Enhancing Application Robustness in Cloud Data Centers

Madalin Mihailescu, Andres Rodriguez, Cristiana Amza
University of Toronto

Dmitrijs Palcikovs, Gabriel Iszlai, Andrew Trossman, Joanna Ng
IBM Center of Advanced Studies

Abstract

We propose OX, a runtime system that uses application-level availability constraints and application topologies discovered on the fly to enhance resilience to infrastructure anomalies for cloud applications. OX allows application owners to specify groups of highly available virtual machines, following component roles and replication semantics. To discover application topologies, OX monitors network traffic among virtual machines, transparently. Based on this information, OX builds on-line topology graphs for applications and incrementally partitions these graphs across the infrastructure to enforce availability constraints and optimize communication between virtual machines. We evaluate OX in a realistic cloud setting using a mix of Hadoop and YCSB/Cassandra workloads. We show how OX increases application robustness, by protecting applications from network interference effects and rack-level failures.

1 Introduction

Large-scale virtualized data centers hosting many concurrent applications form the foundation for cloud computing. Current Infrastructure-as-a-Service (IaaS) clouds, such as Amazon EC2 [1], rely on data centers holding tens to hundreds of thousands of servers [19]. Servers are virtualized using Xen [2] or VMware [3], and are shared by multiple virtual machines (VMs); VMs host components of distributed applications deployed by cloud tenants.

As data centers grow in size and complexity, hosted applications become increasingly vulnerable to dynamically occurring cloud infrastructure anomalies. Anomalies within large data centers can be diverse, ranging from hardware component failures to hardware bottlenecks; these anomalies are hard to predict and diagnose and can impact application uptime, functionality and/or performance severely.

In particular, the network communication fabric in large-scale data centers is notoriously hard to scale and manage [19]. Current data center networks are built hierarchically, in a multi-tier architecture, with servers organized into racks, and each server connected to a rack-level switch. The servers in the same rack can typically communicate at the full rate of their interfaces, e.g., 1 Gbps; intra-rack CPU, memory and disk resources are also normally plentiful. On the other hand, inter-rack links are usually oversubscribed, creating potential performance bottlenecks. Oversubscription factors of 10 on inter-rack links are common, i.e., 4 Gbps of uplink for 40 servers in a rack. This is especially problematic with network intensive MapReduce [16] workloads; such workloads require high network performance from the cloud infrastructure. They also possibly compete with co-hosted applications with stricter QoS guarantees, such as, Web applications, creating network interference for these applications. Finally, inter-rack network paths involve one or more switches, hence are higher latency, and more vulnerable to failures than within-rack network links.

Higher density interconnect architectures [19][12], while claiming high bandwidth connectivity between any two servers, increase infrastructure and management costs significantly. As a result, in current data centers, applications are especially vulnerable to inter-rack network interference, and inter-rack network link failures, and can benefit from VM placement within a rack. This type of placement not only makes the application robust to any type of inter-rack network anomalies; it also reduces application latency due to shorter network paths between components. This placement is not always possible, however, mainly due to high availability constraints of applications. Specifically, while robust to inter-rack anomalies, an application may now become too vulnerable to intra-rack failures. Statistics from Google [15] report approximately 20 rack unit failures (e.g., 40-80 servers disconnecting, with 1-6 hours to recover) during a one year period, for a data center size in the 1000s of servers. In general, racks are considered a significant failure domain to avoid [18] when deploying highly available distributed applications. The degree of application disruption depends on the application's fault tolerance, reconfiguration capabilities, and redundancy of VM components affected by, or inaccessible due to, the respective failures.

For instance, application-level VM replication techniques would be ineffective in case of any rack-type failure e.g., server, rack switch, or rack power unit failures, unless VM replica placement is on different racks. Moreover, the desirable placement configuration for VM replicas may be different for different replication semantics. For instance, simply placing the two VMs of a replicated primary-backup database tier on two different racks shields the application from server and rack-level failures. On the other hand, other replication schemes, such as, chained declustering, quorum based, or other combinations between data partitioning and replication would potentially have different goals, and resulting VM placement requirements. Therefore, i) VM placement constraints are usually dependent on application semantics, and ii) these constraints need to be known and respected by the VM placement algorithm.

In this context, we introduce OX[1], a novel semantic-aware VM management framework that optimizes VM mapping to racks, incrementally and dynamically. OX leverages application-level semantic knowledge, i.e., *availability constraints* for redundant VMs, and VM *communication patterns*, to optimize placement and enhance resilience to cloud infrastructure anomalies for hosted applications. OX allows tenants to specify high-level groups of highly available VMs *without* the need to expose the underlying physical infrastructure architecture to applications. To discover each application's topology, OX monitors network traffic among its hosted VMs, transparently. Based on each application's availability requirements and its traffic matrix, OX builds per-application topology graphs. OX then applies an *incremental* graph partitioning algorithm to optimize placement decisions for each individual application, and migrates VMs accordingly across server racks. OX's partitioning algorithm proceeds from current application placement configurations and gradually moves towards the final placement. This incremental adaptation uses a minimum number of VM migration operations to optimize application placement. Assuming sufficient resources exist, OX evolves towards a stable per-application VM to rack placement that reduces application traffic on inter-rack links, while respecting application high availability constraints.

Our placement optimization algorithm is simple, intuitive, and lightweight. OX focuses on optimization opportunities in a per-application, incremental fashion. This localized per-application, per-link, fine-tuning focus avoids the pitfalls of global-reshuffle optimizations. As long as per-application semantics and patterns are relatively stable, OX optimizes the number of VM migrations in-flight while converging over time to stable application configurations. Per-application robustness to both inter-rack network bottlenecks and intra-rack failures is the natural result.

We implemented a prototype of OX as a set of extensions to an in-house Infrastructure-as-

[1]As in "strong as an ox"

a-Service (IaaS) cloud management framework. To evaluate the benefits, we construct realistic scenarios with cloud tenants instantiating and configuring VMs as part of various modern distributed applications, e.g., Hadoop, Cassandra. We show how OX optimizes VM placement and shields applications from *inter-rack network interference effects* and *rack-level failures.*

2 OX Overview

OX enhances robustness for distributed applications hosted on Infrastructure-as-a-Service cloud offerings, e.g., Amazon EC2, through availability and network communication optimizations. OX implements an algorithm for semantic-aware mapping of application components (VMs) onto the virtualized infrastructure.

