VMware ESX. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[20] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32, 2007.

[21] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 265–278, 2007.

[22] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[23] J. T. Peter Svärd, Benoit Hudzia and E. Elmorth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceeding of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[24] M. Powell and B. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating System Principles*, 1983.

[25] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[26] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating Systems Principles*, 1985.