

Application-Assisted Live Migration of Virtual Machines with Java Applications

Kai-Yuan Hou^{†*} Kang G. Shin[†] Jan-Lung Sung^{*}

[†] University of Michigan ^{*} Oracle America, Inc.

{karen.hou,janlung.sung}@oracle.com

kgshin@eecs.umich.edu

Abstract

Live migration of virtual machines (VMs) can consume excessive time and resources, and may affect application performance significantly if VM memory pages get dirtied faster than their content can be transferred to the destination. Existing approaches to this problem transfer memory content faster with high-speed networks, slow down the dirtying of memory pages by throttling the execution of applications, or reduce the amount of memory content to be transferred, for example, using compression. However, these approaches incur high resource costs or application performance penalties. In this paper, we propose to skip the transfer of VM memory pages that need not be migrated for the execution of running applications at the destination, by exploiting applications' assistance. We have designed a generic framework for application-assisted live migration and then used it to build and evaluate *JAVMM*, which migrates VMs running various types of Java applications skipping the transfer of garbage in Java memory. Our experimental results show that *JAVMM* can reduce the completion time, the network traffic of transferring memory pages, and the application downtime of Java VM migration, all by up to over 90%, compared to the vanilla Xen VM migration, without incurring noticeable performance penalty to applications.

*Work done during Kai-Yuan Hou's attendance of the University of Michigan.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04...\$15.00.
<http://dx.doi.org/10.1145/2741948.2741950>

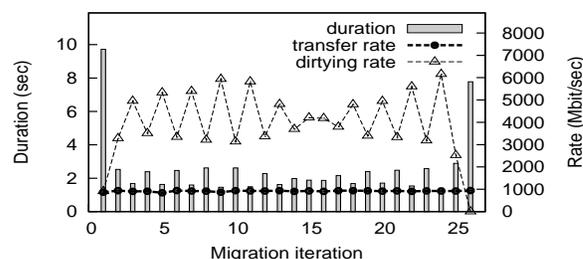


Figure 1. Live migration of a 2GB Xen VM running the Apache Derby database workload from SPECjvm2008.

1. Introduction

Live migration [13, 30] is to move a running virtual machine (VM) from a physical host to another with minimal disruption to the execution of the VM. It has been used for load-balancing [34, 37], fault-tolerance [28, 32], power savings [11, 14, 29], and performance enhancements [12].

To migrate VMs within a LAN, such as within a data-center, the primary task is to migrate the contents of VMs' memory; VM disk contents can be stored in a shared storage. Most migration tools transfer VM memory by using a *pre-copy* approach. While a migrating VM continues to run on the source host, its memory pages are iteratively transferred to the destination host. All pages are sent in the first iteration, and at each following iteration, only those pages dirtied during the previous iteration are sent. Ideally, dirty pages should be transferred faster than new pages get dirtied, and the number of dirty pages pending transmission should decrease iteratively. When the VM is paused for the last iteration, a small number of dirty pages remain to be transferred. After this short *stop-and-copy*, the VM resumes execution in the destination, and the migration completes.

However, this ideal migration is not always achievable, since the underlying network can become a bottleneck. Figure 1 shows live migration of a 2GB database Xen VM over a gigabit Ethernet. Since the database application dirties memory pages much faster than the pages can be transferred, the

number of dirty pages to be transferred does not decrease iteratively; hence the iterations do not keep becoming shorter. Migration cannot finish with a short stop-and-copy, but is forced to enter the last iteration after generating excessive network traffic (a total of 7GB). It incurs a long completion time (66 secs), causes a noticeable VM downtime (8 secs), and degrades application performance (by over 20%).

To alleviate the network bottlenecking problem during migration and its undesirable consequences, approaches have been proposed to speed up memory transfer using high-speed networks [21], slow down memory dirtying by throttling application execution [13], or reduce the amount of memory contents to be transferred, e.g., by using compression [35]. However, these approaches incur high resource costs or application performance penalties. The OS’s knowledge can also be utilized to reduce the amount of memory transfer by not sending clean page cache pages and free pages [24], but the benefit is limited. Page cache misses may degrade application performance at the destination, and skipping free pages may only benefit the migration of lightly-loaded VMs.

In this paper, we propose to reduce the amount of memory transfer by exploiting application semantics. Specifically, we design *application-assisted live migration*, which skips transfer of VM memory pages that need not be migrated for the execution of running applications at the destination. We build a framework for the proposed approach based on Xen. Our framework places a paravirtualized stub in the guest VM to enable collaboration between the migration tool and applications in the guest. We ask the applications, which know best their semantics, to identify areas in their memory that need not be migrated. Based on the applications’ inputs, we maintain a *transfer bitmap* that guides the migration tool to transfer or skip over VM memory pages.

Using the proposed framework, we build *JAVMM*, which migrates *Java VMs*—VMs running various types of Java applications—without transferring garbage¹ in Java memory. Targeting Java applications does not restrict JAVMM’s applicability, since these applications are nearly ubiquitous; with over 9 million developers worldwide, Java has become the global standard for web-based content and enterprise software, and runs in 89% of computers in the U.S.[3]. Java applications are increasingly being deployed and run in VMs for flexible resource sharing and easy deployment. Various types of Java cloud services are being widely used [5], many of which are provisioned based on VMs for elasticity. To migrate Java VMs fast and with little performance impact is therefore an important task.

In JAVMM, Java Virtual Machine (JVM)² assists in migration on behalf of Java applications. Right before a mi-

grating Java VM is paused for the last iteration, the running JVM performs a garbage collection. After the last iteration is completed, the VM resumes execution at the destination in a post-collection state: in the *Young generation* of the Java heap, only one survivor space may contain live objects, which survived the collection. JAVMM migrates the surviving objects in the last iteration, and skips transfer of the entire Young generation throughout migration.

We prototyped JAVMM using HotSpot JVM [2] in the proposed framework, and evaluated it in terms of three metrics commonly considered for live migration: the time and resources used by migration, and the impact of migration on running applications’ performance. Our experimental results show that compared to Xen live migration, which is agnostic of application semantics, JAVMM can reduce the completion time, network traffic and application downtime caused by Java VM migration, all by more than 90% when the running Java application has a high object allocation rate and needs a large Young generation space, without incurring noticeable performance degradation to the application.

The primary contributions of this paper are as follows.

- We propose application-assisted live migration, and establish a generic framework to skip migration of VM memory selectively based on application semantics.
- Using the proposed framework, we build JAVMM to migrate Java VMs skipping over garbage with JVM’s assistance, without customizing each Java application.
- Via an in-depth evaluation of JAVMM, we demonstrate the utility of application-assisted live migration.

The remainder of this paper is organized as follows. Section 2 reviews existing approaches to alleviating network bottleneck during live migration. Section 3 presents our approach, a generic framework for application-assisted live migration. Section 4 describes JAVMM, built based on the proposed framework for efficient migration of Java VMs. Section 5 evaluates JAVMM experimentally. We discuss future extensions of this work in Section 6, and conclude the paper with Section 7.

2. Related Work

To alleviate network bottlenecking during live migration, approaches have been proposed to send dirty memory pages faster, generate dirty memory pages slower, or send less data for the dirty memory pages generated.

Huang *et al.* [21] proposed to transfer memory pages faster using high-speed networks capable of Remote Direct Memory Access (RDMA) like InfiniBand. The network remains a potential bottleneck even with high-speed links, though, considering the increasing computation power of individual VMs and the fact that multiple VMs may be migrated at the same time.

Clark *et al.* [13] proposed to slow down the memory-dirtying rate by moving processes to a wait queue after they

¹*i.e.*, Java objects that are no longer used

²Note the difference between a JVM, the application-level VM that executes Java bytecode, and a Java VM, a general-purpose VM in which Java applications and their JVMs run.

generate more than a certain number of dirty pages. This may degrade application performance, and as the authors noted, one must be careful not to throttle interactive services.

Our approach falls in the third category, which transfers less data for the dirty memory pages generated. Compression [22, 35] and deduplication [15, 16, 38] are popular in this category, trading CPU for network bandwidth. Our approach skips transfer of selective memory pages, performing no computation on the contents of the pages skipped and incurring a minimal CPU overhead.

