Live *VM* Migration under Time-Constraints in Share-Nothing *IaaS*-Clouds

Konstantinos Tsakalozos, Vasilis Verroios, Mema Roussopoulos, and Alex Delis

Abstract—Live *VM* migration helps attain both cloud-wide load balancing and operational consolidation while the migrating *VM*s remain accessible to users. To avoid periods of high-load for the involved resources, *IaaS*-cloud operators assign specific time windows for such migrations to occur in an orderly manner. Moreover, providers typically rely on share-nothing architectures to attain scalability. In this paper, we focus on the real-time scheduling of live *VM* migrations in large share-nothing *IaaS* clouds, such that migrations are completed on time and without adversely affecting agreed-upon SLAs. We propose a scalable, distributed network of brokers that oversees the progress of all on-going migration operations within the context of a provider. Brokers make use of an underlying special purpose file system, termed *MigrateFS*, that is capable of both replicating and keeping in sync virtual disks while the hypervisor live-migrates *VM*s (i.e., RAM and CPU state). By limiting the resources consumed during migration, brokers implement policies to reduce SLA violations while seeking to complete all migration tasks on time. We evaluate two such policies, one based on task prioritization and a second that considers the financial implications set by migration deadline requirements. Using our *MigrateFS* prototype operating on a real cloud, we demonstrate the feasibility of performing migrations within time windows. By simulating large clouds, we assess the effectiveness of our proposed broker policies in a share–nothing configuration; we also demonstrate that our approach stresses 24% less an already saturated network if compared to an unsupervised set up.

Index Terms—Distributed Systems, Cloud Computing, *IaaS* Clouds, Virtual Machine Migration

1 Introduction

Large *IaaS* cloud providers offer high quality services by constantly adjusting resource usage and balancing the load in their infrastructure [1], [2]. Since *IaaS* providers predominantly offer virtual machines (*VM*s), load-balancing is usually achieved through *VM* migration, i.e., transferring a *VM* from one physical machine (*PM*) to another. *VM* movement helps offload congested physical nodes, can enhance the utilization of the underlying resources, and can ultimately achieve an improvement in the quality of provided services.

In this paper, we focus on *live VM* migrations for *IaaS* cloud providers that rely on a *share-nothing* infrastructure. In a *share-nothing* infrastructure, *PM*s use their resources (e.g., memory, disk storage) independently, instead of accessing common resources via synchronization layers (e.g., common storage layer). The main advantage of *share-nothing* infrastructures is scalability, as less synchronization means fewer bottlenecks (more in Section 2).

A *live VM* migration is performed while the migrating *VM* remains on–line and involves a short downtime hardly noticeable by users interacting with the *VM* [3]–[5]. Practically, in a *share-nothing* infrastructure, a *live VM* migration requires that a virtual disk on the order of multiple *GBytes* be transferred from the source to target *PM* (used for hosting the *VM*), while the *VM* remains fully accessible to its users.

Furthermore, *VM* migrations should not coincide with high-load periods for the involved *VM*s and network resources. To this end, *IaaS*-cloud providers rely on usage statistics to decide upon a *time window* for each migration to take place. Failing to complete a migration within the time window will most likely degrade the *QoS* experienced by the affected users and may lead to a number of Service Level Agreement (SLA) violations.

The problem we study in this paper, is the real–time scheduling of *live VM* migration tasks in *share-nothing IaaS*-clouds. Given a new *PM* host and a time window for each *VM* migration (decided by a cloud reallocation policy), a real–time scheduling mechanism must:

- control the resources allocated to each migration task, based on the *QoS* degradation and the SLA violations that any affected *VM* may experience. For example, the network bandwidth consumed by a migration task can lead to an SLA violation for a *VM* hosted on the migration's target *PM*: an efficient scheduler must adjust the network resources allocated to the migration task, to avoid such SLA violations.
- limit the migration side-effects experienced by the users of a migrating *VM*. Ideally, there should be no restrictions on the usage of a migrating *VM*. However, letting a migrating *VM* constantly write on blocks that need to be re-transferred to the target *PM*, can extend the migration's duration beyond its assigned time window. To prevent a migration from extending beyond its time window, a real-time scheduler may have to limit the *VM*'s write rate. An efficient scheduler should take into account the tradeoff between side-effects and the migration's duration and be as non-interventional as possible while still preventing a migration from running beyond its window.
- prioritize concurrent migrations and impose limitations that minimize the overall cost, taking into account potential SLA violations and implications on the *QoS*.

Prior work on *on-demand virtual disk synchronization* [6]–

- *Konstantinos Tsakalozos is with Canonical Ltd, London, SE1 0SU, UK. E-mail: konstantinos.tsakalozos@canonical.com*
- *Vasilis Verroios is with Stanford University, Stanford, CA 94305-9040. E-mail: verroios@stanford.edu*
- *Mema Roussopoulos and Alex Delis are with the University of Athens, Athens, GR15703, Greece. E-mail: {mema, ad}@di.uoa.gr*

[9] focused on synchronizing individual virtual disks. In contrast, we focus on the *multiple simultaneous migrations* setting, as discussed above, and we leverage results from prior work to form the foundation of our approach.

Our approach consists of *MigrateFS*, a low-level special-purpose file system, and higher-level resource allocation policies, designed to accommodate large numbers of simultaneous migration tasks.

The *MigrateFS* file system runs on every *PM*: instances of *MigrateFS* communicate over the network and jointly control the transfer of a virtual disk image between any two *PM*s. *MigrateFS* continuously adjusts the consumption of both *VM* and migration resources based on hints provided by performance monitoring tools [10], [11]. In particular, *MigrateFS* tunes two rates during *VM*–disk shipment: *a)* disk throughput available to the *VM*'s internal processes that access the virtual disks during migration, and *b)* network throughput used for the purposes of migration. In this way, *MigrateFS* is able to accurately estimate the completion time of a migration. Estimates also allow *MigrateFS* to delay a migration (while still completing it before the end of the assigned time window) when the cloud experiences heavy workloads.

Resource allocation policies are implemented by a coordinating *Migrations Scheduler* and a distributed *network of Brokers*. Essentially, resource allocation policies allow for prioritization of migration tasks while taking into account the network status so that "hot" physical network links are not further stressed by virtual disk shipments. *Brokers* apply such policies and drive the operation of *MigrateFS* instances by indicating how the network and disk throughput must be restricted, in each case.

Our evaluation, based on both a *MigrateFS* prototype and simulation of large infrastructures, shows up to 24% less stress on saturated *PM*s during migration. The main cost in our approach comes from the need to keep track of I/O operations. In our evaluation, we show how *MigrateFS* is outperformed by local file systems, yet it is more efficient than network storage solutions that enable *VM* migration.

