Supercloud: A Library Cloud for Exploiting Cloud Diversity

ZHIMING SHEN, QIN JIA, GUR-EYAL SELA, WEIJIA SONG, HAKIM WEATHERSPOON, and ROBBERT VAN RENESSE, Cornell University

Infrastructure-as-a-Service (IaaS) cloud providers hide available interfaces for virtual machine (VM) placement and migration, CPU capping, memory ballooning, page sharing, and I/O throttling, limiting the ways in which applications can optimally configure resources or respond to dynamically shifting workloads. Given these interfaces, applications could migrate VMs in response to diurnal workloads or changing prices, adjust resources in response to load changes, and so on. This article proposes a new abstraction that we call a *Library Cloud* and that allows users to customize the diverse available cloud resources to best serve their applications.

We built a prototype of a Library Cloud that we call the *Supercloud*. The Supercloud encapsulates applications in a virtual cloud under users' full control and can incorporate one or more availability zones within a cloud provider or across different providers. The Supercloud provides virtual machine, storage, and networking complete with a full set of management operations, allowing applications to optimize performance. In this article, we demonstrate various innovations enabled by the Library Cloud.

CCS Concepts: • **Computer systems organization → Maintainability and maintenance**; *Reliability*; *Availability*; Fault-tolerant network topologies;

Additional Key Words and Phrases: Cloud computing, nested virtualization, Cloud Federation

ACM Reference format:
Zhiming Shen, Qin Jia, Gur-Eyal Sela, Weijia Song, Hakim Weatherspoon, and Robbert van Renesse. 2017. Supercloud: A Library Cloud for Exploiting Cloud Diversity. *ACM Trans. Comput. Syst.* 35, 2, Article 6 (October 2017), 33 pages.
https://doi.org/10.1145/3132038

1 INTRODUCTION

The Infrastructure-as-a-Service (IaaS) cloud paradigm is popular, both as a public service (Amazon EC2, Google Compute Engine, RackSpace, Windows Azure, etc.), as well as for use in private clusters (VMware vSphere, Eucalyptus, IBM Cloud, OpenStack, etc.). But current cloud providers violate a basic systems design principle as formulated by Butler Lampson: *Don't Hide Power*

Q. Jia and G.-E. Sela were at Cornell University while conducting this research. Q. Jia is currently at Google. G.-E. Sela is currently at University of California, Berkeley.

This work was partially funded and supported by a SLOAN Research Fellowship received by H. Weatherspoon, DARPA MRC (FA8750-10-2-0238, FA8750-11-2-0256), DARPA CSSG (D11AP00266), NSF CSR-1422544, NSF CNS-1601879, NSF CAREER (1053757), NSF TRUST (0424422), NIST Information Technology Laboratory (60NANB15D327,70NANB17H181), Cisco, Intel, Facebook, and Infosys.

Authors' addresses: Z. Shen, W. Song, H. Weatherspoon, and R. van Renesse, Department of Computer Science, Gates Hall, Cornell University, Ithaca, NY, 14853; emails: {zshen, weijia, hweather, rvr}@cs.cornell.edu; Q. Jia, Google Inc. 76 Ninth Avenue, New York NY 10011; email: qinjia@google.com; G.-E. Sela, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94720; email: ges@berkeley.edu.

(Lampson 1983), or, in other words, do not hide desirable interfaces. Cloud providers hide many powerful management interfaces, including virtual machine (VM) placement and migration, CPU capping, memory ballooning, page sharing, I/O throttling, and network routing. Without such interfaces, applications are severely limited in the ways they can optimally configure resources or respond to dynamically shifting workloads.

Given additional management interfaces, applications could easily migrate VMs across availability zones in response to diurnal workloads or changing prices, adjust the resources given to particular VMs in response to load changes, deploy applications across multiple cloud providers for increased fault tolerance, and so on. Control over storage and network routing would further improve customizability for optimal performance.

In this article, we introduce the *Library Cloud* abstraction, inspired by the concept of Library Operating Systems (OSes) (Engler et al. 1995; Madhavapeddy et al. 2013; Porter et al. 2011). In the Library OS architecture, services such as the file system that are traditionally implemented in the OS kernel are instead implemented in a user space library and linked directly with applications, simplifying customization. The OS kernel is needed only to provide basic low-level resource management and isolation, such as disk blocks, memory pages, and so on. Similarly, in a Library Cloud, cloud providers are only needed to provide basic resources for computation, storage, and networking. A Library Cloud implements an entire cloud stack on top of these resources, as it were, in user space. Moreover, these resources can be allocated at multiple cloud providers.

Like a private IaaS cloud, a Library Cloud supports traditional cloud services for allocating VMs. But a Library Cloud also provides all of the administrative APIs that are not available to users of public cloud providers. Because a Library Cloud supports management operations such as user-level migration and memory consolidation, it enables application solutions not easily achieved with disjoint cloud providers or even federated clouds based on standard interfaces. Two examples of this are management of geographically shifting workloads and transparent consolidation of VM resources. We describe and evaluate these examples in this article.

We built a prototype Library Cloud that we call the *Supercloud*. Building the Supercloud, we had to overcome various challenges related to computation, storage, networking, and management. Although some cloud providers, such as Amazon, allow customers to rent dedicated physical servers, most cloud providers do not expose physical resources directly. Supercloud leverages nested virtualization technology (Williams et al. 2012), eliminating the need for VM management support from underlying providers. Xen-Blanket unifies machine virtualization on different platforms and provides the same interfaces that are supported in the Xen hypervisor.

We designed and implemented a new distributed storage system optimized for wide-area cross-provider VM migration. It decouples providing strong consistency from update propagation, minimizing overhead and optimizing performance. VMs run in a high-performance Software-Defined Network (SDN) built with Open vSwitch and VXLAN tunnels crossing cloud boundaries. We also designed and evaluated various solutions to deal with migrating services that use public IP addresses. The Supercloud runs the OpenStack platform and appears to users as a single, private OpenStack cloud.

This article makes the following contributions:

- We propose a Library Cloud abstraction for user-level resource management.
- We show how nested virtualization can be leveraged to handle heterogeneity and implement a Library Cloud without the support of underlying cloud providers.
- We propose storage and networking solutions for supporting efficient VM migration in a Library Cloud and show how they can be implemented efficiently.
- We evaluate a prototype of the Library Cloud abstraction—the Supercloud.

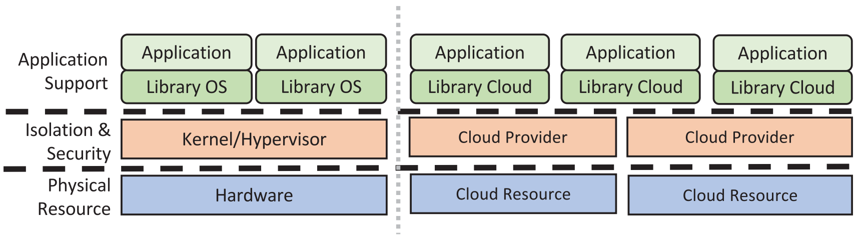


Fig. 1. Library OS vs. Library Cloud.

- We demonstrate how applications can benefit from the Library Cloud using case studies of geographically shifting workloads and VM consolidation.

2 TOWARDS THE LIBRARY CLOUD

2.1 The Library Cloud Abstraction

A *Library OS* (Engler et al. 1995; Madhavapeddy et al. 2013; Porter et al. 2011) is an application of the end-to-end principle (Saltzer et al. 1984) to OSes. A traditional monolithic OS kernel packs many functions into kernel space and presents complex high-level abstractions to user programs, such as a hierarchical multi-user file system and network sockets. Although such high-level abstractions simplify development of applications, they also limit customization and optimization, as they force applications to build on top of modules designed with specific requirements in mind. The Library OS concept, in conjunction with an *Exokernel*, addresses this problem. The Exokernel is a tiny kernel that implements only resource multiplexing and security isolation; other traditional OS functionalities are implemented in Library OSes specialized for different applications. This approach gives applications more flexibility to choose a proper abstraction for optimal performance and security.

The Library Cloud abstraction introduced in this article is conceptually similar to a Library OS: users and applications get control over a full cloud software stack and can customize it for their own purposes; the underlying cloud providers only need to provide basic resource multiplexing and isolation. The Library Cloud abstraction further extends the notion of Library OS: a single Library Cloud can span across multiple cloud providers. Figure 1 compares the Library OS and Library Cloud designs.

Applications running in a Library Cloud interact using a *Library Cloud API* that can be much richer than a typical cloud API. A Library Cloud API not only supports typical Cloud API methods, such as creating VMs and managing networks, but also enables methods that are usually only available to cloud providers, such as live VM migration, consolidation, checkpointing, and dynamic scaling. Section 3 presents an example of the Library Cloud API. Figure 2 illustrates the difference between traditional clouds and Library Clouds.

Today, a Library Cloud can be implemented on top of any existing cloud API. We are expecting that Cloud APIs will emerge with lower abstractions so that unnecessary abstraction layers can be removed. For example, HIL (Hennessey et al. 2016) is an Exokernel-like layer for data centers that exposes physical resources directly to different cloud users.

2.2 Innovations Enabled by the Library Cloud

A Library Cloud abstraction supports innovations that are hard to realize in current cloud infrastructures. In this section, we will describe use cases that benefit from a Library Cloud like the Supercloud.

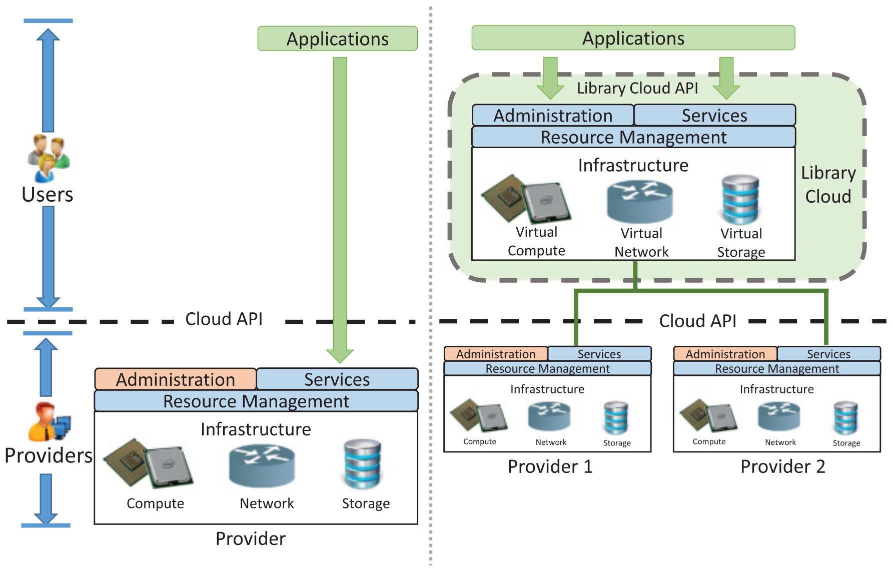


Fig. 2. Traditional Cloud (left) vs. Library Cloud.

2.2.1 Follow the Sun. For a service that has global users in different time zones, it would be ideal if it could "follow the sun"—that is, continuously shifting to a location where the majority of users experience the lowest possible latency. This is a challenging task even for distributed applications that can migrate by adding and removing nodes, not to mention legacy applications that cannot be easily scaled dynamically.

Distributed applications typically adopt a distributed consensus or transaction protocol to support data replication, distributed locking, distributed transactions, and so on. These protocols are designed to be fault tolerant. However, adding and removing nodes while tolerating failure is a fundamental and challenging problem—it requires changing the "membership" of the system, and thus subsequent requests following the membership change must be processed with a different configuration. In addition to the complexity of reaching agreement on configuration, adding and removing nodes triggers complicated internal state transfer and synchronization protocols, and membership reconfiguration is not transparent to clients. As a result, tolerating failure while providing good performance during reconfiguration is non-trivial.

Because membership reconfiguration is generally considered a rare event, applications use ad hoc mechanisms to achieve it with unpredictable performance. For example, MongoDB does not allow changing the sharding key on the fly, and the whole database must be exported and then imported again to change key distribution. As another example, re-configuring a cluster running an old version of ZooKeeper (without the new dynamic reconfiguration feature) requires a "rolling restart"—a procedure whereby servers are shut down and restarted in a particular order so that any quorum of currently running servers includes at least one server with the latest state (Shraer et al. 2012). If not performed correctly, one server with outdated state might be elected as the new leader and the whole cluster might enter an inconsistent state. Although ZooKeeper recently added support for dynamic reconfiguration, it is inefficient for geographic migration (see Section 4.2.2). A key-value store such as Cassandra can easily add and remove a node and adjust token distribution, but doing these for geographic server migration triggers unnecessary data replication and load rebalancing, and can affect service availability if a failure occurs at the same time (see Section 4.2.1).

