vMocity: Traveling VMs across Heterogeneous Clouds

Cheng Cheng[*], Zhui Deng[†], Zhongshu Gu[‡], Dongyan Xu[§]
[*][§]*Department of Computer Science, Purdue University*
[†]*Apple Inc.*
[‡]*IBM T.J. Watson Research Center*
Email: {[*]chengcheng,[§]dxu}@cs.purdue.edu, [†]dengd03@gmail.com, [‡]gzs715@gmail.com

Abstract—**Current IaaS cloud providers typically adopt different underlying cloud infrastructures and are reluctant to provide consistent interfaces to facilitate cross-cloud interoperability. Such status quo significantly complicates inter-cloud virtual machine relocation and impedes the adoption of cloud services for more enterprises and individual users. In this paper, we propose vMocity, a middleware framework enabling VM relocation across heterogeneous IaaS clouds. vMocity extends the principles of cold migration and decouples VM's storage stack from their underlying virtualization platforms, which presents a homogeneous view of storage to cloud users. We deploy our prototype system across three representative commercial cloud platforms — Amazon EC2, Google Compute Engine, and VMware vSphere-based private cloud. Compared to existing approaches on both synthetic and real-world workloads, vMocity can significantly reduce the disruption time, up to 27 times shorter, of relocated services and boost the recovery time, up to 1.8 times faster, to pre-relocation performance level. Our results demonstrate that vMocity is efficient and convenient for relocating VMs across clouds, offering freedom of choice to customers when facing a market of IaaS clouds to align with business objectives (cost, performance, service availability, etc.)**

1. Introduction

Infrastructure as a Service (IaaS) clouds provide significant flexibility and versatility of resource provisioning to help organizations manage their IT infrastructures at a low cost. Due to such technical and economic benefits, increasing number of enterprises and individual users are migrating their in-house applications to public/private cloud infrastructures (e.g., Amazon EC2, Google Compute Engine, VMware vCloud Air, IBM SoftLayer, Rackspace, OpenStack, and Microsoft Azure). According to a cloud computing market report, by 2017 the value of cloud services will increase to \$127 billion (compared to \$37.8 billion in 2010).

IaaS clouds typically use a *pay-as-you-go* business model to ensure customers pay for the computing resources they consume. However, such standard IaaS model binds each virtual machine (VM) instance tightly with a specific cloud provider, which may lead to several dependability and flexibility problems. For example, a sudden service disruption (caused by natural disasters, malicious cyber-attacks, system mis-configurations, etc.) may cause its tenants to lose accesses to their already-paid resources. This eventually interrupts the business continuity of tenants' online services. Moreover, customers wish to benefit from being able to relocate workloads across multiple cloud providers to:

a) Avoid Vendor Lock-in. Since application stack customized for a specific cloud infrastructure is typically locked into its cloud provider, rebuilding the same application stack for a different cloud takes non-trivial engineering efforts, even if the administrator leverages some predefined image as a starting point. Therefore, enterprise users are interested in better inter-cloud application portability to assuage their fears of cloud lock-in. When deploying applications on clouds, being able to relocate VMs among cloud providers grants users the flexibility to switch to the most appropriate cloud providers with respect to reliability, functionality, performance, and cost at any time.

b) Support Cloud Brokerage. While there exist many IaaS cloud providers on the market, none offers a common platform for cooperative inter-cloud usage. Cloud broker service [1] [2] [3] is introduced for mediating between different cloud providers to facilitate the use of cross-provider cloud offerings in accordance with service-level agreement (SLA) and in alignment with business objectives (e.g., cost, service availability, etc.). The capability to relocate VMs rapidly among competitive cloud providers is the key enabling mechanism so that the cloud broker service can execute its decision and relocate VMs seamlessly among different cloud providers.

Unfortunately, it is difficult, if ever possible, to enable such desired inter-cloud relocation under current technologies due to following challenges:

a) Limited Cloud User Privilege. In current IaaS clouds, users only have restricted operation privileges and are not able to directly manipulate or relocate their VM image files, making the process of exporting or importing VM images to arbitrary destinations difficult. A few IaaS cloud providers, such as Amazon EC2 and Google Compute Engine, provide APIs/tools for users to import/export VMs, but those APIs are cloud-specific and fundamentally limited, resulting in non-trivial administrative efforts when users decide to relo-

Deng and Gu contributed to the work while at Purdue University.

cate their VMs to a different cloud provider.

b) Lack of Interoperability. IaaS clouds are not interoperable. The significant differences in VM abstractions or underlying hypervisor services make it infeasible to transparently relocate a VM (or a set of VMs) between competing public clouds, or between a private cloud and a public cloud. Depending on the underlying virtualization techniques (e.g., Amazon EC2 is based on Xen [4]; while Google Cloud uses KVM [5]), VM images are stored and accessed in different ways. At the same time, standardization, or provider-centric homogenization is unlikely to be adopted by all clouds. For instance, Amazon Elastic Block Store (EBS) offers persistent block level storage volumes to serve Amazon EC2 instances, while Google provides several types of data disks that can be used as persistent storage for VM instances in the Google Compute Engine.

c) Relocation Efficiency. In a local-area network, relocating a VM does not necessarily require relocating VM images because hosts can be configured to share storage devices. However, remotely mounting and sharing storage over wide-area network cannot achieve satisfactory I/O performance [6]. Moreover, VMs created by users are highly customized and may be extremely heavy-weight. Relocating a whole VM from the source cloud to the destination cloud may incur significant service disruption time, especially when VM image transportation is performed over the wide-area network with limited bandwidth and high latency. Such service disruption time severely impacts the feasibility of inter-cloud VM relocation. It is necessary to develop new techniques to move VMs efficiently, while minimizing application performance degradation.

There exist some research efforts to facilitate workload relocation across cloud providers. Nested virtualization techniques, such as HVX [7] and Xen-Blanket [8], use a second-layer virtualization to provide a homogeneous virtual platform interface. However, they limit the choice of virtualization platforms and impose non-trivial performance overhead. Lightweight (OS-level) virtualization technologies, such as Linux Containers [9], provide workload relocation at the process level. However current available techniques are mostly Linux based and they heavily rely on the kernel version of the operating system that hosts the containers, thus greatly limiting their flexibility. Moreover, it is infeasible to use a VM image across different cloud providers without in-depth adaptation due to their different image formats, which eventually leads to a high administrative and monetary cost. Some other efforts (e.g., OpenStack) are also trying to create homogeneity for various cloud services, such as alerting, API, authentication, and billing.