For large cloud providers, there are cases where the SLAs of different *VM*s vary widely: violating the SLA of one *VM* will impose a financial cost that may be orders of magnitude greater if compared to the SLA violation of another *VM*. For such cases, we propose a simplified, yet general, model that quantifies the cost different SLA violations entail. This model inspires a cost-driven resource allocation policy that runs across the distributed network of Brokers assisting *MigrateFS* to deliver substantial gains: our experiments indicate a $2x$–improvement over the baseline approach that is unaware of the differences among SLAs.

The rest of the paper is organized as follows: I) we provide an overview of *VM* migration in *IaaS* clouds: we discuss different architectures with respect to *VM* migration and how our approach fits in this space (Section 2), II) we suggest a comprehensive solution based on brokering of migration resources which is empowered by our *MigrateFS*; the latter offers the means to control resource consumption (Sections 3 and 4), III) we propose two resource allocation policies based on migration task prioritization and cost-based SLA compliance (Section 5), IV) we demonstrate through prototype and simulation experimentation that our *MigrateFS*-based approach offers significant gains in shipping *VM*s with diverse SLAs in real-time (Section 6).

2 Live *VM* Migration – Motivation

We first discuss suspend-resume and live *VM* migration and then we offer an overview of how different cloud architectures support *VM* migration. In the end of this section, we describe how the proposed approach addresses the high-level challenges in supporting large-scale live *VM* migration.

2.1 *VM* Migration

Virtualization enables cloud administrators to migrate *VM*s across the physical nodes and control the resources the clients of the cloud provider can access. *VM* migration is a key operation in current cloud installations as it assists administrative tasks (e.g., remove *VM*s from a failing physical machine) and serves high level resource management policies such as load balancing (i.e., moving *VM*s of stressed physical nodes to under-loaded ones).

There are two alternatives to *VM* migration: suspend-resume and live migration. When the suspend-resume method is used, the operation of the *VM* is stopped (suspended), the contents of the *VM*'s RAM, the virtual disks and the virtual devices' state are copied from the source physical machine to the target and, finally, the operation of the *VM* is resumed on the target physical node. The main drawback in this *VM* migration method is the *VM* downtime; the downtime period is typically dominated by the time to copy the virtual disks across physical nodes.

Live migration reduces downtime to a degree that is hardly noticeable to humans interacting with the *VM*. During live migration, RAM contents are copied progressively from the source *PM* to the target while the *VM* remains online. In a single short step, the *VM*'s operation is suspended, any remaining dirty RAM pages along with the virtual devices' state are copied to the target *PM*, and the *VM*'s operation is resumed. This short downtime is on the order of milliseconds [3]. Live migration is currently a task undertaken by *VM* Monitors (*VMM*s) [11]–[13] alone. Therefore, the two hosting physical systems should be set up with the same *VMM*, trusting each other and, most importantly, sharing the same persistence layer (storage) in which the virtual disks of the *VM*s reside. In other words, the architecture and setup of the *IaaS*-cloud largely affect the ability to support live migration.

Next, we discuss in more detail the shared-storage-layer and shared-nothing architectures with respect to live *VM* migration:

Dedicated Shared Storage Systems: Current *IaaS*-clouds that support live *VM* migration often use dedicated shared storage systems. Such storage solutions offer enhanced reliability and availability features through RAID arrays and hot swappable disks. However, having *PM*s clustered around powerful storage pools has the following drawbacks:

1) As storage nodes are used to serve multiple *VM*s they become a single point of failure.
2) The storage nodes may become a performance bottleneck as they may not be able to effectively (in terms of disk and network bandwidth) handle high load peaks.
3) To address the high performance and reliability requirements of the storage nodes, non-commodity hardware must be utilized. This, however, is not aligned with

the use of commodity hardware across the cloud that allows large cloud providers to be cost-efficient.

4) Storage nodes only partially solve the problem of load balancing. Load can only be exchanged amongst *PM*s accessing the same storage node. Consequently, this approach does not scale to the number of *PM*s found in typical large data centers.

Distributed File Systems over a Share-Nothing Archirecture: Clouds will typically have to follow a share-nothing architecture and exploit commodity hardware so as to scale in a cost-effective way. To tackle inefficiencies of the dedicated shared storage approach, cloud providers can employ distributed file systems such as *GFS* [14] and *GlusterFS* [15]. Such file systems allow for several *PM*s to collectively form a persistence layer, while at the same time enhance resistance to node failures through data replication. Yet, scalability issues are also present here. In addition, highly scalable file systems are forced to relax their semantics (e.g., offer non-POSIX API) and thus are inappropriate for hosting *VM*s.

On-demand Virtual Disk Synchronization over a Share-Nothing Archirecture: The requirements driving the design decisions of distributed file systems differ from those of a file system targeting only live migration. A general purpose distributed file system aims to present the *same* view of *all* files to *all PM*s at *all* times. Instead, a special-purpose file system built solely to assist migration, needs to keep a virtual disk synchronized only during the respective *VM*'s migration [6]–[8].

2.2 *MigrateFS* over a Share-Nothing Archirecture

We conclude this section by briefly discussing how the approach we propose, addresses the challenges for live *VM* migration on large-scale cloud infrastructures.

Essentially, attaining load balancing and resource consolidation in a large-scale *IaaS* cloud is a challenging task due to mainly two factors: a) the size of the physical substrate calls for solutions that scale horizontally in an cost–effective manner and b) the complexity of task orchestration can effectively render other known solutions impractical [8], [16]. Our approach builds on the success of on-demand disk synchronization [7], as this migration solution fits the scalability requirements of large clouds. *MigrateFS* is a special purpose file system that empowers our on-demand disk synchronization and so, it enables live migration between any pair of *PM*s in a cloud infrastructure; this design characteristic allows clouds to be based on a share-nothing architecture. Through our own *MigrateFS* implementation, we offer key performance tuning mechanisms that can be used to implement high level, migration-oriented, resource allocation policies.

To accommodate the high level goal of load balancing and timely migrations, *MigrateFS* can monitor and limit resource consumption. In effect, our approach works towards real-time scheduling of migrations in *IaaS*-clouds by respecting deadlines on copying disk images and limiting the impact on the network and *VM* operation. *Brokers* schedule migrations under deadlines and try to ameliorate any resource shortages that may appear. The resource consumption policies enforced by the *brokers* reduce SLA violation. In addition, as migration tasks are on-going, resource consumption is continuously tuned so as both the deadlines and the *VM* SLAs are met.

The decisions regarding which *VM*s should be migrated and the time window for each migration, are taken at a level where a view of the entire physical infrastructure -and its performance- is available. We consider such decisions as input to *MigrateFS* and we do not discuss them in detail, in the rest of the paper.

