Time-Bound, Thread-Based Live Migration of Virtual Machines

Kasidit Chanchio
Department of Computer Science
Faculty of Science and Technology,
Thammasat University
Patumtani, THAILAND 12121
kasiditchanchio@gmail.com

Phithak Thaenkaew
Large Scale Simulation Research
National Electronics and Computer
Technology Center (NECTEC)
Pathumtani, THAILAND 12120
phithak.thaenkaew@gmail.com

Abstract*— Live migration of virtual machines is the ability to move running virtual machines between two computers with minimal downtime. Although various migration mechanisms such as pre-copy, post-copy, and state compression have been proposed, they may suffer long migration times when the migrating virtual machines run large computation and memory intensive workloads. This paper presents the design and implementation of a novel Time-bound, thread-based Live Migration (TLM) mechanism, where additional threads are added to the pre-copy live migration algorithm to handle virtual machine state transfers within a bounded time period. In the time-bound principle, the upper-bound migration time of a virtual machine is proportional to the size of the virtual machine's memory. We propose a CPU over-committing mechanism to minimize migration downtime and avoid performance impacts to other virtual machines when the migration threads are in operation. We have implemented a prototype implementation of TLM on KVM, and conducted experiments by migrating virtual machines running a number of Class D OpenMP and MPI NAS parallel benchmarks. Experimental results showed the following: (i) TLM finished live migration in a bounded time period. Users are able to measure progress of migration operation. (ii) The CPU over-committing mechanism can be used to minimize live migration downtime. However, communication performance of virtual machines during live migration also declined as the number of over-committed CPUs reduced. The patterns of decline depended on execution behaviors of the applications on the virtual machines. (iii) The execution time increases of the OpenMP and MPI versions of the MG and IS benchmarks in our experiments were approximately equal to the migration times of TLM. (iv) We evaluated our CPU over-committing mechanism against the auto-convergence mechanism recently developed in kvm-1.6. We found that both mechanisms have their pros and cons, and their performance results are varied with application. Based on these results, we believe that the TLM design is practical for live migration of virtual machines running memory-intensive workloads, and the time-bound principle is an important new feature for pre-copy live migration optimization.

Keywords-live migration, virtualization, cloud computing

I. INTRODUCTION

Cloud computing has become an increasingly common platform for high performance computing and large-scale data analysis applications. Cloud service providers have been offering large scale Virtual Machine (VM) instances for users. Google Compute Engine, for example, offers VMs with up to 8 CPU cores and 52 GB of memory in its "high-memory" instance type [1]. Amazon Web Services has been offering "CPU-optimized" and "memory-optimized" instances that may have up to 32 vcpus and 244 GiB of memory [2]. In data centers that offer such services, efficient VM migration is critical for resource utilization and fault-resiliency: The faster the migration, the better the adaptability of resource utilization. Fast VM migration can be used to quickly balance workloads among servers in a server pool, relocate VMs closer to data, and move VMs out of partially malfunctioning computers or away from disaster-affected areas.

Despite these advantages, developing a mechanism to efficiently migrate large, memory intensive VMs is a challenging task. Mainstream hypervisors such as Vmware, KVM, Xen, and Hyper-V implement a pre-copy live migration mechanism [3] that lacks an optimization feature to quickly relocate a VM running large, memory-intensive applications. In its default operation mode, the pre-copy mechanism may take a long time to migrate the memory-intensive VM. In many cases, the migration does not complete until after the applications finish. To alleviate the problem, hypervisors such as KVM allow users to customize certain migration parameters; however, determining the right parameter values for VMs with different workloads is not trivial. Using wrong migration parameters, live migration may take a long time. Recently, the idea has been introduced of using additional threads to increase VM migration capability while slowing down VM computation to reduce VM state size [4,5]. Our work also takes this approach. However, the design principles we propose are fundamentally different from those of existing live migration mechanisms.

This paper presents a novel pre-copy live migration mechanism, namely the *Time-bound thread-based Live Migration* (TLM) mechanism, based on the following new design principles: (i) VM migration must complete within a bounded period of time proportional to the size of VM memory, (ii) the migration mechanism is entitled to use appropriate amounts of computing and network resources to satisfy the first principle, and finally (iii) VM computation can be slowed down for migration optimization. These principles are fundamentally different from those of pre-copy migration and other approaches. In this paper, we define the migration time as the duration from the start of a VM migration to its completion, the downtime as a time period during which the migration operation stops the VM to allow data transfer to take place, and the live migration time as the difference between the migration time and the downtime.

Based on the first principle, additional threads are created in the hypervisor to handle live migration. On the source computer, TLM starts dirty page tracking and creates two threads at the beginning of the migration, to transfer VM state while the VM is running. The first TLM thread sequentially scans VM memory from first to last page and transmits memory pages that have *not* been modified by VM computation to the destination computer. On the other hand, the second thread is responsible for transmitting *dirty* pages. After the first thread finishes scanning VM memory, TLM enters the *downtime* stage. In this stage VM computation and its two threads are stopped, and the rest of the dirty pages and VM state are transferred. On the destination machine, TLM also creates two threads to receive VM state information from the source.

For the second principle, we propose a CPU over-committing mechanism to provide computing resources for these additional threads. In a multi-core VM, every CPU core is mapped to a set of physical CPU cores of the host. During migration, TLM reassigns every CPU of the VM to a subset of the original set, and frees up some CPU cores to be used by the additional threads created for live migration. This design can also help reduce the interference of the migration operation with the computation of other VMs on the same host.

In satisfying the third principle, we use the same CPU over-committing mechanism to minimize migration downtime. Since TLM forces migration completion after the first thread finishes scanning memory, a large number of dirty pages may remain to be transferred during the downtime stage. This mechanism aims to slow down VM computation to reduce the amount of dirty memory pages generated by the VM

We have integrated the TLM mechanism with qemu-kvm 1.0 [6, 17]. For the sake of brevity, we will refer to *kvm* instead of *qemu-kvm* throughout this paper. To evaluate TLM, we conducted a number of experiments migrating VMs running OpenMP and MPI versions of the NAS Parallel Benchmark (Class D) programs [7] between two server computers over a 10 Gbps network. In our experiments, we found that the migration time across every experiment was less than 120 seconds. Without downtime minimization, the downtimes varied from 1.33 to 25 seconds depending on the benchmark programs. With downtime minimization, TLM reduced downtime to less than 2 sec in most cases. In the experiment on the MG benchmark that produced a downtime of 25 seconds using TLM without optimization, this was reduced to below 10 seconds when optimization was applied. Experimental results have led us to the following findings:

First, TLM always completes the migration of a VM within a bounded time period (hereafter "time-bound"). Assuming that the two TLM threads run at full capacities and have the same priority to utilize computing and networking resources, the completion of VM migration under TLM depends on how fast the first thread scans the VM's memory. The progress of live migration can also be measured using the progress of the first TLM thread.

Second, TLM downtime minimization can significantly reduce migration downtime. The length of downtime depends on three factors: the amount of dirty pages generated during live migration, the memory update behavior of the VM, and the dirty page transfer bandwidth during the migration. The higher the dirty page generation rate, the longer the downtime. By over-committing VM CPUs to host CPUs, we can reduce the amount of dirty pages generated, and consequently downtime. In our experiment, we found that the downtime reduction patterns of each VM varied depending on the memory update behaviors of the benchmark program. Although CPU over-committing can effectively reduce migration downtime, the communication bandwidth capability of the migrating VM during live migration also reduces.

Third, we found that the execution time of applications increases, by an amount which varies with the application. These increases are due to two factors: delays caused by migration activities, and the execution behavior of the VM after migration. In our experiments, the execution time increases under the MG and IS benchmarks were dominated by the former and were generally low due to low TLM migration times.