To follow the sun, applications must be migrated several times a day, and, as we have discussed, implementing migration by adding and removing nodes is sub-optimal. An alternative approach is to use live VM migration since it can be performed transparently to the application and even to clients. For example, if live VM migration were supported, we can migrate the ZooKeeper leader and servers to locations where the leader and a majority of servers are geographically close to most active users, without the need for changing a single line of source code in ZooKeeper. Often this only involves migrating the leader: if the majority of clients reside in two different regions, such as the United States (US) and Asia, and there are $2f + 1$ servers, including one leader, then with f servers running in one location, the US, and f servers running in another, Asia, only the leader needs to migrate back and forth once a day. Live migrating a VM comprises multiple phases (Clark et al. 2005). First, the image is copied while the VM is still running at the old location. Pages that are written by the running VM after they have been copied have to be copied again, and thus this phase might require multiple rounds of copying. When the number of dirty pages is below some minimum, the VM is suspended for a short period of time to finish copying the remaining state. After this, the VM at the new location resumes execution. The application downtime corresponds to this relatively short final copying phase plus the time required for the network layer to adjust routing paths for the migrated VM. The downtime due to live migration of a VM between the US and Asia is less than 1 second, whereas the total migration time for 1GB memory (which proceeds in the background) is about 100 seconds. The downtime is small enough such that TCP connections do not break nor cause broken sessions or leader re-election.

2.2.2 Dynamic Resource Scheduling. VMware vSphere Distributed Resource Scheduling (DRS) redistributes, invisible to applications, VMs among a pool of hardware servers to optimize utilization, and it is an important technique to meet business goals. Public clouds provide services such as Amazon Auto Scaling (Amazon 2016) to dynamically add and remove nodes for distributed applications. But these services require users to reconfigure their application on the fly to deal with a changing number of VMs. The Supercloud supports a new paradigm for resource scaling that we call *Supercloud Dynamic Resource Scheduling* (SDRS). SDRS opens up new opportunities to minimize cost without compromising application performance or changing application configurations.

SDRS leverages hypervisor-level mechanisms enabled by the Supercloud, such as VM migration, CPU capping (Shen et al. 2011), memory ballooning (Waldspurger 2002), page sharing (Gupta et al. 2008), and I/O throttling (Kim et al. 2009) to share resources efficiently while ensuring that different VMs do not interfere with each other. When load increases, nested VMs are migrated to separate provider VMs with more available resources to maintain application performance. This is analogous to a cloud provider consolidating VMs on a smaller number of physical machines and migrating VMs to an increased number of physical machines when load increases. When application load is low, nested VMs can be consolidated on a single-provider VM, greatly reducing cost. Section 4.3 evaluates this using the TPC-W benchmark.

2.2.3 Smart Spot Instances. Amazon EC2 provides a spot market, offering cheap VMs in underused availability zones. Microsoft Azure has a similar offering. Such instances are difficult to use effectively in practice: prices of spot market instances change rapidly and can even significantly exceed on-demand VM instances. More problematically, Amazon EC2 reserves the right to terminate instances when an availability zone is no longer under-utilized. We have designed and built *Smart Spot Instances* (Jia et al. 2015), achieving both low cost and high availability by migrating VMs rapidly to the cheapest and most available locations. The technique takes into account the network charge for transferring VM images across availability zones. In our experience, we are able to provide a long-running smart VM instances at approximately a third of the price of a normal instance. A similar idea was developed in SpotCheck (Sharma et al. 2015). The Supercloud

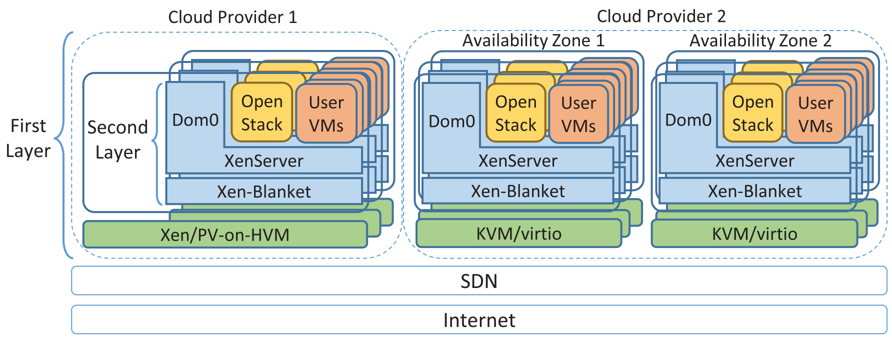


Fig. 3. Example deployment of the Supercloud.

makes it easy to develop such innovations and, going beyond SpotCheck, generalizes to multiple cloud providers.

3 THE SUPERCLOUD: A LIBRARY CLOUD IMPLEMENTATION

The Supercloud is an instance of a Library Cloud that is designed to operate without any specific support from underlying cloud providers. It embodies control of compute, networking, storage, and management that can be customized by users (see Figure 2). Importantly, a Supercloud can span multiple availability zones of the same provider, as well as availability zones of multiple cloud providers and private clusters (Figure 3). To accomplish this, there are two layers of hardware virtualization. The bottom layer, called the *first layer*, is the infrastructure managed by an IaaS cloud provider such as Amazon EC2 or Rackspace, or managed privately. It provides VMs, cloud storage, and networking. Another layer of virtualization on top of this, called the *second layer*, is the IaaS infrastructure managed by the Supercloud. It leverages resources from the first layer and provides a single uniform virtual cloud interface. Importantly, the second layer is completely controlled by users.

In case of compute resources, the first layer has a hypervisor managed by the underlying cloud provider and a collection of hardware virtual machines (HVMs). We refer to these as *first-layer hypervisors* and *first-layer VMs*.

The second layer, exposed to Supercloud users, is similarly separated into a hypervisor and some number of guest VMs that we call the *second-layer hypervisor* and *VMs*.[1] We use Xen-Blanket (Williams et al. 2012) for the second-layer hypervisor. Xen-Blanket provides a consistent Xen-based para-virtualized (PV) interface. In Xen, one VM is called *Domain-0* (aka *Dom0*) and manages the other VMs, called *Domain-Us* (aka *DomUs*). A second-layer Dom0 multiplexes resources such as I/O among the DomUs.

We use OpenStack (OpenStack 2016) to manage user VMs and provide the administrative OpenStack API as the Library Cloud API to existing applications. In particular, a XenServer runs within the second-layer Dom0 and allows OpenStack to manage all second-layer DomU VMs.

A common practice in VM migration is using shared storage to serve VM images so that a VM migration only needs to transfer the memory. This is essential to good performance because migrating a VM with the disk image takes a long time. XenServer offers two options for sharing storage: an NFS-based solution and an iSCSI-based solution. Both of these approaches use a centralized storage repository. Although simple, this can lead to significant latencies, low bandwidth,

[1]User VMs, user VM instances, second-layer VMs, and second-layer DomU VMs are used interchangeably.

and high Internet cost for VMs that access the disk through a wide area network (WAN) when migrated to another region or cloud. Previous works have proposed different mechanisms and optimization for wide-area VM migration with disk images (Bradford et al. 2007; Mashtizadeh et al. 2011; Nicolae and Cappello 2012). However, migrating the whole image file incurs significant Internet traffic, which is typically charged by cloud providers. To support efficient VM migration in the WAN, we developed a geo-replicated image file storage that seeks a good balance between performance and cost.

To provide the illusion of a single virtual cloud, the control services (including XenServer and OpenStack) and user VMs need to communicate in a consistent manner no matter where the endpoint VMs reside. A migrated user VM expects its IP address to remain unchanged. To accomplish this, the Supercloud network layer is built using an SDN overlay based on Open vSwitch, VXLAN tunnels, and the Frenetic SDN controller (Foster et al. 2010). Such an overlay network gives control over routing for the second layer and enables compatibility with heterogeneous first-layer networks.

We support all major hypervisors including Xen, KVM, Hyper-V, and VMware, so a Supercloud instance can span all major cloud providers, including Amazon EC2, Rackspace, Windows Azure, Google Compute Engine, and VMware vCloud Air.

The Supercloud, like any other IaaS cloud, provides three types of resources: computing, networking, and storage. In the following, we present each one in more detail.

3.1 Computing

3.1.1 Nested Virtualization. Nested virtualization is essential in the Supercloud to provide a uniform interface and support privileged operations such as live VM migration. However, nested virtualization is not natively supported by most cloud providers. Solutions that require special support in the first-layer hypervisor, such as the Turtles project (Ben-Yehuda et al. 2010), cannot be run in public clouds. We use Xen-Blanket (Williams et al. 2012), a nested virtualized Xen hypervisor that runs on various widely supported HVMs and provides a PV interface to second-layer VMs.

In a full virtualized environment, the underlying hypervisor is completely transparent to the system running in the VM. When virtual devices are emulated, a Xen hypervisor can run without modification in an HVM. However, emulated devices have poor performance. Thus, modern clouds typically adopt PV devices for HVM instances (e.g., `PV-on-HVM` devices on Xen, `virtio` devices on KVM, and `enlightened I/O` devices on Hyper-V). In this model, a device driver in the VM works together with the underlying hypervisor to improve I/O performance. This solution breaks transparency and brings new challenges when we want to run Xen as a second-layer hypervisor:

- A standard PV device driver cannot work properly because a second-layer Dom0 is no longer running in the `Ring-0` privileged domain;
- The notion of a "physical address" in a Xen-based VM no longer matches the "machine address" in the HVM.

Xen-Blanket opens new interfaces to enable communication between a PV device driver in the second-layer Dom0 and the first-layer hypervisor, and leverages different *blanket drivers* to deal with diverse hypervisors. It then multiplexes these PV devices and provides a consistent Xen device interface. Hence, the underlying infrastructure becomes transparent to the second-layer user VMs. The bulk of the effort in enabling Xen-Blanket in a cloud infrastructure is porting the blanket driver. Once enabled, second-layer user VMs can run on Xen-Blanket without any modification.

The performance overhead of Xen-Blanket has been thoroughly evaluated (Williams et al. 2012). To summarize, Xen-Blanket is able to match native network I/O performance and incur about 12% overhead for disk I/O performance compared to a native PV instance. We also tested a sysbench

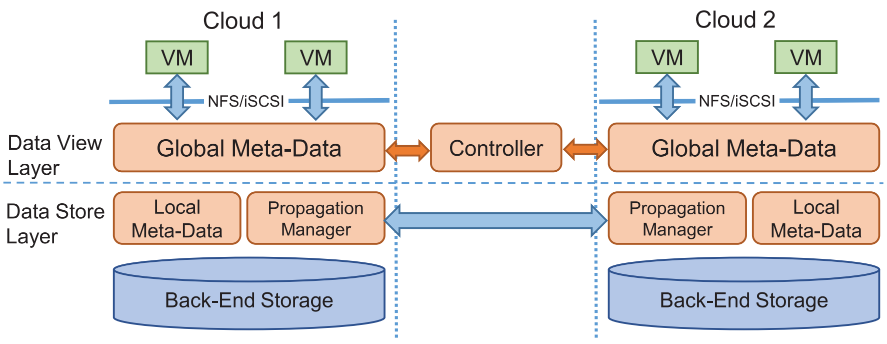


Fig. 4. Storage architecture of the Supercloud.

CPU benchmark, which shows that the completion time difference between first- and second-layer VMs is within 10%.

3.1.2 Handling Heterogeneity. Different underlying CPUs might have different capabilities, preventing a VM from working properly after being migrated. We can record the CPU features exposed to a second-layer VM when it is created and only migrate it to a first-layer VM that support compatible CPU features. However, this places a limit on the placement of the VM.

For applications that require more flexibility on VM placement, we can expose a subset of CPU features that are common in modern CPUs, which is sufficient for supporting most applications. This requires a solution to control which CPU feature we want to expose. Hardware solutions such as Intel VT FlexMigration and AMD-V Extended Migration trigger a fault when guest VMs execute a `CPUID` instruction, thus providing the virtual machine monitor (VMM) an opportunity to intercept this instruction and report a consistent set of CPU features to the VMs. Unfortunately, `CPUID` faulting is not virtualized. As a result, a second-layer hypervisor running in an HVM loses the ability to intercept the `CPUID` instruction.

Fortunately, because all second-layer VMs running on top of Xen-Blanket are PV, in most cases they will invoke the `pv_cpuid` hook in the hypervisor instead of executing the `CPUID` instruction directly. Thus, we modified the `pv_cpuid` hook in Xen-Blanket to only expose a desired subset of CPU flags. Applications running in second-layer VMs will see a set of consistent CPU flags when checking the Linux */proc* interface. Note that this solution does not prevent user VMs from running the `CPUID` instruction directly. Processor vendors may eventually support virtualized `CPUID` faulting so that a second-layer hypervisor could intercept the instruction and control CPU flags exposed to second-layer VMs.

3.2 Storage

3.2.1 Consistency and Data Propagation. Geo-replicated storage solutions sometimes adopt a weaker consistency model, such as *eventual consistency*, in which applications reading different replicas may see stale results. This is not suitable for an image storage on which a migrated VM expects up-to-date data. Using a strongly consistent geo-replicated storage requires synchronous data propagation, resulting in low write throughput. A key observation is that a running application typically does not require all data in the image. Our storage system thus decouples consistency from data propagation.

As shown in Figure 4, the storage service consists of two layers: a data view layer provides a required consistency guarantee, and a data store layer stores and propagates data. Images are divided into blocks with a constant size (4KB). The global meta-data in the data view layer includes a version number for each block. When a VM is reading a block, the version number is compared to the version number in the local meta-data in the local data store. If the latest version of the block is available locally, data is returned immediately. Otherwise, the data store checks the location of the block in the global meta-data and fetches the data remotely. Updating a block increases its version number by one, and the updated global meta-data is then propagated to other replicas.