3 Overview of Our Approach

Each migration task is defined by: *a)* the *VM* to be migrated, *b)* the source and target *PM*s involved, and *c)* the time window within which the migration has to complete. Policies [1], [2], [17] that help determine whether, when, and where a *VM* should migrate, typically consider the average and current load in both *PM*s and *VM*s, projections on future *VM* resource consumption, and the SLAs to be satisfied.

In most cases, the migration of a *VM* is scheduled in low activity periods for *all* involved *VM*s. As a toy example, consider a migrating *VM* v_a that migrates from a source *PM* that also hosts a second *VM* v_b to a target *PM* that also hosts a third *VM* v_c. In addition, consider that all three *VM*s interact with thousands of users per minute, however, these users are located in different geographical areas: most users of v_a are located in California, most users of v_b are located in Europe, and most users of v_c are located in China. Because of the time difference, we can expect that the low activity period for the three *VM*s will be a window of a few hours or minutes. In practice the number of involved *VM*s may be much larger. Thus, the low activity window may be extremely limited and a migration extending beyond this window may critically affect the *QoS* for the involved *VM*s.

Our approach assumes the existence of a queue where all migration tasks arrive. We aim to manage resources in a way that all migrations complete within their respective time-constraints while not failing the offered SLAs.

Figure 1 shows the key components of an *IaaS*-cloud that our approach deploys. At the top of Figure 1 lays the *Cloud Middleware* such as OpenStack or OpenNebula [18], [19]. *Migration Tasks* produced by a *Load Balancing Policy* [1], [2], [17] in the context of the middleware, are dispatched to the underlying physical infrastructure. In Figure 1, there are three physical systems, each one featuring its own local physical disk and a *VM* hypervisor [12], [13]. The *VM*s hosted on each system place their data on virtual disks stored as files on physical disks. All physical systems communicate through a networking layer represented as a switch/router at the bottom of Figure 1.

Our approach entails three components: the *Migrations Scheduler*, the *Brokers*, and a special-purpose file system *MigrateFS*. *MigrateFS* offers resource management facilities -denoted as *Disk* and *Network Throughput Control Points* in Figure 1- exploited by the *Brokers* during *VM* migration. Next, we describe the three components in more detail.

3.1 The *Migrations Scheduler* and its *Brokers*

The *Migrations Scheduler* takes as input a number of migration tasks along with their respective time-constraints. Tasks can be prioritized based on the cost of violating their time-constraints. The distributed network of *Brokers* oversees the resource consumption for the transfer of the *VM*s' disk images. A low cost communication policy is

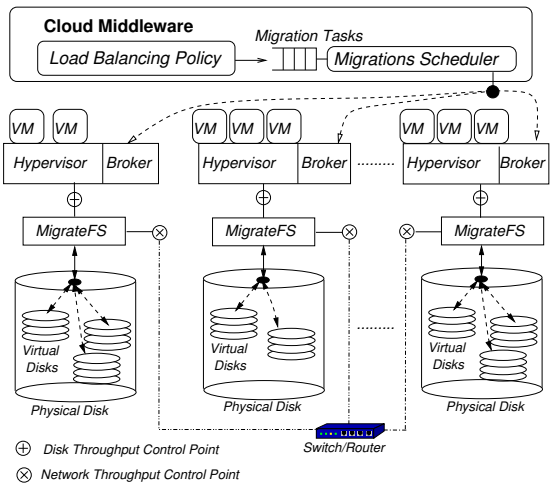


Fig. 1. High level view of our approach.

used for the interaction between *Brokers* so as to ensure the scalability of our approach. As soon as the virtual disks are transferred across the *PM*s, the *Migrations Scheduler* contacts the hypervisor and initiates the last stage of the migration; which seamlessly passes the control for the migrated *VM* to the new *PM* host. The synergy of the *Migrations Scheduler* and the *Brokers* ensures the timely movement of all *VM*s pending action.

To honor the migration time constraints, while respecting the SLAs offered, the *Brokers* have to manage two types of resources. First, the network bandwidth consumed for transferring virtual disk images must not hamper the performance of other *VM*s. Second, the virtual disk I/O bandwidth (available to the migrating *VM*) must be limited since this disk I/O translates to dirty disk pages that ultimately have to be (re-)transferred over the network. The usage of both resources (disk and network) can be constrained through the facilities offered by *MigrateFS*. The consumption of these two resources is continuously adjusted so that no SLAs fail in the dynamic cloud environment where the migrations take place (details in Section 5).

3.2 *MigrateFS* and Resource Consumption Restrictions

Constraining resource consumption, so as to comply with time restrictions on migration tasks, has to be assisted by low-level cloud facilities. Such facilities must function outside the *VM*s as the abstractions enforced by the cloud, hide the *VM*'s internal operations from the cloud administration. In our approach, migration is assisted by a special purpose file system, *MigrateFS*, that *traps all I/O operations* targeting the virtual disk images of the migrating *VM*s. *MigrateFS* introduces a layer between the *VM* Monitor (*VMM*)/hypervisor and the physical device where the *VM*'s virtual disks are stored (Figure 1). Instances of *MigrateFS* collaborate in moving *VM* disk images without interrupting the operation of the *VM*s. During migration, blocks of the migrating *VM*'s disk images are copied to the target *PM*. When all blocks are transferred, the two copies of virtual disk images -in both the source and target *PM*s- are kept synchronized while the *VM* hypervisor completes the migration task by moving RAM contents and devices' states. It is in the context of *MigrateFS*, where the bandwidth of disk and network, are placed under restrictions.

• *Disk bandwidth:* As *MigrateFS* transfers blocks from the source to the target *PM*, the still on-line *VM* may write over already transferred blocks. These dirty blocks need to be transferred again. If the rate at which virtual disk blocks get dirty is higher than the rate blocks are transferred through the network, the migration task will not finish. Yet, the migration task must come to its completion under certain time constraints. To this end, an authorized *Broker* may contact *MigrateFS* and limit the rate at which the *VM* writes to its virtual disk. The decision on when and if such an I/O rate must be limited is based on the migration time constraints and the current network and disk I/O rates reported by *MigrateFS* (details in Section 5).

• *Network bandwidth:* We need to ensure that no migration task will deplete the network resources of the *IaaS*-cloud. *MigrateFS* enables the *Brokers* to limit the network bandwidth consumed during migration. In this way, saturated network links, shared among migrating and non-migrating *VM*s, do not cause network SLA failures. By effectively limiting the network bandwidth, we are able to distribute resource consumption throughout the entire migration's time window (details in Section 5).