Finally, we evaluated our CPU over-committing mechanism against the auto-convergence mechanism of kvm-1.6 [5]. The auto-convergence mechanism was developed at the same time we developed TLM and was recently released. It enables high-speed VM state transfer and automatically reduces computation speeds of VMs by repeatedly freezing the VM for a few milliseconds when live migration does not converge. From our evaluation, we found that that TLM migration with CPU over-committing achieved total migration times 0.35 to 0.69 times those of kvm-1.6 migration with auto-convergence on the MG, IS, and SP benchmarks. On the other hand, the auto-convergence achieved lower downtimes and better communication bandwidths than those of CPU over-committing on the MG benchmark. In summary, both mechanisms have their pros and cons, and their performance results varied with application.

This paper is organized as follows. Related works are discussed in Section 2. The TLM design and implementation are presented in Section 3. Section 4 describes experimental setups and experimental results. Finally, Section 5 concludes the paper and makes suggestions for future work.

II. RELATED WORKS

We believe that VM migration is a key mechanism that can significantly improve the efficiency and fault-resiliency of the cloud. Our beliefs are inspired by the theoretical foundation given in the work of Harchol-Balter and Downey [8], which conducts trace-driven simulation to demonstrate the benefits of preemptive migration over non-preemptive migration in distributed computing even when the memory transfer costs are high. However, efficient migration mechanisms are needed to realize such the distributed environments.

A. Pre-copy migration

The pre-copy VM migration mechanism was introduced in [3]. It has been implemented in mainstream hypervisors such as Xen, vmware, and KVM. This mechanism operates in three stages: the startup, pre-copy, and stop-and-copy stages. At the startup stage, the mechanism instructs the KVM kernel module to track modification to VM memory. The pre-copy stage repeatedly scans VM memory and performs several

iterations of memory transfer. For each iteration, the pre-copy mechanism periodically transfers a number of dirtied memory pages to destination. On each memory transfer, the hypervisor transmits a subset of dirty pages to destination in which the subset's size is calculated from the *maximum transfer rate limit* parameter value. At the end of the memory transfer, if the length of time to transmit remaining dirty pages *does not exceed* the *tolerable downtime* parameter value, the hypervisor enters the stop-and-copy stage, where the VM is stop and all remaining dirty pages as well as device state are transferred to the destination computer. After the transfer completes, the VM computation is resumed.

In case a VM has multiple SMP CPUs (vcpus), multi-threading is used in the hypervisor's implementation. The implementation of the kvm hypervisor consists of an io-thread and a number of computing threads, each representing a vcpu of a VM. These threads are scheduled to run on physical CPU cores of the host computer by Linux scheduler. In earlier versions of kvm, the io-thread is responsible for performing the migration operation and managing user commands. In kvm-1.4 [4] and later, an additional thread was added to specifically handle VM migration. This allows migration operation to perform faster and help relieve the migration task off the io-thread. We have also implemented this approach in our previous work [9]. We have evaluated the migration mechanisms of kvm-1.2 and kvm-1.6 in Section IV (A.1-2). We found that the migration times vary by different tolerable downtime values. With appropriate tolerable downtime parameter values, decent migration performances are obtained. However, identifying the right downtime parameter is not trivial. Live migration may take too long when the tolerable downtime value is not suitable for an exceedingly high rate of dirty page generation of an application.

This problem was resolved by the migration auto-convergence mechanism of kvm-1.6 [5]. In this mechanism, if migration does not converge, kvm-1.6 will slow down VM computation by repeatedly (every 40 msec) pausing each VM's vcpu for a short period of time (3 msec) until live migration completes. Kvm-1.6 determines a migration non-convergence by performing a series of checks to see if the amount of dirtied bytes significantly exceeds the amount of bytes transferred. If such a condition occurs more than 4 times, auto-convergence is performed. While kvm-1.6 uses the auto-convergence mechanism, TLM uses CPU over-committing to slow down VM computation. We evaluated the CPU over-committing against the auto-convergence mechanisms in Section IV.C.

B. Post-copy migration and other techniques

As opposed to pre-copy, Hine et al. proposed post-copy live migration [10] in which the processor state is sent to the destination to start the computation first and transfer memory pages to the destination machine later when page-faults occur. To minimize migration time, the source performs one-round memory scan and sends memory pages to the destination. Like TLM, the post-copy finishes the migration in a bounded time period. However, their memory transfer mechanisms are fundamentally different. The post-copy approach performs the memory scan and transfer after VM computation is switched over to the destination, while TLM does so before the switching over takes place. The main drawback of the post-copy is reliability due to residual dependency. If the destination fails before the migration completes, the VM computation on the source can resume. However, the VM computation on the destination would be lost. To alleviate the problem, the post-copy mechanism must transmit the updated VM state on the destination to the backup copy on the source regularly during its one-round memory scan and transfer from the source to destination. This reliability problem is not an issue for TLM and pre-copy migration. If failures on the destination occur, the source simple aborts the migration and lets VM computation continues.

The post-copy mechanism may suffer a long downtime due to page faults when migrating a VM running memory intensive applications. To reduce page faults, the guided-copy mechanism [11] extends post-copy by letting the VM on the source runs concurrently with the VM on the destination to determine an optimal sequence of memory pages to transfer to the destination. This scheme can significantly reduce the number of page faults on the destination VM. Thus, during live migration, the computation speed of the VM on the destination will depend on data delivery bandwidths.

On the other hand, since TLM is based on pre-copy, it attempts to transfer the memory efficiently during the live migration stage (see Section III). In case dirty page generation rate is highly greater than the network bandwidth between the source and destination, TLM and kvm-1.6 employs CPU over-committing and auto convergence mechanisms, respectively, to lower dirty page generation to fit the available network bandwidth.

There are a number of proposed migration optimization techniques based on pre-copy live migration. Ibrahim et al. [12] propose an on-line algorithm to optimize the pre-copy migration for VMs running computation and memory-intensive workloads. Their algorithm uses memory update information to automatically determine the tolerable downtime parameter. Wood et al. [13] propose a VM migration system over WAN that optimizes live migration by monitoring the number of pages sent and dirtied, and picking the optimal point (pre-copy iteration) to stop the migration. Svärd et al. [14] propose delta compression techniques to reduce the amount of data transfer for pre-copy live migration. The work in [13] also uses Content-based Redundancy and page deltas to reduce the amount of WAN data transfer. Compression requires computing powers especially for memory-intensive VMs. Song et al. [15] present a parallelizing live migration mechanism where a number of threads are created on the source and destination to transfer state information in parallel. Their implementation uses threads to improve data transfer as in TLM and kvm-1.6. Jo et al. [16] optimize pre-copy migration by tracking memory pages residing on shared storage to reduce the amount of data transfer during migration.

TLM is different from these pre-copy mechanisms. While the termination conditions of live migration in these works focus on finding the VM execution point that best satisfies the tolerable downtime, TLM requires live migration to complete within a bounded time period and performs "best effort" downtime minimization.

III. TIME-BOUND, THREAD-BASED LIVE MIGRATION

TLM consists of multiple threads running alongside the VM threads in the hypervisors at the source and destination computers of a migration as illustrated in Figure 1. The io-threads on both ends work as the controller of the migration operation. At the source, TLM creates the memory page transmitter (**mtx**) thread and dirty page transmitter (**dtx**) thread. At the destination, TLM creates the dirty page receiver (**drx**) thread. We assume that before a migration takes place, a destination VM is already running and waiting for state transfer on the destination machine, and that both the source and destination VM access the same set of VM disk images on a shared file system.