To complement the above solutions, we propose vMocity, a novel VM persistent storage hosting framework, to support seamless inter-cloud VM relocation. Instead of leveraging live migration techniques, vMocity extends the principles of cold migration. By decoupling VM's persistent storage access interface from the underlying virtualization platform, vMocity enables cloud users to switch VM hosting platforms instantly and transparently, which opens a novel operating paradigm in the cloud era. Instead of relying on cloud providers to build a homogeneous storage access infrastructure, we advocate a *customer-centric* view of storage homogenization by deploying vMocity framework as a lightweight middleware layer, allowing users to run their VMs on any cloud platform without complex storage adaptation. Our evaluation demonstrates the effectiveness of vMocity in both controlled and real environments. For example, using vMocity to relocate a MySQL server VM reduces service disruption time by a factor of 27 and resumes the VM to offer stable performance 1.8 times faster than state-of-the-art approaches.

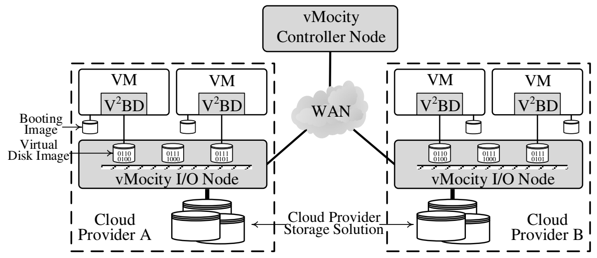


Figure 1: Architecture of vMocity Framework.

The remainder of the paper is organized as follows. Section 2 presents the design of vMocity framework. The implementation details are in Section 3. Detailed performance evaluation in both controlled and real cloud environments using diverse sets of workloads is presented in Section 4. Finally, Section 5 surveys related work and Section 6 concludes the paper.

2. Design of vMocity

We begin with an architectural overview of vMocity. Next, we examine the internals of the *vMocity I/O Node* with various design supports for efficient relocation and the virtual block driver used for connecting VMs to vMocity.

2.1. Architecture Overview

Figure 1 depicts the overall architecture of the vMocity framework, which mainly consists of three components: *vMocity Controller Node*, *vMocity I/O Node*, and *vMocity Virtual Block Driver (V^2BD)*. The *Controller Node*, the central control point, keeps disk mapping and *I/O Nodes* topology information. The *I/O Nodes* running in different clouds form a virtual disk image store and expose a unified block interface to V^2BD for accessing virtual disk image. On a VM instantiation, V^2BD queries the *Controller Node* and establishes connections with appropriate *I/O Node*. V^2BD can also switch its backing storage transparently.

Under vMocity, a VM is composed of two types of images: a *vMocity Booting Image* and one or more *vMocity Virtual Disk Images*. The *Booting Image* is a provider-specific VM image, which contains a bootloader, a kernel, and a customized init ramdisk. It gets deployed to a specific cloud and is used for instantiating an individual VM. The *Virtual Disk Image*, hosted in the image store exposed by *I/O Nodes*, is a user-centric virtual disk image populated

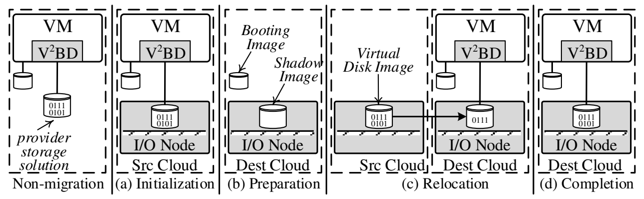


Figure 2: Different Phases of Migrating a VM with vMocity.

by vMocity, which stores the actual VM data, such as the root file system. A *Virtual Disk Image* is composed of an array of fixed-sized blocks. Those blocks are stored as object files and managed by the underlying object store [10] [11] expanding through all *I/O Nodes*. Using object stores significantly simplifies managing and accessing objects across *I/O Nodes*.

Figure 2 illustrates a concrete example of VM relocation across two clouds using vMocity framework. In the *non-migration phase*, users can directly use virtual disks offered by the cloud provider without introducing additional monetary and performance cost. vMocity framework is only required during the relocation phases. Starting from the *initialization phase*, the VM to be relocated is running in the source cloud, within which the VM's virtual disk image is hosted on the corresponding *I/O Node*. Once the user issues a request to the *Controller Node* for initiating the VM relocation, the running VM is going to be terminated safely on the source cloud. In the *preparation phase*, The *Controller Node* extracts VM configuration metadata (number of vCPUs, memory size, network capacity, etc.) of the original VM from the source cloud, and prepares a new *Booting Image* along with a set of empty shadow virtual disk images, which are going to be filled with the content from its peer in the source cloud, in the destination cloud for the new VM. The VM can be powered on instantly in the *relocation phase*. It loads V^{2}BD module during the initialization process from the attached *Booting Image*, generates contextual information, and sends queries to the *Controller Node* for connecting the appropriate *I/O Node*. The contextual information includes the hosting cloud provider and location (e.g., regions, zones) details of the VM instance. Based on the contextual information, the *Controller Node* directs V^{2}BD to establish iSCSI connection with an appropriate *I/O Node* for accessing virtual disk image. The content from the source virtual disk image are relocated and filled to the shadow virtual disk image in the destination cloud. Once the relocation process finishes (*completion phase*), the virtual disk image on the *I/O Node* of the source cloud can be safely destroyed without incurring extra storage for duplicated virtual disks. Thanks to the flexibility offered by V^{2}BD, the backing storage device of the VM can be transparently switched between vMocity and native provider's storage solutions.