The consistency model seen by applications is completely determined by the data view layer. Since we only have global meta-data in this layer, it is relatively cheap to implement strong consistency. Section 3.2.2 describes how we can further optimize the synchronization of global meta-data by relaxing the consistency model. Since the data store layer is decoupled, data propagation can be optimized separately without concern about consistency.

Our current design optimizes performance and minimizes traffic cost. The back-end storage can tolerate failures within a single cloud. However, after several migrations the image may have the latest version of blocks scattered in different clouds, and a cloud-level failure before an updated block is propagated may affect the availability of the whole image. If an image is really critical and would like to tolerate cloud failures, it is possible to pass a hint to the propagation manager so that each write is synchronously propagated to different clouds, at significant cost to application performance.

3.2.2 Global Meta-Data Propagation. Due to the long latency in the WAN, meta-data transfer affects application performance significantly if this propagation is in the read/write critical path. We make two key observations:

- If an image file is open for writing, it can only be accessed by a single VM. This happens when the image is private to a VM.
- If an image file is shared by multiple VMs, it is read-only. This happens when the image is a snapshot or a base image file.

These observations indicate that although a migrated VM is expecting strong consistency from the image storage, multiple replicas of the same image file do not need to be identical all the time. It suffices to provide VMs with a "close-to-open" consistency: after a file is closed, reads after subsequent opens will see the latest version. We remove the global meta-data propagation from the read/write critical path by committing the meta-data update locally, and only flushing the global meta-data to a centralized controller when closing the image file. Subsequent opens of the file need to sync the global meta-data with the controller first. Note that the controller is involved only when opening or closing the image file.

3.2.3 Data Propagation Policies. By decoupling the data view and data store layer, a VM can see a consistent disk image no matter where it is migrated. However, fetching each block on demand through the WAN significantly degrades read performance. It is useful to proactively propagate a block before migration if we can predict that it is going to be accessed in another place. Intuitively, a block that is read frequently and updated rarely should be aggressively propagated. Yet propagating a block that is updated frequently is a waste of network resources. Because Internet traffic is typically charged to the user, we need to be careful about proactive propagation.

The data store layer monitors the access pattern of the VM and selectively propagates those blocks that are most likely to be accessed after migration and least likely to be updated after propagation. To this end, we maintain a priority queue of updated blocks for each image file. The priority of each block is updated on each read or write. We use the read/write ratio $F = f_r/f_w$ to calculate the priority of propagating a block, where f_r and f_w are the read and write frequency of a block, respectively. When two blocks have the same value of F, we first propagate the block with a higher read frequency. Thus, the formula of calculating the propagation priority P^b for a block b is

$$P^b = \begin{cases} K \cdot f_r^b / f_w^b + f_r^b & \text{if } f_r^b / f_w^b \geq S \\ -1 & \text{if } f_w^b = 0 \text{ or } f_r^b / f_w^b < S. \end{cases}$$

The value K determines how much we want to favor the read/write ratio. In our prototype, we use $K = 1,000$. The constant S determines the aggressiveness of the propagation. The lower S is, the more data will be propagated. When $f_w^b = 0$, which means that the block has never been updated, or when $f_r^b / f_w^b < S$, we set the priority to -1 to indicate that this block is not to be propagated proactively. In our current prototype, S is set to 1.

Applications might have different requirements for when to propagate data updates. Depending on the workload or the tradeoff between application performance and traffic cost, the propagation should be tuned separately for each VM on the fly. Our storage layer provides parameters K and S as tuning knobs for this purpose. Designing an automatic online tuning policy is left as future work.

After deciding which block to propagate, the next question is where to propagate the block. Since the Supercloud can be deployed to many places even across clouds, propagating each block to all destinations wastes Internet traffic and money. To further optimize data propagation, a transition probability table is added into the image file's meta-data, indicating the probability that a VM is moved from one place to another. Each data block is randomly propagated according to the probability in the transition table so that the destination to which the VM is most likely to be migrated will receive most propagated blocks. In our current prototype, the transition table is provided as a hint by the user when creating the image file. It is also possible to train the table on the fly if the VM is migrated many times.

3.3 Networking

3.3.1 High-Performance VPN. To enable communication between control services (including XenServer and OpenStack services) and VMs, we place them into a Virtual Private Network (VPN). Good performance requires minimizing the number of hops. Existing VPN solutions such as OpenVPN (2016) use a centralized server to forward traffic, which causes high latency and poor throughput. The *tinc* VPN (Tinc 2016) implements an automatic full mesh peer-to-peer routing protocol, minimizing the number of hops traversed between endpoints. However, we found that tinc imposes high-performance overhead, mostly caused by extra data-copy and kernel/user mode switching.

To build a high-performance VPN solution for the Supercloud, we use Open vSwitch (2016), VXLAN tunnels, and the Frenetic SDN controller (Foster et al. 2010). Open vSwitch implements data paths in kernel mode and supports an OpenFlow-based control plane. Each virtual switch uses an uplink to connect to the VMs running on the same first-layer VM and a set of VXLAN tunnels to connect to all other switches, as illustrated in Figure 5. We create a full mesh network here because we want to always forward packets directly to their destinations. We use VXLAN over Generic Routing Encapsulation (GRE) tunnels because this approach is based on UDP instead of a proprietary protocol, and thus is better supported by different firewalls.

In a hierarchical topology, switches running in a private network cannot set up VXLAN tunnels directly with other switches outside the network. A gateway switch is required to forward packets in this case (see Figure 5). The node running the gateway switch is in more than one network. To implement the gateway switch, we create one switch for each of the networks, inter-connected with an in-kernel *patch port*. Each switch builds full mesh connections with other switches in its own network as before and treats the patch port as an uplink.

Switches connected in a full mesh form loops. Ordinarily, one would run a spanning tree protocol, but with a network topology demonstrated in Figure 5, a spanning tree cannot minimize the number of hops for every pair of nodes. To route packets efficiently, switches in the Supercloud

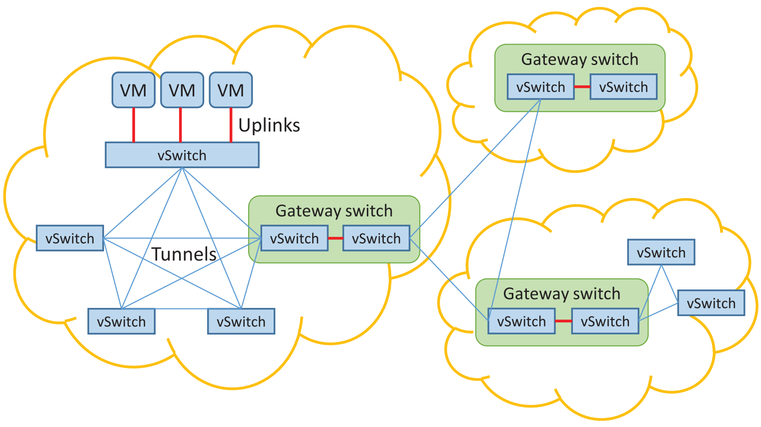


Fig. 5. Network topology of the Supercloud.

VPN are connected to a centralized SDN controller implemented with Frenetic (Foster et al. 2010). The controller learns the topology of the network by instructing the switches to send a "spoof packet." On receiving the spoof packet, switches report to the controller on which port the packet is received so that the controller can record how switches are connected. The controller implements a MAC-learning functionality for each switch. The MAC address, IP address, and location of a VM is learned when it sends out packets.

Address Resolution Protocol (ARP) packets are forwarded directly to the destination instead of broadcast. When routing a packet, the controller calculates the shortest path on the network and installs OpenFlow rules along the path to enable the communication. This is done only once for each flow, and subsequent data transportation does not need to involve the controller anymore.

3.3.2 Supporting VM Live Migration. Supporting VM live migration is another challenge. To keep the IP address of a migrated VM unchanged, the underlying VPN needs to adjust the routing path and re-direct traffic. However, the adjustment cannot be done immediately when the migration is triggered, because at this point the VM is still running in the original location and existing network flows should not be affected.

To know at which point the routing path should be adjusted without adding a hook into the hypervisor, before triggering the migration, the controller is notified with the source and destination of the migrated VM. The controller then injects a preparation rule in the destination switch so that it can get an immediate report when a packet with the migrated VM's MAC address is received. When migration is finished, the migrated VM will send out an ARP notification. This is captured by the controller so that it knows that the migration has finished. The controller then updates all switches that have the migrated VM's MAC address in the MAC table, avoiding the usual ARP broadcast.

3.3.3 Supporting Public IP Addresses. A public IP address is required to expose a service to the public network. We currently support addressing a second-layer VM from the public network by allocating a public IP address to a first-layer VM in the Supercloud network (i.e., a public IP front-end), then applying port forwarding to map certain ports to the second-layer VM. This solution has good performance because packets can be routed in the public network to the VM directly.

A challenge arises when this VM is migrated to a different cloud provider. Without specific support from Internet Service Providers (ISPs) such as anycast, mobile IP, or multi-homing, traffic sent to the public IP address needs to be re-directed by the same first-layer VM, no matter where the second-layer VM currently resides. Although this works, it can lead to high latency.

To address this issue, we adopt an idea from Content Distribution Networks (CDNs): instead of giving the same public IP address to clients all over the world, we give each client the public IP address of a front-end server in a nearby data center. Taking this extra but short hop, the routing from the front-end to the second-layer VM is always optimized in the public network, performing much better for most clients than a centralized public IP solution.

For applications that do not want to pay the additional cost of the front-end servers, multiple public IP addresses, and an additional hop, the Supercloud has its own dynamic DNS service and can update the DNS mapping after migration. Note that clients might see an out-of-date public IP address before the DNS cache is expired. For services that use protocols such as HTTP, SOAP, and REST, we deploy an HTTP redirection service on the original public IP front-end and respond to clients with an HTTP redirection response.

3.4 Management and Scheduling Framework

A Library Cloud includes a resource management platform that manages underlying computation, storage, and network resources, and provides an easy-to-use Library Cloud API to access them. To this end, we adopted the widely used OpenStack. OpenStack provides interfaces to manage cloud resources through a Web-based *OpenStack dashboard*, or via the *OpenStack API*. To incorporate OpenStack with Xen-Blanket, we run XenServer (2016), a server virtualization platform that provides feature-rich APIs to communicate with the Xen hypervisor.

One technical challenge that we faced was that XenServer requires a direct installation with an ISO image of the complete software stack, including the Xen hypervisor and Dom0 OS, which cannot be easily ported to different clouds. To overcome this, we use xenserver-core (2016), a set of core components of XenServer that can be installed in a standard CentOS installation. We ported Xen-Blanket to Xen 4.2.2 and Linux Kernel 3.4.53 to run xenserver-core.

Applications in the Supercloud can make migration decisions by themselves and issue migration commands through the OpenStack API. To facilitate the process of deciding optimal placement, the Supercloud provides a scheduling framework for the user. Users can customize the scheduling policy for different applications by implementing some interfaces. The Supercloud scheduler periodically evaluates current placement of the application and automatically adjusts it when a better placement is found.

Suppose that the Supercloud is deployed to N different data centers: $d_1, d_2, \ldots, d_N$. We denote a placement plan for an application with k nodes as $P = \{p_1, p_2, \ldots, p_k\}$, where p_i is the location of the i^{th} node. Periodically, the Supercloud measures end-to-end latency between all different data centers and stores the results in a latency matrix L, where $L(i, j)$ is the Round-Trip Time (RTT) from d_i to d_j. The workload of the application is also captured periodically in a workload statistics report S. S is application specific, so it should be monitored in the application level and passed to the scheduling framework as an opaque handle. Applications can pack more information into S if needed, such as the current placement, topology, and workload history.

To evaluate a placement plan, the application has to provide (1) an evaluation function $f(P, S, L)$ that evaluates a placement plan under a certain workload and returns a score and (2) a threshold T that specifies the minimal score change that can trigger VM migration. Using f, the scheduler iterates through all possible placement plans and gets a set of candidate placement plans D that maximize the score and outperform the current placement plan $P_{current}$ by at least T, that is:

$$D = \arg\max_{P}\{f(P, S, L) | f(P, S, L) \geq f(P_{current}, S, L) + T\}.$$

To choose a placement plan in D, the scheduler compares each placement plan with the current placement $P_{current}$ and selects the one that requires the fewest migrations.

In the following, we present case studies to demonstrate policies for two different types of applications.

3.4.1 Non-Distributed Applications. For a single VM running a non-distributed application such as a MySQL database, a placement plan is simply the location of the VM: $P = \{p\}$. We can deploy a set of service front-ends located in different data centers to collect user requests and forward them to the VM. This architecture makes the location of the VM transparent to clients and can help with simplifying placement evaluation (Section 3.3.3).

The goal of placement is to minimize average latency for all front-ends. Here we only consider latency in the network and ignore processing time for different types of requests. S is defined as

$$S = \{(s_1, d_{s_1}), (s_2, d_{s_2}), \ldots, (s_n, d_{s_n})\},$$

where n is the number of front-ends, s_i is the number of active clients of the i^{th} front-end, and d_{s_i} is its location.