There are also other approaches that skip transfer of selective memory pages, according to different criteria than ours. Some skip over frequently dirtied pages during live iterations, but those pages must be transferred in the last iteration [10, 20, 25, 26], risking a long VM downtime. Page cache pages can be skipped over in all iterations if the storage has an identical copy of the pages; the contents skipped need to be reproduced [23], or VM performance may degrade after migration [24]. Post-copy migration skips over all memory pages and removes the pre-copy stage. To run the VM in the destination, pages are fetched from the source [18, 19], incurring performance penalties. Exploiting knowledge of the migrating OS, free pages can be skipped over and not fetched upon access [24], but only in lightly-loaded VMs we may find a considerable number of free pages to be skipped.

Our approach exploits knowledge of the migrating applications to skip transfer of selective memory pages during migration. Targeting Java applications, JAVMM skips transfer of garbage in the frequently-dirtied Young generation of the Java heap. It need not reproduce the contents skipped, and does not degrade application performance. Application-Level Ballooning (ALB) [31] is another system that targets Java applications and leverages application knowledge in a hypervisor-level mechanism. It extends memory ballooning to JVM, to move Java memory in and out of the hosting VM as application demand changes. ALB may be used to shrink the Java heap before migration begins and send less dirty data during migration, with the tradeoff of potentially lower application performance; application performance may degrade as the heap becomes smaller since garbage collection may be triggered more frequently.

The work described in this paper is closest to the memory deprotection technique discussed in RemusDB [27], a VM-based high-availability system for databases. RemusDB continuously replicates checkpoints of VMs running databases. To reduce system overhead, the authors explored omission of selective memory contents from VM checkpoints based on application inputs, although data structures to be suitably omitted by this technique are yet to be identified.

3. Application-Assisted Live Migration

We take a *white-box* approach to reducing the amount of memory transfer for efficient VM live migration: we propose

to skip transfer of selective VM memory based on application semantics, by exploiting applications' assistance.

3.1 What Memory to Skip Migrating?

Generally, memory contents that are reproducible or not required for correct application execution need not be transferred during migration; these contents also need no replication in high-availability systems [27].

Examples of reproducible contents include those recoverable from application logs and intermediate results that can be recomputed. It may be beneficial to skip migrating these contents if regenerating them in the destination is faster than transferring them from the source.

Memory contents not required for correct application execution include caches and garbage. Caches of various kinds, e.g., web cache and database buffer pool, need not be migrated if the performance drop caused by empty caches at the destination can be mitigated or is acceptable. Garbage is memory content that is no longer being used. It is a good candidate to skip migrating, since in its absence at the destination, applications can execute correctly and without performance degradation. Garbage exists in any applications written in languages that do not deallocate memory explicitly; Java, C# and most scripting languages, e.g., Python and Ruby, fall in this category.

3.2 Challenges and Design Principles

To skip migration of selective application memory, the key challenge is to let the migration tool and running applications collaborate. The migration tool needs to know which memory pages to skip transferring. For the memory contents not transferred, the applications need to recover or not access them in the destination host.

Traditionally, the migration tool and an application in the guest VM are unaware of the execution of each other. They do not, and have no existing channel to, communicate. They also address memory differently: the migration tool transfers VM memory pages based on Page Frame Numbers (PFNs), *i.e.*, the page numbers in the VM's contiguous memory space, while the application executes based on Virtual Addresses (VAs). For the migration tool and the application to collaborate, the *communication gap* and *semantic gap* between them must be bridged.

We design a framework to enable their collaboration and be able to skip migration of selective application memory, following three principles; each principle describes the responsibility of one software component in our framework.

- The guest kernel provides system-level support for bridging the communication gap and semantic gap between the migration tool and running applications. It coordinates between the migration tool and the applications as they perform migration collaboratively, so that the migration tool need not interact with each application individually.

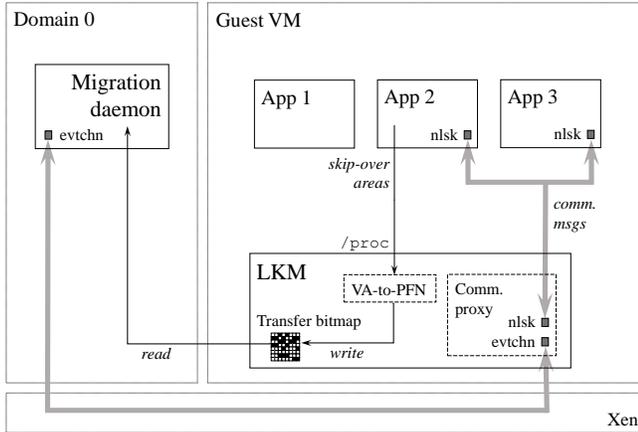


Figure 2. A generic framework for application-assisted live migration, prototyped based on the Xen hypervisor and a Linux guest. A Loadable Kernel Module (LKM) runs in the guest to facilitate collaboration between the migration daemon and running applications. It bridges the communication gap between them by relaying messages for them, using Xen’s event channel (evtchn) and Linux’s netlink socket (nlsk). It bridges the semantic gap by performing VA-to-PFN translations, and manages a transfer bitmap that guides the migration daemon to skip transfer of selective application memory.

- A running application identifies which areas of its memory need not be migrated, and informs the migration tool. We ask the application to do this, since the application knows best the semantics of its memory, e.g., what each memory area is used for and when the content is needed.
- The migration tool needs to know which memory pages to skip transfer, without incorporating application semantics. This way the tool becomes generic (i.e., application-independent), and can thus be used for different applications without modification. This also minimizes changes to existing live migration mechanisms.

3.3 A Generic Framework

Figure 2 provides an overview of our framework, prototyped based on Xen 4.1; our guest VM runs Linux 3.1. We added a Xen management command to invoke application-assisted live migration. Once invoked, our migration daemon executes. Our migration daemon is a modified version of Xen’s. It communicates with applications in the guest through the guest kernel, and skips transfer of memory pages guided by a *transfer bitmap*. We provide guest kernel support in a Loadable Kernel Module (LKM).

3.3.1 Bridging the Communication Gap

Our LKM serves as a communication proxy between the migration daemon and the applications in the guest. It interacts with the migration daemon using event channel,

the event notification primitive provided by Xen. A special event channel port is created when the guest VM is created, through which the migration daemon can communicate with the LKM throughout the migration process.

The LKM interacts with the applications using netlink sockets, a special socket family for communication between kernel- and user-space processes. We use netlink because it is bi-directional, asynchronous and capable of multicasting. Upon loading, the LKM creates a netlink socket, and associates it with a multicast group, which the applications subscribe to. The migration daemon communicates with the applications simply by contacting the LKM, and the LKM multicasts a netlink message to notify all subscriber applications. The LKM also relays messages from the applications to the migration daemon.

3.3.2 Bridging the Semantic Gap

The applications identify areas in their memory that the migration daemon can skip transfer. They specify each *skip-over area* by a VA range, and pass the VA range to the LKM via a `/proc` entry. The LKM finds the PFNs of the skip-over area by page table walks, while the application continues its normal execution. The LKM may consider a smaller VA range than that specified by the application. It aligns the start and end VAs of the specified range to the immediate next and previous page boundaries, respectively, to ensure pages found in the skip-over area can be skipped by the migration daemon in their entirety.

3.3.3 Skipping Transfer with a Transfer Bitmap

The LKM records the PFNs of the skip-over areas in a *transfer bitmap*. When transferring VM memory, the migration daemon examines the transfer bitmap, in addition to the dirty bitmap maintained by the hypervisor.

The transfer bitmap is created in the guest when the LKM is loaded, and is shared with the migration daemon when migration begins. It uses one bit per VM memory page (PFN), based on the same page size used by the dirty bitmap; assuming 4KB pages, the transfer bitmap uses 32KB per GB of VM memory, incurring a negligible memory overhead. Each transfer bit is either set (1) or cleared (0). A set transfer bit indicates the page needs to be migrated; the migration daemon transfers the page if its marked dirty in the dirty bitmap. A cleared transfer bit indicates migration of the page can be skipped; the migration daemon does not transfer the page, even if it is marked dirty.

3.3.4 Updating the Transfer Bitmap

The transfer bitmap is initialized with all bits set; by default, memory pages are transferred if they are marked dirty. Figure 3 illustrates how the LKM updates the transfer bitmap to record the PFNs of the skip-over areas, so that these pages are not transferred during migration.