2.2. Internals of vMocity I/O Node

vMocity I/O Node, the building block of vMocity framework, is deployed within the same availability zone and exposes virtual disks to VMs for connections. *vMocity I/O Nodes* hide the heterogeneity of diversified IaaS cloud storage solutions. On one hand, *vMocity I/O Nodes* decouple the dependency between the storage virtualization stack and the underlying hypervisor platform by providing a unified block interface for accessing VM virtual disk image. On the other hand, to be compatible with various cloud providers, *vMocity I/O Nodes* still utilize cloud providers' native storage solutions as the backing storage devices for hosting image stores. For instance, a *vMocity I/O Node* running in Amazon EC2 is configured to use EBS volume as the storage backing device, whereas a *vMocity I/O Node* running in Google Compute Engine adopts Google's version of persistent block storage. Hosting and accessing virtual disk images within vMocity framework allows users to manage and relocate virtual disk images conveniently. In addition, *vMocity I/O Nodes* also collect performance metrics for the hosted virtual disks and send to a *vMocity Controller Node*, allowing cloud users to monitor their virtual disk status from a central point.

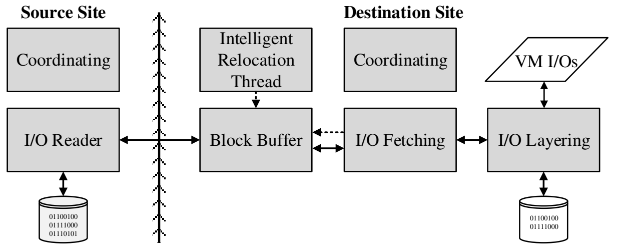


Figure 3: Components of vMocity I/O Handling Subsystem.

When a VM tries to access the data in its virtual disk image, if the data is already on the local *vMocity I/O node*, all I/O accesses are served locally. Otherwise, vMocity framework relocates the content of the virtual disk image from the source site, as shown in Figure 2(c). Intuitively, this can be achieved by setting up a copy-on-write shadow image on the destination site. However, all read I/Os to not-yet-written blocks would have to be naively forwarded to the source site, causing significant performance overhead. For a better relocation efficiency, *vMocity I/O Node* employs *I/O handling subsystem* with various design supports to handle virtual disk image relocation transparently.

2.2.1. I/O handling Subsystem. The I/O handling subsystem running in the *vMocity I/O Node* is responsible for handling all I/Os during the entire *relocation phase*. Figure 3 shows the main components involved in the inter-cloud relocation for both source and destination sites. When receiving a relocation request, the *Coordinating Module* first sets up an empty shadow virtual disk image on the destination site, and launches the *Intelligent Relocation Thread*. The *Intelligent Relocation Thread* enqueues requests of bulk fetching remote blocks to the *Block Buffer* in background. All blocks fetched from the remote site are stored temporarily in *Block Buffer* before being committed into the shadow virtual disk image. The *I/O Layering* module interposes on all accepted virtual disk I/Os and remaps these I/Os to the corresponding blocks on the shadow virtual disk image.

An I/O can be served immediately if the corresponding blocks have already been relocated from the source site. Otherwise, this I/O will be passed to the *I/O Fetching Module* for further handling. For a read I/O, the *I/O Fetching Module* first checks whether the *Block Buffer* contains blocks with the requested bytes. If not, the *I/O Fetching Module* immediately fetches the requested bytes from the source site so they can be accessed as soon as possible; In the meantime, it also enqueues requests to relocate the whole blocks that contain the requested bytes for this specific read I/O. Such not-yet-relocated blocks will be relocated asynchronously based on bandwidth availability. The *I/O Fetching Module* is also responsible for building and managing a byte-map for each individual block to be relocated, ensuring no duplicate byte is transmitted from source site. A block is reassembled in the *Block Buffer* once all its bytes are successfully transmitted to the destination site. For a write I/O, *I/O Fetching Module* first fetches all requesting blocks to the *Block Buffer*, and then applies changes to these blocks before committing to the shadow virtual disk image. As a result, in vMocity I/O handling subsystem, access to any byte in a not-yet-relocated block always triggers the relocation of the whole block.

Intelligent Relocation Thread Although fetching bytes in an on-demand manner from the remote site can immediately serve the requesting I/O, such mechanism still incurs significant delay, especially when bytes are transferred over the wide-area-network. To reduce such delay, we designed the *Intelligent Relocation Thread* to accurately relocate blocks in advance by leveraging the block access locality prediction information.

vMocity Intelligent Relocation Thread uses the VM's virtual disk accessing order from previous run on the source site to predict the first-time block-accessing order on the destination site. Based on our observation, instantiating and bootstrapping the VM on the destination reloads blocks following the similar order to previous run.

vMocity Intelligent Relocation Thread traces the virtual disk accesses, records the order of the first access to each block to an accessing order list, and saves this list as metadata into virtual disk image header. The maximum length of the accessing order list is equivalent to the total number of blocks of a virtual disk image. Once slots in the accessing order list are completely filled or the virtual disk image is disconnected from accessing, *vMocity Intelligent Relocation Thread* stops tracing the virtual disk accesses. *vMocity Intelligent Relocation Thread* also differentiates relocation I/O requests from normal I/O requests and does not record relocation I/O requests in the access ordering list. When setting up the shadow virtual disk image on the destination site, the ordered list is copied from the source virtual disk image along with other metadata.

Divergence of accessing virtual disk blocks from the VM's execution during previous run is unavoidable. It is possible for some blocks, which are not accessed during the previous run, to be accessed on the destination site. As the accessing order list does not have such accessing order information, we use block address space locality to predict accessing locality. This technique assumes that if block X and Y are adjacent in the block address space, then a block access to block X or Y is a good predictor that the VM will also access the other block.

vMocity I/O handling subsystem enforces priority guarantee between on-demand I/Os and background relocation I/Os. On-demand I/Os are associated with higher priority compared to background relocation I/Os. Without such enforcement, on-demand I/Os may be severely starved by intensive background relocation I/Os because the amount of data transferred for relocating the whole block is larger than that of serving individual bytes for an on-demand I/O.

Memory Caching When hosting and providing accesses to virtual disk images, the image store, running as a user space service, does not provide an extra layer of memory caching. Accepted I/Os are directly mapped to blocks and then processed upon corresponding object files by the lower object store layer. This greatly simplifies the system design while still guarantees the efficiency. On the other hand, the file system containing object files can still take advantage of Linux page cache. When an object is first read from or written to the storage backing devices, Linux kernel also stores the object data in unused areas of memory, which acts as a cache. If this object is read again later, it can be quickly read from this cache in memory. vMocity inherits both benefits and drawbacks from Linux page cache. It improves I/O performance if the blocks accessed can fit into *I/O Node's* available memory and aggregates I/Os to save operating cost as some cloud providers also charge per I/O usage. At the same time, it also suffers from data inconsistency due to reasons such as sudden hardware failures. Whether to enable memory caching for vMocity can be decided by users based on their specific requirements.