Each front-end is assigned a weight, which equals the number of requests it receives. The score of a placement plan is the weighted average latency of all front-ends (negated so lower latencies result in higher scores)—that is,

$$f(P, S, L) = -\sum_{i=1}^{n} s_i \cdot L(d_{s_i}, p).$$

3.4.2 Distributed Applications With Replicated State. Distributed applications typically maintain replicated state using some consensus protocol. In the following, we use ZooKeeper as an example. ZooKeeper is implemented using a replicated *ensemble* of servers kept consistent using ZooKeeper Atomic Broadcast (Zab) (Junqueira et al. 2011). In Zab, one server acts as a leader that communicates with a majority of servers to agree on a total order of updates. Read requests are handled by any node in the ensemble, whereas write requests must be broadcast by the leader and agreed upon by the majority of the ensemble.

For a ZooKeeper ensemble with m nodes, a placement plan is the location of all nodes: $P = \{p_1, p_2, \ldots, p_m\}$, where p_l is the location of the leader. Again, we assume that there are n front-ends in different data centers to collect and forward client requests. To evaluate a placement plan, we need to consider read and write requests separately. Thus,

$$S = \{(r_1, w_1, d_{s_1}), (r_2, w_2, d_{s_2}), \ldots, (r_n, w_n, d_{s_n})\},$$

where r_i and w_i are the number of read and write requests received by the i^{th} front-end and d_{s_i} is its location.

The goal of placement is again to minimize average network latency for all front-ends, ignoring request processing time and only considering network delay. For the purpose of load balancing, ZooKeeper clients are typically connected randomly to one node in the ensemble. Read requests can return immediately. For the i^{th} front-end, the expected read latency is

$$R_i = \underset{j=1\ldots m}{\text{avg}} L(d_{s_i}, p_j).$$

Write requests are processed in three steps:

- *Step 1*: A write request from the i^{th} front-end goes randomly to one of the ZooKeeper nodes. The average latency is
$$W_i^{(1)} = \underset{j=1\ldots m}{\text{avg}} L(d_{s_i}, p_j).$$

- *Step 2*: The write request is then forwarded to the leader of the ensemble. The average latency for this step is

$$W_i^{(2)} = \underset{j=1\ldots m}{\text{avg}} L(p_j, p_l).$$

- *Step 3*: The leader broadcasts the request twice in a protocol similar to two-phase commit, and for each broadcast it must wait until at least half of the ensemble replies. The average latency for this step is

$$W_i^{(3)} = 2 \times \underset{j=1\ldots m, j\neq l}{\text{median}} L(p_l, p_j).$$

Thus, the expected network latency for a write request from the i^{th} front-end is $W_i = W_i^{(1)} + W_i^{(2)} + W_i^{(3)}$. The evaluation function for ZooKeeper calculates the weighted average network latency of all requests, assuming that we give read and write requests a weight α and β, respectively:

$$f(P, S, L) = -\sum_{i=1}^{n} (\alpha \cdot R_i \cdot r_i + \beta \cdot W_i \cdot w_i).$$

3.5 Discussion

The benefits of the Supercloud do not come without costs. For example, nested virtualization imposes performance overhead including CPU scheduling delay and I/O overhead. Users can evaluate this tradeoff based on migration frequency and performance requirements, and it is application dependent. We are currently in the process of building support for containers into the Supercloud. Container technology (Soltesz et al. 2007) provides another way to homogenize different cloud platforms. However, compared to live VM migration, which is mature and widely used, container migration (Mirkin et al. 2008) technology is preliminary and involves a checkpointing/resume mechanism that might cause a relatively large performance hiccup. Even with mature container migration, the challenges of building a well-performing Supercloud remain essentially the same.

4 EVALUATION

The Library Cloud represents a powerful abstraction, and its implementation via the Supercloud represents a new and unique capability: A distributed service can migrate live, and incrementally or whole, between availability zones and heterogeneous cloud providers. In this section, we investigate four research questions enabled by and enabling this abstraction and capability:

(1) How effective is the abstraction and its scheduler in enabling applications to follow the sun?
(2) Is VM migration a viable approach to follow the sun?
(3) How effective is SDRS in saving the cost of running an application?
(4) What is the efficacy of Supercloud storage and networking in supporting live VM migration?

4.1 Follow the Sun

In this set of experiments, we use a distributed application, ZooKeeper (Section 4.1.1), and a database, MySQL (Section 4.1.2), to investigate application performance benefits due to following the sun. Results demonstrate that the Supercloud scheduler was able to automatically follow the sun and migrate resources geographically and across heterogeneous clouds, enabling high performance to be maintained.

4.1.1 *A ZooKeeper Ensemble.*

ZooKeeper writes require a majority of ZooKeeper servers (aka, the ensemble) and a ZooKeeper leader to coordinate and order all writes. High network latency between ZooKeeper servers or between clients and the ZooKeeper leader causes high end-to-end service latencies. For good performance, the majority of ZooKeeper servers have to be located where most active clients are, and the ZooKeeper leader must be part of that majority.

Experimental setup. To evaluate the efficacy of following the sun using the scheduler described in Section 3.4, we used a ZooKeeper distributed application and measured its ability to respond to clients in two different regions: Virginia and Taiwan. The clients used a workload corresponding to a week-long trace of active MSN connections (see Figure 5(a) in Chen et al. (2008)). The trace does not specify the start time of the MSN workload—we made an educated guess based on the diurnal pattern in the data and assumed that the workload started at 12 am at the beginning of a Monday. Nor does the trace specify the absolute workload—we scaled the workload so that the peak workload can be served by our ZooKeeper cluster when latency is low. We varied load based on location and time: we circularly shifted the start of the trace by 12 hours ahead to produce an identical workload where the start of the trace from Virginia was 12 hours before starting in Taiwan. The ZooKeeper clients randomly connected to one ZooKeeper server and submitted blocking read and write requests in a ratio of 9:1. Each read operation obtained a 64-byte ZooKeeper *znode*; each write operation overwrote a 64-byte znode. The clients ran on first-layer VMs in Virginia on Amazon VMs and in Taiwan on Google VMs.

For the ZooKeeper ensemble, we deployed the servers in the Supercloud simultaneously spread across Amazon Virginia and Google Taiwan regions. The type of first-layer VMs used in our experiments was `m3.xlarge` in Amazon and `n1-standard-4` in Google, both of which had four vCPUs and 15GB memory. The ZooKeeper ensemble ran on three second-layer VMs, denoted as `zk1`, `zk2`, and `zk3`. Each VM had 1GB RAM and one CPU core. We used one first-layer VM in each region to serve as the Supercloud storage server. The data of the ZooKeeper nodes was stored on disk and propagated automatically by the storage servers.

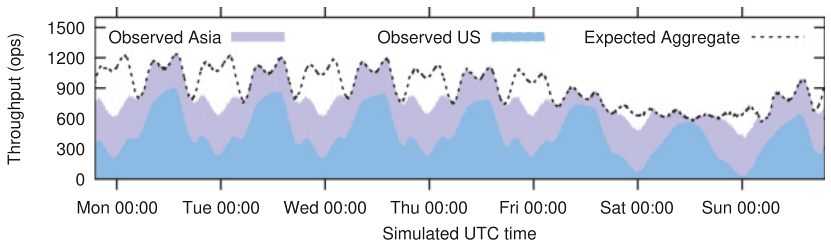
We implemented the scheduling evaluation functions for ZooKeeper discussed in Section 3.4. ZooKeeper server VMs reported VM load to the scheduler. How quickly the scheduler reacts to workload changes depends on the frequency of workload monitoring and evaluation. For this experiment, scheduler placement evaluation was triggered every minute. Once a VM migration was started, the scheduler waited until it finished before considering a new placement. Migrations were performed in parallel, so going from one placement plan to another was fast.

We evaluated ZooKeeper in three scenarios:

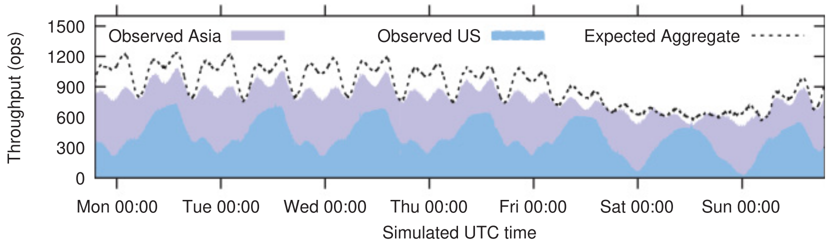
(1) *US Ensemble*: All three ZooKeeper nodes, `zk1`, `zk2`, and `zk3`, were in Amazon Virginia.
(2) *Global Ensemble*: `zk1` and `zk2` were in Amazon Virginia, and `zk3` was in Google Taiwan.
(3) *Dynamic Ensemble*: The Supercloud scheduler automatically placed VMs according to the workload.

To simplify experimentation and save cost, we sped up the trace by a factor of 30 so that the 1-week trace could be replayed in less than 6 hours. The performance of the US Ensemble and Global Ensemble was not affected by the speed-up, whereas the performance of the Supercloud was slightly degraded since migration was not sped up correspondingly.

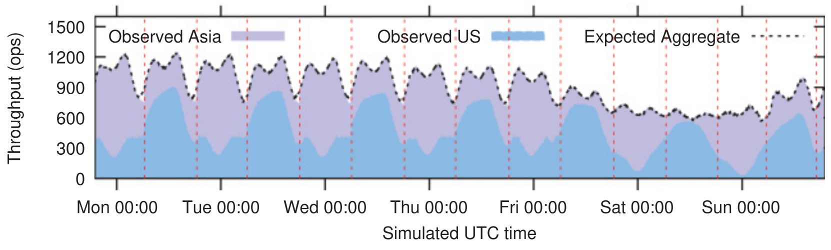
Experimental results. Figure 6 shows the throughput (in operations per second) for the three scenarios: US, Global, and Dynamic ensembles. Each scenario displays throughput measured for the US clients in Virginia ("Observed US"), throughput measured for Asian clients in Taiwan ("Observed Asia"), and throughput if latency were negligible ("Expected Aggregate"). The throughput results observed by US and Asian clients are stacked on top of one another (with US below Asia)



(a) US Ensemble



(b) Global Ensemble

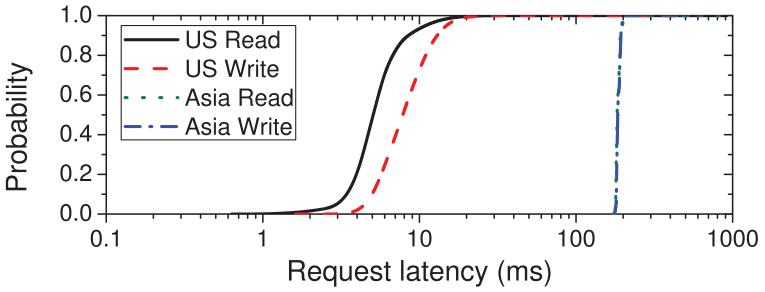


(c) Dynamic Ensemble

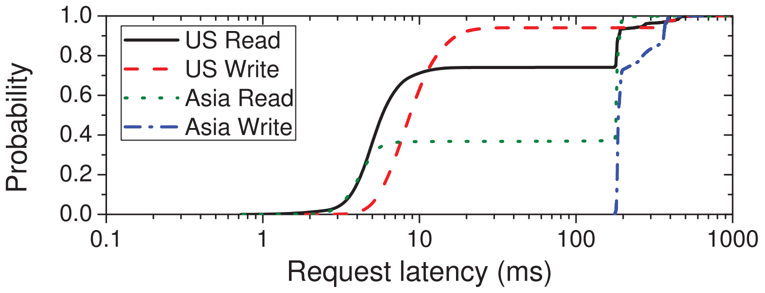
Fig. 6. ZooKeeper throughput (vertical dashed lines indicate the end of the migrations).

so that the top of the throughput results for the clients in Asia show the total aggregate throughput that was observed by all clients. For the US Ensemble scenario (Figure 6(a)), US clients were able to reach their maximum throughput, but clients in Asia suffered from high latencies and, as a result, experienced poor throughput. For the Global Ensemble (Figure 6(b)), by placing one node in Taiwan, one third of Asian clients experienced increased read throughput and total throughput was improved. However, two thirds of the read requests and all write requests from Asian clients still experienced poor throughput. Moreover, US clients were not able to reach their maximum throughput during peak times because one third of clients were connected to the ZooKeeper server in Taiwan. Finally, Figure 6(c), shows the result for the Supercloud case where throughput remained high and matched the expected performance as the scheduler was able to automatically migrate ZooKeeper servers to the region where load was high.