Tenants of the cloud infrastructure create VM instances and configure them into application-level clusters, e.g., Apache, Cassandra, MySQL, with various availability requirements. After performing an initial VM mapping, OX incrementally optimizes the mapping to conform to tenant-provided availability constraints and discovered application topologies. OX adapts the VM mappings to *dynamic changes* in availability constraints, failed VMs, VMs dynamically provisioned to absorb application load bursts, and changes in VM network traffic patterns, over time. Finally, OX proactively reduces network utilization on inter-rack links, whenever the opportunity arises. In this way, OX provides application flexibility in case of pattern changes, as well as protection against *unpredictable* infrastructure anomalies.

The overall goals are to (1) increase application availability and (2) improve application resilience to anomalies in the network communication fabric, e.g, network interference. To address the former, redundant VMs are placed on separate server racks, according to tenant-defined availability constraints. For the latter, any two VMs that communicate frequently are hosted on the same rack, as long as there is no availability constraint between them.

Towards this, OX models placement optimization as an *incremental* graph partitioning problem. Each distributed application is represented as a graph. Graph vertices are VM instances composing each application, while edges are defined based on availability constraints or communication patterns between VMs. The OX optimization component runs periodically and selects VMs for partition re-assignment as follows. Each VM, which by its current placement either *violates* an availability constraint, or is involved in more inter-partition traffic than intra-partition traffic, is selected. The OX algorithm then *re-assigns* the selected VMs to new partitions, in a greedy fashion, based on the latest traffic and availability constraint graphs. For relatively stable application traffic patterns and constraints only a small subset of application VMs require re-assignment.

Once a new *logical* application partition configuration is determined, OX uses a partition-to-rack assignment algorithm to decide *partition placement* onto racks. This assignment algorithm aims to *minimize* VM migration costs while also considering available *rack resources.* Finally, OX completes the assignment by *live migrating* VMs across racks. Migrations are performed in parallel as long as they are disjoint, e.g., they do not incur network traffic across the same rack switch. The end result is segregation on different server racks of VMs part of a high availability group, while VMs that might be impacted by anomalies in the network communication fabric, such as, network interference, are colocated on the same rack.

In the following, we describe the components of OX and the interface it offers for accessing cloud resources.

3 Accessing Cloud Resources

3.1 Basic Interface

IaaS cloud providers facilitate tenant access to cloud resources through a basic command interface. In Table 1, we show a restricted subset of commands, similar to Amazon EC2's API [1]. From a tenant's viewpoint, there are three main building blocks: root filesystem images (*images*), *VM instances*, and block devices (*volumes*).

The cloud typically provides a repository of pre-installed *images* for various applications, such as Hadoop [4], Apache, MySQL, etc. Tenants can also upload images for any operating system and with any pre-installed software packages. Using images in the cloud image repository, tenants instantiate *VMs* as components of tenant applications hosted on the cloud infrastructure. VMs can be created with various resource quotas, e.g., CPU/RAM; tenants are usually billed based on the VM resources requested. Configuration of VM instances as part of distributed applications is done after boot, based on dynamic IP addresses and information about other related VMs. *Volumes* are used to store data on behalf of applications. For example, an instance running a DBMS can use a volume to store the actual database. In our cloud system we offer two types of volumes: *local* and *remote*. Local volumes are created on the local server disk drives where the VM is hosted; these volumes are tied to the respective server location for the lifetime of the VM. Remote volumes reside on a network storage system and are typically more reliable than local volumes.

3.2 OX Semantic Interface

We enrich the basic interface with a collection of commands to allow tenants to make application semantics explicit, as well as to inquire about and use the logical VM locations assigned by OX. A description of the proposed semantic extensions is given in Table 1.

Tenants can employ high availability groups to define availability constraints among application VMs. Availability constraints should be used to separate *redundant* VMs – that is, VMs that implement a *common function*. To create high availability groups, we introduce a *create-hagroup* command. Application VMs can be placed into a high availability group in two ways: (1) by specifying the *hagroup-id* when calling *run-instances*, or (2) by using the *addto-hagroup* command. Tenants can remove VMs from a high availability group through *rmfrom-hagroup*.

We also introduce the notion of a tenant-defined *segregation degree* per high availability group. The default semantic-aware placement policy is to host each VM from a high availability group on a distinct rack, therefore, achieving full segregation. This default policy corresponds to a segregation degree of 1. Tenants, however, can specify lower values for the segregation degree in order to create opportunities for OX to strengthen application resilience to other infrastructure anomalies, e.g., network interference effects. For instance, by defining a segregation degree of 0.5, tenants instruct OX to split the VMs across two racks, only.

Some applications replicate functionality across components by taking VM location into account – see Section 3.3. Through the command *get-location*, tenants can query OX for *logical* VM locations. Tenants can thus discover how VMs are separated on the cloud infrastructure and incorporate this information into the application. VM location information matches the application's logical partitioning across the infrastructure, with no reference to the physical infrastructure architecture.

3.3 Application Semantics

Cloud tenants provision VMs with various functions, ranging from components in web applications – e.g., LAMP stacks, to elements in platforms for data analytics, such as Hadoop.

Web applications and *data analytics platforms* normally consist of several tiers, each tier composed of multiple VMs. For instance, a web application can include a load balancer tier – HAProxy, a web server tier – Apache, a caching tier – Memcached and a data tier – a relational database system such as MySQL, or a noSQL system, e.g., Cassandra [20]. Similarly, data analytics applications, such as Hadoop, rely on a Map-Reduce [16] computational framework to administer job execution and a distributed filesystem, e.g., HDFS [5], for storing data.

By and large, availability constraints for *stateful tiers* – e.g., application server tier, data tier, follow the underlying data replication semantics. Common approaches for replicating data fall into one of the following categories: mirroring, i.e., primary/backup, chained declustering as employed by distributed hash table (DHT)-based systems, e.g., Cassandra, or random replica placement, like

Basic Interface

describe-images	returns list of images available in the repository
run-instance -i *image-id* -s *size*	instantiate VM from image *image-id*; *size* defines resources, i.e., CPU/RAM
create-volume -s *size* -t *type* [-i *instance-id* -d *device*]	create volume of *size* and *type*; *type* is local or remote; *device* is local VM device, e.g., */dev/sdb*; returns new *volume-id*

Semantic Interface

create-hagroup [-s *seg-degree*]	create a high availability group; *seg-degree* defines group segregation – default is 1 (full segregation)
run-instance -i *image-id* -s *size* [-g *hagroup-id*]	instantiate VM from *image-id*; *hagroup-id* enables tenants to specify availability constraints at VM creation time
addto-hagroup -i *instance-id* -g *hagroup-id*	add VM *instance-id* to high availability group *hagroup-id*
rmfrom-hagroup -i *instance-id* -g *hagroup-id*	remove VM *instance-id* from high availability group *hagroup-id*
get-location -i *instance-id*	get *logical* location for VM *instance-id*

Table 1: **Cloud Tenant Interface.** We augment the basic cloud interface with commands for making application availability constraints explicit.

in HDFS.