The LKM makes the first bitmap update when migration begins. It queries the applications for skip-over areas. For

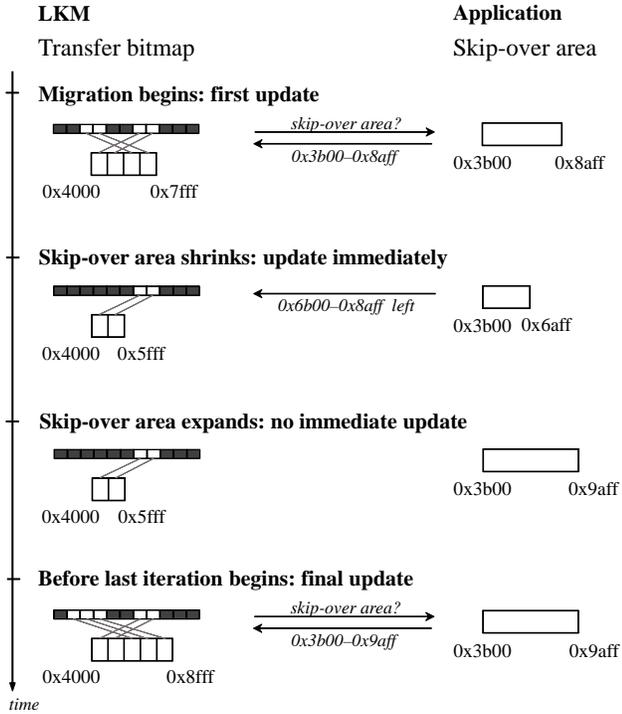


Figure 3. An example of transfer bitmap updates; in the illustration, filled bits in the transfer bitmap are set. When migration begins, the LKM clears the transfer bits of the memory pages in the skip-over area. If the skip-over area shrinks during migration, the LKM sets the transfer bits of the pages leaving the area immediately, but when the area expands, it defers clearing the transfer bits of the pages joining the area until in the final bitmap update, which is performed right before the last iteration begins.

each area in the applications' responses, it remembers the VA range, finds the associated PFNs by page table walks, and clears the corresponding bits in the transfer bitmap. Therefore, the migration daemon does not transfer the pages in the skip-over areas even if they are dirtied.

In parallel with, and after, the first bitmap update, the VM continues to run, and each skip-over area may expand or shrink, *i.e.*, VA ranges and the associated PFNs may join or leave the area. Subsequent updates to the transfer bitmap may be needed.

A skip-over area is expected to shrink infrequently and by a small amount during migration, or the benefit of skipping its migration decreases. When a skip-over area shrinks during migration, the application should notify the LKM of the VA ranges leaving the area. The LKM updates its memory of the area's VA range, and immediately, sets the transfer bits of the PFNs leaving the area. These pages may later get dirtied in a memory area requiring migration. Setting their transfer bits immediately ensures transfer of their dirty con-

tents in the iteration following the dirtying, and guarantees the correctness of migration.

Given the VA ranges leaving a skip-over area, the LKM does not find the PFNs leaving the area via page table walks; if the area shrinks due to deallocations, the PFNs leaving the area are reclaimed and can no longer be found in the page tables. Instead, the LKM caches PFNs as they are found in a skip-over area and their transfer bits are cleared. It queries the PFN cache by the VA ranges leaving the skip-over area to quickly find the PFNs that must have their transfer bits set. After setting their transfer bits, it removes the PFNs from the cache. The PFN cache uses little memory: 1MB per GB of skip-over area with 4-byte entries (a 0.1% overhead).

When a skip-over area expands, transfer bitmap updates are not required. Not clearing the transfer bits of the PFNs joining the area does not affect the correctness of migration, although the pages may be unnecessarily migrated. To reduce runtime overhead, the application does not notify the LKM when a skip-over area expands during migration. The LKM does not clear the transfer bits of the PFNs joining the area until in the final bitmap update, which is performed right before the last iteration begins. Dirty pages in the expanded space of a skip-over area will be skipped in the last iteration to reduce VM downtime.

In the final bitmap update, the LKM queries the applications again for skip-over areas. It compares the VA ranges replied by the applications with those in its memory. For any expanded space, it finds the PFNs joining the areas via page table walks, and clears their transfer bits. For any shrunk space, it sets the appropriate transfer bits based on the cached PFNs. The skip-over areas should not shrink in the short window of the final bitmap update, to ensure the transfer bits of all the pages leaving the areas are set; if necessary, the applications may be paused to meet this requirement. Once the final bitmap update is completed, the VM is paused, and the last iteration begins.

In our current implementation, if a PFN joins or leaves a skip-over area with no changes in the area's VA range, the transfer bitmap is not updated. This happens when a virtual page in a skip-over area has its PFN mapping changed, in three possible ways: (1) from *null* to *p*, when a page frame is allocated; (2) from *p_{old}* to *p*, when the page is remapped due to page sharing, compaction and migration (within the guest); and (3) from *p_{old}* to *null*, when the page is swapped out. For (1), migration finishes correctly without clearing the transfer bit of the allocated page joining the skip-over area. For the events in (2) and (3), we currently assume their absence in skip-over areas during migration.

We considered an alternative approach that updates the transfer bitmap for all of the above cases, working as follows. The LKM identifies all the pages that join or leave the skip-over areas in the final bitmap update, by walking the page tables again to find the PFNs of the skip-over areas and comparing them with those found in the first bitmap up-

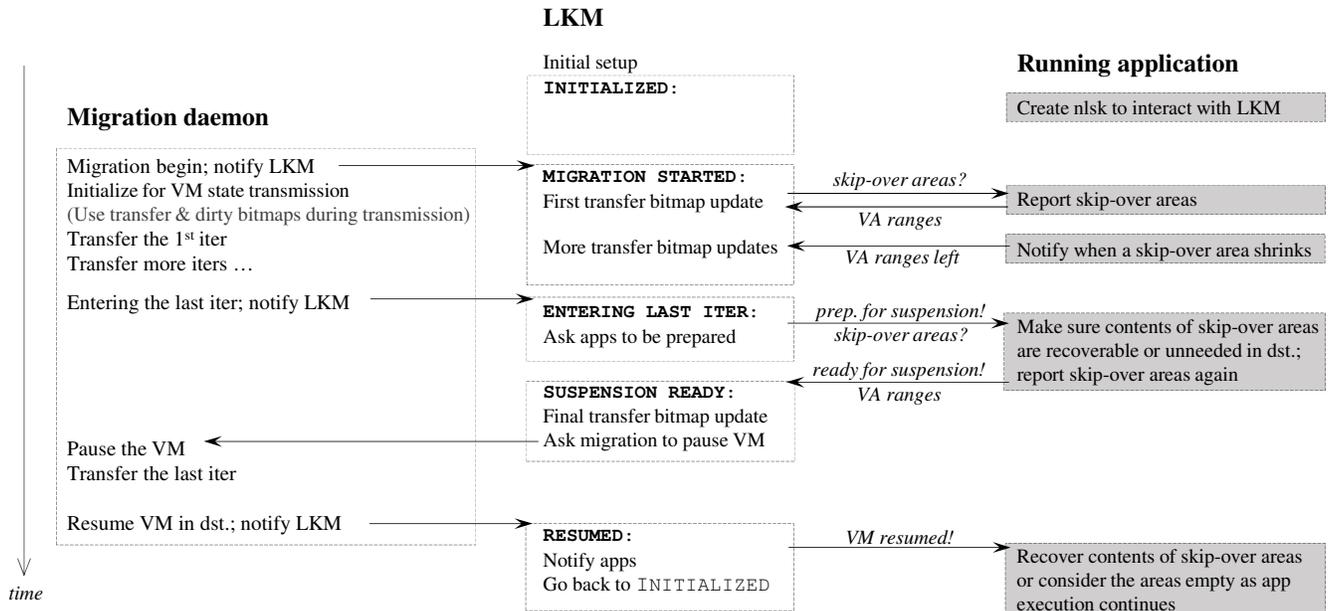


Figure 4. The workflow of application-assisted live migration; time increases vertically. The LKM coordinates between the migration daemon and application as they collaborate through different stages of migration. The horizontal arrows show the messages exchanged between them. The gray boxes describe the requirements on application for assisting in live migration in our framework.

date. In the last iteration, the migration daemon makes sure to transfer any pages dirtied after leaving a skip-over area, by considering all the pages dirtied during migration. This approach does not require applications to notify skip-over area shrinkage and performs no updates to the transfer bitmap between the first and the final updates. However, walking the page tables of all the skip-over areas slows down the completion of the final bitmap update, during which the applications may be paused. We thus defer implementing this approach while exploring its acceleration by using parallelism.