Other Optimizations To improve proficiency of VM relocation, vMocity framework also employs other common optimization mechanisms in the I/O handling subsystem, which include compression, skipping zero block relocation, and skipping zero writes.

2.3. vMocity Virtual Block Driver (V^2BD)

vMocity brings VM mobility to cloud users, as they can flexibly relocate VMs among any clouds. However, using vMocity also introduces additional monetary cost and performance degradation after relocation is done, which may dilute the gained advantage. In particular, if a user prefers to settle down on a particular cloud provider for a foreseeable period after relocation, directly using virtual disks offered by the cloud provider would be the best option. Therefore, we design a virtual block driver called V^2BD in vMocity framework which allows users to transparently choose storage services best fitting their needs.

Figure 4 depicts the internal of V^2BD and its placement in Linux storage stack. V^2BD populates a special virtual block device upwards, which can be used similar to a regular disk. V^2BD has two operating modes: vMocity mode and native mode. In vMocity mode, all I/Os are handled through iSCSI driver for accessing vMocity disk image. When operating in the native mode, V^2BD is mapped to the virtual hard disk exported by the underlying hypervisor.

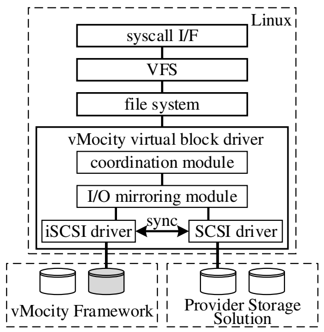


Figure 4: vMocity virtual block driver and its placement in Linux architecture.

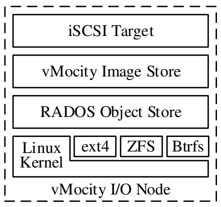


Figure 5: I/O Node internal architecture

To enable seamless backing device switchover between vMocity and native modes, the coordination module and the I/O mirroring module work collaboratively to synchronize blocks from one side to another. To initiate the backing device switchover, the coordination module synchronizes the two backing devices block by block. In the meantime, the I/O mirroring module interposes all write I/Os and mirrors those I/Os to the destination device synchronously. When the two backing devices have identical content, the coordination module redirects the I/O path to the target backing device.

Fairness during Switchover. Within V^2BD, there are two types of in-flight I/Os during the switchover stage: normal I/Os accepted from upper layer and synchronization I/Os generated by coordination module. V^2BD employs a queue to hold all in-flight I/Os. To provide fairness between I/Os, this queue is split into two portions, one for holding normal I/Os and the other portion for synchronization I/Os. Such queue is drained in a weighted round-robin fashion. Without this, two types of I/Os can severely impact one another when one is more intense than the other.

3. Implementation

The vMocity image store builds on the scalable RADOS architecture [10]. We added around 5200 lines of code to boost the VM relocation capability without introducing new bottlenecks. Our prototype implementation of V^2BD is built upon Linux RAID-0 and open-iscsi [12] projects, which involve about 600 lines of code change. The control protocol communicating between vMocity Controller Node and V^2BD is extended from iSNS [13]. We implement our control logic in iSNS project with 500 lines of code change. Although the prototype implementation of V^2BD and *vMocity Booting Image* are developed for Linux, they can be adopted for Windows as well, making vMocity an OS-agnostic solution.

3.1. vMocity I/O Node Stack

Figure 5 depicts the internal stack of *vMocity I/O Node*. *I/O Node* is built upon a Linux based VM, exposing a unified block interface to V^2BD for accessing virtual disk image. In our prototype, virtual disk images are populated as storage targets of iSCSI protocol, which is a block-level protocol that encapsulates SCSI commands into TCP/IP packets. By placing VMs and *I/O Nodes* within the same availability zone, iSCSI protocol can achieve performance close to local disk access [14]. Another reason for adopting iSCSI in our prototype system is its compatibility among a wide range of operating systems.

We modified RADOS Block Devices (RBD) [15] for building our image store. Since a *vMocity Virtual Disk Image* is composed of object files, which are retrievable from any endpoint within the object store, relocating an image can be accomplished by migrating all corresponding object files. Various underlying file systems can be used as backend file systems, leaving opportunities for exploiting more advanced functionalities such as file system level encryption, deduplication, and compression.

3.2. vMocity Booting Image

A VM is instantiated from its corresponding *vMocity Booting Image*, which is derived from the predefiend VM templates specific to each cloud provider. In our prototype system, we implement a Linux version of *Booting Image* (as shown in Figure 6), consisting of a Linux kernel, a customized init ramdisk and a bootloader to be compatible with both full virtualization and para-virtualization hypervisor. Although vMocity does not require any specific modification to the Linux kernel, the Linux kernel does have to be "virtualization friendly" by satisfying two prerequisites: 1) the kernel should contain necessary device drivers for popular hypervisors; 2) the kernel is patched for allowing operation in a para-virtualized guest VM. Fortunately, mainline Linux kernels since version 3.0 satisfy the above two prerequisites. The init ramdisk contains the V^2BD with metadata, which includes the unique ID to identify vMocity virtual disk image and the associated authentication key. During the booting process, the V^2BD generates contextual information by trying to query cloud providers' specific APIs for determining the current cloud provider and location. Based on the contextual information and metadata, the V^2BD can initiate the iSCSI connection with appropriate *I/O Node* thereafter.

4. Evaluation and Analysis

We evaluated vMocity using an I/O tester micro-benchmark and two application benchmarks. Fio [16] is a versatile I/O workload generator which has fine-grained control over I/O workload. Sysbench OLTP benchmark is a database benchmark for MySQL server. CloudStone is a Web 2.0 benchmark that includes a Web 2.0 social-events application (Apache Olio) and a client implemented using the Faban workload generator.

To evaluate the performance of vMocity framework and the benefits of various relocation optimizations, we first deployed vMocity in our local private cloud environment and conducted micro-benchmark and sysbench OLTP benchmark in this controlled environment. We also demonstrated the practical usability of vMocity framework by relocating an Olio server from our local private cloud to Google Compute Engine, from Google Compute Engine to Amazon EC2, and from Amazon EC2 back to our local private cloud.