4 Operational Aspects of *MigrateFS*

The I/O operations trapped by the *MigrateFS* layer can be replayed on remote *PM*s so as to create and maintain synchronized copies of virtual disk images. To this end, an instance of *MigrateFS* has to be installed on each *PM* that plays the role of either the source or target hosting node in a migration.

Each *MigrateFS* instance listens on a port for connections from either three sources: *a)* the *Migrations Scheduler* requesting the transfer of a *VM*, *b)* another *MigrateFS* instance sending data blocks and remotely replaying I/O operations, or *c)* a *Broker* overseeing the resource consumption of a migration task. *MigrateFS* presents an interface through which a *Broker* can query the progress of a migration task and set limits on the disk and network bandwidth. Table 1 summarizes the functionality *MigrateFS* offers for handling and monitoring a migration task.

TABLE 1
MigrateFS network API for a *VM* migration task

Operation	Input/Output
Start a Migration	**In:** Target *PM*, disk image
Set Network Limit	**In:** Bandwidth in KB/sec
Set Disk Limit	**In:** Bandwidth in KB/sec
Query Progress	**In:** Selectively query the Network or Disk rate **Out:** The respective bandwidth in KB/sec
Query Completion	**Out:** True or False

As we show in Figure 2, *MigrateFS* is a user-space file system functioning as an intermediate between the *Linux* kernel and any underlying file system on the physical disk. *MigrateFS* is thus mounted over an already existing file system and mirrors its contents. The underlying file system is used to store virtual disk images. When an I/O request is issued by a process within a *VM* (I/O op. in Figure 2) it is forwarded through the virtual file system (VFS) API of the hypervisor to the file system mounted on the path where

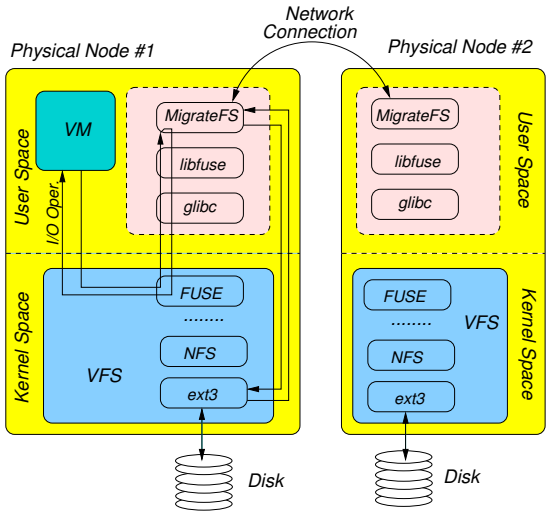


Fig. 2. Routing I/O calls through the layers of our approach.

the virtual disks are stored. The response to the I/O call is routed from the underlying file system through *MigrateFS* and the hypervisor to the *VM*. Our decision to mirror an already existing path in the physical disk storage greatly reduces the effort to set up *MigrateFS* on an already operational *IaaS*-cloud. An in-kernel file system implementation might yield higher performance but does so at the expense of a more intrusive and less deployable approach.

The main operation of *MigrateFS* is to synchronize a virtual disk image across any two *PM*s. To initiate a *VM* migration, the *Migrations Scheduler* has to contact the *MigrateFS* instance of the source *PM* where the *VM* is currently hosted and specify: a) the hostname of the target *PM*, b) the port on which the *MigrateFS* instance of the target *PM* listens, and c) the filenames of the disk images of the migrating *VM*. As soon as the request for a *VM* migration is received, the *MigrateFS* instance deployed on the source *PM* contacts the *MigrateFS* instance on the target and starts sending blocks of the virtual disk image over the network. Synchronizing a virtual disk across two *PM*s is a three-phase process:

- *Phase 1:* Iterate once over all blocks of the virtual disk and send them to the target *PM*. As the *VM* remains on-line, in-*VM* processes may write over a portion of the blocks already transferred; these blocks are marked as dirty.
- *Phase 2:* Dirty blocks are transferred through the network. Here, the operating *VM* may continue to alter the content of blocks already transferred, thus rendering them dirty again. Sending blocks stops (phase 2 ends) as soon as all dirty blocks are transferred. For this to happen, the rate at which blocks get dirty must be less than the rate at which blocks are transferred through the network.
- *Phase 3:* During this phase, the virtual disks remain synchronized across the target and source *PM*s. Each write operation performed on the virtual disk residing in the source *PM* is replayed on the one residing on the target *PM*.

The purpose of this 3-phase copy of the virtual disks is to ameliorate the performance penalties of migration. The *VM* can operate with no restrictions during the first and second phases. In these two phases, the disk addresses of dirty blocks are kept in a thread-safe array protected through proper locking mechanisms. Dirty blocks are cleaned out during the second phase by sending them over the network. To reduce the high network latency penalty, the transfer block size (512 *KB*) is larger than the block size used by the local disk file systems.

In phase 3, we have no dirty blocks. Each write operation is performed in both replicas of the virtual disk. During this phase, there is a significant impact on the I/O performance of the *VM* due to the network latency. Here, the block size is equal to the size of the I/O request. We expect the third phase to be short. During this phase, the hypervisor completes the live migration task by transferring the *VM*'s memory and devices' state. The details of moving the *VM*'s RAM and devices' state from the source to the target *PM* while minimizing the downtime are hypervisor-specific. With respect to the file system, the hypervisor sends a *sync* I/O system call as the last I/O operation right before the *VM* starts operating in the target *PM*. When the *sync* call is received, *MigrateFS* flushes all buffers (network and disk) so that the *VM* on the target *PM* will find the virtual disk in a consistent state.

The *Broker* overseeing the migration process *tunes* the consumed network and disk resources through *MigrateFS*. In the following sections we show how the *network of Brokers* collectively coordinates action towards attending outstanding migrations and thus, adjusts the cloud load in a timely fashion.

5 *VM*-Migration Resource Management

Management of cloud resources during migrations ensures that *VM* movements are fulfilled within the specified time-constraints. Efficient resource handling should impose minimal overheads so that it can be applied to large, share-nothing clouds. In addition, resource handling should not impose special hardware requirements as the scalability of large infrastructures is based on the use of commodity hardware. We achieve the above through the distributed network of *Brokers* that interact using a low communication cost policy.

Pending *VM* migrations -produced by *VM* placement policies [1], [2], [17]- are delivered to the *Migrations Scheduler*. As soon as the latter decides that a migration task should start, it instantiates a *Broker* on the migration's source *PM*; the task of resource handling in shipping the *VM* is assigned to that *Broker*. The *Broker* aims at keeping the rate at which blocks are moved (*Net_Rate*) from the source to the target *PM* greater than the rate at which blocks get dirty (*Dirty_Rate*), so that the virtual disk network copy completes within the designated time frame (*finish_time*) and still leaves enough time (*VMM_time*) for the hypervisor to successfully complete the migration. In other words, the *Broker* acts so that Expression 1 remains true for the designated migration task:

$$\frac{Disk\_size}{(Net\_Rate - Dirty\_Rate)} + VMM\_time \leq finish\_time \quad (1)$$

Both the *Dirty_Rate* and the *Net_Rate* are periodically queried from *MigrateFS*. Increasing the querying frequency

comes at the expense of higher overhead in the CPU, memory and network resources.