Cloud tenants can leverage the semantic interface to make replication semantics explicit; conversely, cloud tenants can instruct OX to segregate VMs and then configure the high-level replication mechanism according to the segregation information provided by OX.

For example, mirroring can be used to implement large-scale data tiers by partitioning data and deploying a primary/backup pair of VMs for each partition, for availability purposes. In this case, the application can define a high availability group for each primary/backup partition. Chained declustering can be viewed as a generic case of mirroring, with primaries and backups chained together. Finally, with randomized schemes, data replicas are usually spread randomly across VMs, at a finer data granularity, based on available capacity. Due to the higher vulnerability to failures for data tiers relying on random placement, tenants can conservatively place all tier VMs into a single high availability group with a segregation degree of 1, to achieve full segregation. Alternatively, these schemes may inquire about and incorporate VM location information when storing data replicas.

In *stateless tiers* – e.g., load balancer, web server, caching tiers, the functions implemented by individual VM components are typically interchangeable. Therefore, the failure of one VM in the tier has a reduced overall effect on application performance and availability, as load is easily rebalanced across the remaining components. Furthermore, stateless VMs are easier to recover and reintegrate into the application. Tenants can either create high availability groups for a subset of VMs in a stateless tier, or create whole-tier availability groups with lower segregation degrees. Tenants can thus ensure overall tier availability when facing unpredictable hardware failures.

Cloud tenants can define/update availability constraints for owned VMs at any time. All availability constraints specified by tenants are considered by OX when optimizing placement for hosted applications.

4 Optimizing Placement

4.1 Initial VM Placement

At the infrastructure level, the placement of new VMs complies with availability constraints specified by tenants at VM instantiation time. In addition, VMs are initially placed in the proximity of other VMs of the same application.

OX, then, periodically reorganizes application VMs across server racks according to application dynamic changes, to satisfy application availability requirements and reduce application exposure to anomalies in the inter-rack communication fabric, such as network interference effects.

4.2 Applications as Graphs

Availability Constraint Graphs: OX uses tenant-defined availability constraints to construct application constraint undirected graphs. Graph vertices are application VM instances and edges are defined based on high availability group specifications. Namely, for a high availability group with a segregation degree of 1, an edge exists between any two VM instances in the availability group. For lower segregation degrees, OX splits clusters of VMs across racks, according to the degree; an edge in the constraint graph exists between any two VMs in different clusters.

In the following, we use $CG = (V, CE)$ to represent the constraint graph, where VM_i is vertex V_i and an availability constraint between VM_i and VM_j is defined by edge $CE_{ij} = (V_i, V_j)$.

Communication Graphs: OX discovers application topologies at the infrastructure level by monitoring virtual machine network traffic. We employ a number of open source tools to gather and analyze network traffic statistics in real-time, using the NetFlow interface (described in Section 5).

Based on the latest statistics, a traffic analyzer daemon generates application topologies as undirected traffic graphs. For each graph, vertices are VM instances composing the application. An edge exists between two vertices if the two corresponding VMs exchanged traffic during the last period. Edges are labeled with the current network traffic bandwidth.

In the following, $TG = (V, TE)$ denotes the traffic graph, where VM_i is vertex V_i and edge $TE_{ij} = (V_i, V_j)$ is shared by two communicating VMs.

4.3 OX Partitioning Algorithm

OX models application placement optimization as an incremental graph partitioning problem based on the current partition (placement) configuration, P, constraint graph, CG, and the traffic graph, TG. To solve the problem, OX employs an approximation algorithm structured into two phases. In **phase 1**, OX uses P, CG, and TG, to verify, for every VM, whether there are any violated availability constraints or better placement opportunities in terms of network traffic. VMs that require placement optimizations are logically de-partitioned from P. **Phase 2** improves partition composition by re-integrating the de-partitioned VMs, in a greedy breadth first search fashion, according to TG and CG. The output is the new partition configuration.

De-partitioning VMs: Phase 1 proceeds as follows. For each application VM, V_i, let $P_a \in P$, such that $V_i \in P_a$ – that is, V_i is currently assigned to partition P_a. Availability constraints are violated when VMs in the same partition are part of the same high availability group. Therefore, if there exists an edge in CG from V_i to another VM, V_j, with $V_j \in P_a$, V_i and V_j are marked for removal from P_a.

Changes in communication patterns among application VMs can also give rise to opportunities for improved placement. To discern these opportunities, OX computes a per-VM *non-constrained inter/intra-partition traffic ratio*, for every VM not yet marked for removal. For $V_i \in P_a$, *intra-partition traffic* is traffic within a rack between V_i and the other VMs in P_a. Intra-partition traffic is derived by summing up labels (network bandwidth values) of edges (V_i, V_j) in TG, where $V_j \in P_a$. *Inter-partition traffic* represents *non-constrained* traffic between V_i and VMs from partitions other than P_a, i.e., other racks. This traffic aggregate is derived for every partition $P_b \in P$, P_b != P_a, by summing labels of edges (V_i, V_j) in TG, where $V_j \in P_b$ and no edge (V_i, V_j) exists in CG.

OX uses inter/intra-partition traffic ratios when deciding to de-partition VMs from P. Specifically, for $V_i \in P_a$, if there exists a $P_b \in P$, P_b != P_a, such that the corresponding V_i inter-P_b/intra-P_a traffic ratio is greater than one, V_i is marked for removal. A ratio greater than one indicates that V_i exchanges more expensive inter-rack traffic with VMs in P_b than intra-rack traffic with VMs in P_a.

All VMs marked for removal due to either violated availability constraints, or changes in network traffic patterns, are temporarily *de-partitioned* from P. In the following, we use D to denote this set of de-partitioned VMs.

Re-partitioning VMs: New logical partition composition is constructed by re-integrating the VMs from D, in a greedy breadth first search fashion, based on TG and CG. Re-integration is performed iteratively, by growing partitions with de-partitioned VMs in decreasing order of network traffic magnitude. Moreover, a partition grows with VMs that have no availability constraints with current partition nodes.