Local Experimental Testbed. As shown in Figure 7, our local cluster consists of four Dell R210 servers running VMware ESX 5.5. Each server had a quad-core 3.2 GHz Intel Xeon E3-1230 processor, 16 GB of RAM, a dual port Broadcom 1GbE network adapter and two 7200 RPM hard disks. We placed these servers onto two separate racks, and deliberately configured the network to emulate wide-area-network environment. In the following set of experiments, the inter-rack network latency and bandwidth were set to around 40 ms and 200 Mb/s, while the intra-rack network latency and bandwidth were set to sub-millisecond and 1 Gb/s, respectively. The intuition for this network configuration is based on the iperf measurement between our local datacenter and Amazon datacenter in West Virginia.

In a VM's lifecycle, the time spending on relocation is much less than the time settled for normal operations. For convenience, we differentiate the VM lifecycle into relocation phase and non-relocation phase.

We evaluated the efficiency of vMocity relocation mechanism on our local testbed with three relocation strategies. The first strategy, serving as the baseline, relocates the whole VM image to the destination site before cold starting the VM. This strategy simulates the performance of relocating a VM without using vMocity framework. Vanilla, the second strategy, demonstrates the performance of using vMocity framework without *vMocity Relocating Subsystem*. Under this scenario, the relocation is handled by the copy-on-write mechanism of the RBD image store. The last optimized strategy makes use of *vMocity Relocation Subsystem* and employs various optimizations. This configuration shows the total relocation performance benefits of vMocity framework.

4.1. Micro-benchmark

VM workloads tend to be very complex, making it difficult to pinpoint the source of performance differences. We chose fio as the benchmark tool because it has control over every aspect of I/O workload.

Performance during Relocation Phase. To compare the performance of different relocation strategies, we ran the fio application in the client VM on the destination site to access a 10GB VM image, which was being relocated from the source site. In this experiment, the fio client VM and *vMocity I/O Nodes* were all configured with 2 vCPUs and 4 GB memory. The results of six types of I/O workloads were collected.

As shown in Figure 8, compared with vanilla relocation strategy, using optimized relocation strategy always achieves higher IOPS and lower latency during relocation for all I/O types. For sequential and random read I/O workloads, it increases IOPS by 180% and reduces latency by 60%, which are the largest improvements among all six types. Such significant improvements are mainly attributed to the

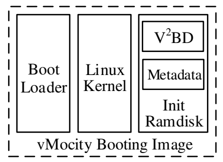


Figure 6: Booting Image Components

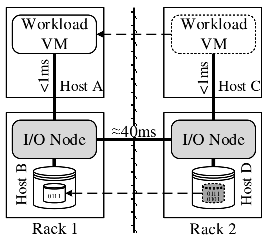


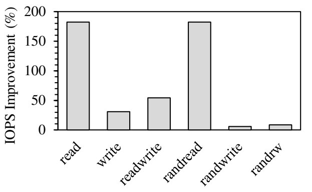
Figure 7: Experiment Deployment of vMocity framework

following facts: (1) accessing any byte in a not-yet-relocated image block triggers the relocation of the whole block, causing subsequent reading of the bytes in the same block to be served locally after the block is relocated from the source site; (2) relocated blocks are also cached in the *I/O Node*, which further improves the IOPS and reduces the latency as some portion of subsequent I/Os are served directly from cache without hitting hard disk; (3) the intelligent background relocation process can accurately predict and prefetch blocks that are going to be accessed, resulting in higher probability of serving I/O requests from local *I/O Node*.

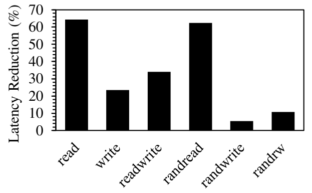
For sequential and random write I/O workloads, the I/O paths of the optimized relocation strategy are similar to the vanilla relocation strategy. However, we still observe improvements of IOPS and latency due to the compression and the intelligent background relocation mechanism.

Performance during Non-Relocation Phase. In current public cloud environments, VM images are directly backed by cloud provider's specific solution. In vMocity, we add one additional layer of storage abstraction. To evaluate the performance implication of hosting VM images on vMocity framework during the non-relocation phase, we conducted an experiment on our local cluster. In the vMocity setup, the *I/O Node* was connected with a 7200 RPM HDD for backing VM images, and populated the VM image to fio client VM via iSCSI protocol. In the non-vMocity setup, the same HDD was directly attached to the VM running fio client. In both setups, the HDD was connected to the corresponding VM via raw device mapping (RDM) [17], allowing a VM to directly access and use the physical storage device.

Figure 9 shows the normalized IOPS and latency of vMocity setup with respect to non-vMocity setup. For sequential types of I/O workloads, hosting VM images on vMocity framework incurs some performance degradation due to the overheads of object store and accessing via iSCSI protocol. Once the VM has "settled down" in the destination cloud, the user can switch to provider's native storage solution with V^2BD to avoid this temporary overhead. The performance will get back to the original level without using vMocity when the switching is done. The user can choose to turn off *vMocity I/O Nodes* to avoid additional cost afterwards. On the other hand, the encouraging result shows that using vMocity achieves improved IOPS and reduced latency for I/O workload involving random access. This is mainly due to the benefits from extra caching layer provided by



(a) IOPS Improvement



(b) Latency Reduction

Figure 8: IOPS improvement and latency reduction of optimized relocation strategy with respect to vanilla relocation strategy during relocation phase for fio benchmark (the higher the better).

the *I/O node* outweighs the overhead introduced by vMocity framework, as long as the blocks of a VM image can fit into the cache. Moreover, I/O workload involving write performs worse than read-only I/O workload in vMocity framework. This is because a single write I/O request issued by fio has to be accomplished by a series of read/write operations in an *I/O Node* due to the use of object store.

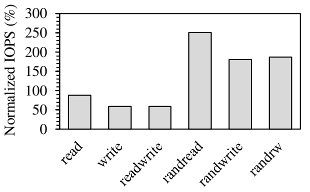
4.2. Application Benchmark

In addition to the I/O tester micro-benchmark, we also evaluated the performance of vMocity framework for two representative application benchmarks — Sysbench OLTP benchmark and CloudStone. Sysbench is a system performance benchmark that includes an OnLine Transaction Processing (OLTP) test profile. CloudStone [18] is a Web 2.0 benchmark to evaluate the suitability, functionality, and performance of Web technologies.