3.3.5 Migration Workflow

Figure 4 shows the workflow of application-assisted live migration. Our LKM coordinates between the migration daemon and applications in the guest as they collaborate through different stages of migration. To ease its job of coordination, the LKM transitions between states of operation based on the messages exchanged with the migration daemon and the applications, and takes different actions in each state as described next.

Before migration. Once the guest VM is created, the LKM may be loaded in preparation for possible migration. Upon loading, the LKM sets up the communication proxy and the transfer bitmap, and then enters the `initialized` state, ready for migration. If an application has memory areas that need not be transferred during migration, it creates a netlink socket as it runs in the VM, to communicate with the LKM and assist in migration.

Migration begins. The migration daemon connects with the LKM once it is started. The LKM enters the `migration started` state, and multicasts a netlink message to query running applications for skip-over areas. Based on the applications' responses, it performs the first transfer bitmap update, clearing the transfer bits of the memory pages in the skip-over areas. As the VM continues execution, the migration daemon transfers memory pages based on both the transfer bitmap and the dirty bitmap. The applications are required to notify the LKM when a skip-over area shrinks. The LKM updates the transfer bitmap immediately for the pages leaving the area, as described in Section 3.3.4.

Entering the last iteration. The migration daemon contacts the LKM again before pausing the VM and entering the `last iteration`. The LKM multicasts a netlink message, asking the applications to prepare for VM suspension. This message also queries the applications for the current VA ranges of the skip-over areas, which are needed in the final transfer bitmap update.

To prepare for VM suspension, the applications are required to ensure that when the VM resumes running in the destination, the contents of their skip-over areas, which are not transferred to the destination, are recoverable or unneeded. For example, they may need to execute to a known recoverable state, flush caches or collect garbage. Once completing the actions required, they notify the LKM, passing along the current VA ranges of the skip-over areas.

Knowing that the applications are `suspension-ready`, the LKM performs the final transfer bitmap update for any

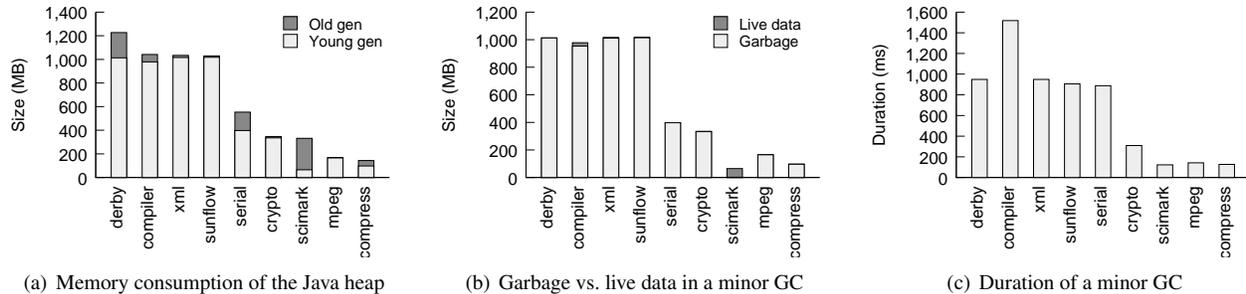


Figure 5. Java heap usage and GC behavior of sample workloads from SPECjvm2008 running in a 2GB VM; see Table 1 for workload descriptions. The Young generation of the Java heap is allowed to use at most 1GB memory.

Workload	Description
derby	Apache Derby [6] database with business logic
compiler	OpenJDK 7 front-end compiler [7]
xml	Apply style sheets to XML documents
sunflow	An open-source image rendering system [9]
serial	Serialize and deserialize primitives and objects
crypto	Sign and verify with cryptographic hashes
scimark	Compute the LU factorization of matrices
mpeg	MP3 decoding
compress	Compression by a modified Lempel-Ziv method

Table 1. Description of the SPECjvm2008 workloads used in our experiments.

expanded or shrunk space of the skip-over areas. Once completing the update, it notifies the migration daemon to suspend the VM and proceed with the last iteration. The contents of the skip-over areas should remain recoverable or unneeded until VM suspension is completed.

Migration finishes and VM resumed. After the last iteration finishes, the migration daemon activates the VM at the destination, and notifies the LKM that VM execution has resumed. The LKM in turn notifies the applications, which then recover the contents of their skip-over areas, or consider those areas empty as they continue to run. The LKM returns to the initialized state in preparation for the next migration.

4. JAVMM: Java-Aware VM Migration

Using the proposed framework for application-assisted live migration, we have designed and implemented JAVMM, which migrates VMs running Java applications assisted by JVM.

In designing JAVMM, we considered skipping transfer of both the JVM code cache and garbage in the Java heap. The code cache stores native code compiled for performance enhancements. If it is not migrated, applications can resume running interpreted in the destination, but we have observed a non-trivial performance drop in such a case. Since the code cache is small relative to the Java heap, we decided

to migrate it as usual, and focus on skipping the transfer of garbage in the Java heap.

4.1 Background on Java Heap Management

As a Java program runs, objects are created in the heap of its JVM. Most implementations of JVM (e.g., Oracle’s HotSpot and JRockit and IBM’s JVM) use a *generational* heap based on the *weak generational hypothesis* [36], i.e., most objects die young. The remainder of this paper is presented in the context of HotSpot, based on which JAVMM is prototyped. The general principles and our design of JAVMM are also applicable to other JVM implementations.

In HotSpot, the heap is divided into *Young* and *Old* generations. The Young generation is further divided into three spaces: *Eden* and two survivor spaces, *From* and *To*. Most objects are allocated in the Eden. When the Eden gets filled up, JVM performs a *minor* garbage collection (GC) to reclaim memory from garbage in the Young generation. Java (application) threads execute to a *Safepoint* [1] and pause for a minor GC, so that GC threads can move objects in the heap in a consistent manner. A minor GC copies live data in the Eden to the *To* space. Live data in the *From* space are either copied to the *To* space, or promoted to the Old generation if they have survived a number of minor GCs. At the end of a minor GC, the Eden is completely empty. The *From* and *To* spaces swap roles: *From* becomes the one that holds live data, and *To* becomes empty.

4.2 Garbage in Java Heap

To understand Java heap usage, we experiment with the SPECjvm2008 suite [8], a benchmark suite for measuring the performance of Java runtime environments. We run one workload from each benchmark category for 10 minutes in a 2GB VM, using HotSpot and its parallel garbage collector; Table 1 describes the workloads used. HotSpot is allowed to grow the Young generation to the maximum size of 1GB and the Old generation to use the rest of the VM memory.

Figure 5(a) shows the average memory consumption of the Java heap. For 8 of the 9 workloads evaluated, the Young generation grows faster and uses more memory than the Old generation; up to 98% of the heap memory is consumed

by the Young generation. Only scimark uses more memory in the Old generation, since that workload has more long-lived than short-lived objects. We observed that for derby, compiler, xml and sunflow, the Young generation quickly grows to the maximum size of 1GB to accommodate the large number of objects created by the workloads. These workloads have high object allocation rates.

A large portion of the Young generation memory may contain garbage when the lifetime of the objects is short. For all workloads except scimark, over 97% of the Young generation memory is garbage collected in a minor GC, as shown in Figure 5(b). The amount of garbage is significant for the four workloads using a 1GB Young generation. We observed that these workloads fill the Young generation and trigger a minor GC frequently, every 3 seconds or so; each minor GC reclaims almost all of the Young generation memory. Throughout workload execution, this pattern repeats, and the entire Young generation is continuously dirtied.

Figure 5(c) shows the average time required to collect Young generation garbage by a minor GC. Our results suggest that it may be faster to collect the garbage than to transfer them over a bottleneck network link. This applies to all workloads except scimark, which has exceptionally small amounts of garbage. Even for compiler, which has the longest GC duration of the workloads, its 950MB of garbage takes 1.5 seconds to be collected, but would take more than 7 seconds to be transferred over the gigabit Ethernet link in our testbed. Note, however, that for Old generation garbage, collection may not be faster than transmission. In our experiments, a full GC can take as long as 4 seconds to collect only 93MB of garbage in the Old generation.

In summary, for a wide range of Java workloads we have made the following observations.

Observation 1. The Young generation can be large and continuously dirtied, due to the high object allocation rate of the workload.

Observation 2. A significant portion of the Young generation memory may contain garbage, due to the workload’s use of short-lived objects.

Observation 3. Collecting Young generation garbage may be faster than sending them over a bottleneck network link.

4.3 JAVMM

The Young generation can generate a large number of dirty pages during the migration of a Java VM, yet many of the dirty pages may contain garbage (Observations 1 and 2). JAVMM thus skips transfer of the garbage with assistance of JVM, which knows where garbage objects are located in memory.