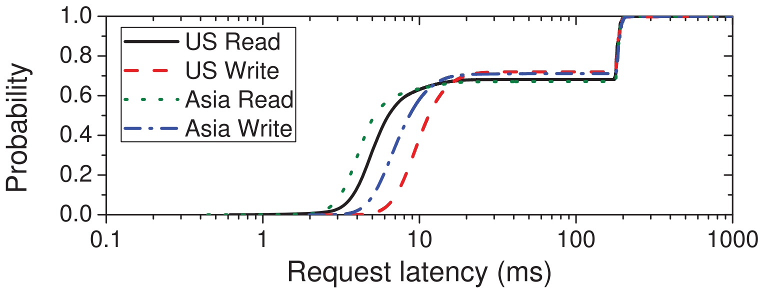
Figure 7 shows the cumulative latency distributions for read and write operations. We can see that for the US Ensemble (Figure 7(a)), 90% of US clients observed less than 15ms latency for read and write operations, whereas client latencies in Asia were close to 200ms. (We repeated the experiments with an Asia Ensemble and observed symmetric results.) In the Global Ensemble (Figure 7(b)), the ZooKeeper quorum was in the US. The server in Taiwan helped Asian clients gain better read throughput, although write latency did not improve: 37% of reads from Asian clients completed in 15ms because they were handled by the server in Taiwan. The high tail latency of US



(a) US Ensemble



(b) Global Ensemble



(c) Dynamic Ensemble

Fig. 7. ZooKeeper latency CDF.

read/write performance was because some requests went to the server in Taiwan. (A symmetric experiment with the quorum in Asia observed the same results.)

Finally, in the Dynamic Ensemble (Figure 7(c)), the scheduler automatically figured out that placing all three nodes in one location was the best placement for the workload. This is because clients connect to servers randomly and the read/write ratio is fixed. For example, leaving one node in Asia can only improve performance of one third of read requests from Asia, but also make one third of read requests from the US suffer from long latency. This was not beneficial when the US workload was higher than the Asia workload. Therefore, it always migrated the whole ensemble together: the scheduler migrated the ensemble every 12 hours between Amazon Virginia and Google Taiwan to make the ZooKeeper cluster close to most clients. The scheduler placement resulted in low latency: 61% of writes and 69% of reads were less than 15ms. About 30% of requests had longer latency since they were from clients accessing the servers remotely. However, on average, the Dynamic Ensemble experiment achieved significantly lower and more balanced latency across all clients than the US and Global ensembles.

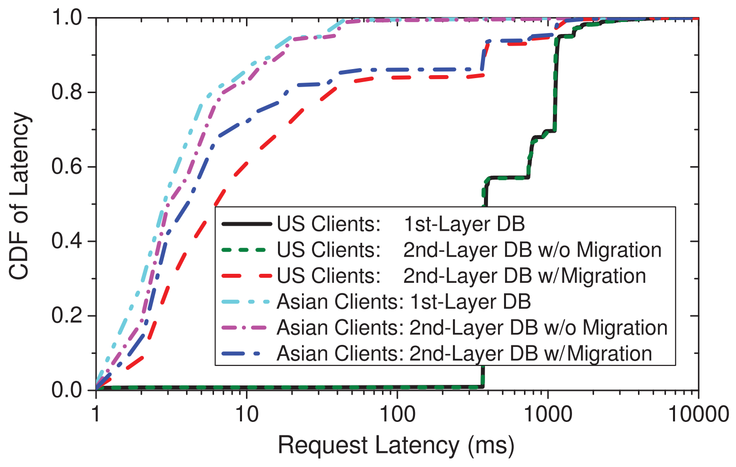


Fig. 8. TPC-W Client Latency CDF.

4.1.2 *A Single MySQL Database.*

Experimental setup. We used the `TPC-W` benchmark to stress a single MySQL database and deployed Web servers in two first-layer VMs in the Amazon Virginia and Google Taiwan regions. The TPC-W benchmark clients performed 80% browsing (read) and 20% ordering (write).

The MySQL database ran in the Supercloud in a second-layer VM that had three vCPUs and 2GB memory. The first-layer VM could run in the Amazon Virginia or Google Taiwan regions with instances described in Section 4.1.1. The storage engine of MySQL was set to `MEMORY`.

For comparison, we evaluated three scenarios:

(1) *First-layer database*: The database was deployed in a first-layer VM located in Taiwan.
(2) *Second -layer database without migration.* The database was deployed in a second-layer VM located in the Taiwan.
(3) *Second-layer database with migration*: The database was deployed in a second-layer VM, and the Supercloud scheduler automatically migrated the VM back and forth between Amazon Virginia and Google Taiwan regions according to the workload.

Experimental results. Figure 8 shows the results: cumulative latency distributions for Web interactions. There are a several points to observe. First, for the first- or second-layer databases without migration, 80% of the clients in Taiwan ("Asian Clients") observed less than 10ms latency, whereas latencies for clients in Virginia ("US Clients") were close to 200ms. Comparing with the latency of Asian clients in the first two scenarios, we can see a slight difference in the curve caused by the small overhead that the Supercloud presents.

In the third case, "2nd-Layer DB w/Migration," the Supercloud scheduler automatically migrated the MySQL database every 12 hours between Taiwan and Virginia to place the database where load was high. As a result, the aggregate throughput to match the workload. Figure 8 shows that latencies for 70% Web interactions were less than 10ms. The tail latency was caused by requests issued in the night of the corresponding region when the database was far away. The small plateaus in the curves were caused by the 20% ordering requests that write to the database and incur a higher latency. And the Asian clients observed a slightly shorter latency, which indicated that the Google VMs experienced lower latencies than the Amazon VMs.

4.2 Comparing Migration Approaches

In this set of experiments, we use two popular distributed applications—Cassandra and ZooKeeper—to investigate the viability of relying on live VM migration via the Supercloud to enable distributed applications to follow the sun. The results demonstrate that not only can distributed applications benefit from the Supercloud, but they can also do so without any change to the application and can outperform other approaches that require application modification.

4.2.1 Cassandra Migration. Cassandra (2016) supports adding and removing nodes, and automatically handles data replication and load distribution. When following the sun, it makes little sense to move only some of the Cassandra nodes. Migrating an entire Cassandra cluster can be implemented by adding nodes in the destination location and removing nodes in the source location.

Experimental setup. To compare migration approaches—"application" and "Supercloud"—we started a three-node Cassandra cluster in the Amazon Virginia region, then migrated the whole cluster to the Google Taiwan region. As a result, the migration was across geographically separated clouds. We deployed a three-node Cassandra cluster with replication factor of 2 in second-layer VMs with one vCPU and 2GB memory. For the first-layer VMs, we used `m3.xlarge` instances in the Amazon Virginia region and `n1-standard-4` instances in the Google Taiwan region.

For the application migration approach, a Cassandra cluster was initially running in Virginia. To move all nodes to Taiwan, we followed the process specified in the Cassandra documentation for replacing running nodes in the cluster. Three new nodes in Taiwan joined the cluster with a configuration file pointing to a seed node and automatically propagated their information through Cassandra's gossip protocol. The key space then spread evenly across all six nodes, and the data associated with the key moved automatically. After the three new nodes joined the cluster, we performed "node decommissioning" in each of the three original nodes. As a result, the Virginia nodes copied their data to the Taiwan nodes. The gossip protocol automatically updated each node with the cluster information transparently to the clients.

For the Supercloud migration, we set up a Supercloud across the Amazon Virginia and Google Taiwan regions using the same first- and second-layer VM configurations described earlier.

For the workload, data was stored in memory and moved explicitly with the application approach or migrated with the VM with the Supercloud approach. We populated the database with 30,000 key-value pairs, each of which had a size of 1KB. We started one client in each region. The clients read and wrote to the database continuously with a ratio of 4:1; each read operation obtained the value of a random key, and each write request updated the value of a random key. The consistency level was set to `ONE` (default), indicating eventual consistency.

Experimental results. Figure 9(a) shows the throughput of application migration (in operations per second) for both Amazon Virginia and Google Taiwan clients. Although not easy to see in this case, the throughput results for both types of clients are again stacked on top of one another, with the throughput of the Asia clients below that of the throughput of the US clients. The top of the graph therefore shows the total aggregate throughput. During migration, the throughput dropped dramatically and remained low for about 200 seconds. Even after the migration completed, it took another 200 seconds to restore performance to the original throughput due to the overhead of data replication.

With the VM migration mechanism in the Supercloud, we migrated the three Cassandra VMs from Amazon Virginia to Google Taiwan in parallel. Total migration time was around 2 minutes, but note that most of this time happens in the background without affecting the application. Figure 9(b) shows that performance impact was small with a downtime of around 5 seconds. (Downtime could be reduced further by synchronizing the migration finishing time—a project

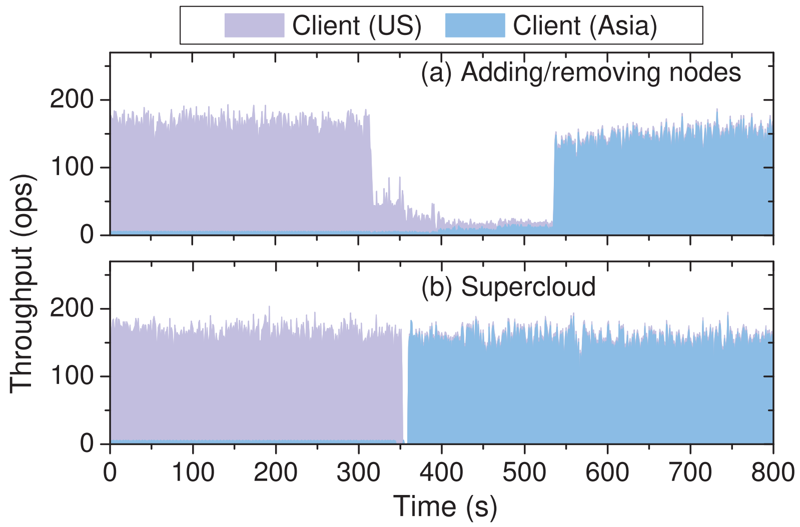


Fig. 9. Comparison of different migration mechanisms for moving a Cassandra cluster.

we are planning for future work.) The Supercloud maintained the same IP addresses and network topology without triggering any unnecessary data replication or key shuffling.

4.2.2 ZooKeeper Migration.

Experimental setup. To evaluate different follow-the-sun approaches for ZooKeeper, we set up a Supercloud using Amazon EC2 `m3.xlarge` instances (four vCPUs, 15GB memory) in the Virginia and Tokyo regions. We deployed a ZooKeeper ensemble in three second-layer VMs with one vCPU and 1GB memory. The leader was initially in Virginia, with one follower in Virginia and another in Tokyo. We started one client in each region generating a constant workload with read/write ratio 9:1. Each read operation obtained a 1KB ZooKeeper znode; each write operation overwrote a 1KB znode.

We compared three approaches.

(1) A "two-step reconfiguration" moved a majority of the servers from one region to another: we first added a new node in Tokyo, then removed the original leader in Virginia.

    Unfortunately, the two-step reconfiguration does not guarantee that a node in Tokyo will be elected as the leader. Instead, it was more likely that one of the Virginia nodes would become the leader, which was not desirable since the majority of the ensemble is in Tokyo.

(2) A "three-step reconfiguration" ensured that the leader ended up in Tokyo while maintaining the same level of fault tolerance: we added two nodes in Tokyo first, then removed both nodes from Virginia. After the new leader was elected in Tokyo, we added a new node in Virginia and removed one of the nodes from Tokyo.

(3) Using the Supercloud, we transparently migrated a second-layer VM running the leader from Virginia to Tokyo. Neither ZooKeeper nor clients required any modification.

Experimental results. Figure 10 shows the stacked throughput (in operations per second) of clients in both Virginia and Tokyo regions before and after leader migration using the three migration approaches discussed previously. After the two-step reconfiguration shown in Figure 10(a),

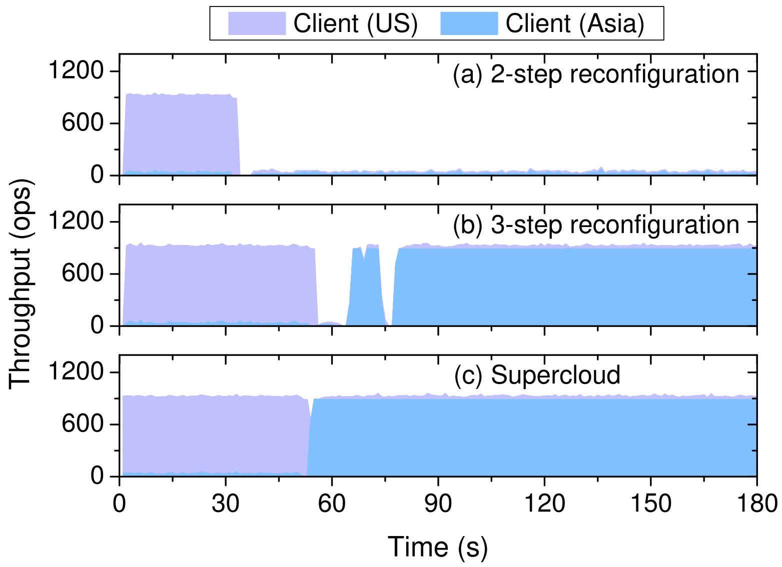


Fig. 10. Comparison of different migration mechanisms for moving the ZooKeeper leader.

the leader role was switched to a Virginia node while the two followers were in Tokyo, which was inefficient, and as a result, the throughput dropped significantly for both clients. Using the three-step reconfiguration in Figure 10(b), the leader was successfully moved to Tokyo, so good throughput for the Tokyo clients were achieved. Unfortunately, the three-step reconfiguration took 20 seconds, during which performance was inconsistent and low. Finally, Figure 10(c) shows the performance of the Supercloud: the drop in performance was less than a second and transparent to the ZooKeeper application and its clients.