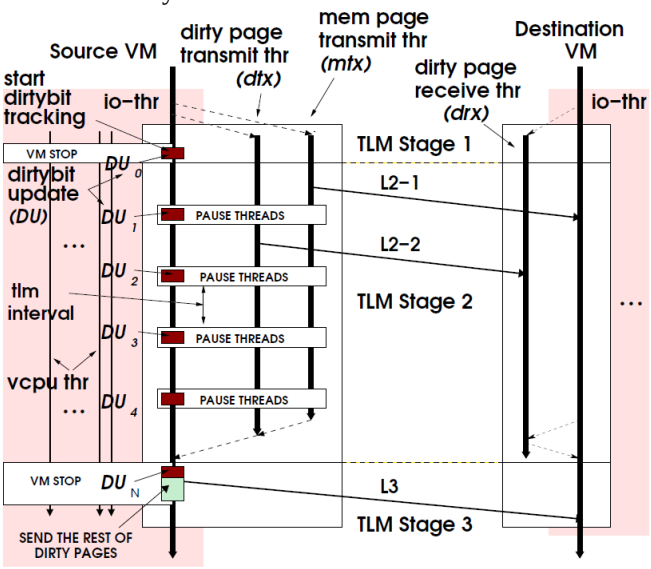


Figure 1. TLM Design

The TLM mechanism can be described in three stages as shown in Figure 1. At the first stage, the io-thread of the source VM creates the **mtx** and **dtx** threads. After that, it creates TLM dirty bit array, initializes every array element to a **clear** status, and instructs the KVM kernel module to start dirty bit tracking mechanism. Every data element of the array represents TLM's record of the past modification of each memory page. A value in each element is either **dirty** or **clear**. The **mtx** and **dtx** also create two TCP connections (L2-1 and L2-2 in Figure 1) with the io-thread and **drx** threads on the destination respectively. All vcpu threads of source VM are paused during Stage 1. At Stage 2, the vcpus and io-thread of the source VM resume execution. At the same time, the **mtx** and **dtx** proceed. The **mtx** thread is responsible for sending memory pages 0 to the *last memory page* of the VM sequentially via the L2-1 link, while the **dtx** thread transmits newly generated dirty pages over the L2-2 link.

In order to decide which memory pages are dirty, TLM lets the io-thread perform a *Dirty bit Update (DU)* operation at the end of every epoch (the *tlm interval* in Figure 1). The *tlm interval* value is configurable by users. It is set to 3 seconds by default in our implementation. In Figure 1, we denote the dirty bit tracking and initialization at stage 1 as DU_0, and represent subsequent DUs as DU_i, where $0 \leq i \leq N$ and $N \geq 0$.

During a DU operation, the io-thread calls the KVM kernel module to identify memory pages that have been modified in *the last tlm interval* and set the corresponding dirty bit array elements of those pages to **dirty**. After the DU operation finishes, the io-thread resumes its original duty with the hypervisor. To prevent race conditions on the dirty bit array, the **mtx** and **dtx** are blocked during DU.

The dirty bit array is the key data structure that the **mtx** and **dtx** use to determine if they will transmit a memory page to the destination. The **mtx** will send a memory page over the L2-1 link *iff* the dirty bit array element corresponding to that page has **never** been set to **dirty** by a DU before. On the other hand, the **dtx** will transmit a page *iff* the value of its dirty bit array element is **dirty**. After the **dtx** transmits that page, it will set the corresponding array element to **clear**. It is worth nothing that pages that have already been transmitted by **dtx** will never be sent by **mtx** again even though their dirty bits are **clear**. In other words, TLM uses **mtx** to send the snapshot content of memory pages at the moment the migration starts, and employs **dtx** to capture and transmit contents of newly modified memory pages thereafter.

On the destination VM, the **drx** and io-thread threads receive memory page contents from the **dtx** and **mtx**, respectively. Since the contents received by **drx** are newer than those received by the io-thread, the **drx** has a higher priority than the io-thread in placing the contents to any memory page. On the other hand, the io-thread can put contents to a memory page *iff* that page has **never** been assigned any content from the **drx** before.

The key part of TLM that satisfies the **time-bound principle** is the execution lifespan of the **mtx**. The progress of live migration can also be measured using the **mtx**'s progress. After the **mtx** finishes sending the last page of the VM, it informs the **dtx** of its completion, and terminates. The **dtx** in turn forwards the completion notification to the io-thread, and terminates as well. The io-thread will learn of the completion of **dtx** and **mtx** at the next DU event, marking the end of Stage 2.

Finally, TLM enters Stage 3, where it wraps up the migration by sending the rest of the dirty pages of the source VM to destination. When the DU operates at this stage, it detects the completion notification from **dtx** and stops the VM to prevent further generation of dirty pages. After the DU updates the dirty bit array, the io-thread will send the contents of every **dirty** memory page and the state of every device via L3 to the destination. It is worth noting that the L3 link can reuse either L2-1 or L2-2. After finishes, TLM stops the VM on the source and resumes the execution on the destination VM.

In terms of performance, the vcpu threads of the VM are stop only during Stage 1 and Stage 3. During Stage 2, the io-thread would be interrupted periodically every *tlm interval* to perform DU. In general, the operational time of Stage 1 is negligible. The duration of Stage 3, on the other hand, could be large depending on the number of dirty pages remaining after **mtx** and **dtx** complete. From the TLM design, we can define *TLM migration time*, *live migration time*, and *downtime* as: *TLM migration time* equals to the total amount of time TLM performs Stage 1, 2, and 3, *TLM live migration time* is equal to the amount of time TLM performs Stage 1 and 2, and *TLM downtime* equals to the duration of Stage 3.

TLM has three modes of operation. They are: TLM, TLM.1S, and TLC. The TLM mode performs the TLM

mechanism described earlier. The TLM.1S is a variation of TLM designed for performance comparisons in our experiments. In this mode, most of the operations are the same as those of TLM except that the DU does not do anything during TLM Stage 2. As a result, the **dtx** will not transmit any data, and all VM dirty pages will be accounted for and transferred on a single shot at Stage 3. The TLC mode represents the Time-bound, thread-based Live Checkpointing mechanism, not covered in this work.

A. *Downtime Minimization*

In order to minimize downtime, the number of dirty pages generated by the guest VM execution must be reduced. We have devised a method based on CPU over-committing to reduce dirty page generation during TLM live migration (Stage 2). In this method, users have to specify two parameter values: the reduced number of host CPU cores assigned to the VM (*reduced-cores*) and the point of CPU affinity adjustment (*adjust-point*).

The *reduced-cores* number must be less than the number of CPU cores originally assigned to the VM. TLM will use this number to generate a new set of host CPU core ids that is a subset of the original, and pass it to the **sched_setaffinity** system call to assign this new set of CPU cores to the VM. The adjust-point indicates when the **sched_setaffinity** system call will be invoked. This number is the percentage of memory pages the **mtx** thread has already transferred to destination.

This dirty page reduction mechanism is added to the end of DU operation. It will calculate the **progress** of the **mtx** from the current number of pages the **mtx** has transferred to the migration destination so far. This percentage number is used to compare with the *adjust-point*. If the current progress is greater than or equal to the *adjust-point*, the DU will stop the guest VM, invoke the **sched_setaffinity** system call to adjust CPU assignment, and resume the guest VM execution. For example, suppose the size of the original set of CPU ids is 8, by setting the reduced-cores to 4 and adjust-point to 0.2, TLM will reduce the number of CPU cores used by the VM from 8 to 4 when the **mtx** has transferred at least 20 percent of VM memory to destination. Note that **mtx** makes only one round of memory transfer and will skip pages that have already been transferred by **dtx**.