The *re-partitioning phase* progresses iteratively, as follows. First, for the VMs that are still in P, OX aggregates edges to VMs in D. That is, given P_a in P and V_i in D, OX replaces all traffic edges from V_i to VMs in P_a with a single edge; this edge is labeled with the sum of all respective edge labels. Edges with at least one end (VM) in D are then sorted in decreasing order of labels (bandwidth) and added to a queue, Q. At each iteration, the algorithm considers the highest edge in Q, $TE_{max} = (V_i, V_j)$, for expansion. If V_i and V_j are both in D, they are assigned to a new partition P_b. Otherwise, if V_i (or V_j) represents a partition P_a, V_j (or V_i) is assigned to P_a, if the respective VM has no availability constraints with VMs in P_a. Lastly, if both V_i and V_j represent intermediate partitions P_a and P_b, respectively, the two partitions are merged only if there are no availability constraints between them. Upon re-integration into a partition, a VM is removed from D. Analyzed edges are removed from Q, while labels for remaining edges in Q are updated accordingly.

The iterative loop finishes when Q is empty or when the algorithm was not able to grow a partition in the current iteration, due to availability constraints. The set of VMs not assigned to a partition along with the corresponding traffic/constraint subgraphs are used as inputs to a new instance of the re-partitioning algorithm.

4.4 Partition Placement

Based on new logical partition configurations, OX reshuffles application VMs across racks such that partitions are segregated and all VMs in a partition are hosted on the same rack. Partitions are first assigned to racks based on minimal VM migration costs and available rack resources, e.g., memory. OX then finalizes the reshuffling process by migrating VMs according to the partition-to-rack assignment.

Assigning Partitions to Racks: In order to respect availability constraints or alleviate inter-rack network traffic, each VM partition has to be allocated to a different rack. OX assigns partitions to server racks such that VM migration costs are minimized. The partition assignment algorithm also considers available rack memory, as it is by far the scarcest resource in most environments.

Migrating Virtual Machines: The assignment algorithm outputs a mapping of partitions to racks and a set of VM instance migrations per application. OX optimizes the migration process for all applications, by executing migrations in parallel, while reducing network congestion caused by reshuffles. To parallelize VM migrations, while not generating network interference, OX employs a simple heuristic policy: no two concurrent migrations are allowed to/from the same rack.

5 Prototype

We implemented a prototype of OX as a set of extensions to an in-house Infrastructure-as-a-Service (IaaS) cloud management framework. While we rely on our own IaaS cloud implementation, the concepts presented in this paper can be easily integrated into existing IaaS cloud solutions, e.g., Eucalyptus [6].

5.1 Cloud Management Software

Our cloud management software allows cloud tenants to get access to infrastructure resources, by instantiating VMs, creating storage volumes, and defining VM availability constraints. Our management software consists of: (1) agents running on each virtualized server, handling requests, (2) a storage server maintaining cloud metadata and exporting root filesystem images and remote storage volumes, and (3) a DHCP server assigning IP addresses dynamically to VMs.

We select one of the server agents to act as global cloud manager. Cloud tenant requests are first submitted to the cloud manager, who forwards them to server agents or the storage server, accordingly. The storage server maintains the cloud metadata, e.g., information about VM instances, images, volumes, high availability groups, in a local database. The storage server also runs an NFS server, to export images to NFS clients running on virtualized servers. To multiplex a single root filesystem image to multiple VM instances, we use the QCOW2 format [7] to create Copy-on-Write VM images from a base image. Finally, to provide remote storage volumes to VMs, the storage server uses the NBD implementation of the Network Block Device protocol [8].

To exemplify the management software workflow, suppose a tenant first creates a high availability group, *ha-xyz*, by issuing the *create-hagroup* command – Table 1. The request is sent to the cloud manager who forwards it to the storage server. The storage server then creates an entry in the local database for *ha-xyz*. Next, the tenant starts two VMs from image *img-abc* as part of *ha-xyz*, by calling *run-instance*. The cloud manager uses the availability constraints, location of other VMs, and available resources, to decide the initial location for the two VMs. Based on the placement decision, the cloud manager forwards requests to the server agents. Server agents create COW images based on *img-abc*, instantiate the VMs and reply back with the unique VM instance ids. The cloud manager then sends instance metadata, *(instance-id, location, ha-groups)*, to the storage server, to be added to the local database, and replies to the tenant.

5.2 Discovering Communication Patterns

OX discovers application topologies at the infrastructure level by monitoring virtual machine network traffic. We use a number of open source tools to gather and analyze network traffic statistics in real-time. Each virtualized server in the data center runs *Open vSwitch* [9]. Open vSwitch is a virtual switch implementation that supports standard management interfaces (e.g. NetFlow), and is open to programmatic extension and control. We configure each virtual switch to export flows to a central collector, periodically, using the NetFlow interface.

The collector component is deployed in a virtual machine on the cloud infrastructure and is based on the *flow-tools* package [10]. Flow-tools provides a set of programs used to collect, send, process, and generate reports from NetFlow data. In OX, a *flow-capture* daemon collects flows from the virtual switches and writes them to disk, periodically, at a one minute interval. A second traffic analyzer process runs every minute in the background and consists of three phases. First, the analyzer daemon uses *flow-nfilter* to filter VM traffic. Second, it calls *flow-report* and *flow-print* to generate statistics based on the latest collected data. For each flow, OX extracts source ip address, destination ip address, flow start time, flow end time and bytes transferred. Based on the latest statistics, the traffic analyzer daemon generates application topologies as undirected traffic graphs and forwards this information to the optimization component.

5.3 OX Optimization Component

The optimization component process is also deployed in a virtual machine on the cloud infrastructure. It receives up-to-date per-application traffic graphs from the traffic analyzer daemon and it fetches availability constraints, along with VM location information, from the storage server. The optimization process then executes the graph re-partitioning algorithm and, following the new partition configuration, the partition-to-rack assignment algorithm. Lastly, to complete the new placement, it issues *migrate-instance* commands to the cloud manager, according to the migration schedule. The *migrate-instance* admin command takes as arguments the identifier of the VM to be migrated and the new rack hosting the VM.

6 Evaluation

In this section, we evaluate the benefits OX provides to applications running on the cloud infrastructure. In the evaluation, we construct realistic small-scale scenarios with cloud tenants instantiating and configuring VMs as part of various popular distributed applications, e.g., Hadoop, Cassandra. Tenants specify availability constraints both at VM instantiation time and VM runtime. We show how OX shields applications from *inter-rack network interference effects* and *rack-level failures*, through smart VM placement/migration operations.

In the following, we first describe the cloud infrastructure and benchmarks. Then, we present the experimental results with a mix of tenant applications running on the cloud infrastructure.