4.3 Dynamic Resource Scheduling

To examine the efficacy of SDRS, we evaluate it in a private cloud using the TPC-W benchmark. The first-layer VMs have 8GB memory and eight CPU cores, but we only leave 2.5GB memory and two CPU cores for the second-layer Dom0 and OpenStack DomU, and always restrict user VMs to use one other CPU core and 1GB memory. This mimics an environment where first-layer VMs have three CPU cores and 3.5GB memory in total. Each second-layer user VM has 1GB memory and one CPU core.

To decide whether VM consolidation should be performed, the SDRS scheduler needs to estimate future workload of applications. Previous studies have demonstrated that workloads for enterprise applications typically show a periodicity that is a multiple of hours, days, or weeks (Gmach et al. 2007). It is a common practice to schedule a VM consolidation plan based on the repeating pattern of application workloads. Workload prediction and modeling techniques (Shen et al. 2011; Song et al. 2014; Wood et al. 2007; Xiao et al. 2013) can also be applied to SDRS. In this experiment, we assume that the workload has a predictable recurring pattern. Three TPC-W clients follow a synthetic workload and feed requests to three TPC-W servers running in different second-layer VMs, respectively. This workload is carefully chosen so that neither the client nor the network can be overloaded.

We compare three scenarios: *no over-subscription (MAX)*, when three second-layer VMs running the TPC-W servers are in separated first-layer VMs so that resources are always guaranteed; *always*

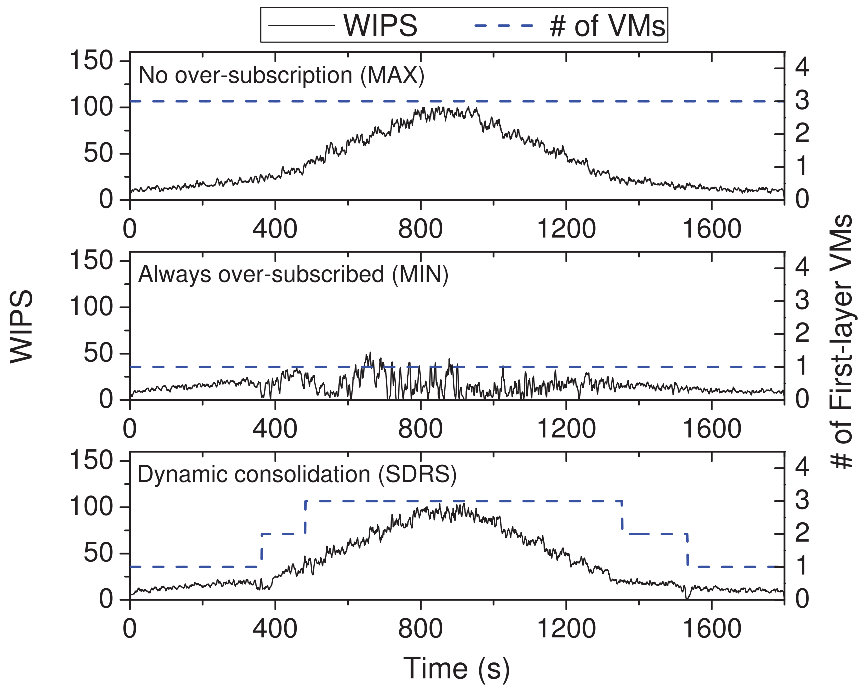


Fig. 11. Evaluation of SDRS. Dashed lines show the number of first-layer VMs being used.

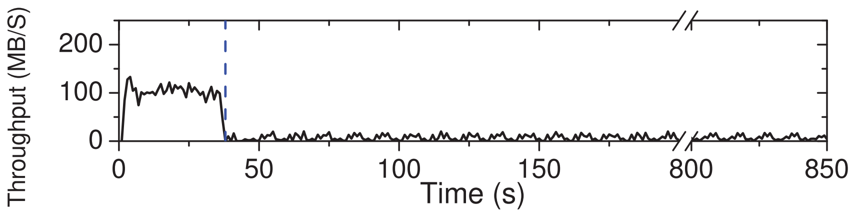
over-subscribed (MIN), when all three second-layer VMs are packed into one first-layer VM using memory ballooning; and *dynamic consolidation (SDRS)*, when SDRS dynamically migrates VMs. Since the workload pattern is known in advance, we use a pre-defined scheduling plan. Memory size of the VMs is adjusted on the fly so that VMs can use as much memory as possible.

Figure 11 shows the 5-second moving average of aggregate Web interactions per second (WIPS) of all three TPC-W servers. MAX wastes a lot of resources, whereas MIN significantly hurts the performance. In contrast, SDRS on average uses only 2.13 first-level VMs throughout the experiment, by paying only 1.5% performance degradation. That translates to 29% lower cost than MAX during a half-hour experiment.

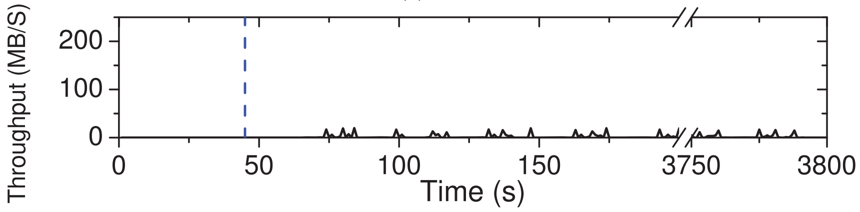
4.4 Storage Evaluation

Experimental setup. To evaluate storage performance under migration, we deployed a Supercloud setup with two `m3.xlarge` instances in Amazon Virginia and Northern California regions. The ping latency between two VMs in these regions was 75ms. We started a user VM with one virtual CPU core and 512MB memory and ran the `DBENCH` (DBENCH 2016) benchmark in it, which simulated four clients generating 500,000 operations each on the file system based on the standard `NetBench` benchmark. We configured the DBENCH clients so that they ran at the fastest possible speed. Without migration, a VM using local storage finished the benchmark in 2 minutes.

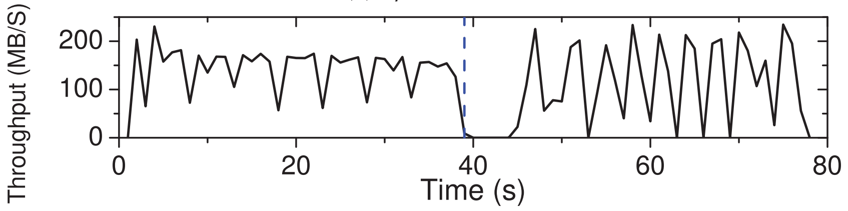
In the following experiments, immediately after starting the benchmark, we triggered a VM migration from Virginia to Northern California. The full migration took 40 to 50 seconds (most of which was in the background while the VM kept running). During most of the full migration, in particular during the pre-copy phase, the benchmark kept running at the old location. In fact, before the VM was actually moved to Northern California, one third of the workload had finished.



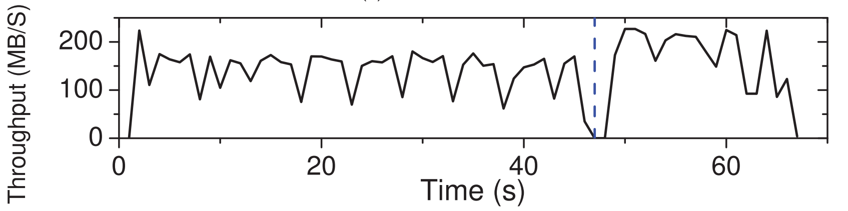
(a) NFS



(b) Sync-on-Write



(c) On-demand



(d) Supercloud

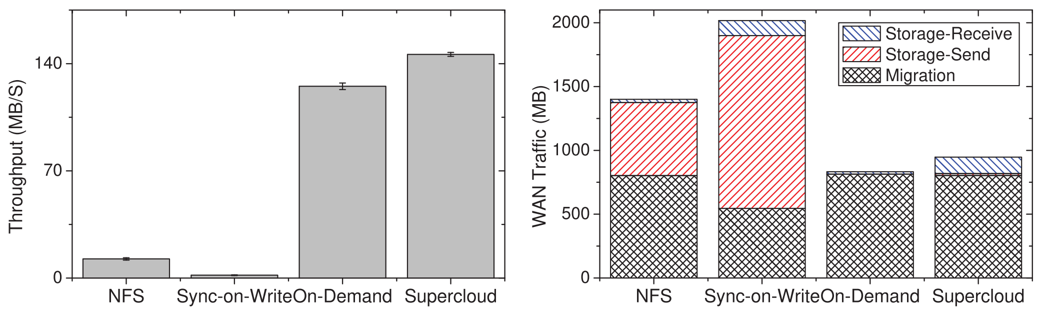
Fig. 12. Average throughput per second of the DBENCH benchmark in the migrated VM. For clarity, the *x*-axes are on different scales.

We compared the performance of the benchmark with four different underlying storage systems:

- *NFS*: A traditional NFS server deployed in Amazon Virginia.
- *Sync-on-Write*: A strongly consistent geo-replicated store that synchronously propagates each write.
- *On-demand*: Supercloud storage with proactive propagation turned off.
- *Supercloud*: Supercloud storage with proactive propagation in the background.

Except for NFS, all schemes were implemented in the Supercloud's propagation manager for fair comparison.

Experimental Result. Figure 12 shows the results: the average throughput in each second of the DBENCH benchmark. The vertically dashed lines indicate when migration was finished. Figure 12(a), NFS, shows that throughput dropped significantly after the clients were migrated. Low performance resulted from remote disk accesses that incurred high latency. Figure 12(b),



(a) DBENCH throughput (b) WAN traffic on the migration destination

Fig. 13. Average throughput and total WAN traffic of the DBENCH benchmark in the migrated VM.

Sync-on-Write, shows that a strongly consistent geo-replicated storage incurs high-performance overhead both before and after migration because each write needs to be propagated through the WAN. Figure 12(c), Supercloud On-demand, eventually achieved good average throughput after migration since all writes could be committed locally. However, read throughput was low right after migration since no blocks had been copied ahead of time. Finally, Figure 12(d), Supercloud proactive propagation, shows that most read and write requests could be served locally. Consequently, the corresponding benchmark took the least time.

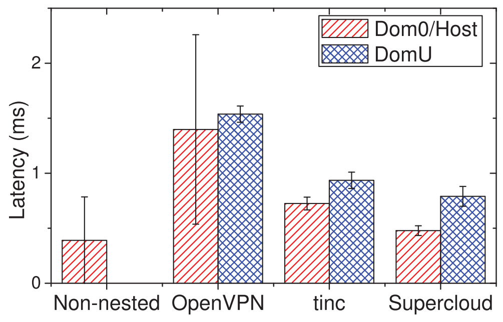
Figure 13(a) shows the average throughput for each case. We repeated the same experiment three times and report the average number with the standard deviation. The Supercloud proactive propagation achieved the highest throughput.

Figure 13(b) shows the total WAN traffic measured at the migration destination (Northern California). The migration caused 850MB of WAN traffic for NFS, On-demand, and Supercloud, but only 570MB of traffic for Sync-on-Write. Unsurprisingly, the migration traffic depended on the rate at which the VM dirties memory pages. A benchmark running at a higher speed caused more dirty memory pages to be copied during migration. Sync-on-Write storage results in low throughput in the benchmark, thus lowering the page dirty rate of the whole VM. Both NFS and Sync-on-Write incur a lot of outgoing traffic when accessing storage because each disk write needed to go through the WAN. Sync-on-Write also had higher incoming traffic, which was generated when the benchmark was running in Virginia. On-demand achieved the lowest WAN traffic since it fetched data remotely only when necessary, and no update propagation was needed. The Supercloud incurred more incoming traffic than On-demand because it pushed some data that was not needed, but it achieved higher throughput and predictable performance.

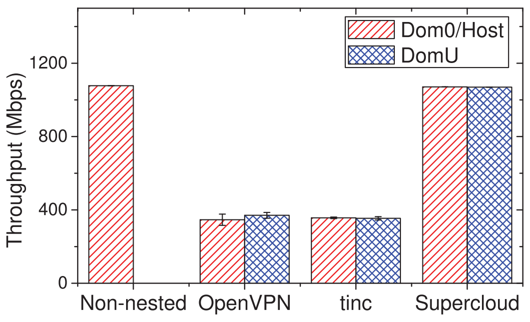
4.5 Network Evaluation

To evaluate the network performance of the Supercloud, we deployed the Supercloud in two `m3.xlarge` instances in the Amazon Virginia region and measured UDP latency (using UDP ping) and TCP throughput (using `netperf`) between second-layer Dom0 VMs and DomU VMs, respectively. For comparison, we ran the same benchmark using the following settings:

- *Non-nested*: A setup where we ran the benchmark directly in the first-layer VMs for baseline purposes
- *OpenVPN*: A VPN solution using a centralized controller, also running in Amazon Virginia
- *Tinc*: A P2P VPN solution, which implements full-mesh routing
- *Supercloud*: The Supercloud implementation based on Open vSwitch.



(a) UDP latency



(b) TCP throughput

Fig. 14. Network performance evaluations.