B. *Correctness*

We discuss the correctness of TLM in two aspects: the compliance with the time-bound principle and validity of memory pages on the destination VM.

The time-bound principle would hold for the Stage 2 operations under the following conditions. First, starvation must not occur to the **mtx** thread because the completion of the **mtx** marks the end of Stage2. Since the **dtx** and **mtx** and every thread in the VM run under Linux CFS scheduler and have the same priority, the **mtx** will progress and does not starve. Second, the communication bandwidth given to **mtx** must allow the **mtx** thread to transfer data without waiting indefinitely. Since the TCP connections L2-1 and L2-2 in Figure 1 equally share the underlying network, **mtx** will not wait indefinitely for network availability.

Since memory pages of the source VM are transmitted while the VM is running, a particular memory page could be modified by VM computation and read by the **mtx** or **dtx** at the same time. Thus, contents of memory pages transmitted to the destination might be corrupted. However, the corrupted page would be marked dirty by the dirty bit tracking mechanism on the next DU operation. Therefore, the correct page contents would either be transmitted to the destination by **dtx** during the next *tlm-interval* or be transmitted by io-thread at Stage 3.

C. *Implementation*

We have implemented the TLM mechanism based on the pre-copy migration mechanism of kvm-1.0. We name our prototype implementation **tlm-kvm** (Figure 2). In our current implementation, we assume the source computer has enough CPU and memory resources to run the **mtx**, **dtx**, and a migrating VM's threads without affecting other VMs on the same host. We define the default *tlm interval* value to 3 seconds because this value yields high VM communication bandwidths and low downtimes and migration times in our experiments.

To facilitate thread synchronization during data transfer, we have extended KVM's data structures that represent dirty bit information to incorporate the TLM dirty bit array. To prevent race conditions on the dirty bit array by the **mtx**, **dtx**, and **io-thread**, mutual exclusion variables are added. Since it is too costly to allocate a mutual exclusion variable for each memory page, we allocate 4096 mutual exclusion variables per gigabyte of VM memory. The mapping of memory page addresses to a mutual exclusion variable is similar to that of direct mapped cache.

IV. EXPERIMENTS

We have conducted a number of experiments to evaluate the TLM prototype. These experiments are categorized by type of workload on migrating VMs: OpenMP and MPI.

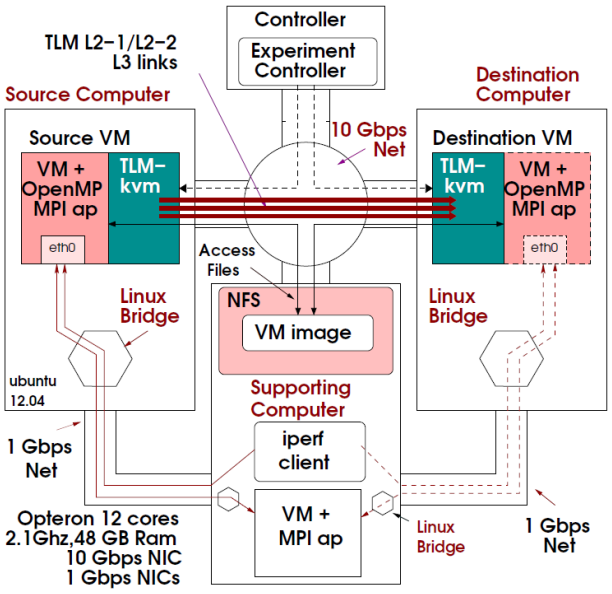
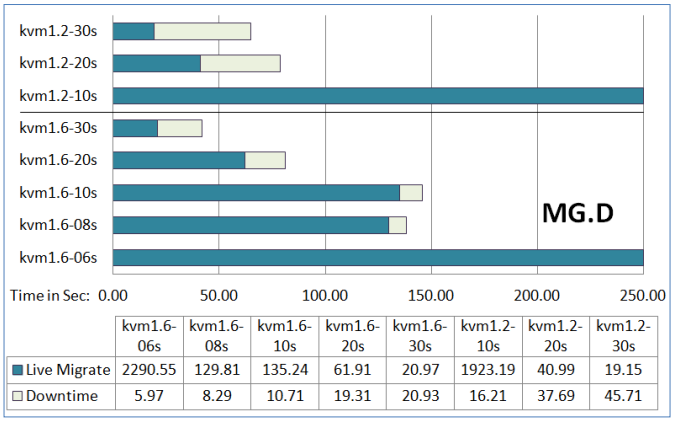


Figure 2. TLM Experimental Setups

Figure 2 shows our experimental setups. Our testbed consisted of 4 HP Proliant DL385 servers with AMD Opteron 21.0GHz 12 cores, 48 GB Ram, and 900GB 15K RPM SAS hard drive. All machines ran Ubuntu 12.04 (Linux 3.2.0) as the host OS and connected to a 10 Gbps and a 1 Gbps Ethernet network. Files and VM images on



	kvm1.6-06s	kvm1.6-08s	kvm1.6-10s	kvm1.6-20s	kvm1.6-30s	kvm1.2-10s	kvm1.2-20s	kvm1.2-30s
Live Migrate	2290.55	129.81	135.24	61.91	20.97	1923.19	40.99	19.15
Downtime	5.97	8.29	10.71	19.31	20.93	16.21	37.69	45.71

Figure 3. Live migration performance (under various tolerable downtime parameter values) of the original kvm-1.2 and kvm-1.6 on a VM running the MG Class D benchmark. Each data point is from a single run.

the supporting computer were shared via NFS v4 over the 10 Gbps network. The Controller computer ran the Experimental Controller program written in Python.

We configured the **Source Computer** to run a VM and later migrate to the **Destination Computer**. The Guest OS of the VM running OpenMP programs was Ubuntu 10.04 (Linux 2.6.32), and the Guest OS of the VM running MPI (MPICH2) programs was Ubuntu 12.04 (Linux 3.2).

We installed NAS Parallel Benchmark 3.3 [7] on the Guest VMs. TLM-kvm performed all data communication to and from VMs via virtio and a Linux bridge on each host, which, in turn, transmitted data over a 1 Gbps network.

To measure network bandwidth during the migration of a VM running an OpenMP benchmark, we ran an iperf client on the **Supporting Computer** to send data packets to the iperf server on the Guest OS of the migrating VM for 2,000 seconds starting when the benchmark was launched. The bandwidth was reported every 0.5 sec. For the MPI workload experiments we replaced the iperf client program with a VM running a part of communicating processes of the same MPI benchmark programs with the migrating VM. Data communication between the supporting computer and a migrating VM was performed over the 1 Gbps network. Every data point reported here is an average of 10 runs unless otherwise stated.

A. OpenMP Workloads

In this set of experiments, we investigated the migration of VM running OpenMP benchmarks by measuring migration performance, impact on IO bandwidths, and application execution time. Every VM was configured to use 8 SMP vcpus; however, their memory sizes were varied to match application workloads. In this paper, we focus on four OpenMP "Class D" benchmarks: MG, IS, SP, and BT. We chose these benchmarks because they have large working set sizes (WSS) and, at the same time, were not too big for the available resources. The VM memory size was configured to be bigger than the WSS of each benchmark. Since the WSS sizes of the MG, IS, SP, and BT benchmarks are 27.3GB, 34.1GB, 12.1GB, and 11.8GB, we set the VM memory sizes to 36GB, 36GB, 16GB, and 16GB, respectively.