Garbage objects are scattered among live data, and their locations keep changing as objects become unreferenced. It is impractical to keep track of the locations of garbage objects in order to skip their migration. Instead, JAVMM enforces a minor GC to collect garbage for efficient migra-

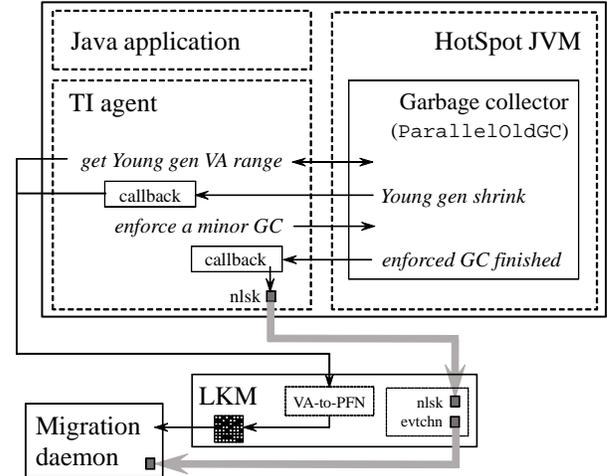


Figure 6. An overview of JAVMM, built based on the generic framework for application-assisted live migration. It is a zoom-in view of Figure 2 with Java application being the running application. JVM collaborates with the migration daemon through the LKM on behalf of Java application. We use HotSpot JVM in our prototype, and provide most of the functionalities required of JVM in a loadable agent written by JVM Tool Interface (TI).

tion, since collection may be faster than network transmission (Observation 3).

Built based on the proposed application-assisted live migration framework, JAVMM enforces a minor GC only once during migration, when running applications are notified by the LKM to prepare for VM suspension. After the enforced GC completes, the VM is suspended. In the Young generation, the Eden and To spaces are empty. The From space may contain live data, which are the data surviving the enforced GC. These live data are the only Young generation data that will be used when the VM resumes running in the destination.

JAVMM makes sure to transfer these live data in the last iteration, and throughout migration, it skips transfer of the memory pages in the Young generation, even if they are dirtied. JAVMM is thus beneficial for migrating Java VMs with a large and frequently-dirtied Young generation; this typically happens when the running Java applications are characterized by high object allocation rates.

4.3.1 System Overview

In JAVMM, JVM provides all the assistance needed for VM migration on behalf of Java applications; no modifications to Java applications are required. Figure 6 shows how JAVMM is built based on the proposed application-assisted live migration framework; our prototype uses HotSpot JVM (OpenJDK 7) and its parallel garbage collector.

We enable JVM to communicate with our LKM and collaborate with the migration daemon through the LKM.

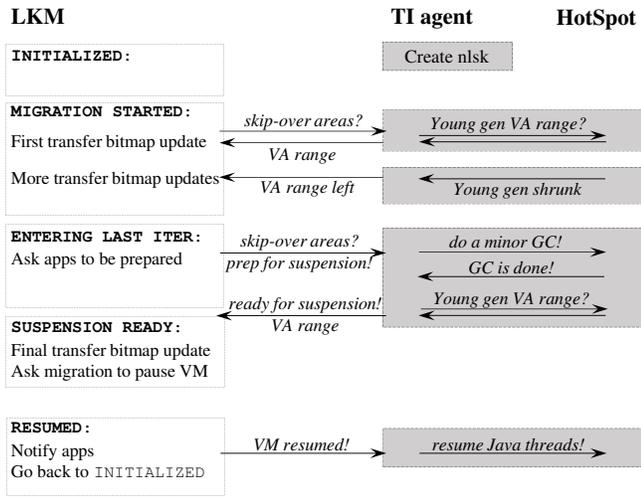


Figure 7. The workflow of JAVMM, with details of JVM’s and our TI agent’s actions to fulfill the requirements on application assisting in migration in our framework, shown in the gray boxes of Figure 4.

To provide most of the functionalities required of JVM as pluggable modules and minimize modifications to the core HotSpot code, we implemented an agent using JVM Tool Interface (TI) [4], a native programming interface for inspecting and controlling JVM. The TI agent compiles to a dynamic library to be loaded as Java applications run; it runs in the same OS process as the JVM/Java applications. JVM interacts with the LKM through the TI agent. When the functionality required is beyond the current scope of TI, we extend TI with the necessary modifications to HotSpot.

4.3.2 Workflow of JAVMM

Figure 7 shows the workflow of JAVMM. It details how JVM accomplishes the actions required of an application assisting in migration using our framework, sketched in the gray boxes of Figure 4.

As a Java application runs, our TI agent is loaded. It creates a netlink socket to communicate with the LKM.

The agent is notified by the LKM when migration begins, and is queried for skip-over areas. It obtains the VA range of the Young generation from JVM, and tells the LKM. Based on the agent’s response, the LKM performs the first transfer bitmap update. It clears the transfer bits of the Young generation pages, so the pages will not be transferred even if they are dirtied.

During migration, the agent notifies the LKM when memory pages leave the Young generation, so that the transfer bitmap can be updated. In HotSpot, memory pages may be freed from the Young generation at the end of a GC. We slightly modify HotSpot to notify when this happens, based on TI’s notification interface of GC events. A callback in our agent is invoked to pass to the LKM the VA range with mem-

ory pages freed, and the LKM immediately sets the transfer bits of the pages leaving the Young generation.

The agent is notified by the LKM again when migration is about to enter the last iteration, and is asked to prepare for VM suspension. It enforces a minor GC to collect Young generation garbage; we modify HotSpot to ensure that this GC is not silently ignored.³

As usual, Java threads execute to a Safepoint and pause, and JVM performs a collection. Once the collection is finished, a callback in our agent is executed; at this time, the Eden and To spaces are empty, and the Java threads are still paused. Without giving JVM control to release the Java threads from the Safepoint and resume their execution, the agent notifies the LKM that the application is ready for VM suspension. The Java threads are thus prevented from using the heap, and this ensures the Eden and To spaces remain empty until VM suspension is completed.

Along with the notification of the application being suspension-ready, the agent passes to the LKM the current VA range of the Young generation and also that of the occupied From space, which contains the live data surviving the enforced GC. Based on these information, the LKM performs the final transfer bitmap update. It considers the occupied From pages “leaving” the Young generation, and sets their transfer bits, in order to ensure transfer of live Young generation data in the last iteration.

Once the final transfer bitmap update is completed, the migration daemon suspends the VM, and finishes migration with the last iteration. When the VM resumes in the destination, our agent is notified by the LKM. It returns control to JVM, which in turn releases the Java threads from the Safepoint. The Java application then resumes execution with all live data available in the destination.

5. Evaluation

We now evaluate JAVMM in comparison with Xen VM live migration, which is a traditional pre-copy approach that is agnostic of the applications running in the migrating VM.

5.1 Experimental Setup

Our evaluation uses both real-life applications and benchmarks from SPECjvm2008 [8], the same benchmark suite used to profile Java heap usage in Section 4.2.

We run each workload for 10 minutes in a VM configured with 2GB memory and 4 vCPUs. Halfway through the workload execution, we migrate the VM, between two HP Proliant BL465c blades in the same gigabit Ethernet LAN; each blade is equipped with two dual-core AMD Opteron 2.2 GHz CPUs and 12GB RAM.

Alongside each workload, we run a custom analyzer that sends out the number of operations completed by the work-

³HotSpot may ignore GC requests when several requests are enqueued at about the same time due to simultaneous allocation failures in multiple threads—only one of these requests needs to be executed.

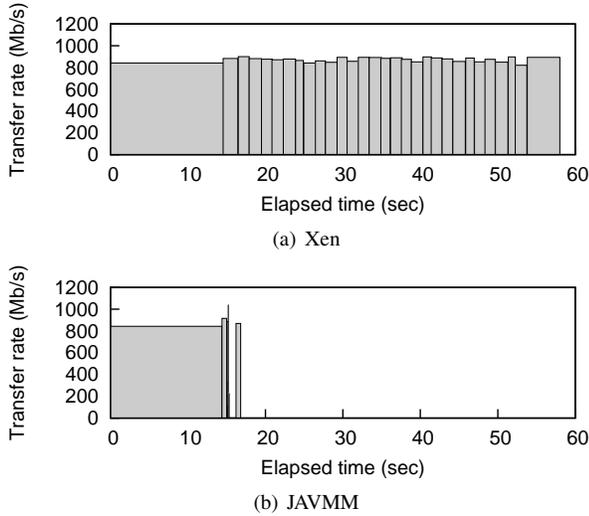


Figure 8. Progress of migrating a VM running the compiler workload from SPECjvm2008. Each box represents a migration iteration; the width shows the duration and the area shows the amount of traffic sent. In (b), the second last iteration of JAVMM generates little network traffic while waiting for the workload to execute to a Safepoint (0.7 sec) and a minor GC to be done (0.1 sec).

load once every second. We observe workload throughput from outside of the VM using a time source that is not affected by temporary suspension of the VM, which happens before completing migration.