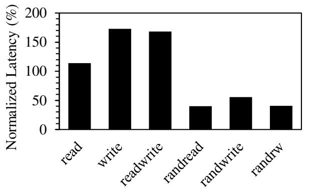
4.2.1. OLTP Workload. To understand the performance of vMocity framework, we conducted sysbench OLTP test both for relocation phase and non-relocation phase. Unlike some synthetic OLTP tests, the OLTP test included in the sysbench is a practical database-backed benchmark conducting transactional queries to an instance of a MySQL server. To simulate database workload in real-world scenarios, we ran the test in a complex R/W mode with various query types.

Performance during Relocation Phase. We created a MySQL server VM on our local cluster. The VM was configured with 2 vCPUs, 4 GB memory, and 20 GB virtual hard disk. Hard disks were attached to *vMocity I/O Nodes* via RDM for hosting the vMocity image store. The *vMocity I/O Nodes* were configured with 2 vCPUs and 4 GB memory. In this experiment, we defined 30 tables (each with 1 million rows), resulting in a 7.5 GB database of 30 million entries. We ran the sysbench OLTP workload generator in a separate physical host on the same rack as the server hosting the MySQL server VM. The evaluation was performed under three relocation strategies: baseline, vanilla, and optimized.

Figure 10 shows the trend of transaction rate during the first 1500 seconds under different relocation strategies with detailed service disruption time breakdown. All relocation technologies strive to achieve minimal service disruption time. In this experiment, the service disruption time is the duration of time before the newly relocated MySQL server VM responding to the first query on the destination site. As



(a) Normalized IOPS

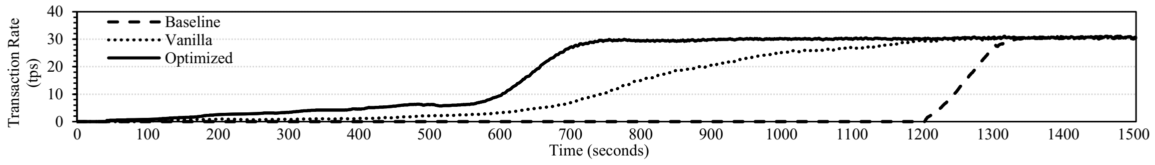


(b) Normalized Latency

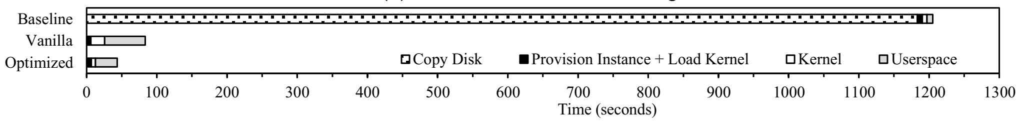
Figure 9: Normalized IOPS and latency of vMocity setup with respect to non-vMocity setup for fio benchmark (the higher the better).

shown in Figure 10a, with the baseline relocation strategy, the MySQL server requires 1206 seconds to respond to the first query. Using vMocity framework, the first query is responded in 84 seconds for the vanilla relocation strategy, and 44 seconds for the optimized relocation strategy. Optimized relocation strategy significantly reduces the service disruption time by a factor of 27 compared to the baseline relocation strategy, and a factor of 1.9 compared to the vanilla relocation strategy. On the other hand, both vanilla and optimized relocation strategies suffer non-trivial performance degradation at the beginning. This performance degradation is acceptable as blocks are transfered from remote site. Once the migration of blocks is finished, the performance will resume to the previous level. For optimized strategy, the duration of this performance degradation only lasts 700 seconds, while vanilla strategy takes 1200 seconds.

Figure 10b further shows the breakdowns of service disruption time in four phases. For the baseline relocation strategy, the whole VM virtual disk image has to be first copied from the source site, which is a time-consuming process. In comparison, the vanilla and optimized relocation strategies do not require such a step, thus the VM on the destination site can start instantly. The instance provisioning and kernel loading phase denotes creating, initializing and powering of a VM instance. This phase takes 7 seconds on average in our local testbed. After being powered on, the VM instance loads kernel and init ramdisk into memory, as well as performs kernel initialization to load various system modules and supporting libraries into memory. We call this phase the kernel initialization phase. In this phase, the V^{2}BD retrieves connection metadata from the controller node and connects to the designated *I/O Node* to mount the root file system. This phase takes 19.5 seconds for the vanilla relocation strategy, while using the optimized strategy reduces the time to 5.8 seconds. As system modules and libraries reside on non-contiguous blocks, the optimized strategy can relocate needed blocks more accurately, thus the initialization phase is less likely to be blocked by storage accessing. The final phase is the userspace initialization phase, in which the bootstrap process loads modules and libraries from the root file system, and starts various userspace services thereafter. For the baseline relocation strategy, since all image blocks have been relocated over and are hereby accessed locally, this userspace initialization phase only takes 8.4 seconds. The optimized and vanilla relocation strategies, however, require 31 seconds and 58 seconds to complete this phase, respectively.



(a) Transaction rate trend during relocation.



(b) Service disruption time breakdowns.

Figure 10: Performance results of relocating MySQL server in local testbed.

Another important metric in VM relocation is the time required to resume stable performance. As shown in Figure 10a, by applying various optimization techniques, MySQL server VM is capable of serving queries stably in 760 seconds. The optimized relocation strategy reduces the time duration before resuming normal transaction rate by a factor of 1.8 compared with the baseline relocation strategy, and a factor of 1.7 compared with the vanilla relocation strategy. One of the main reasons for such a reduction is that the intelligent background relocation thread can locate blocks that will be accessed by the workload and relocate those blocks with high priority from the remote site.