A.1) Migration Performance of kvm-1.2 and kvm-1.6

First, we consider live migration performance of kvm-1.2. and kvm-1.6 under various *tolerable downtime* parameter values. In Figure 3, the label *kvm1.2-30s* denotes the kvm-*1.2* with the tolerable downtime value of *30* seconds. In the experiment, we run the MG class D benchmark and wait until it reach peak memory usage (120 seconds) before launching a migration. The *maximum transfer rate limit* in this experiment is set to 10 Gbps. From the Figure, the migration times vary by different tolerable downtime values. With tolerable downtimes of 10 or 20 seconds, live migration performs shortly. On the other hand, when the tolerable downtime is lower, the pre-copy mechanism takes longer time to perform live migration (Stage 2) until the amount of remaining dirty pages is low enough to be transfer within the tolerable downtime. In cases of *kvm1.2-10s* and *kvm1.6-6s*, the migration times take longer than the average execution time of the MG benchmark, 1363 sec, to complete. In the experiment, we also found that kvm-1.6 performs generally better than kvm-1.2 due to the addition of a thread to handle migration.

With appropriate tolerable downtime parameter (kvm-1.6-20s, kvm1.6-10s, kvm1.6-8s), decent migration performance can be achieved. In fact, these performances are better than TLM (see Figure 4) in terms of downtimes. Their live migration times are also appropriate comparing to application execution times.

However, identifying the appropriate tolerable downtime parameters is not trivial. The work in [12] has proposed algorithms to automatically determine the tolerable downtime. However, their migration results do not always converge. The migration behaviors similar to those of *kvm1.2-10s* and *kvm1.6-6s* may occur. TLM, on the other hand, is time-bound by design and presents lower migration time on its MG experiment. However, TLM has to pay for that with a higher downtime value.

A.2) Migration Performance of TLM

Next, we consider live migration performance under various TLM modes of operation. We performed a TLM migration on VMs after launching the MG, IS, SP, and BT benchmarks for 120, 180, 90, and 90 seconds, respectively. These times were chosen to ensure that the benchmarks used peak amounts of VM memory before migration.

Figure 4 shows live migration times and downtimes (in sec) of the benchmark programs. The label **OFFLINE** represents offline migration time, which is equal to downtime. The label **TLM.1S** denotes the performance of the TLM.1S mode. (Note that the TLM.1S can also be considered *equivalent* to performing kvm-1.6 pre-copy mechanism for single iteration during Stage 2 and then stop to transfer the remaining dirty pages to destination.) The label **TLM** represents the performance of the TLM operation that does not perform CPU-over-committing and has *tlm interval* value of 3 seconds. The remaining labels in the formats of ***x-(y)*** represent TLM performance when CPU over-committing was applied. For example, **0.2-(4)** means TLM performed CPU over-committing by assigning all vcpus of the migrating VM to run on **4** host CPU cores after the **mtx** thread finished scanning **20%** of the VM's memory.

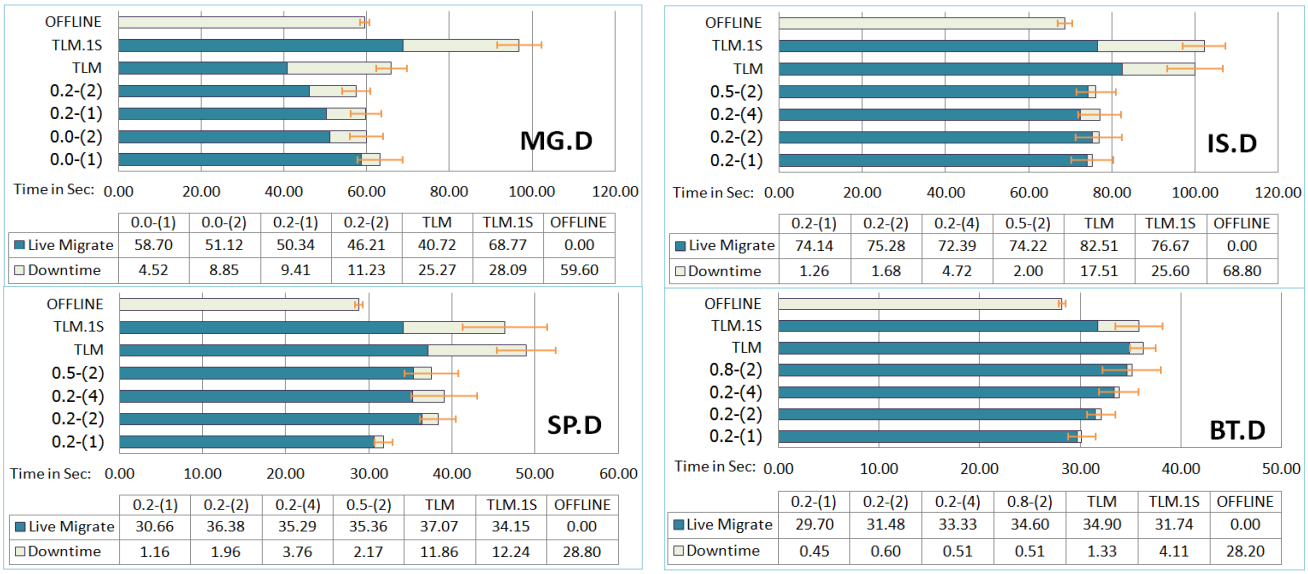


Figure 4. Live migration time, downtime, and total migration time of VMs running OpenMP MG, IS, SP, and BT Class D

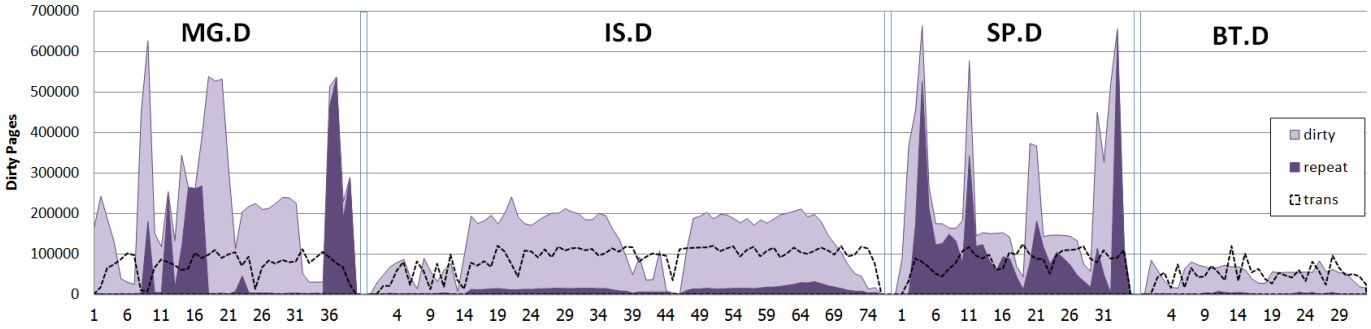


Figure 5. Number of generated dirty pages (**dirty**), repeating dirty pages (**repeat**), and transmitted pages (**trans**) per second

From the figure, it can be seen that TLM significantly reduced downtime of every benchmark compared to that of the offline migration. When we migrated a VM running the MG, IS, SP, and BT benchmarks, the TLM downtimes were 0.36, 0.26, 0.36, and 0.03 times that of the benchmark's offline downtime, respectively. This is because the **mtx** and **dtx** threads transfer most VM memory contents to the destination during Stage 2. TLM only has the most recently updated set of dirty pages to transfer during downtime.