Each experiment is repeated at least three times. Unless otherwise mentioned, we report the average of the measurements, and show 90% confidence intervals in bar graphs.

5.2 Progress of Migration

We begin by analyzing how a Java VM is migrated iteratively by Xen and JAVMM, respectively. We use a VM running the compiler workload from SPECjvm2008 as an example; see Table 1 for the workload description. Figure 8 plots the progress of migrating the VM in an experimental run. We plot each iteration by a box, and show the duration and the amount of traffic sent by the width and area of the box, respectively.

In the first iteration, Xen and JAVMM perform equally well. They both skip sending about 500MB of memory, as shown in Figure 9. Xen skips over pages that are dirtied before transmission, since these pages will be transferred in the next iteration, which makes transferring them in the current iteration redundant. Prototyped on Xen, JAVMM also skips over pages that are already dirtied, and in addition, all Young generation pages. The workload is using a 512MB Young generation when migrated, and most of the space is skipped over by both Xen and JAVMM in the first iteration.

Xen and JAVMM start to progress differently from the second iteration. Although they both have more than 500MB

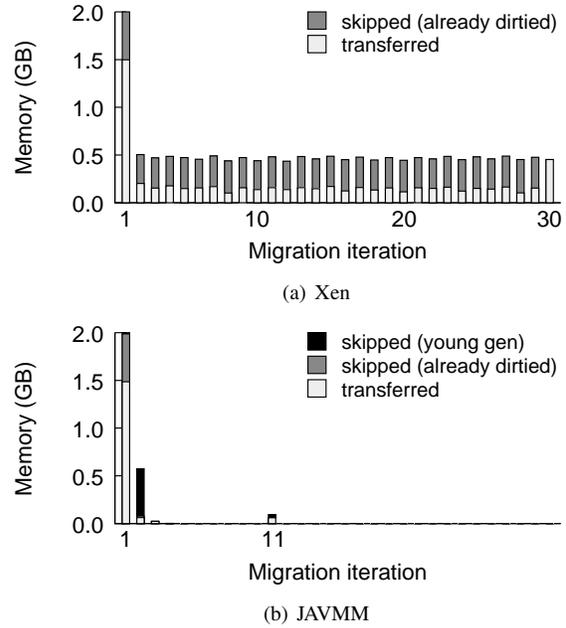


Figure 9. Amount of memory processed when migrating a VM running the compiler workload from SPECjvm2008. In (b), the 4–10th iterations of JAVMM each processes less than 2MB of dirty memory.

of dirty memory pending transmission in the second iteration, they transfer different amounts. JAVMM sends only 64MB of the dirty memory, skipping over both repeatedly dirtied pages and Young generation pages. Xen has to send more than 200MB of the dirty memory, since it can only skip over repeatedly dirtied pages.

Since JAVMM sends less dirty data, it finishes the second iteration faster, during which less memory gets dirtied. As a result, it has even less dirty data to send in the third iteration. JAVMM reduces the amount of memory transfer effectively as iterations progress. After 10 iterations, little dirty memory remains to be sent. JAVMM then finishes migration with a short stop-and-copy at the 11th iteration, using 17 seconds and sending 1.6GB of network traffic.

However, for Xen, the amount of memory transfer does not decrease over the iterations. Migration is forced to enter stop-and-copy when it reaches the maximum 30 iterations allowed by Xen’s default. The stop-and-copy takes long, since over 400MB of dirty memory remains to be sent. Xen finishes migration taking 58 seconds and sending 6.1GB of network traffic, i.e., over 3x longer time and more traffic than JAVMM.

5.3 Performance of Migration

Next, we evaluate JAVMM for workloads with different characteristics of Java heap usage.

Workload characterization. When profiling sample workloads from SPECjvm2008 in Section 4.2, we found the

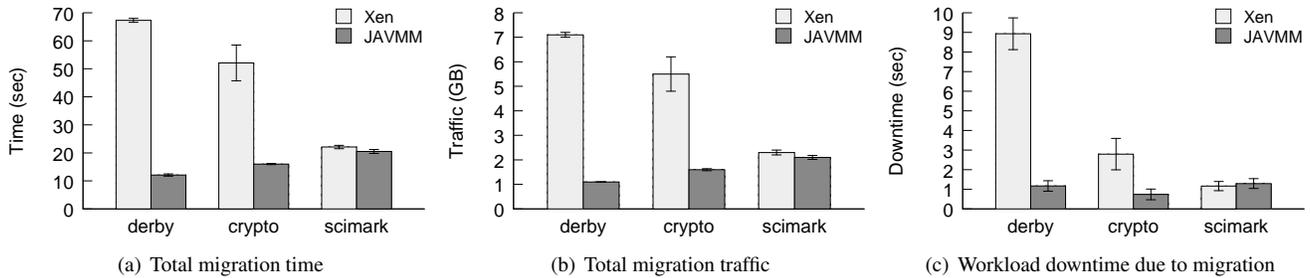


Figure 10. Performance of JAVMM and Xen live migration for workloads with different characteristics of Java heap usage.

Workload	Max allowed Young gen (MB)	Observed when migrated Young gen (MB)	Old gen (MB)
derby	1024	1024	259
crypto	1024	456	18
scimark	1024	128	486

Table 2. Experimental settings of workloads with different characteristics of Java heap usage.

workloads fall in the following three categories according to Java heap usage; see Table 1 for description of the workloads.

- **Category 1.** The workload has a high object allocation rate, and uses mostly short-lived objects. As a result, the Young generation quickly grows to the maximum size. The derby, compiler, xml and sunflow workloads are in this category.
- **Category 2.** The workload has a medium object allocation rate, and uses mostly short-lived objects. The Young generation grows faster than the Old generation, albeit not maximally utilized. The serial, crypto, mpeg and compress workloads are in this category.
- **Category 3.** The workload has a low object allocation rate, and uses mostly long-lived objects. It thus has a small Young generation and a large Old generation. Scimark is the only workload in this category.

Our observations on object allocation rates are consistent with the measurements by other researchers [33].

We evaluate JAVMM using one workload from each category. For Category 1, which is the most favorable workload scenario for JAVMM, we evaluate derby; in the workloads of this category, derby uses the largest Old generation, which JAVMM has to transfer. For category 2, we evaluate crypto. For category 3, which is the least favorable workload scenario for JAVMM, we evaluate scimark.

Derby, crypto and scimark are all CPU-intensive workloads. They use up 90% of CPU, and perform no network I/Os. Table 2 shows their experimental settings. While each workload can use a maximum 1GB Young generation, when

migrated, the Young generations of derby, crypto and scimark are using 1GB, 0.4GB and 0.1GB of memory, respectively.

How fast does JAVMM migrate a Java VM? Figure 10(a) shows the time required to migrate the VMs running the three workloads. JAVMM migrates the derby VM fastest, taking only 12 seconds. Compared to Xen, which takes over a minute to migrate the VM, JAVMM reduces the migration time by 82%. JAVMM also achieves a 69% reduction of migration time for the crypto VM. For scimark, JAVMM can skip over little Young generation memory. It migrates the VM using a comparable amount of time as Xen.

How much resource does JAVMM use for migration? Figure 10(b) shows the amount of network traffic transferred to migrate the VMs. For derby and crypto, JAVMM migrates the VM sending even less traffic than the VM size, while Xen sends up to 3.5x the VM size of migration traffic. Compared to Xen, JAVMM reduces migration traffic for derby and crypto by 84% and 72%, respectively. For scimark, JAVMM achieves a 10% reduction of migration traffic.

Thanks to the reduced data transfer, JAVMM also uses up to 84% less CPU time than Xen in migrating the VMs. In these experiments, JAVMM uses at most 1MB of memory for the transfer bitmap and PFN cache.