Performance during Non-Relocation Phase. We also evaluated the performance of hosting VM images on vMocity framework with OLTP workload during non-relocation phase. We used the setups similar to the ones in Section 4.1. The MySQL server VM was configured with 2 vCPUs and 2 GB memory. For the non-vMocity setup, the virtual disk of the VM was backed by a physical HDD via RDM; for the vMocity setup, the virtual disk was populated to the VM by an *I/O Node*, which was also configured with 2 vCPUs. We configured the *I/O Node* with varying sizes of memory to limit the availability of Linux page cache. In this experiment, we defined 120 tables (each with 1 million rows), resulting in a 30 GB database of 120 million entries.

For OLTP workloads during non-relocation phase, the transaction rates for hosting VM's virtual disk on vMocity framework incur about 7% to 12% performance degradation, with *I/O Node* memory configured to various sizes (4 GB, 2GB and 1GB). This degradation is as expected due to the extra complexity (e.g., object store and iSCSI protocol overhead) introduced by the additional layer of storage abstraction. With the flexibility of V^{2}BD, such degradation can be mitigated by switching to the provider's native storage solution. We also notice that increasing *I/O Node* memory size from 1 GB to 4 GB, which in turn increases the available Linux page cache size, only improves the transaction rate by 5%. The reason for such a small improvement is due to the I/O randomness of the benchmark workload.

Performance during Switchover Phase. V^{2}BD enables users to transparently choose storage services best fitting their needs. During the switchover phase, it limits the background synchronization rate between two sources of backing block devices and strives to minimize the negative performance impact on the running workload. To demonstrate the effectiveness of our design, we conducted an experiment using a running MySQL server VM on our local test bed to switch the backing storage device from the virtual disk populated by *vMocity I/O Node* to HDD connected via RDM. The VM was configured with 2 vCPUs and 2 GB memory, and the size of the block device is 30 GB. We limited V^{2}BD to only use 1/3 of overall bandwidth for background synchronization during switchover phase.

Figure 11 presents the OLTP workload transaction rate in a duration of 6000 seconds. The graph shows the impact on the MySQL server throughput during switchover phase. In the first 1000 seconds, the MySQL server VM used virtual disk populated from *vMocity I/O Node* as backing storage device, and can achieve the throughput of 53 transactions per second. When starting switching backing storage device, the transaction rate dropped to 40 transactions per second and the switchover phase lasted for 3250 seconds, during which the two sources of backing storage devices were being synchronized at a rate around 10 MB/s. Once the switchover was done, the throughput increased to 60 tps since the native HDD does not introduce performance overhead as vMocity does.

4.2.2. Apache Olio. Web serving is one of the fundamental applications for IaaS cloud. CloudStone includes Olio, which implements a social-event calendar web application that provides representative functionality of Web 2.0 applications prevalent in clouds — user-generated metadata, social networking functions, and rich AJAX-based GUI.

To demonstrate the application of vMocity framework in the real environment, we conducted an experiment to relocate an Olio server VM among three different clouds. More specifically, the steps for relocating the VM are as follows: (1) we first relocate the VM running in our private cloud (located at our university) to Google Compute Engine; (2) from Google Compute Engine, the VM is relocated

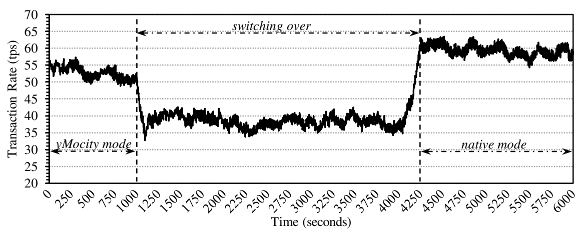


Figure 11: Transaction Rate during switchover phase for OLTP workloads

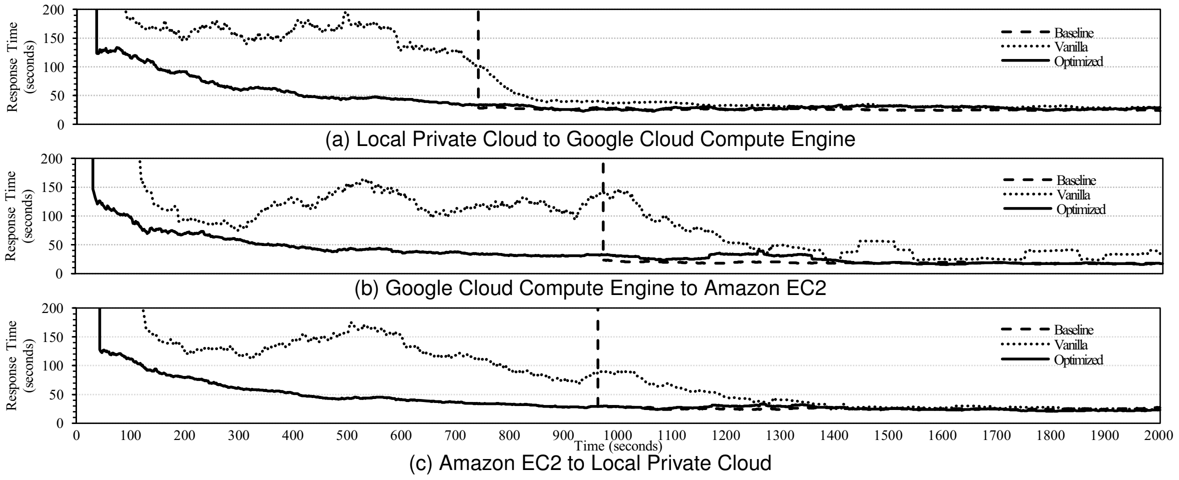


Figure 12: Response time of the Olio server when relocating between different clouds.

to Amazon EC2; (3) finally we relocate the VM from Amazon EC2 back to our private cloud. The evaluation was still performed under three relocation strategies: baseline, vanilla, and optimized.

In this experiment, we adopted the PHP version of CloudStone and built an Olio server VM containing three components: (1) a web server to process user requests; (2) a MySQL database instance to store user information; (3) and an NFS server to store images and documents. On each cloud, the Olio server VM was configured with 2 vCPUs, 7.5 GB memory, and 20 GB virtual disk populated by vMocity framework. In our private cloud, the user requests were sent from the workload generator running in a separate physical host. In Google Compute Engine and Amazon EC2, we ran the workload generator in a separate VM residing in the same availability zone as Olio server. We configured the workload generator to simulate 800 concurrent users. Each user sent one request every five seconds. The peak CPU utilization of the Olio server is about 70%, which is very close to the CPU load in real-world cloud environments.