In order to understand TLM downtime, we first consider the downtime of each benchmark under the TLM.1S migration. The effectiveness of TLM in reducing downtime can be measured by TLM.1S performance. In the TLM.1S mode, no dirty pages are transmitted during Stage 2; therefore, the downtime straightforwardly represents the transfer of *all dirty pages* generated during the live migration stage of each benchmark. Since TLM uses the **dtx** thread to reduce the number of remaining dirty pages at Stage 3, the effectiveness of the **dtx** can be measured by comparing the downtimes of TLM to that of TLM.1S. The TLM downtimes of the MG, IS, SP, and BT benchmarks were 0.89, 0.68, 0.96, and 0.32 times of their TLM.1S counterparts (Figure 4). We can see from the above that the dtx thread did not perform well in the MG and SP cases. However, it performed moderately well on the IS experiment, and better on the BT.

To analyze this further, another set of TLM migration experiments was conducted. In these experiments, we configured the *tlm interval value* to 1 second and recorded the following values: (i) the number of dirty pages generated by the migrating VM in the last second (denoted **dirty** in Figure 4), (ii) the number of dirty pages in (i) that were marked as dirty earlier but had not yet been transferred to the destination (**repeat**), and (iii) the number of dirty pages that had been transferred (and marked **clear** in the dirty bit array) by the **dtx** in the last *tlm interval* seconds (**trans**). The *tlm interval* was changed from 3 to 1 second because we want DU to record these three values at a finer time step. We use three graphs to illustrate these values (Figure 5). The dirty page generation data were taken from a representative run out of 10 runs, in which the downtime value was closest to the average TLM downtime of each benchmark in Figure 4.

The downtime of each benchmark depends on the amount of remaining dirty pages at Stage 3 of TLM. From the **dirty** graphs in Figure 5, it can be observed that large numbers of dirty pages were generated during the live migration of the MG, IS and SP benchmarks (over 600,000 per second, in some runs). On the other hand, the numbers of pages transferred per second by the **dtx** thread of both benchmarks (presented in their **trans** graphs) were much smaller (approximately 100,000 pages). Therefore, the numbers of dirty pages remaining at Stage 3 of these benchmarks are high. For the BT, since its dirty page generation rate was close to the data transfer rate, the number of remaining dirty page at Stage 3 was low.

In terms of migration time, TLM performance depends on the memory update behavior of each benchmark. Figure 4 shows that TLM live migration time was higher than the offline downtime on every benchmark except the MG. Since the memory update behaviors of the IS, SP,

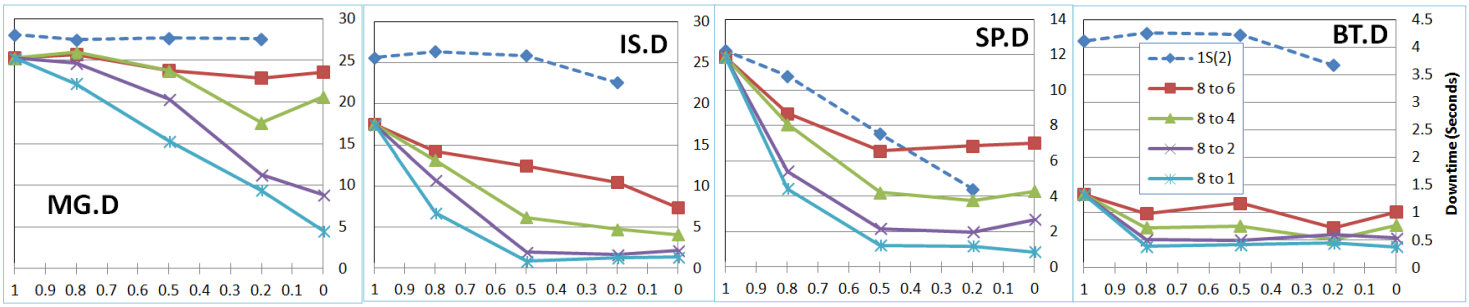


Figure 6. Downtimes of TLM under various migration parameters. The x axis represents the *ad-just points*.

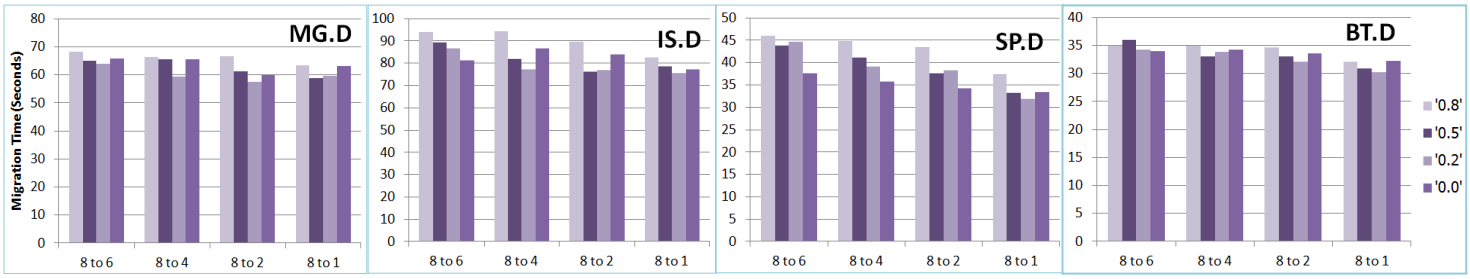


Figure 7. Migration time of TLM under various migration parameters. The x axis represents the *CPU change* labels.

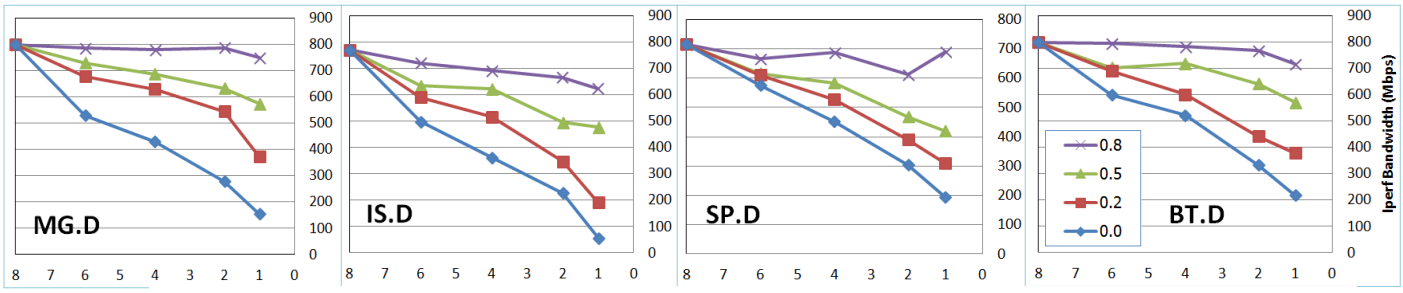


Figure 8. Iperf bandwidths during TLM live migration under various parameters. The x axis represents the *reduced-cores*.

and BT are highly localized, the **mtx** thread have to transfer most memory contents of the VM. At the same time, some repeated dirty page information is also transferred by the **dtx**. As a result, the total amount of memory pages transmitted to destination during Stage 2 by both threads of TLM became larger than the total amount of VM memory.

The MG's unique memory update pattern causes its TLM live migration time to be different from the rest of the benchmarks. From Figure 4, it can be observed that the MG's live migration time was 0.68 times the offline downtime, while those of other benchmarks were higher than theirs. Since the MG performs extensive short and long range memory updates which spanned over large address spaces of the VM's memory [7], the **mtx** skips transferring a large number of dirty pages during Stage 2.

A.2) Downtime Minimization

This section investigates the downtime minimization mechanism described in Section 3. From the experiments, we observed that the average bandwidth values reduced as the number of host CPU cores on which the VM CPUs were running reduces. Figure 4 shows TLM live migration time and downtime performance of the MG, IS, SP, and BT benchmarks under various parameters. It can be seen that CPU over-committing can significantly reduce TLM migration times and downtimes.