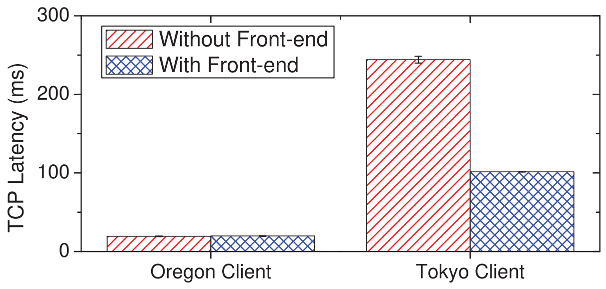
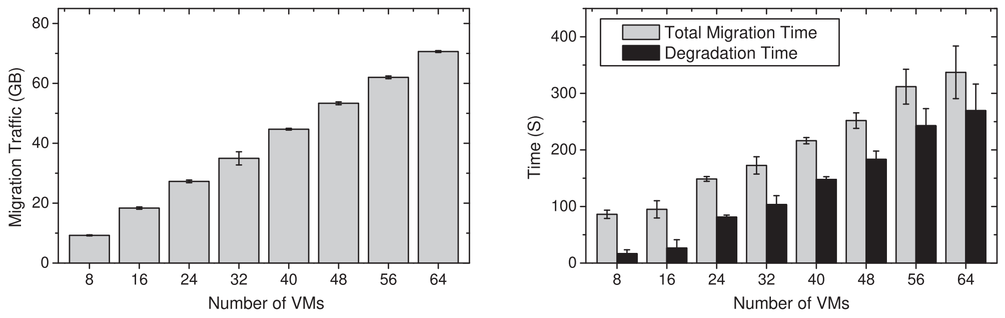


Fig. 15. Impact of using public IP front-ends.

For the latency test, we ran 50 UDP pings (1 ping per second). Figure 14(a) shows the average latency and standard deviation across all runs. Although our Open vSwitch–based virtual network slightly increases the UDP latency, the overhead was much smaller than either OpenVPN or tinc. The latency for Dom0 was smaller than DomU because network packets to DomU go through Dom0 first. OpenVPN had the highest latency because all packets travel through the centralized controller.

To measure TCP throughput, we enabled jumbo frames in all setups and repeated a `netperf` TCP stream benchmark with default options 10 times for each setup. As shown in Figure 14(b), the throughput of non-nested instances, Dom0, and DomU in the Supercloud all achieved 1Gbps with small variance. In contrast, tinc and OpenVPN could only achieve 300Mbps. Because both tinc and OpenVPN use a tap device connected to a user-level process, the extra memory copy and kernel-user mode context switching incurred significant overhead.

To evaluate the impact of using public IP front-ends, we deployed a second-layer VM as a server running in the Amazon Oregon region. We had two clients running in Google Compute Engine Oregon and Tokyo regions, respectively, and used the `hping` tool to measure TCP latency from clients to the server with and without public IP front-ends. Each test was repeated 15 times. Without public IP front-ends, the clients used the public IP address of the first-layer VM directly to communicate with the server. With public IP front-ends, each client communicated with a front-end running in a nearby Amazon data center. Figure 15 shows the average TCP latency from different clients to the server in both cases. For the Oregon client, the overhead of adding a front-end was negligible. Interestingly, the Tokyo client measured lower latency by going through the public IP front-end in Tokyo. This was because the front-end was running in an Amazon data center, and the latency from Amazon Tokyo to Amazon Oregon was significantly less than the latency from Google Tokyo to Amazon Oregon.



(a) Migration traffic (b) Migration time and period of performance degradation

Fig. 16. Migration time and traffic with different number of VMs.

4.6 Scalability

In this section, we evaluate scalability of Supercloud live migration by migrating a Cassandra cluster with varying numbers of nodes. We set up a Supercloud in the Amazon EC2 Oregon and Tokyo regions. In each region, we deployed 16 `c4.xlarge` nodes (four vCPUs, 7.5GB memory) for running second-layer VMs. We chose 16 as the cluster size in each region because for an ordinary user Amazon imposes a default limit of 20 on the number of EC2 instances in a single region, and we also had to run other instances for storage and workload generators.

We started 8 to 64 second-layer VMs for running Cassandra. Each second-layer VM had 1GB memory and one vCPU. The key space was evenly distributed across the whole cluster, and the replication factor was set to 2. The workload generator was the same as what was used in Section 4.2.1: each region had a client generating read and write operations on random keys with a ratio of 4:1. The Cassandra cluster was initially in the Oregon region.

Figure 16 shows the network traffic and time taken to migrate the entire Cassandra cluster from the Oregon region to the Tokyo region while running the workload generator. Network traffic increases approximately linearly with the number of second-layer VMs because we need to copy at least 1GB memory for each second-layer VM. It is possible to reduce total migration time and traffic by de-duplicating the copied memory pages, which is left for future work.

"Degradation Time" shows the time during which the total throughput of all clients dropped by more than 50%. We can see that the degradation time increased with the number of VMs. This is because the workload generator we used issued synchronous requests one by one to random Cassandra nodes. Any time the cluster was split in two different locations, throughput of the workload was affected significantly. Figure 17 shows the stacked workload trace of the US and Asian clients when the cluster size was 64. The vertical dash lines indicates the time when the first and last VM were migrated. As we can see, the total throughput dropped until the whole cluster was moved. As part of future work, we want to develop an "ensemble live migration" mechanism where the foreground stop-and-copy phase of the migration is synchronized across VMs. Doing so should significantly reduce the time during which performance is degraded.

5 RELATED WORK

5.1 Multi-Cloud Deployment

Multi-cloud deployment is attractive to users because of its high availability and cost effectiveness. SafeStore (Kotla et al. 2007), DepSky (Bessani et al. 2011), HAIL (Bowers et al. 2009), RACS (Abu-Libdeh et al. 2010), SPANStore (Wu et al. 2013), Hybris (Dobre et al. 2014) and SCFS (Bessani

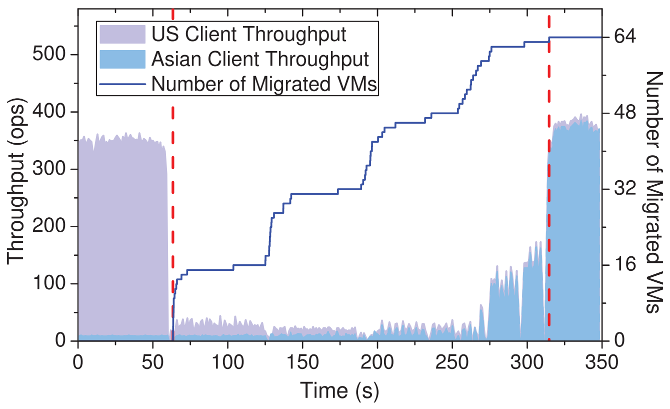


Fig. 17. Workload trace of migrating a 64-VM Cassandra cluster.

et al. 2014) have demonstrated benefits of multi-cloud storage, but they do not support computational resources. Docker (2016) deploys applications encapsulated in Linux containers (LXC) to multiple clouds. Although light-weight, LXC is not as flexible as a VM because all containers must share the same kernel, and it has poor migration support. Ravello (2016) leverages a nested hypervisor to provide an encapsulated environment for debugging and development distributed applications, but it does not addresses the challenges of providing storage and network support for wide-area application migration. Platforms like fos (Wentzlaff et al. 2010), Rightscale (2016), AppScale (2016) TCloud (Verissimo et al. 2012), IBM Altocumulus (Maximilien et al. 2009), and Conductor (Wieder et al. 2012) enable multi-cloud application deployment, but none of them provides the generality and flexibility of a multi-cloud IaaS.

5.2 Wide-Area VM Migration

VM live migration (Clark et al. 2005) has been widely used for resource consolidation and workload burst handling (Bryant et al. 2011; Hermenier et al. 2009; Nguyen et al. 2013). A traditional VM live migration only involves memory transfer and assumes that the disk image is shared. The pre-copy strategy (Clark et al. 2005) is the most widely used memory transfer technology. Post-copy (Hines and Gopalan 2009) has been proposed to eliminate duplicated transmission in pre-copy. However, VM live migration is not exposed to end users of public clouds.

Migrating a VM in the WAN faces long latency in accessing a shared disk image. To address this problem, Bradford et al. (2007) propose to iteratively copy the image in parallel with the memory. CloudNet (Wood et al. 2011) presents optimizations that minimize the cost of storage transfer and memory. Hirofuchi et al. (2009) combine on-demand fetching and a background copy after the memory is migrated. Mashtizadeh et al. (2011) describe various solutions used for live storage migration in VMware ESX. Zheng et al. (2011) optimize storage migration by leveraging temporal and spatial locality. Nicolae and Cappello (2012) propose a hybrid local storage transfer scheme for live migrating VMs with I/O intensive workloads—it combines pre-copy, post-copy, and prioritized prefetching based on access frequency. CloudSpider (Bose et al. 2011) combines VM image replication and scheduling—it replicates the VM image asynchronously in the background, but the image needs to be synced before migration is finished. Seagull (Guo et al. 2012) facilitates cloud bursting by determining which applications can be transferred into the cloud most economically—it uses opportunistic pre-copy to transfer an incremental snapshot of a VM's disk state. Our proactive approach attempts to only transfer data that is critical to performance.

VMFlockMS (Al-Kiswany et al. 2011) exploits similarity among VM images. VMFlockMS focuses on offline VM migration, requiring to shut down the VM first. FVD (Tang 2011) is a new VM image format that supports copy-on-write, copy-on-read, and adaptive prefetching that can be used for optimizing the performance of migrating a VM with the disk image. However, FVD can prefetch data only after the VM resumes on the destination. In contrast, our Supercloud storage proactively propagates data before migration is triggered.

5.3 Dynamic Resource Scaling

Resource scaling has been widely studied. Fine-grain resource scaling solutions adjust CPU, memory, and I/O resource allocation based on control theory (Kalyvianaki et al. 2009; Padala et al. 2009; Zhu et al. 2008), workload prediction (Chandra et al. 2003; Shen et al. 2011), or workload modeling (Doyle et al. 2003; Stewart et al. 2008). Coarse-grain capacity scaling schemes dynamically adjust the number of nodes in a distributed system (Lim et al. 2010). VM cloning (Lagar-Cavilla et al. 2009) and live migration are also widely used for resource consolidation or workload burst handling (Bryant et al. 2011; Hermenier et al. 2009; Nguyen et al. 2013). Techniques such as fine-grain CPU capping or VM live migration are typically not available to cloud end users. In addition, due to the complexity of different applications, it is impractical for cloud providers to provide services that can fit all requirements of any application. In contrast, users have much more knowledge about the application and its requirements. In the Supercloud users have control over their applications and have the flexibility and control in choosing resource scaling solutions. All of these techniques are compatible with the Supercloud and indeed are easier to support than in traditional clouds, which would require the cloud provider to expose such functionality.

5.4 Nested Virtualization

Nested virtualization was studied and theoretically analyzed in the 1970s (Goldberg 1973, 1974; Popek and Goldberg 1973). Belpaire and Hsu (1975a) created a formal model for recursive VMs that needs a centralized supervisor. Lauer and Wyeth (1973) removed this requirement and proposed to leverage nested virtual memories to create nested virtualization environment. Belpaire and Hsu (1975b) presented a formal model of hardware/software architecture that can be applied to recursive VM systems. The IBM z/VM hypervisor (Osisek et al. 1991) is the first practical implementation of nested virtualization, which relies on multi-level architectural support. These solutions typically require hardware mechanisms and corresponding software support that bear little resemblance to today's x86 architecture and OSes.

A traditional x86 architecture has only a single level of architectural support for virtualization. Ford et al. (1996) proposed to enable nested virtualization on x86 platforms by modifying the software stack at all levels based on a microkernel. Their goal is to enhance OS modularity flexibility and extensibility rather than virtualizing legacy OSes. As new hardware extensions are being added into x86 platforms such as Intel-VT (Intel 2016) and AMD-V (AMD 2016), people start searching for nested virtualization solutions that can utilize hardware primitives at different levels. The Turtles project (Ben-Yehuda et al. 2010) extended KVM with nested virtualization support on Intel processors. It also improves the I/O performance by enabling multi-level device assignment. Graf and Roedel (2009) implemented nested virtualization based on AMD processors in KVM. Recently nested virtualization supports have been proposed for the Xen hypervisor (He 2009). CloudVisor (Zhang et al. 2011) protects the privacy and integrity of customers' VMs on commodity virtualized infrastructures by introducing a tiny security monitor underneath the commodity VMM and exposing nested virtualization support for guest VMs. Although these nested virtualization solutions share a lot of motivations with Xen-Blanket, their major focus is how to support multi-level full virtualization with current hardware primitives. In contrast, Xen-Blanket seeks for a practical solution for nested para-virtualization, which has technical challenges fundamentally different from that of exposing hardware support to multiple levels.

Blue Pill (Garfinkel et al. 2007) is a root-kit emulating VMX to remain functional and avoid detection when a hypervisor is installed in the system. It is loaded during boot time by infecting the disk master boot record (MBR). Its nested virtualization support is minimal since it only needs to remain undetectable. In contrast, a comprehensive nested virtualization platform must efficiently multiplex the hardware across multiple levels of virtualization and manage all CPU, MMU, and I/O resources. Berghmans (2010) proposes another nested x86 virtualization platform where a software-only hypervisor is running on a hardware-assisted hypervisor. Different from this approach, Xen-Blanket does not assume any hardware virtualization support.