5.1 Priority-Based Resource *Brokers*

The network of *priority-based Brokers* requires two types of cloud operational information: *1)* Real-time notification of saturated network switches (hot-spots): such notifications can be provided by cloud-monitoring tools [10] that detect stressed network links of the fixed (often tree-based) physical network topology [20]. *2)* The path of *VM* shipment: it is straightforward to compute such paths in a fixed network topology with static routing rules. The *Migrations Scheduler* computes migration paths and marks path sections shared among multiple migration tasks.

The *Migrations Scheduler* provides the *VM* shipment path upon the *Broker's* instantiation. The *Broker* registers for notifications on saturated switches to the corresponding cloud monitoring tools. As soon as the *Broker* projects that the designated migration time-constraint will be violated, it needs to request other *Brokers* to release (if possible) some of the network bandwidth they occupy[1]. As the *Migrations Scheduler* is aware of all migrating *VM* disks sharing network paths any *Broker* can exploit this information to notify only those *Brokers* with which it shares saturated network links. Since *Brokers* exchange messages directly with each other in a peer-to-peer fashion there is no single message exchange hub. The distributed nature of *Broker* communication allows our approach to scale to the size of large cloud installations.

The cloud administration is allowed to specify which of the migrating tasks are important. *Brokers* responsible for such tasks should be the first to *a)* ignore network congestion and *b)* signal other *Brokers* to temporarily suspend their migration tasks whenever they face the danger of violating the migration's time frame. There are two thresholds, termed *danger* and *warning*, that are used in task prioritization. Both thresholds are percentages expressing the ratio between the time the migration has to be finished (*timetodeadline*) and the projected time the migration task will actually last (*timeleft*). Both threshold percentages are expected to be greater than 100% and the *danger* percentage to be less than the *warning* percentage as we first get a warning and then we face the danger of violating a constraint.

The operation of a *Broker* is described in Algs 1 and 2. Alg. 1 shows *when* the *Broker* chooses to limit the disk and network resources in the context of a single migration task, while Alg. 2 shows *how* this network limit is set. Network bandwidth consumption due to migration may cause other *VM*s to fail to uphold their SLAs. To address failing network SLAs, we should limit the network resources used by the migration process. When limiting the disk transfer rate, we ensure that SLAs will not fail due to the resources consumed for migration, but we do not commit to any migration time frame. Constraints on the migration time – given the available network bandwidth does not decrease– are honored when we limit the disk bandwidth available to the processes inside the *VM*. Limiting both transfer and disk rates enables us to offer *Deadline Scheduling* for the migration. *Deadline Scheduling* can be achieved even if we unbound the network transfer usage. In this case, we take the risk of failing network SLAs, yet, we potentially reduce the migration time.

Alg. 1 estimates the *timeleft* and the *timetodeadline* (lines 3 and 4) based on the disk blocks (*task.diskleft*) and two rates (*diskrate* and *netrate*) queried through the *Query Progress* call of *MigrateFS* (Table 1). Should a warning of no compliance with a designated time-constraint be received (line 9), we limit the available disk bandwidth. *setDiskLimit* uses Ineq. 1 to compute the *Dirty_Rate* based on the current network utilization and remaining disk blocks. In line 6, we assess the danger of failing to migrate the *VM* on time and if so, we limit the used network rate (*setNetworkLimit* call in line 7). Similarly, in line 9, we get a warning of violating a time constraint if the ratio of *timeleft* to *timetodeadline* is greater than the *warning*. The higher the values of the *danger* and *warning* parameters, the sooner our algorithm will take action to secure the time constraints.

Algorithm 1 PriorityBasedMigrationsManagement

Input: *task*: The migration operation
period: Time between monitoring iterations
danger: Threshold indicating high chances of loosing the migration deadline
warning: Threshold indicating miss of the migration deadline

```
1:  while (task.completed == false) do
2:    sleep(period);
3:    timeleft := computeTimeLeft(task.diskleft, task.netrate,
      task.diskrate);
4:    timetodeadline := task.deadline - now;
5:    backoff := shouldBackOff(task.source, task.target);
6:    if ((timetodeadline/timeleft ≥ danger)
          and (backoff == true)) then
7:      setNetworkLimit(backoff, task);
8:    end if
9:    if timetodeadline/timeleft ≤ warning then
10:     setDiskLimit(task);
11:   end if
12: end while
```

Limiting the network bandwidth is not based only on the *danger* threshold. We also limit the network usage rate if we detect a network contention, as indicated by the *backoff* flag. This flag is set to true under two conditions: *a)* the migration process causes a network SLA failure of a running *VM*, *b)* the network bandwidth consumed should be given to another migration task that is about to violate its time constraint. The *shouldBackOff* function detects saturated network links on the path between the source and target hosting *PM*s.

Alg. 2 depicts how this network limit is set. The *Broker* takes into account the current network rate available through the input parameter *task*, the *backoff* flag and the number of consecutive periods with no backoff request indicating no network congestion. Inspired by TCP, our approach reduces the limit of the network bandwidth usage by dividing the current network rate by two and increases the network limit linearly. Two factors influence our decision: the last decision to increase or decrease the network limit (outer **if-then-else** statement), and any request for releasing bandwidth made through the *backoff* flag. If we had limited the network usage and we still observe network congestion

1. Network congestion may not be caused by *Brokers*. Nevertheless, *Brokers* can only release resources they alone reserve.

(**if** statement in line 2), we use the *task.netrate* divided by two as the new limit. If we had previously reduced the network rate and we now have no back off request, we mark the current network rate as one that causes no network congestion (line 5). This mark, stored in *task.lastOKnetlimit*, is used in line 10 where we have previously increased our network limit and we just got a back off request. Assuming that our migration task caused the network congestion, we quickly revert back to a network rate that did not cause any congestion in a previous period. Further *backoff* requests will cause lowering even more the network limit. The **else** clause of lines 11 to 18 handles the case where we continuously increase our network limit. We raise the network limit linearly, yet, there might be the case that a lot of bandwidth has become unexpectedly available due to an event that we are not informed of (e.g., another migration task has just finished). In this case, we need to probe the network availability and quickly take advantage of the extra bandwidth (lines 13 to 15).