6.1 Testbed

Cloud Infrastructure: Our evaluation infrastructure consists of three racks, each composed of ten servers and a rack switch. Servers are connected to the rack switches through 1Gbps network links. Thus, rack switches deliver 1Gbps network bandwidth for intra-rack server communication. We use a third switch, the cluster switch, to interconnect the rack switches. The rack-to-cluster switch links are 1Gbps. Therefore, the rack-to-cluster links are *oversubscribed* by a factor of 10. The physical servers are Dell PowerEdge SC1450 and we deployed Xen 3.4.3 as virtualization technology. The switches are Extreme Networks Summit 400-48t.

Benchmarks: *HADOOP* – We use Apache's Hadoop 0.21.0 distribution based on the HDFS distributed filesystem and MapReduce framework [4]. HDFS is composed of a NameNode, maintaining metadata, and DataNodes, storing the data. The MapReduce framework enables distributed processing of large data sets on compute clusters. It consists of a JobTracker and multiple TaskTrackers. Processing jobs are submitted to the JobTracker who creates map/reduce tasks for each job and assigns them to TaskTrackers. Map tasks process data stored in HDFS. A data shuffling phase feeds outputs from map tasks as inputs to reduce tasks. Reduce tasks generate the final output and store it in HDFS. As Hadoop workload, we use the *sort benchmark* [11]. In our experiments, this benchmark takes as input a 10GB random data set stored in HDFS and produces the sorted data using the MapReduce framework.

We create a root filesystem image, *img-hadoop*, based on the Hadoop software for tenants to instantiate VMs running the Hadoop system. In the following, each Hadoop VM is configured 3GB of RAM and 3 vCPUs. HDFS data is stored on local storage volumes of size 20GB. The local volumes are created by the cloud management software on the servers hosting the Hadoop VMs. In general, all Hadoop VMs act as HDFS DataNodes / Map-Reduce TaskTrackers, with one VM also acting as HDFS NameNode and one as Map-Reduce JobTracker.

YCSB-Cassandra – YCSB [14] is a framework for evaluating different cloud database systems. A cloud database is typically a scalable key-value store, such as Amazon's Dynamo [17] or Cassandra [20]. YCSB targets serving systems, which provide online read/write access to data. That is, a web user is waiting for a web page to load, and reads and writes to the cloud database are carried out as part of the page construction and delivery. YCSB consists of a workload generating client and a package of standard workloads with various characteristics, e.g., read-intensive, write-intensive, to be executed by the client. The framework also comes with database connectors for different cloud database systems. To evaluate OX, we use Cassandra as cloud database and we configure YCSB clients to run read-intensive workloads against the Cassandra system.

We create two root filesystem images, *img-ycsb* and *img-cassandra*, based on the YCSB 0.1.2 and Cassandra 0.6.4 releases, respectively. Tenants instantiate YCSB VMs from *img-ycsb* and Cassandra VMs from *img-cassandra*. In the following scenarios, each YCSB VM is configured with 512MB of RAM and 2 virtual CPUs, while each Cassandra VM has 1GB of

RAM and 2 vCPUs. Application data is stored on remote storage volumes of 1GB in size, one per Cassandra VM.

6.2 Network Interference Scenario

We first construct a realistic scenario to illustrate how OX protects applications from network congestion on oversubscribed inter-rack links. In this scenario, we use *two racks* to host three tenant applications.

Tenant Applications: $Tenant_1$ deploys a *YCSB-Cassandra* distributed application, app_1, composed of two YCSB VMs, $a_1{:}y_1$ and $a_1{:}y_2$, and two Cassandra VMs, $a_1{:}c_1$ and $a_1{:}c_2$. The Cassandra tier employs 2-way replication, and the tenant requires the tier to be highly available. He creates a high availability group with segregation degree 1, $ha\text{-}c_1$, and starts $a_1{:}c_1$ and $a_1{:}c_2$, as members of $ha\text{-}c_1$. $a_1{:}y_1$ and $a_1{:}y_2$ are instantiated with no availability constraints. Later on, while app_1 is running, $tenant_1$ decides to create a high availability group for the YCSB tier, $ha\text{-}y_1$, to increase tier availability. He then adds $a_1{:}y_1$ and $a_1{:}y_2$ to $ha\text{-}y_1$. This is an example of a dynamic change in availability constraints.

$Tenant_2$ also deploys a *YCSB-Cassandra* application, app_2, composed of two YCSB VMs and two Cassandra VMs. He creates two high availability groups with the default segregation degree of 1, $ha\text{-}y_2$ for YCSB and $ha\text{-}c_2$ for Cassandra. The tenant then instantiates the YCSB VMs, $a_2{:}y_1$ and $a_2{:}y_2$, as part of $ha\text{-}y_2$, and the Cassandra VMs, $a_2{:}c_1$ and $a_2{:}c_2$, as members of $ha\text{-}c_2$.

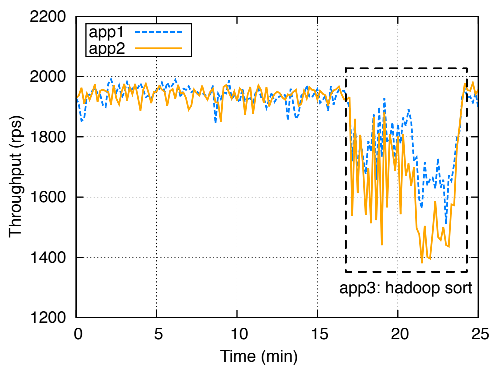
$Tenant_3$ starts a *Hadoop* application, app_3, composed of ten Hadoop VMs, $a_3{:}h_1$ ... $a_3{:}h_{10}$. $Tenant_3$ wants to configure HDFS with a replication degree of 3 – that is, each data block is replicated on 3 Hadoop VMs. Moreover, the tenant requires at least 2 replicas to be separate. With OX, he creates a high availability group, $ha\text{-}h_1$, with segregation degree 0.5. Then, he instantiates the ten Hadoop VMs as part of $ha\text{-}h_1$. At the infrastructure level, the above availability requirements ensure that the Hadoop VMs are balanced across the two racks. A common replica placement policy employed by HDFS, rack-aware, makes use of VM location information. Hence, the tenant queries OX for logical VM locations and uses this information when configuring HDFS.

For the *network interference scenario*, we are interested in evaluating application performance, when the three applications run concurrently on the cloud infrastructure. For app_1 and app_2, YCSB clients constantly issue requests to the Cassandra system. For app_3, a sort job on the 10GB data set is infrequently submitted to the MapReduce system. We evaluate OX against a random semantic-aware VM placement approach.