6 CONCLUSION

A Library Cloud is a cloud that can seamlessly span multiple cloud providers and support user-level resource management. The Supercloud is an instance of a Library Cloud and presents a complete cloud software stack under the user's full control that can seamlessly span multiple availability zones and cloud providers, including private clouds. It features live migration, as well as shared storage, virtual networking, and automated scheduling of workloads, placing and migrating VM resources as needed. Spanning availability zones and cloud providers, the Supercloud provides maximal flexibility for placement. Using our automated schedulers, we demonstrate continuous low latency for diurnal workloads that it is important for global cloud services to be able to follow the sun.

AVAILABILITY

The code and data of the Supercloud project are publicly available at http://supercloud.cs.cornell.edu.

REFERENCES

Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 229–240. DOI: http://dx.doi.org/10.1145/1807128.1807165

Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. 2011. VMFlock: Virtual machine co-migration for the cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC'11)*. ACM, New York, NY, 159–170. DOI: http://dx.doi.org/10.1145/1996130.1996153

Amazon. 2016. Auto Scaling. Retrieved August 18, 2017, from http://aws.amazon.com/autoscaling.

AMD. 2016. Virtualization. Retrieved August 18, 2017, from http://www.amd.com/en-us/solutions/servers/virtualization.

AppScale. 2016. Home Page. Retrieved August 18, 2017, from http://www.appscale.com.

Gerald Belpaire and Nai-Ting Hsu. 1975a. Formal properties of recursive virtual machine architectures. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*. ACM, New York, NY, 89–96. DOI: http://dx.doi.org/10.1145/800213.806526

Gerald Belpaire and Nai-Ting Hsu. 1975b. Hardware architecture for recursive virtual machines. In *Proceedings of the ACM Annual Conference (ACM'75)*. ACM, New York, NY, 14–18. DOI: http://dx.doi.org/10.1145/800181.810258

Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 1–6. http://dl.acm.org/citation.cfm?id=1924943.1924973

Olivier Berghmans. 2010. *Nesting Virtual Machines in Virtualization Test Frameworks*. Master's thesis. Department of Mathematics and Computer Science of the Faculty of Sciences, University of Antwerp.

Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2011. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*. ACM, New York, NY, 31–46. DOI: http://dx.doi.org/10.1145/1966445.1966449

Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. 2014. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14).* 169–180. https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani.

Sumit Kumar Bose, Scott Brock, Ronald Skeoch, and Shrisha Rao. 2011. CloudSpider: Combining replication with scheduling for optimizing live migration of virtual machines across wide area networks. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID'11).* IEEE, Los Alamitos, CA, 13–22. DOI:http://dx.doi.org/10.1109/CCGrid.2011.16

Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09).* ACM, New York, NY, 187–198. DOI:http://dx.doi.org/10.1145/1653662.1653686

Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. 2007. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07).* ACM, New York, NY, 169–179. DOI:http://dx.doi.org/10.1145/1254810.1254834

Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. 2011. Kaleidoscope: Cloud micro-elasticity via VM state coloring. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11).* ACM, New York, NY, 273–286. DOI:http://dx.doi.org/10.1145/1966445.1966471

Cassandra. 2016. Replacing a Dead Node. Retrieved August 18, 2017, from https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_replace_node_t.html.

Abhishek Chandra, Weibo Gong, and Prashant Shenoy. 2003. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 11th International Conference on Quality of Service (IWQoS'03).* 381–398. http://dl.acm.org/citation.cfm?id=1784037.1784065

Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-aware server provisioning and load dispatching for connection-intensive Internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08).* 337–350.

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05).* 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

DBENCH. 2016. DBENCH benchmark. Retrieved August 18, 2017, from https://dbench.samba.org/.

Dan Dobre, Paolo Viotti, and Marko Vukolić. 2014. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14).* ACM, New York, NY, Article 12, 14 pages. DOI:http://dx.doi.org/10.1145/2670979.2670991

Docker. 2016. Docker. Retrieved August 18, 2017, from https://www.docker.com.

Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. 2003. Model-based resource provisioning in a Web service utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03).* 5. http://dl.acm.org/citation.cfm?id=1251460.1251465

D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr.1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95).* ACM, New York, NY, 251–266. DOI:http://dx.doi.org/10.1145/224056.224076

Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. 1996. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96).* ACM, New York, NY, 137–151. DOI:http://dx.doi.org/10.1145/238721.238769

Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. 2010. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO'10).* ACM, New York, NY, Article 6, 6 pages. DOI:http://dx.doi.org/10.1145/1921151.1921160

Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS'07).* Article 6, 6 pages. http://dl.acm.org/citation.cfm?id=1361397.1361403

Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization (IISWC'07).* IEEE, Los Alamitos, CA, 171–180. DOI:http://dx.doi.org/10.1109/IISWC.2007.4362193

R. P. Goldberg. 1973. Architecture of virtual machines. In *Proceedings of the June 4–8, 1973, National Computer Conference and Exposition (AFIPS'73).* ACM, New York, NY, 309–318. DOI:http://dx.doi.org/10.1145/1499586.1499669

Robert P. Goldberg. 1974. Survey of virtual machine research. *Computer* 7, 6, 34–45. DOI:http://dx.doi.org/10.1109/MC.1974.6323581

Alexander Graf and Joerg Roedel. 2009. Nesting the virtualized world. In *Proceedings of the Linux Plumbers Conference.*

Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. 2012. Seagull: Intelligent cloud bursting for enterprise applications. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 33. http://dl.acm.org/citation.cfm?id=2342821.2342854

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 309–322. http://dl.acm.org/citation.cfm?id=1855741.1855763

Qing He. 2009. Nested virtualization on Xen. In *Proceedings of Xen Summit Asia.*

Jason Hennessey, Sahil Tikale, Ata Turk, Emine Ugur Kaynar, Chris Hill, Peter Desnoyers, and Orran Krieger. 2016. HIL: Designing an Exokernel for the data center. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. ACM, New York, NY, 155–168. DOI: http://dx.doi.org/10.1145/2987550.2987588

Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. 2009. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. ACM, New York, NY, 41–50. DOI: http://dx.doi.org/10.1145/1508293.1508300

Michael R. Hines and Kartik Gopalan. 2009. 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 (VEE'09)*. ACM, New York, NY, 51–60. DOI: http://dx.doi.org/10.1145/1508293.1508301

Takahiro Hirofuchi, Hidemoto Nakada, Hirotaka Ogawa, Satoshi Itoh, and Satoshi Sekiguchi. 2009. A live storage migration mechanism over WAN and its performance evaluation. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing (VTDC'09)*. ACM, New York, NY, 67–74. DOI: http://dx.doi.org/10.1145/1555336.1555348

Intel. 2016. Intel Virtualization Technology. Retrieved August 18, 2017, from http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2015. Supercloud: Opportunities and challenges. *SIGOPS Oper. Syst. Rev.* 49, 1, 137–141. DOI: http://dx.doi.org/10.1145/2723872.2723892

Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN'11)*. IEEE, Los Alamitos, CA, 245–256.

Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC'09)*. ACM, New York, NY, 117–126. DOI: http://dx.doi.org/10.1145/1555228.1555261

Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. 2009. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. ACM, New York, NY, 101–110. DOI: http://dx.doi.org/10.1145/1508293.1508308

Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'07)*. Article 10, 14 pages. http://dl.acm.org/citation.cfm?id=1364385.1364395

Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, NY, 1–12. DOI: http://dx.doi.org/10.1145/1519065.1519067

Butler W. Lampson. 1983. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'83)*. ACM, New York, NY, 33–48. DOI: http://dx.doi.org/10.1145/800217.806614

Hugh C. Lauer and David Wyeth. 1973. A recursive virtual machine architecture. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, New York, NY, 113–116. DOI: http://dx.doi.org/10.1145/800122.803951

Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. 2010. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*. ACM, New York, NY, 1–10. DOI: http://dx.doi.org/10.1145/1809049.1809051

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 461–472. DOI: http://dx.doi.org/10.1145/2451116.2451167

Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. 2011. The design and evolution of live storage migration in VMware ESX. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 14. http://dl.acm.org/citation.cfm?id=2002181.2002195

E. Michael Maximilien, Ajith Ranabahu, Roy Engehausen, and Laura Anderson. 2009. IBM Altocumulus: A cross-cloud middleware and platform. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*. ACM, New York, NY, 805–806. DOI: http://dx.doi.org/10.1145/1639950.1640024

Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. 2008. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, Vol. 2. 85–90.

Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 69–82. https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen.

Bogdan Nicolae and Franck Cappello. 2012. A hybrid local storage transfer scheme for live migration of I/O intensive workloads. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. ACM, New York, NY, 85–96. DOI: http://dx.doi.org/10.1145/2287076.2287088

OpenStack. 2016. Home Page. Retrieved August 18, 2017, from http://www.openstack.org/.

OpenVPN. 2016. OpenVPN. Retrieved August 18, 2017, from https://openvpn.net/.

Open vSwitch. 2016. Home Page. Retrieved August 18, 2017, from http://openvswitch.org

D. L. Osisek, K. M. Jackson, and P. H. Gum. 1991. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Syst. J.* 30, 1, 34–51. DOI: http://dx.doi.org/10.1147/sj.301.0034

Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, NY, 13–26. DOI: http://dx.doi.org/10.1145/1519065.1519068

Gerald J. Popek and Robert P. Goldberg. 1973. Formal requirements for virtualizable third generation architectures. In *Proceedings of the 4th ACM Symposium on Operating System Principles (SOSP'73)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/800009.808061

Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, 291–304. DOI: http://dx.doi.org/10.1145/1950365.1950399

Ravello. 2016. Home Page. Retrieved August 18, 2017, from http://www.ravellosystems.com/.

RightScale. 2016. Home Page. Retrieved August 18, 2017, from http://www.rightscale.com.

J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst.* 2, 4, 277–288. DOI: http://dx.doi.org/10.1145/357401.357402

Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, Article 16, 15 pages. DOI: http://dx.doi.org/10.1145/2741948.2741953

Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*. ACM, New York, NY, Article 5, 14 pages. DOI: http://dx.doi.org/10.1145/2038916.2038921

Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. 2012. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Technical Conference*. 1. http://dl.acm.org/citation.cfm?id=2342821.2342860

Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. 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 (EuroSys'07)*. ACM, New York, NY, 275–287. DOI: http://dx.doi.org/10.1145/1272996.1273025

Weijia Song, Zhen Xiao, Qi Chen, and Haipeng Luo. 2014. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Trans. Comput.* 63, 11, 2647–2660. DOI: http://dx.doi.org/10.1109/TC.2013.148

Christopher Stewart, Terence Kelly, Alex Zhang, and Kai Shen. 2008. A dollar from 15 cents: Cross-platform management for Internet services. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*. 199–212e3. http://dl.acm.org/citation.cfm?id=1404014.1404029

Chunqiang Tang. 2011. FVD: A high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*. 1. http://dl.acm.org/citation.cfm?id=2002181.2002199

Tinc. 2016. Home Page. http://www.tinc-vpn.org/.

P. Verissimo, A. Bessani, and M. Pasin. 2012. The TClouds architecture: Open and resilient cloud-of-clouds computing. In *Proceedings of the IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W'12)*. DOI: http://dx.doi.org/10.1109/DSNW.2012.6264686

Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*. ACM, New York, NY, 181–194. DOI: http://dx.doi.org/10.1145/1060289.1060307

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 3–14. DOI: http://dx.doi.org/10.1145/1807128.1807132

Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2012. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 27. http://dl.acm.org/citation.cfm?id=2228298.2228335

Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. 113–126.

T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. 2011. CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*. ACM, New York, NY, 121–132. DOI: http://dx.doi.org/10.1145/1952682.1952699

Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. 2007. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*. 17. http://dl.acm.org/citation.cfm?id=1973430.1973447

Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 292–308. DOI: http://dx.doi.org/10.1145/2517349.2522730

XenServer. 2016. Home Page. Retrieved August 18, 2017, from http://www.xenserver.org/.

Xenserver-core. 2016. Xenserver-core. Retrieved August 18, 2017, from https://github.com/xenserver/buildroot.

Zhen Xiao, Weijia Song, and Qi Chen. 2013. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Trans. Parallel Distrib. Syst.* 24, 6, 1107–1117. DOI: http://dx.doi.org/10.1109/TPDS.2012.283

Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 203–216. DOI: http://dx.doi.org/10.1145/2043556.2043576

Jie Zheng, Tze Sing Eugene Ng, and Kunwadee Sripanidkulchai. 2011. Workload-aware live storage migration for clouds. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*. ACM, New York, NY, 133–144. DOI: http://dx.doi.org/10.1145/1952682.1952700

Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 2008. 1000 Islands: Integrated capacity and workload management for the next generation data center. In *Proceedings of the 2008 International Conference on Autonomic Computing (ICAC'08)*. IEEE, Los Alamitos, CA, 172–181. DOI: http://dx.doi.org/10.1109/ICAC.2008.32

Received November 2016; revised May 2017; accepted August 2017