Algorithm 2 setNetworkLimit

Input: *task*: The migration task
backoff: True, if we are to reduce the network bandwidth, False otherwise
n: # of consecutive periods with no *backoff* request indicating no network contention
C: Step of network rate increase, in MB
Output: The network limit

```
1: if task.lastAction == "reduce bandwidth" then
2:   if backoff == true then
3:     return task.netrate / 2;
4:   else
5:     task.lastOKnetlimit := op.netrate
6:     return task.netrate;
7:   end if
8: else if task.lastAction == "increase bandwidth" then
9:   if backoff == true then
10:    return task.lastOKnetlimit;
11:  else
12:    task.lastOKnetlimit := task.netrate
13:    if consecutivePeriodsWithoutBackOff(task,backoff) > n
       then
14:      return +∞ /*no network limit*/
15:    else
16:      return task.netrate + C
17:    end if
18:  end if
19: end if
```

Alg. 2 implements a policy that requires no communication with the other consumers of the network bandwidth. Priority is given to *VM*s failing their SLAs and to migration tasks in danger of violating their time constraints. In this context, we opt for a low-cost communication policy among *Brokers* as we target large cloud infrastructures. *Brokers* need only to announce the danger of violating the time constraint of a *VM* migration to a well specified subset of other *Brokers* so that the *shouldBackOff* call yields valid back-off requests.

5.2 Cost-Driven *Brokers*

Cloud providers often have to deal with very diverse SLAs. As a consequence, the eventual cost due to an SLA violation considerably varies depending on the *VM* type and/or the customer. Imposing the same network or disk rate restrictions on diverse *VM*s may significantly affect the quality of services delivered. For instance, consider a *VM* supporting hundreds of database transactions per minute. The financial cost incurred when restricting the dirty page production rate of such a *VM* (as defined in the respective SLA) can be much greater than the cost of limiting the bandwidth of a group of (other) *VM*s. Similarly, violated time constraints of different migrations, incur different end results.

In our *cost-driven* policy, the network of *Brokers* is enhanced to take into account the financial penalty inflicted by the resource restriction decisions. *Cost-driven Brokers* aim at cloud providers that can largely benefit from fine-tuning the migrations' disk/network-rate restrictions, based on the different SLAs. To this end, we use a simple model (Eqn. 2) applicable to any type of SLA and cloud infrastructure. The factors involved in our cost model are similar to the ones involved in the model used in papers [21] and [22].

Eqn. 2 captures the costs involved in a migration task: *a)* the impact on the QoS due to slowing down the "dirty" data rate of the migrating *VM* (*DiskCost*), *b)* the overhead of the migration traffic to the network (*NetCost*), and *c)* the consequences of a possible violation of the migration's time-constraint (*MissRisk*).

$$Cost = DiskCost + NetCost + MissRisk \quad (2)$$

Our rationale for the three cost factors is as follows:

1) SLA violations occur when limiting the disk bandwidth available to the migrating *VM* (*disklimit*) below the requested bandwidth (*requestedrate*). We model the *DiskCost* to be proportional to the ratio *requestedrate*/*disklimit*.
2) The more network bandwidth a migration consumes, the greater the *NetCost* should be.
3) The probability of missing a migration deadline increases rapidly as the projected time left (*timeleft*) becomes greater than the time-to-deadline (*timetodeadline*). Thus, *MissRisk* is based on the ratio *timetodeadline*/*timeleft*.

DiskCost

We model the *DiskCost* as a polynomial function of the ratio *requestedrate*/*disklimit*:

$$DiskCost(disklimit) = C_{disk} \times (\frac{requestedrate}{disklimit})^q, \quad q \geq 0 \quad (3)$$

In most cases, a linear function, i.e., using an exponent $q = 1$, is sufficient. (In rare cases where the service offered by the migrating *VM* is very sensitive to limiting the available disk bandwidth, a q greater than 1 should be selected.) The constant C_{disk} normalizes the cost in financial terms.

NetCost

The *NetCost* is modelled as a polynomial function of the network bandwidth a migration consumes (*netlimit*):

$$NetCost(netlimit) = C_{net} \times netlimit^p, \quad p \geq 1 \quad (4)$$

As in the *DiskCost*, a linear function ($p = 1$) is sufficient in most cases. The normalizing constant C_{net} depends on the network SLA violations due to migration traffic. In practice, C_{net} can be proportional to the "saturation" of the

switches involved in the migration; where the saturation of a switch reflects how much of the overall switch's bandwidth is consumed.

MissRisk

The *MissRisk* depends on how critical the migration's deadline is and how likely it is to miss that deadline. When the *timetodeadline* is much higher than the *timeleft*, the *MissRisk* should approach zero. On the other hand, when *timetodeadline* is much lower than the *timeleft*, the *MissRisk* should approach the cost incurred by the (inevitable) deadline miss. Hence, we model the *MissRisk* as:

$$\textit{MissRisk(timeleft)} = C_{\textit{risk}} \times e^{-\frac{\textit{timetodeadline}}{\textit{timeleft}}} \tag{5}$$

Note that as

$$\frac{\textit{timetodeadline}}{\textit{timeleft}} \to \infty, \quad e^{-\frac{\textit{timetodeadline}}{\textit{timeleft}}} \to 0$$

and as

$$\frac{\textit{timetodeadline}}{\textit{timeleft}} \to 0, \quad e^{-\frac{\textit{timetodeadline}}{\textit{timeleft}}} \to 1$$

The constant $C_{\textit{risk}}$ expresses the consequences of the deadline miss in financial terms. As discussed in Section 5.1, the projected *timeleft* is a function of *netlimit* and *disklimit*. That is, $\textit{timeleft} = \textit{task.diskleft}/(\textit{netlimit} - \textit{disklimit})$.

Minimize Total Cost Algorithm

Finding the *netlimit* and *disklimit* that minimize the total cost in Eqn. 2 is an optimization problem. In fact, we need to find the optimal *netlimit* and *disklimit* for each migration task and the solution must be periodically adjusted to reflect incidents like changing workloads and/or arrival/departure of customers. As the cost function of Eqn. 2 is convex, we use the steepest descent numerical solution and more specifically, the *Projected Gradient Descent* method. Such a numerical method may pose a computational overhead on the operation of the scheduler. However, the network of *Brokers* allows us to distribute the computations across all involved *PM*s, while using the selected cost formula allows a fast convergence to the optimum. Our evaluation shows that our approach does not hamper the scalability of the infrastructure as the selected method requires less than 30 ms to converge on a *Intel(R) Core(TM)2 Duo E8400 CPU at 3.00 GHz*.