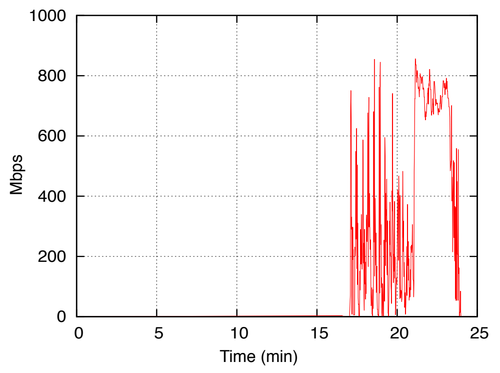
Random Semantic-Aware VM Placement: The cloud management software places the VMs such that availability constraints are satisfied and resource usage is reasonably balanced.

For app_1, $a_1{:}y_1$ and $a_1{:}y_2$ end up on $rack_1$, since the tenant did not specify any initial availability requirements and the VMs are relatively small in size. $a_1{:}c_1$ and $a_1{:}c_2$, however, are part of $ha\text{-}c_1$. Hence, these two VMs are placed on different racks, with $a_1{:}c_1$ on $rack_1$ and $a_1{:}c_2$ on $rack_2$. app_2 initial VM placement follows the availability constraints. Based on $ha\text{-}y_2$, $a_2{:}y_1$ is hosted on $rack_1$, while $a_2{:}y_2$ is on $rack_2$. Likewise, $a_2{:}c_1$ is placed on $rack_2$ and $a_2{:}c_2$ on $rack_1$, due to $ha\text{-}c_2$. For app_3, five VMs are hosted on $rack_1$, with the other five on $rack_2$.

In the following, we use *random semantic-aware VM placement* to term this initial placement. Figure 1(a) shows app_1 and app_2 throughput numbers – requests per second, for an interval of 25 minutes, with the random semantic-aware placement technique. At the 17-minute time mark, $tenant_3$ starts an app_3 Hadoop sort job on the 10GB data set stored in HDFS. From this point onwards, all three applications compete for the oversubscribed inter-rack network bandwidth. app_3 sort job has substantial network bandwidth requirements. The network traffic generated impacts the through-



(a) **app_1 – YCSB-Cassandra, app_2 – YCSB-Cassandra, and app_3.** We show throughput numbers for both applications with and without network interference.



(b) **app_3 – Hadoop Inter-Rack Traffic.** Hadoop sort job stresses the inter-rack network interconnect, particularly during the reduce phase.

Figure 1: **Random Semantic-Aware VM placement.** We show three applications running concurrently on the cloud infrastructure. Application VMs are placed according to availability constraints, with rack resource usage reasonably balanced. app_3 causes network congestion and impacts app_1 and app_2 severely.

put of app_1 and app_2. The drop in throughput is more pronounced during the reduce phase of sort. This is expected, as the data shuffling process exchanges the entire 10GB data set among app_3 Hadoop nodes, stressing the network interconnect.

To better understand the network interference caused by the Hadoop sort job, in Figure 1(b) we plot the inter-rack network bandwidth consumed by app_3. The horizontal axis in Figure 1(b) can be matched against Figure 1(a). During the map phase of sort, the bandwidth has sporadic jumps towards 800Mbps. However, during the reduce phase,

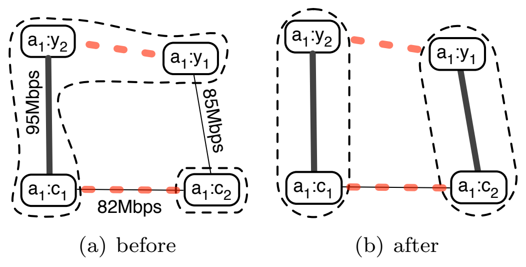


Figure 2: **app_1 – Partition Composition and OX-generated Availability Constraint / Communication Graphs.** In (a), availability constraint between a_1:y_1 and a_1:y_2 is violated (dashed edge), as both VMs are initially on $rack_1$ – same partition. A portion of app_1 VM traffic is inter-rack – solid thin edges. OX produces new, optimal partition composition, as shown in (b)

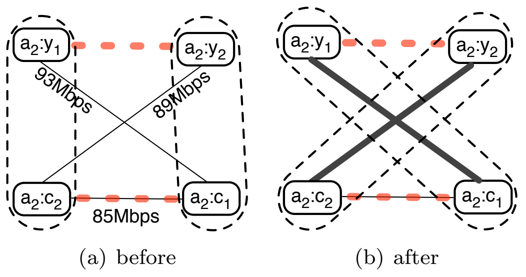


Figure 3: **app_2 – Partition Composition and OX-generated Availability Constraint / Communication Graphs.** In (a), all availability constraints are met – dashed edges. However, all app_2 VM traffic is inter-rack. In (b) we show OX-generated partition composition.

the bandwidth is constantly in the 800Mbps range, causing a sharp interference effect for the two concurrent applications, app_1 and app_2.

OX Placement: OX optimizes application placement following communication patterns and availability constraints. Figure 2 and Figure 3 illustrate the availability constraint/communication graphs produced by OX for app_1 and app_2, along with previous and new partition configurations. The previous partition composition reflects the initial semantic-aware VM placement. Graph vertices are the VM identifiers, while edges are based on availability constraints/network traffic bandwidth between the pairs of VMs.

Availability constraints are shown as dashed edges. For app_1 – Figure 2(a), the availability constraint between $a_1{:}c_1$ and $a_1{:}c_2$ is met, as the two VMs are hosted on separate racks – distinct partitions. On a different note, the constraint between $a_1{:}y_1$ and $a_1{:}y_2$ is violated, as both VMs are on $rack_1$. $Tenant_1$ defined the availability constraint while the VMs were running, hence the violation.

For communication graphs, solid thin edges denote inter-rack traffic, while solid thick edges, intra-rack traffic. For example, as $a_1{:}y_1$ is initially placed on $rack_1$ and $a_1{:}c_2$ on $rack_2$, the two VMs communicate over the oversubscribed inter-rack link at 85Mbps.

For app_2 – Figure 3(a), availability constraints are complied with, while all network traffic is inter-rack.

OX then proceeds with optimizing placement for app_1 and app_2. Note that, due to the strict availability requirements, and, as app_3 VMs use local storage volumes, OX considers the entire application immovable and cannot optimize placement for app_3 – e.g, by hosting all app_3 VMs on a single rack.