How much does JAVMM affect application performance? Figure 11 shows the throughputs of the workloads. For each workload, the VM is migrated after the workload runs for 300 seconds. Using JAVMM, the workload experiences no noticeable throughput degradation during migration, except the short pause before migration finishes. When migrated by Xen, the workload can experience an extended downtime.

Figure 10(c) shows the workload downtime. The downtime includes the time spent in the last iteration and the resumption time. The resumption time is required to reconnect VM devices and activate VM execution in the destination; this time is short, only about 170 ms in our measurements. For JAVMM, the downtime also includes the time required to finish the enforced GC while the workloads pause at a Safepoint, as well as the time required by the final transfer bitmap update. The final bitmap update is completed quickly, within 300 μ s in all our experiments.

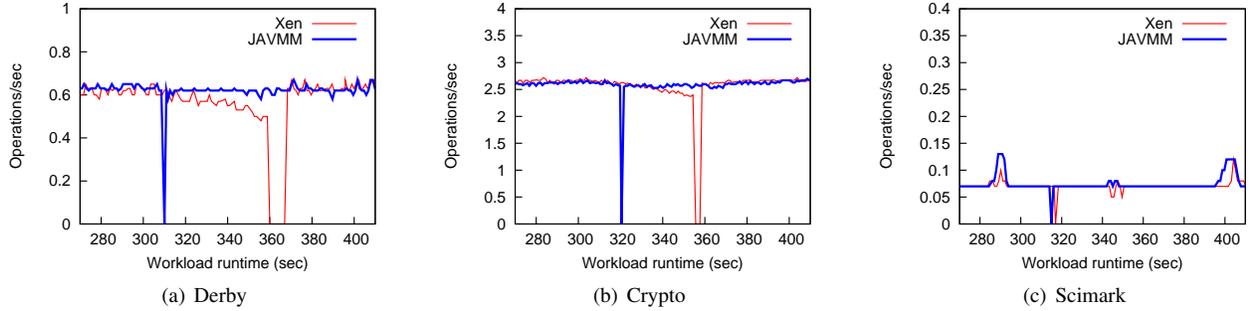


Figure 11. Effect of VM migration on the throughput of running application, *i.e.*, the number of operations completed per second. Migration begins after the application runs for 300 seconds.

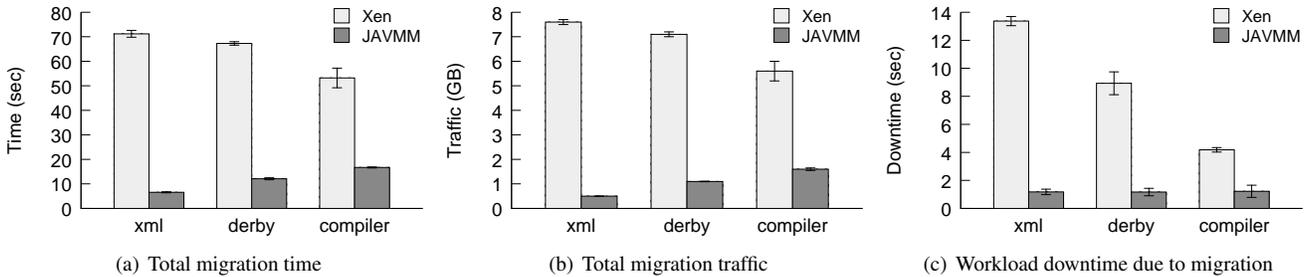


Figure 12. Performance of JAVMM and Xen live migration for Category 1 workloads with different size Young generations.

Derby experiences 1.2 seconds of downtime when the VM is migrated by JAVMM, 83% shorter than the 9-second downtime when the VM is migrated by Xen. Derby dirties the 1GB Young generation rapidly, but JAVMM can still reduce the amount of memory transfer iteratively, by skipping transfer of Young generation pages. In the last iteration, JAVMM sends only 11MB of dirty data skipping over Young generation garbage, while Xen has over 900MB of dirty data to be sent. JAVMM thus reduces the downtime of derby significantly compared to Xen, even though it uses 0.9 second to finish the enforced GC; the GC duration can be further shortened with Java heap fine-tuning or increased parallelism. For crypto, JAVMM also achieves a 73% shorter downtime than Xen.

However, for scimark, JAVMM imposes a 10% longer downtime than Xen. Scimark is paused for 1.2 and 1.3 seconds when the VM is migrated by Xen and JAVMM, respectively. For this workload, JAVMM takes time to perform the enforced GC, but the amount of data to be transferred in the last iteration is not reduced. Most of scimark’s objects are long-lived. They survive the GC enforced, and must be sent in the last iteration.

Summary. JAVMM is advantageous in migrating the VMs running derby and crypto, representatives of workloads with a non-trivial object allocation rate and mostly short-lived objects. Compared to Xen, JAVMM migrates these VMs achieving shorter completion time, smaller network traf-

Workload	Max allowed Young gen (MB)	Observed when migrated Young gen (MB)	Old gen (MB)
xml	1536	1536	28
derby	1024	1024	259
compiler	512	512	86

Table 3. Experimental settings of workloads with a high object allocation rate and mostly short-lived objects.

fic and shorter downtime. Scimark represents workloads with a low object allocation rate and mostly long-lived objects. Compared to Xen, JAVMM migrates this VM with a slightly longer downtime, although it achieves comparable, or slightly better migration time and traffic.

5.4 Impact of Young Generation Size

We conducted a second set of experiments for the workloads most favorable for JAVMM, Category 1 workloads that has a high object allocation rate and mostly short-lived objects. We evaluate the benefit of using JAVMM for these workloads with varying sizes of Young generation, focusing on the same three evaluation questions discussed in Section 5.3.

We experimented with derby and two additional workloads, xml and compiler, from Category 1. All three workloads are CPU-intensive and without network I/Os. We specify different maximum sizes for the Young generations of the

workloads, as shown in Table 3. When migration begins, the Young generations of xml, derby and compiler all reach the maximum sizes. They are using 1.5GB, 1GB and 0.5GB of memory, namely, 75%, 50% and 25% of the VM memory, respectively.

Figure 12(a) shows the time required to migrate the VMs running the three workloads. With high object allocation rates, the workloads dirty the entire Young generation space rapidly. For Xen, the larger the Young generation, the more dirty memory are repeatedly transferred, and the longer it takes to migrate the VM. On the contrary, JAVMM migrates the VMs with larger Young generations faster, since more dirty memory are skipped over. JAVMM thus achieves greater reductions of migration time for the VMs with larger Young generations, than Xen. For the xml, derby and compiler workloads, JAVMM migrates the VM using 91%, 82% and 69% less time than Xen, respectively.

A similar trend is observed for the amount of network traffic sent for migrating the VMs, as shown in Figure 12(b). For JAVMM, the larger the Young generation, the less migration traffic is sent, and it achieves a greater traffic reduction than Xen. JAVMM sends 93% less traffic than Xen to migrate the VM running xml, which has the largest Young generation of the workloads.

Figure 12(c) shows the downtime incurred by the workloads before migration is completed. When the VM is migrated by Xen, the workloads with larger Young generations incur longer downtimes. A large portion of the Young generation keeps getting dirtied until the VM is paused for the last iteration, due to the workloads' high object allocation rates. Xen has up to 1.5GB of data to send in the last iteration, resulting in up to 13 seconds of downtime.

For JAVMM, there is not a direct relationship between the downtime and the Young generation size, since the downtime is also affected by other factors, *i.e.*, the duration of the enforced GC and the amount of surviving data to be sent in the last iteration. The three workloads experience about 1.2 seconds of downtime when the VM is migrated by JAVMM, up to 91% shorter than their respective downtimes incurred when migrated by Xen.

6. Discussions and Future Extensions

When to use JAVMM? JAVMM is most beneficial for the cases which are most problematic to traditional pre-copy approaches—when the VM to be migrated runs Java applications with large Young generations and high object allocation rates.

In some cases, JAVMM should be used with consideration of the resulting application downtime. The first is when the application requires long minor GCs, since the duration of the enforced GC increases downtime. The second is when the application has a high object survival rate. Many objects may survive the enforced GC and must be transferred during stop-and-copy. Scimark is such an example. The third is

when the application is read-intensive, for which traditional pre-copy approaches can reduce downtime effectively; the GC enforced by JAVMM is likely to increase downtime.