Figure 6 shows downtime values against two CPU over-committing parameters: the *adjust-point* and the *reduced-cores* (defined in Section 3). Four graphs for the MG, IS, SP, and BT benchmarks are presented. The x axis of these graphs represents the *adjust-point* as a percentage of the progress of the **mtx** thread. Note that the x value of 0 means over-committing starts at the beginning of migration, and the x value of 1 means no over-committing at all. The label "**x to y**" in the graph represents a *CPU change*, e.g. "8 to 1" denotes the change from 8 vcpus to 1 vcpu. The **1S(2)** (dotted) line depicts the reduction of downtime when migrating a VM under TLM.1S mode and changing its vcpus from 8 to 2 at various *adjust-points*. Note that the VM always resumes using 8 vcpus after migration.

The pattern of downtime reduction of each application depends on its memory usage characteristics. The MG graph in Figure 6 shows linear reduction of downtimes on every line except the **1S(2)** line which stays constant. Since the MG benchmark generated a large number of dirty pages spread widely over VM memory space, different *adjust-points* and *reduced-cores* produce significantly different rates of dirty page generation during TLM's Stage 2. On the other hand, the **1S(2)** downtime stays roughly the same even though the vcpus are reduced to 2 because the dirty pages get accumulated during its Stage 2 to transfer in one shot at Stage 3. The patterns of the **1S(2)** graphs of the IS and BT benchmarks are similar to that of the MG.

In Figure 6, the IS benchmark has a different downtime reduction pattern. The downtime appears to drop sharply when the *adjust-points* values are 0.8 and 0.5. However, when we started over-committing CPU earlier (at the *adjust-points* 0.0 and 0.2), the downtime

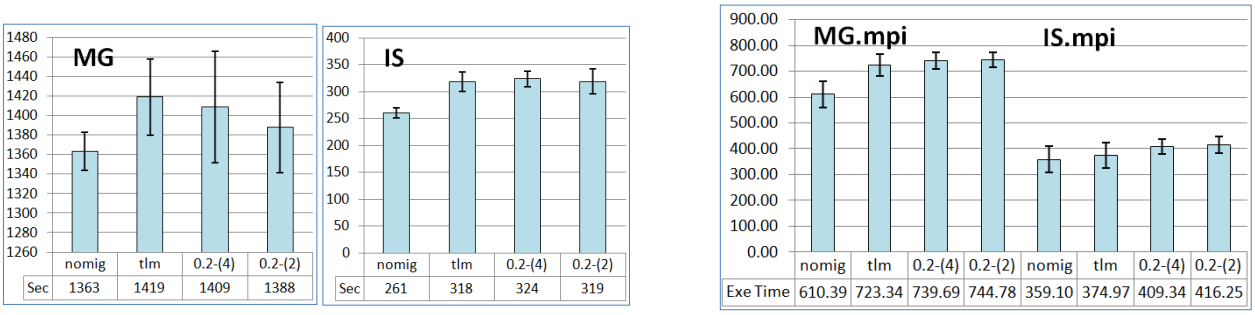


Figure 9. Execution Time (in sec) of the OpenMP version of MG and IS Class D, and the MPI version of MG and IS Class D (MG.mpi and IS.mpi).

values remained close to those at the *adjust-point* 0.5. Since the downtimes at *adjust-points* 0.0, 0.2, and 0.5 are close to one another, the remaining dirty pages at Stage 3 of TLM at these *adjust-point* values must also be similar. From the repeated graph in Figure 5, we believe that these dirty pages represent the *locality set* of memory pages repeatedly updated by the IS program. The higher the number of *reduced-cores*, the bigger the locality set. The SP and BT benchmarks have downtime reduction patterns similar to that of the IS, showing that the benchmarks have high memory update locality.

Figure 7 shows that the migration times of every benchmark when CPU over-committing was applied were generally close to the migration times of TLM without CPU over-committing. The migration time in the figure slightly declines with lower *reduced-cores* values.

A.3) Communication Bandwidth Tradeoff

Although CPU over-commit can help reduce downtime, it has a notable side-effect. The I/O capacity of the VM during TLM live migration (Stage 2) will also reduce. Moreover, the sooner the CPU over-committing starts, the lower the average bandwidth values.

Figure 8 illustrates the reductions of iperf bandwidths reported by iperf clients running on the supporting computer in Figure 2. In Figure 8, every graph shows similar patterns of bandwidth reduction: the bandwidth values of every graph line reduce linearly as the *reduced-cores* values get lower. The label on each graph line in the figure represents an *adjust-point*. When comparing different graph lines with the same *reduced-cores* value in each benchmark, the iperf bandwidths of the graph line with sooner *adjust-points* (e.g. "0.0") are always lower than those of later *adjust-points* (e.g. "0.8").

When the bandwidth values become too low, the VM may be irresponsive to I/O requests during live migration. Although the average IO bandwidths on the "0.0" and "0.2" lines in Figure 8 are in the range of 100 to 200 Mbps on some benchmarks, we found that the distribution of raw bandwidth data is not normally distributed but is skewed. During a live migration period, the bandwidth data points may present several 0 Mbps values in some experiments.

A.4) Application Execution Time

In this section, we evaluate impacts of our TLM implementation on total execution times of the benchmarks under various conditions. We define the *TLM overhead* of an application program as the execution time increase when the VM migrates comparing to the execution time without migration. In Figure 9, the TLM overhead values were labeled **tlm** for a TLM migration, **0.2-(4)** or **0.2-(2)** for a migration with CPU over-committing and **nomig** for no migration. The TLM overheads of the MG and IS were 55 and 58 seconds, respectively. By comparing these overheads to the migration time reports in Figure 7, we found that the TLM overheads of the MG and IS (both under normal TLM and CPU over-committing optimization) are close to TLM migration times.

B. MPI Workloads

We conducted another set of experiments to evaluate TLM performance on two MPI benchmarks: the MG and IS "Class D". Each benchmark consists of 8 MPI processes. We ran four processes on a VM on the source computer (Figure 2) and another four processes on another VM on the supporting computer. Each VM was

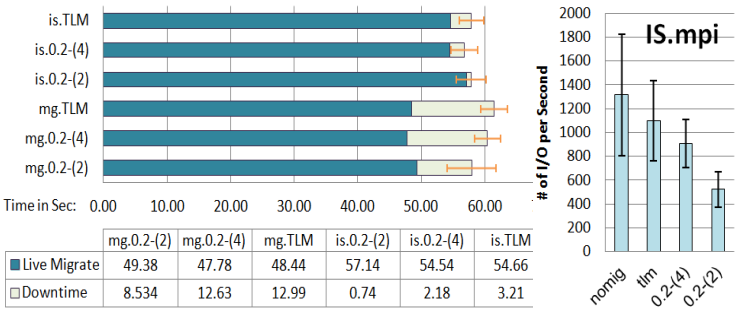


Figure 10. (a) Migration performance of MPI MG and IS Class D and (b) I/O performance of MPI IS Class D.

configured to use 8 vcpus and 36GB of memory. After launching the MG and IS benchmarks, we waited for 90 and 180 seconds respectively to let memory usage of the VM on the source computer reach peak working set sizes before performing a migration. The working set size of the MG on each VM is 14 GB and that of the IS is 12 GB.

Figure 10(a) shows TLM migration performance of the VM running MG and IS benchmarks under normal TLM, 0.2-(4), and 0.2-(2) migration parameters. Every experiment ran with a 3 second *tlm interval*. The performance results of the MPI benchmarks are consistent with those of the OpenMP benchmarks in Figure 4, where TLM is time-bound and spends most of the time on live migration. The downtimes also reduce under CPU over-committing in the same way as the OpenMP benchmarks.