For app_1, Figure 2(b) illustrates the new partition configuration generated by OX: $a_1{:}P_1 = (a_1{:}y_2,\ a_1{:}c_1)$, and $a_1{:}P_2 = (a_1{:}y_1,\ a_1{:}c_2)$. The partition-to-rack assignment algorithm assigns $a_1{:}P_1$ to $rack_1$ and $a_1{:}P_2$ to $rack_2$. To realize the partition assignment, OX migrates $a_1{:}y_1$ from $rack_1$ to $rack_2$. The new app_2 partition composition is: $a_2{:}P_1 = (a_2{:}y_1,\ a_2{:}c_1)$, and $a_2{:}P_2 = (a_2{:}y_2,\ a_2{:}c_2)$. $a_2{:}P_1$ is assigned to $rack_2$ and $a_2{:}P_2$ to $rack_1$. Two VM migrations are required to complete the assignment: $a_2{:}y_1$ from $rack_1$ to $rack_2$, and $a_2{:}y_2$ from $rack_2$ to $rack_1$.

Figure 4 shows app_1 and app_2 throughput numbers for an interval of 45 minutes, with OX optimized placement. Around the 5-minute time mark, OX executes the three VM live migrations sequentially, to optimize VM placement. Each VM migration has a slight impact on application performance. At the 25-minute time mark, $tenant_3$ starts a Hadoop sort job on the 10GB data set stored in HDFS. As in Figure 1(a), all three applications now compete for the oversubscribed inter-rack network bandwidth. However, with OX, network interference is substantially reduced. app_1 and app_2 experience only a small drop in throughput due to the inter-rack network traffic between the Cassandra VMs.

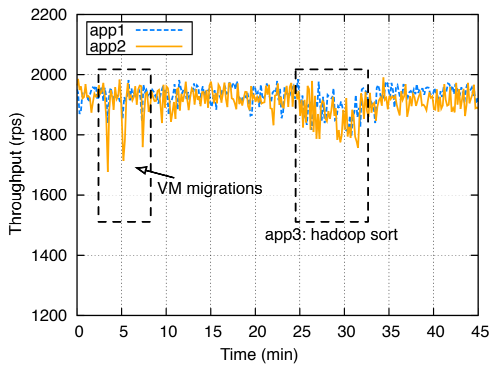


Figure 4: **OX VM placement. app_1 – YCSB-Cassandra, app_2 – YCSB-Cassandra, and app_3 – Hadoop, running concurrently on the cloud infrastructure.** OX optimizes placement for app_1 and app_2 by live migrating VMs according to availability constraints and communication patterns. As a result, network interference effects caused by app_3 are minimized.

Therefore, OX enhanced robustness for hosted applications, by shielding them from unpredictable network interference effects, while adhering to tenant-defined high availability constraints.

6.3 Rack-level Failure Scenario

In the second scenario, we show how OX protects applications from rack-level failures, such as a power outage taking offline an entire rack. We use *three racks* to host two YCSB-Cassandra tenant applications. We evaluate OX against a traffic-aware only VM placement approach.

Tenant Applications: A tenant deploys two identical *YCSB-Cassandra* applications, each composed of three YCSB VMs, y_1, y_2, y_3, and three Cassandra VMs, c_1, c_2, c_3. The Cassandra tier employs 2-way replication based on chained declustering. In addition, the tier supports various levels of *consistency models* – strong consistency or eventual consistency, to ensure availability when dealing with component failures. Thus, the tenant creates three high availability groups per-application, one for each pair of Cassandra VMs, to achieve VM segregation.

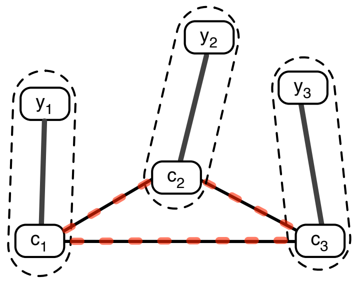


Figure 5: **YCSB-Cassandra Application – Partition Composition and OX-generated Availability Constraint / Communication Graphs.** Following communication patterns and availability constraints, OX produces three partitions to be segregated on three separate racks. The traffic-aware approach ignores availability requirements, placing all VMs on a single rack.

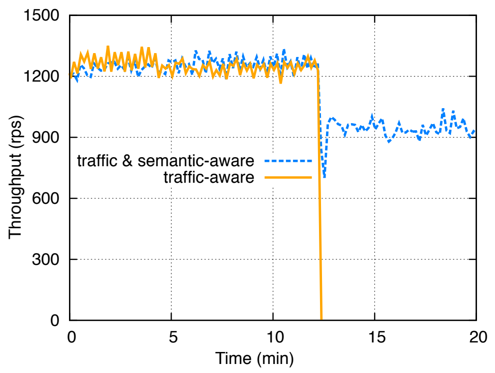


Figure 6: **Rack Failure Scenario.** We show throughput numbers for two identical YCSB-Cassandra applications placed using different policies – OX (traffic & semantic-aware) and traffic-aware only, respectively. At the 12-minute time mark, a rack-level failure (power outage) takes down the entire "traffic-aware only" application.

OX vs. Traffic-Aware: The two identical YCSB-Cassandra tenant applications are hosted on the cloud infrastructure according to different placement policies as follows. For the first application, app_{ox}, VMs are placed using OX, according to communication patterns and availability constraints. For the second application, app_{ta}, we employ a traffic-aware only VM placement policy that ignores availability requirements.

Figure 5 shows the availability constraint / communication graphs for these YCSB-Cassandra applications. The optimal partition configuration generated by OX for app_{ox} consists of three partitions: *(y_1, c_1), (y_2, c_2), (y_3, c_3)*. Consequently, with OX, application VMs are *segregated* across the three racks. On a different note, the traffic-aware policy *colocates* all VMs of app_{ta} on a single rack, $rack_1$, as all VMs communicate frequently and this policy ignores availability constraints.

In Figure 4 we illustrate throughput numbers for the two YCSB-Cassandra applications, during an interval of 20 minutes. For the first 12 minutes, the applications report a similar performance. At the 12-minute time mark, we cause a power outage that fails the entire $rack_1$. As expected, the application hosted using the traffic-aware only policy, app_{ta}, becomes completely unavailable. With OX, however, app_{ox} loses only a third of its VMs and the application incurs just a small drop in performance, with no impact on overall application availability.

Therefore, by placing VMs according to communication patterns and availability constraints, OX protected the application from unpredictable rack-level failures.