Alg. 3 is the equivalent to Alg. 1 for the *Cost-driven Broker*; in the context of a single migration task (*task*). Parameters C_{net}, C_{disk}, C_{risk} and p, q are computed periodically at runtime (line 4) by cloud's administration; taking into account SLA failures, the scarcity of the resources consumed during migration, and the impact of violating the migration's time-constraint. Note that this estimation necessitates the communication of the *Broker* at hand with the network of *Brokers* and the *Migrations Scheduler*. In each period, the *Broker* computes the *netlimit* and *disklimit* rates for *task* in lines 5-7. In line 6, the *netlimit* and *disklimit* rates are readjusted based on the gradient of the cost (∇*Cost*), until they converge to the optimum values. Parameter η is the step size for the Projected Gradient Descent method.

Algorithm 3 Cost–Driven *Broker*

Input: *task*: The migration operation
period: Time between monitoring iterations

1: **while** (*task.completed* == **false**) **do**
2: sleep(*period*);
3: *timetodeadline* := *task*.deadline - now;
4: (C_{net},C_{disk},C_{risk}, p, q) := EstimateCostImportance();
5: **while** notConverged **do**
6: (*netlimit, disklimit*) := *Projection*[(*netlimit, disklimit*) $- \eta * \nabla$*Cost(timetodeadline,*C_{net},C_{disk},C_{risk}, *p,q*];
7: **end while**
8: **end while**

Proof on the Convexity

Here we provide the proof on the convexity of Eqn. 2. To avoid clutter, we switch to a more compact notation:

- we use y instead of *disklimit*, x instead of *netlimit*, rr for the *requestedrate*, ttd for *timetodeadline*, and dl for the *task.diskleft*.
- we use $D(y)$ instead of *DiskCost(disklimit)*, $N(x)$ instead of *NetCost(netlimit)*, and $M(x, y)$ instead of *MissRisk(timeleft)*.

Hence, Eqn. 3, 4, and 5, become:

$$D(y) = C_{\textit{disk}} \times (\frac{rr}{y})^q, \quad q \geq 0$$

$$N(x) = C_{\textit{net}} \times x^p, \quad p \geq 1$$

$$M(x,y) = C_{\textit{risk}} \times e^{-\frac{ttd*(x-y)}{dl}}$$

Lemma 1. *Functions $N(x)$, $D(y)$ and $M(x, y)$ are convex for $x \geq 0$, $y \geq 0$ when $p \geq 1$, $q \geq 0$.*

PROOF. For each function, we examine its Hessian:

$$\begin{pmatrix} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial x\, \partial y} \\ \frac{\partial^2}{\partial y\, \partial x} & \frac{\partial^2}{\partial y^2} \end{pmatrix}$$

The Hessian of $N(x)$ is

$$H_1 = \begin{pmatrix} \kappa p(p-1)x^{p-2} & 0 \\ 0 & 0 \end{pmatrix}$$

where $\kappa = C_{\textit{net}}$. The leading principal minors of H_1 are $\kappa p(p-1)x^{p-2}$ and 0. Since $\kappa p(p-1)x^{p-2} \geq 0$ for $x \geq 0$ and $p \geq 1$, H_1 is positive semidefinite and $N(x)$ is convex.

The Hessian of $D(y)$ is

$$H_2 = \begin{pmatrix} \lambda q(q+1)y^{-q-2} & 0 \\ 0 & 0 \end{pmatrix}$$

where $\lambda = C_{\textit{disk}} * rr^q$. The leading principal minors of H_2 are $\lambda q(q+1)y^{-q-2}$ and 0. Since $\lambda q(q+1)y^{-q-2} \geq 0$ for $y \geq 0$ and $q \geq 0$, H_2 is positive semidefinite and $D(y)$ is convex.

The Hessian of $M(x, y)$ is

$$H_3 = \begin{pmatrix} \mu\xi^2 e^{\xi(x-y)} & -\mu\xi^2 e^{\xi(x-y)} \\ -\mu\xi^2 e^{\xi(x-y)} & \mu\xi^2 e^{\xi(x-y)} \end{pmatrix}$$

where $\mu = C_{\textit{risk}}$ and $\xi = -\frac{ttd}{dl}$. The leading principal minors of H_3 are $\mu\xi^2 e^{\xi(x-y)}$ and 0. Since $\mu\xi^2 e^{\xi(x-y)} \geq 0$, H_3 is positive semidefinite and $M(x, y)$ is convex.

□

Theorem 2. *The cost function of Eqn. 2 is convex for* $x \geq 0$, $y \geq 0$ *when* $p \geq 1$, $q \geq 0$.

PROOF. The cost function of Eqn. 2 is the sum of three convex functions based on Lemma 1. Thus, it is a convex function as well.

□

6 EVALUATION

We evaluate our approach in two ways. First, we use the *MigrateFS* prototype to quantify the overheads involved in hosting *VM*s in our file system. For this, we use a real cloud infrastructure setup in our lab using *Xen* 3.2-1 [12] and OpenNebula [19]. The evaluation on real hardware involving a limited number of nodes shows the feasibility and performance of our approach (Section 6.1). Second, we simulate large infrastructures and show the effectiveness of combining *MigrateFS* with our resource management policies. Based the principles discussed in [23], we implemented our own cloud simulator in *Java* using the proposed *priority-based* and *cost-driven* management policies introduced in Sections 5.1 and 5.2 and provide their evaluation in Sections 6.2 and 6.3 below. The simulation allows us to assess the performance of an *IaaS* cloud given that our proposal is used throughout the physical realm. The simulated scenarios involve contemporary hardware under use cases only enabled through our approach in handling live migration. We set key parameter values following an extensive sensitivity analysis. Our simulation-driven experimentation shows how clouds can handle the -potentially extreme- overhead involved in infrastructure–wide load balancing.

6.1 MigrateFS Overheads

The *MigrateFS* prototype is implemented in *C* using *FUSE* [24] and *pthreads*. As source and target hosting nodes of migrating *VM*s we use two physical systems connected with a 1 *Gbps* Ethernet switch. Each physical node is equipped with 8 *GB* of RAM and an Intel(R) Xeon(R) CPU X3220 at 2.40GHz. The *VM* disk images are stored in an *ext3* file system.

File system benchmarking during normal operation – no migration: Using the *Bonnie++* [25] benchmark, we measure the performance overhead introduced by the additional layer of *MigrateFS*. The benchmark is executed within a paravirtualized *VM* featuring 2 *GB* of RAM and a single CPU core. We compare *MigrateFS* against three methods of accessing the virtual disk:

- *Local:* This method routes I/O system calls through the hypervisor's kernel directly to the file system (*ext3*) where the virtual disk image resides.
- *GlusterFS:* Virtual disks are stored in a distributed file system setup with *GlusterFS* [15]. *GlusterFS* is configured to use two nodes in RAID-1 configuration. With this setup, each write operation is performed on the local file system and on the remote node.
- *Mirror:* This method routes I/Os through the hypervisor to a *FUSE* file system. The *FUSE* file system traps and relays all I/O operations to the underlying file system holding the virtual disk images. In this manner, we quantify the penalty our design incurs by trapping I/O.