Use JAVMM for large VMs with fast networks. Our evaluation has shown benefits of JAVMM by migrating a 2GB VM over a gigabit Ethernet. These benefits remain as VMs configured with tens or hundreds of GBs of memory are migrated over 10 Gbps or faster networks, since in such scenarios, the VM processing power, application memory footprints and memory-dirtying rates likely increase proportionally. As we continue to deploy JAVMM in upgraded environments, the underlying network may remain as much a bottleneck as in our current testbed.

Use JAVMM with other garbage collectors. JAVMM works with garbage collectors that aggregate live data while application threads are paused. With live data aggregated, JAVMM can easily identify and transfer the data surviving the enforced GC in the last iteration. Having application threads paused during the enforced GC helps JAVMM ensure the heap space collected remains empty until the migrated VM resumes in the destination. Such garbage collectors are often referred to as compacting and non-concurrent, and most Young generation collectors fall in this category. We are particularly interested in porting JAVMM to run with collectors that use non-contiguous VA ranges for the Young generation for performance evaluation and optimization. HotSpot's garbage-first garbage collector [17] is one such example.

Apply the proposed application-assisted live migration framework to other applications. In addition to Java applications, JAVMM can be used for applications written in other languages that run on JVM and use JVM's garbage collectors. For example, Jython, an implementation of Python, and JRuby, an implementation of Ruby, can leverage JAVMM as it is. As a matter of fact, the proposed framework can be applied to any application runtime that is GC-based, provided that the runtime has a compacting, non-concurrent garbage collector; the Microsoft .NET framework is one such example. In all these applicable cases, only the application runtime, not every individual application, needs to be modified to run in our framework.

Our framework can also be applied to applications with caching functionality; examples include Apache HTTP server, which uses a web cache, memcached, a distributed caching system, and Oracle Coherence, a proprietary distributed caching system that replicates cache data for backup. The application can specify a portion of its caching memory space to be skipped over by the migration daemon, effectively shrinking the cache in the destination. To reduce the resulting performance impact, when informed by our framework to shrink the cache, the application can purge the least frequently and/or the least recently used cache data; applications with cache data redundancy like Oracle Coherence can consider purging backup cache data. Note that after the

cache shrinkage, the remaining valid data need to be compact in the caching memory space for our framework to work well.

Support large and multiple applications. In our proposed framework, the LKM updates the transfer bitmap on applications' behalf. It can coordinate concurrent bitmap updates from multiple applications, and prevent the applications from manipulating others' memory. While we have implemented the LKM to notify a set of applications by multicast, care is needed to collect responses from all of the applications and handle any straggler. We are also investigating parallelization of transfer bitmap updates to handle large skip-over areas efficiently.

Enhance the proposed framework for security. Since our approach uses applications' assistance during live migration, it does require applications running in the migrating VM to be benign and cooperative. While application authentication is a broad subject that is beyond the scope of this paper, there is room for framework enhancements towards security. For example, in the current framework, if an application does not cooperate by responding to the LKM's queries and requests at different stages of migration, the migration process can incur unbounded delays. One way to prevent this is to incorporate timeouts in the interactions between the LKM and the applications.

Make the proposed framework intelligent. Our focus has been providing *mechanisms* for live migration to use applications' assistance in the proposed framework. We are extending our efforts to add intelligence into the framework, devising *policies* based on which the framework can adapt to workload and resource dynamics for best migration performance.

For example, while an application may report all of its skip-over areas to the LKM, the LKM can make a more informed decision on which ones to skip transferring in a particular migration event, if information such as the sizes of the VM and the skip-over areas, the current network speed and the average memory dirtying rate is available and taken into account. Another example is in the case of JAVMM. We have identified workload scenarios in which JAVMM should be used with consideration of the resulting application downtime, as discussed above. We can incorporate this knowledge back to the system. In the simplest form, we may have the LKM turn off JAVMM and let migration proceed with traditional pre-copying when those workload scenarios are encountered.

Incorporate compression. Compression can reduce the amount of memory transfer during migration, but it is CPU-expensive. To exploit compression at a lower CPU cost, we are extending the framework to compress only the memory pages that have not been skipped over. The transfer bitmap can use multiple bits per VM memory page to indicate the suitable compression methods to apply before sending the page contents over the network.

7. Conclusions

In this paper, we have proposed application-assisted live migration, skipping transfer of selective VM memory pages based on application semantics. We have built a generic framework for the proposed approach, which is then used to build JAVMM, a system that migrates VMs running Java applications skipping transfer of garbage in Java memory. Our experimental results have shown that JAVMM can migrate a Java VM with up to more than 90% less completion time, less network traffic and shorter application downtime than Xen live VM migration, which is agnostic of application semantics. JAVMM also incurs a lower CPU cost than Xen live VM migration and a negligible memory overhead. In JAVMM, JVM is enabled to provide all the assistance needed for migration on behalf of Java applications; no modifications to Java applications are required by JAVMM for efficient migration of a VM.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rong Chen, for comments that improved this paper. This paper is based upon work supported and funded in part by Oracle America, Inc. and the US Airforce Office of Scientific Research (AFOSR) under Grant No. FA9550-10-1-0393.

References

- [1] HotSpot glossary of terms. <http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html>.
- [2] HotSpot virtual machine. <http://openjdk.java.net/groups/hotspot>.
- [3] Popularity of Java applications. <http://www.java.com/en/about>.
- [4] JVM Tool Interface (TI). <http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html>.
- [5] Four Java cloud platforms reviewed. <http://www.javaworld.com/article/2078443/mobile-java/four-java-cloud-platforms-reviewed.html>.
- [6] Apache Derby database in Java. <http://db.apache.org/derby>.
- [7] OpenJDK 7. <http://openjdk.java.net/projects/jdk7/>.
- [8] The SPECjvm2008 benchmark suite. <http://www.spec.org/jvm2008>.
- [9] Sunflow open source rendering system. <http://sunflow.sourceforge.net>.
- [10] J. F. Alamdari and K. Zamanifar. A reuse distance based pre-copy approach to improve live migration of virtual machines. In *Proceedings of the 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, pages 551–556, 2012.
- [11] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: Efficient idle desktop consolidation with partial VM migration. In *Proceed-*

- ings of the 7th ACM European Conference on Computer Systems, pages 211–224, 2012.
- [12] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.
- [13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machine. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.
- [14] T. Das, P. Padala, V. N. Padmanabhan, R. Ramjee, and K. G. Shin. LiteGreen: Saving energy in networked desktops using virtualization. In *Proceedings of the USENIX Annual Technical Conference*, 2010.
- [15] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th International Symposium on High Performance and Distributed Computing*, pages 135–146, 2011.
- [16] U. Deshpande, B. Schlinker, E. Adler, and K. Gopalan. Gang migration of virtual machines using cluster-wide deduplication. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 394–401, 2013.
- [17] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, pages 37–48, 2004.
- [18] M. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 51–60, 2009.
- [19] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi. Enabling instantaneous relocation of virtual machines with a lightweight VMM extension. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 73–83, 2010.
- [20] B. Hu, Z. Lei, Y. Lei, D. Xu, and J. Li. A time-series based precopy approach for live migration of virtual machines. In *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, pages 947–952, 2011.
- [21] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 11–20, 2007.
- [22] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive memory compression. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–10, 2009.
- [23] C. Jo, E. Gustafsson, J. Son, and B. Egger. Efficient live migration of virtual machines using shared storage. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 41–50, 2013.
- [24] A. Koto, H. Yamada, K. Ohmura, and K. Kono. Towards unobtrusive VM live migration for cloud computing platforms. In *Proceedings of the Asia-Pacific Workshop on Systems*, 2012.
- [25] Z. Liu, W. Qu, T. Yan, H. Li, and K. Li. Hierarchical copy algorithm for Xen live migration. In *Proceedings of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 361–364, 2010.
- [26] F. Ma, F. Liu, and Z. Liu. Live virtual machine migration based on improved pre-copy approach. In *IEEE International Conference on Software Engineering and Service Sciences*, pages 230–233, 2010.
- [27] U. F. Minhas, S. R. B. Cully, A. Aboulmaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *PVLDB*, 4(11), 2011.
- [28] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 23–32, 2007.
- [29] 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.
- [30] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2005.
- [31] T. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 337–350, 2013.
- [32] D. P. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D. M. L. Brown, and J. Nieplocha. Transparent system-level migration of PGAS applications using Xen on InfiniBand. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 74–83, 2007.
- [33] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009.
- [34] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.
- [35] P. Svård, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th Conference on Virtual Execution Environments*, pages 111–120, 2011.
- [36] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.
- [37] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, pages 229–242, 2007.
- [38] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 88–96, 2010.