7 OX in Large Scale Environments

In this section, we discuss implementation details for scalability in generic large scale deployments of OX.

We have assumed that, typically, the data center contains more than enough resources within racks to accommodate a stable graph partitioning solution for meeting each application's constraints. As previously mentioned, in this common case, OX monitors and optimizes network traffic per application, and focuses its adaptations unto problem areas of the graph. Under these assumptions, our approach is guaranteed to work by selectively moving application VMs away from oversubscribed links, while respecting their availability constraints.

Therefore, as long as enough resources exist within racks, we expect larger scale scenarios to show similar effects for OX adaptations as our evaluation. This is intuitive because each per-application incremental adaptation, in and of itself, is similar to our scenario; the adaptation fine-tunes the VM placement only for the VMs involved in communication over congested inter-rack links; incremental adaptations for applications are performed independently.

In contrast, we argue that alternative algorithms based on global reshuffle optimizations for VM placement would incur long periods of network instability during VM migrations. In addition, some applications, like Hadoop, cannot be optimized effectively, because of massive traffic exchanges among all nodes in the Hadoop cluster. Therefore, alternative algorithms, which may target global utility in terms of network usage are likely to be costly and ineffective.

Furthermore, we also showed in our evaluation a typical case of incremental evolution of, and dynamic adaptation to availability constraints. As the tenant perceives application vulnerabilities, our interface allows him to create additional availability constraints. OX addresses these additional availability constraints incrementally, through performing selective VM migrations, as needed, to reinforce the additional constraints. In this case as well, we expect that OX will provide robustness for large scale scenarios in a similar fashion as for our small scale scenario.

8 Related Work

A number of recent efforts proposed higher density interconnect architectures using commodity components to decongest data center networks [19][12]. These architectures claim high bandwidth connectivity between any two servers in a data center. However, they require significant architectural changes and incur high management/deployment costs. Our approach, OX, is a low-cost scalable solution that increases application resilience to infrastructure anomalies, such as network interference on oversubscribed network links.

VM placement based on network traffic patterns has been suggested as a method to address network bottlenecks in data centers [21]. However, proposed solutions usually minimize a data center wide objective function in isolation of the current VM placement. In large-scale data centers these global solutions are not scalable. In contrast, OX incrementally repartitions applications, and progresses toward an optimal placement, by minimizing the network disruption due to VM migrations. Moreover, as proposed solutions are oblivious to application semantics, naive implementations increase application vulnerability to internal data center hardware failures. For instance, simplistic solutions could place an entire application on a single rack to maximize network performance. As rack failures are commonplace [15], due to faulty rack switches or rack power units, application uptime could be severely impacted. We describe a holistic solution for dynamic intelligent mapping of application components (VMs) onto the virtualized infrastructure. OX bridges high-level application semantics, e.g., availability constraints, to infrastructure-level metrics, e.g., network traffic statistics, to optimize placement and enhance resilience to cloud infrastructure anomalies for hosted applications.

Infrastructure-as-a-Service providers, e.g., Amazon [1], allow cloud tenants to use multiple availability zones when accessing cloud resources. An availability zone is defined by a geographical subregion, e.g., US East-1, and it typically encompasses an *entire* data center. Our system, OX, introduces high availability groups, an abstraction that enables tenants to define high availability constraints among distributed application components (VMs). OX enforces availability requirements *within* a virtualized data center, through intelligent VM placement, without exposing data center physical infrastructure details. Moreover, OX extends current cloud tenant interfaces and can be easily integrated into existing IaaS cloud solutions.

Live virtual machine migration [13] has been proposed to alleviate overload [22], or colocate VMs for memory affinity [23]. We leverage it to provide high availability guarantees and to minimize exposure to anomalies in the network communication fabric, such as network interference effects, for tenant applications hosted on large-scale cloud infrastructures.

9 Conclusions

We introduce OX, a runtime system that targets on-the-fly virtual machine placement optimizations for enhanced application robustness in virtualized data center architectures. These data centers are built by Infrastructure-as-a-Service (IaaS) cloud computing providers, such as Amazon, to host large numbers of tenant distributed applications, concurrently.

The common structure of cloud infrastructures consists of multiple racks of servers interconnected by a hierarchical networking fabric. Servers are virtualized and are shared by multiple virtual machines (VMs) started by cloud tenants as part of tenant applications. Due to the large scale of cloud infrastructures, anomalies, such as failures or bottlenecks, are commonplace, and impact application uptime and overall functionality. With OX, we address application vulnerability to within-rack failures caused by faulty servers, rack switches, or rack power units, and inter-rack communication anomalies, e.g., network interference on oversubscribed inter-rack network links.

OX provides an interface for tenants to specify application-level availability requirements, following application VM roles and replication semantics. In addition, OX discovers per-application inter-VM communication patterns on the fly. OX uses this information to dynamically implement VM placement optimizations in order to enforce application availability constraints and reduce and/or alleviate application exposure to network communication anomalies, such as traffic bottlenecks. Specifically, OX builds on-line topology graphs, one per application and automatically partitions these graphs in an incremental fashion, according to application availability constraints and communication patterns. OX then performs live VM migrations based on the partitioned graph for each application.

We evaluate OX using a mix of Hadoop and YCSB/Cassandra workloads to create dynamic scenarios of network interference and availability constraints. Our results show that OX enhances application resilience to infrastructure anomalies, by reducing the impact of inter-rack network interference, and by shielding applications from rack-level failures.

References

[1] http://aws.amazon.com/ec2.

[2] http://www.xen.org.

[3] http://www.vmware.com.

[4] http://hadoop.apache.org.

[5] http://hadoop.apache.org/hdfs.

[6] http://open.eucalyptus.com.

[7] http://www.gnome.org/ markmc/qcow-image-format.html.

[8] http://nbd.sourceforge.net.

[9] http://openvswitch.org.

[10] http://code.google.com/p/flow-tools.

[11] http://wiki.apache.org/hadoop/Sort.

[12] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM* (2008).

[13] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *NSDI* (2005).

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *SoCC* (2010).

[15] Dean, J. Large-Scale Distributed Systems at Google: Current Systems and Future Directions. In *LADIS* (2009).

[16] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[17] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP* (2007).

[18] Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in Globally Distributed Storage Systems. In *OSDI* (2010).

[19] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[20] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. In *LADIS* (2009).

[21] Meng, X., Pappas, V., and Zhang, L. Improving the Scalability of Data Center Networks with Traffic-Aware Virtual Machine Placement. In *INFOCOM* (2010).

[22] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI* (2007).

[23] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *VEE* (2009).