Since there are various types of requests sent to the Olio server, we use the weighted mean to represent the mean response time of the seven presented requests. Figure 12 shows the mean response time during the first 3000 seconds when relocating the Olio server VM between different clouds along the itinerary. Although the source site cloud and destination site cloud vary in each section of the itinerary, relocating the Olio server VM with the same strategy exhibits similar trends. Using the baseline relocation strategy to relocate the Olio server VM always suffers long service disruption time. Once the relocation finishes, however, the Olio server VM achieves stably low response time to subsequent requests. With the vanilla relocation strategy, although the Olio server is capable of starting serving requests in around 100 seconds, the response time is high and unstable during a very long period of time. The main reason for such an unstable response time is that the vanilla relocation strategy forwards all read I/Os over the wide-area-network connecting the two clouds. The optimized relocation strategy always outperforms the baseline and vanilla relocation strategies. On one hand, the optimized relocation strategy can achieve the shortest service disruption time; on the other hand, it also allows the response time to drop more rapidly and smoothly. As VM image blocks can be relocated more efficiently with the optimized relocation strategy, more I/Os can be served locally, significantly reducing the response time.

5. Related Work

Live Migration. Clark et al. [19] and Nelson et al. [20] proposed live VM migration in Xen and VMware ESX. With recent advance in storage migration [21], Live VM migration has been extended over WAN [22]. Although live migration enables minimized downtime, all existing live migration approaches require the same hypervisor on the source and destination ends, which does not meet our goal of enabling VM relocation across diversified IaaS.

Nested Virtualization. Nested virtualization employs virtualization on already virtualized resources. This is particularly useful when users cannot directly control the first-layer hypervisor, while a second layer of virtualization offers control, isolation, and homogeneity [23] [24]. However, nested virtualization [25] [8] [7] is a heavy-weight approach and does impose some overhead depending on the operation.

Lightweight Virtualization. Lightweight virtualization technologies [9] have been available for many years. They are getting a fair amount of attention these days [26] [27]. Container provides an abstraction layer between the application and the underlying cloud infrastructure. Once an application is built within a container, users are able to move it between cloud providers. However lightweight virtualization heavily relies on the underlying operating system, which limits its capability to avoid vendor lock-in and the flexibility of choosing operation systems. In comparison, vMocity framework is an OS-agnostic solution. The block level abstraction exposed by *vMocity I/O Node* can be adopted by both Linux and Windows VMs.

Cloud Storage Middleware. Raghavan et al. [28] proposed Tiera, a storage framework that enables the provision of flexible and easy-to-use multi-tiered cloud storage instances for better performance and manageability. Some commercial products, such as Zadara Storage [29] and SoftNAS [30], use special storage instances for delivering enhanced storage services to cloud users.

6. Conclusion

In this paper, we present vMocity, a novel framework to enable efficient VM relocation across different cloud providers. By decoupling the dependency between storage virtualization stacks and their underlying hypervisors, vMocity framework exposes a unified block interface for accessing VM virtual disk images, and hosts virtual disk images on user-maintained VMs to advocate a user-centric view of storage homogenization. Our experiments show that vMocity can significantly reduce the service disruption time during VM relocation and resume stable performance much faster than existing approaches.

7. Acknowledgement

We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF under Award 1219004.

References

[1] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, March 2014.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, April 2010.

[3] D. Petcu, C. Craciun, and M. Rak, "Towards a cross platform cloud api," in *1st International Conference on Cloud Computing and Services Science*, 2011.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. ACM, 2003, pp. 164–177.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[6] Y. Lu and D. Du, "Performance study of iscsi-based storage subsystems," *Communications Magazine*, vol. 41, no. 8, pp. 76–82, August 2003.

[7] A. Fishman, M. Rapoport, E. Budilovsky, I. Eidus *et al.*, "Hvx: Virtualizing the cloud," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, June 2013.

[8] D. Williams, H. Jamjoom, and H. Weatherspoon, "The xen-blanket: virtualize once, run everywhere," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, April 2012, pp. 113–126.

[9] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, March 2007, pp. 275–287.

[10] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*. ACM, 2007, pp. 35–44.

[11] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 307–320.

[12] "Open-iscsi," http://www.open-iscsi.org/.

[13] "Rfc 4171: Internet storage name service (isns)," http://tools.ietf.org/html/rfc4171.

[14] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. J. Shenoy, "A performance comparison of nfs and iscsi for ip-networked storage." in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, 2004, pp. 102–114.

[15] "Rados block devices," http://ceph.com/docs/master/rbd/rbd/.

[16] "fio - flexible io tester," http://freecode.com/projects/fio/.

[17] "Vmware raw disk mapping," https://pubs.vmware.com/vsphere-55/index.jsp#com.vmware.vsphere.storage.doc/GUID-9E206B41-4B2D-48F0-85A3-B8715D78E846.html.

[18] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proceedings of cloud computing and its applications*, 2008.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[20] M. Nelson, B.-H. Lim, G. Hutchins *et al.*, "Fast transparent migration for virtual machines," in *USENIX Annual Technical Conference, General Track*. USENIX Association, 2005, pp. 391–394.

[21] K. Haselhorst, M. Schmidt, R. Schwarzkopf, N. Fallenbeck, and B. Freisleben, "Efficient storage synchronization for live migration in cloud infrastructures," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE, 2011, pp. 511–518.

[22] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty, "Xvmotion: unified virtual machine migration over long distance," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 97–108.

[23] D. Williams, E. Elnikety, M. Eldehiry, H. Jamjoom, H. Huang, and H. Weatherspoon, "Unshackle the cloud," in *Proceedings of the 3th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association, June 2011.

[24] Z. Pan, Q. He, W. Jiang, Y. Chen, and Y. Dong, "Nestcloud: Towards practical nested virtualization," in *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 2011, pp. 321–329.

[25] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 423–436.

[26] "docker," https://www.docker.com/.

[27] "Flocker," https://clusterhq.com/.

[28] A. Raghavan, A. Chandra, and J. B. Weissman, "Tiera: Towards flexible multi-tiered cloud storage instances," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 1–12.

[29] "Zadara storage," https://www.zadarastorage.com/.

[30] "Softnas," https://www.zadarastorage.com/.