In order to measure I/O performance during live migration, we did not use the iperf utility as before because it interferes with MPI communication. Instead, we modified our **tlm-kvm** implementation to count every I/O interrupt that causes the KVM hypervisor to exit from hardware-assisted VM execution mode to handle I/O. Figure 10(b) shows the number of I/O operations per second of the IS benchmark. When using CPU over-commit to reduce downtime, the I/O operations per second drop as the number of vcpus decreases.

The TLM overheads of the MG and IS are different. From Figure 9 and 10, the TLM overheads of all MG experiments are approximately twice as much as the TLM migration time while those of the IS experiments are equal

	kvm-1.6(auto) App-Tx-Dw	Mig (sec)	Down (sec)	Iperf BW (Mbps)	TLM App-Tm-cpu	Mig (sec)	Down (sec)	Iperf BW (Mbps)	*Mig ratio*	*Iperf BW ratio*
1	mg.D-10g-8	138.14	8.42	407.01	mg.D-0.0-(2)	59.97	8.85	277.42	***0.43***	***0.68***
2	mg.D-10g-4	180.98	4.07	374.66	mg.D-0.0-(1)	63.23	4.52	155.26	***0.35***	***0.41***
3	mg.D-10g-1	219.17	1.10	282.31	N/A	N/A	N/A	N/A	***N/A***	***N/A***
4	is.D-10g-2	165.23	2.14	517.26	is.D-0.5-(2)	76.21	2.00	495.25	***0.46***	***0.96***
5	is.D-10g-1	168.23	1.03	517.90	is.D-0.5-(1)	78.60	0.87	477.20	***0.47***	***0.92***
6	sp.D-10g-2	54.29	1.93	418.74	sp.D-0.5-(2)	37.53	2.17	467.12	***0.69***	***1.12***
7	sp.D-10g-1	58.77	1.05	392.84	sp.D-0.5-(1)	33.27	1.21	418.36	***0.57***	***1.06***
8	bt.D-10g-2	28.11	1.85	571.44	bt.D-0.8-(2)	34.60	0.51	767.86	***1.23***	***1.34***
9	bt.D-10g-1	28.88	1.20	562.83	bt.D-0.8-(1)	32.10	0.39	717.13	***1.11***	***1.27***

Figure 11. Performance comparisons between the auto-convergence and CPU over-committing mechanisms. The notation "N/A" at row 3 means TLM cannot achieve one second downtime as kvm-1.6 does.

to or lower. We believe that the higher TLM migration overheads are due to longer communication time or synchronization during the live migration stage of the MG. Such additional communication overheads of the IS during live migration are not as high as those of the MG.

C. Auto-convergence V.S. CPU over-committing

Figure 11 shows the comparisons of migration performance between kvm-1.6 with auto-convergence and TLM with CPU over-committing. For the migration of kvm-1.6, we conducted a number of experiments migrating VMs running the four OpenMP benchmarks we used earlier. In the **kvm-1.6** column in the Table, the label "App-Tx-Dw" represents three migration parameters with auto-convergence enabled. The "App" denotes application name, "Tx" denotes *maximum transfer rate limit* value, and "Dx" denotes the *tolerable downtime* value (in sec). The *maximum transfer rate limit* parameter in these experiments was set to the maximum of 10 Gbps. In the **TLM** column, the label "App-Tm-cpu" represents three TLM parameters, where "Tm" denotes the *adjust-points* and "cpu" denotes *reduced-cores* values. The "Mig" and "Down" labels represent migration time and downtime (in sec). The "Iperf BW" denotes Iperf bandwidth during live migration (in Mbps). The "Mig ratio" represents the ratio of the TLM migration time to the kvm-1.6 migration time. The "Iperf BW ratio" represents the ratio of the average Iperf bandwidth during live migration of TLM to that of kvm-1.6.

In the Figure, we match the performance results of kvm-1.6 and TLM, using results that have comparable downtime values. It can be observed from the MG performance at lines 1 and 2 that TLM migration times are 0.43 and 0.35 times those of kvm-1.6. However, TLM Iperf bandwidths of the MG are lower than those of kvm-1.6. At line 3, the auto-convergence of kvm-1.6 can achieve 1.1 sec downtime while TLM cannot reach such a low value. For the IS and SP (lines 4-7), the TLM migration times range from 0.46 to 0.69 times those of kvm-1.6 while the Iperf bandwidths of the two approaches are close to one another. In case of the BT benchmark (lines 8 and 9), the TLM migration times are 1.23 and 1.11 times those of kvm-1.6. However, the Iperf bandwidths of TLM are a little higher than those of kvm-1.6.

We believe that both mechanisms can be used to minimize downtime. However, their different design principles cause the different performance reported in the table. While the pre-copy migration with auto-convergence mechanism aims to minimize amount of dirty pages to achieve a *mandatory* tolerable downtime, TLM performs *best-effort* dirty page minimization during a bounded time period.

V. CONCLUSION AND FUTURE WORKS

In this paper, we introduce a new design principle for pre-copy VM live migration, which requires migration operation to complete within a bound time period. We have designed, implemented and evaluated the TLM mechanism.

The main advantages of our implementation lie in the simple yet powerful thread-based design and the resource allocation and downtime minimization using CPU over-committing. We believe that TLM is a practical mechanism for migrating computation and memory intensive HPC applications. Our experiments demonstrate that TLM can efficiently migrate the OpenMP and MPI versions of the MG, IS, SP, and BT (Class D) benchmarks over commodity server computers connected to one another over a 10 Gbps network. TLM can achieve lower migration times than other pre-copy approaches. In future works, we will study the mixing of CPU over-committing and auto-convergence mechanisms for downtime minimization. The TLM software is available at [18].

REFERENCES

[1] https://cloud.google.com/pricing/compute-engine

[2] http://aws.amazon.com/ec2/pricing/

[3] C. Clark et al. "Live migration of virtual machines," USENIX NSDI, 2005

[4] http://wiki.qemu.org/ChangeLog/1.4

[5] http://wiki.qemu.org/ChangeLog/1.6

[6] A. Kivity et al. "kvm: the linux virtual machine monitor," *Proc. of Linux Symposium*, 2007.

[7] NPB 3.3, http://www.nas.nasa.gov/publications/npb.html#url

[8] M. Harchol-Balter and A. Downey. "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Transactions on Computer Systems* vol. 15, no. 3, August 1997

[9] V. Siripoonya and K. Chanchio, "Thread-based Live Checkpointing of Virtual Machines," *IEEE NCA*, 2011.

[10] M. R. Hines et al. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev*. 43, 3, July 2009

[11] Jihun Kim et al. "Guide-copy: fast and silent migration of virtual machine for datacenters," *Supercomputing (SC)* 2013

[12] K.Z. Ibrahim et al. "Optimized pre-copy live migration for memory intensive applications," *Supercomputing (SC)* 2011

[13] T. Wood et al. "CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines," *SIGPLAN Not.(VEE'11)* 46, 7, 2011

[14] P. Svärd et al. "Evaluation of delta compression techniques for efficient live migration of large virtual machines," *SIGPLAN Not.(VEE'11)* 46, 7, 2011

[15] X. Song et al. "Parallelizing live migration of virtual machines," *SIGPLAN Not.(VEE'13)* 48, 7, 2013

[16] C. Jo et al. "Efficient live migration of virtual machines using shared storage," *SIGPLAN Not.(VEE'13)* 48, 7, 2013

[17] http://wiki.qemu.org/OlderNews

[18] http://vasabilab.cs.tu.ac.th/projects/TLM.html