Figure 3 shows the read, write, and write-after-read, performance of *MigrateFS* as far as block I/O operations are concerned. *Local* shows the performance attained through sole use of local storage resources; in this configuration, migration is not supported. Here, the performance of *MigrateFS* is evidently almost identical to the one we measure for the *Mirror* file system. This shown that the performance overhead introduced by our approach file system is mainly due to its inherent need to trap I/O operations. Implemented at user-space, the *MigrateFS* prototype uses *FUSE* for monitoring I/O activity[2]. The performance impact of trapping I/O operations is noteworthy if compared to a *Local* setup. In light of non-cached operations we measured a performance penalty of 25% and 22% for read and writes respectively. However, such penalties are significantly less than the ones involved in of network storage solutions that enable *VM* migration. The write performance of *GlusterFS* is significantly hampered by the network latency as all files are replicated in a remote physical node (RAID-1 configuration). In this type of I/O call, *MigrateFS* proves superior. In the case of read operations, *GlusterFS* uses caching at the expense of RAM consumption and thus, it surpasses *MigrateFS*.

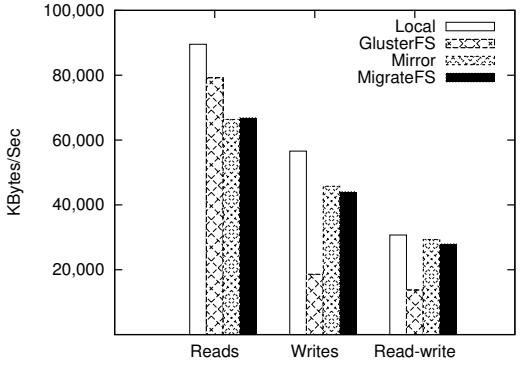


Fig. 3. Comparing block I/O performance of *MigrateFS* to other disk access methods.

Performance during migration: In this experiment, we migrate a virtual disk of 10 *GB* while a process performs write operations on it. The in-*VM* process tries to write blocks at a rate of 20 *MB/sec*. The time constraint we set for the migration is 550 seconds. This forces *MigrateFS* to reduce the disk throughput to 6 *MB/sec* during the first and second disk transfer phases. Figure 4 shows how our approach reduces the disk I/O rate to comply with the designated time constraint. During phase three, each write operation is duplicated in both the target and source *PM*s, while the hypervisor consumes bandwidth to transfer mainly RAM contents. Note that our goal in this experiment is not to measure any potential downtime. This is due to mainly two reasons: a) our work is hypervisor agnostic, while downtime during live *VM* migration is a hypervisor specific property, and b) the downtime is in a different scale (milliseconds)

2. The performance of *FUSE* has improved in versions newer than the one used, yet, these improvements were not available in our testing environment.

from the operations *MigrateFS* performs (minutes). The interested reader can find measurements of downtime for live migration in [12], [26].

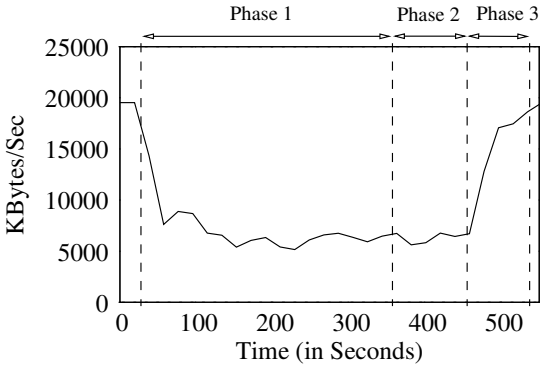


Fig. 4. Limiting virtual disk throughput during migration.

6.2 Priority Based Management of Migration Tasks

Impact of scheduling migrations on SLA failures: In the following two experiments, we simulate an infrastructure made of 500 *PM*s. Each *PM* features a 10 *Gbps* network link towards all other physical nodes. The *VM*s hosted by this physical infrastructure are evenly distributed among the *PM*s. Each *VM* has a 50 *GB* virtual disk and writes into it at a rate of 30 *MB/sec*. This 50 *GB* virtual disk has to be transferred during migration. We choose to ignore the migration overhead involved in copying memory contents as the data volume factor is taken into account through the virtual disk size.

In our simulation we focus only on network bandwidth consumption and we ignore hardware properties that are largely irrelevant to our approach (e.g., CPU architecture). For the disk transfer operation we have the 10 *Gbps* of the *PM* network links at our disposal. However, the operating *VM*s need to use some of the available bandwidth for their normal operation. As we cannot account for the workload of all *VM*s collocated with the migrating ones, we set the available network bandwidth to follow a Gaussian distribution. With this approach we simulate the behaviour of the end physical node and we are not concerned with the network topology.

The evaluation scenario we present here consists of $1,000$ migration tasks. We trigger one migration operation every 100 seconds. Consuming network resources for migrating *VM*s will stress the cloud resources. To quantify the effectiveness of our approach we use two metrics:

- *SLA violations* occur due to network bandwidth shortage. Bandwidth shortage may occur on any randomly selected *PM* as we cannot predict the behavior of each *VM* running. The duration of the resource shortage is set to 300 seconds (five 60 second periods). This shortage (marking an *SLA violation*) is extended for additional periods in case the migration operations consume the entire 10 *Gbps* bandwidth available to the *PM* under stress.
- *The load of the network* is represented by the frequency of violations. That frequency is expressed as the percentage of *PM*s where a shortage occurs within a 60 second period.

Increasing the *load of the network* causes more *SLA violations*. Our goal is to limit the network utilization over stressed links used during migration.

We show how our resource management approach handles migrations with different time constraints. We compare our approach against the currently available cloud setup, denoted as "No Scheduling", where there are no constraints on the resources consumed. The *danger* and *warning* thresholds of Alg. 1 are set to 300% and 400% respectively. The network and disk limits are updated once every 60 seconds (the *period* in Alg. 1).

In Figure 5, we present our policy operating under three different migration completion time values, 200, 350 and 800 seconds, and the "No Scheduling" approach. We measure the periods (y-axis) we observe network SLA violations as we gradually increase the network load from 5% to 25%. High values indicate that we further stress the already limited network resource. The 25% load is an extreme case where in each minute (60 second period), 25% of the *PM*s links fail to satisfy the *VM*'s SLAs and they continue to display high load for the next five minutes.

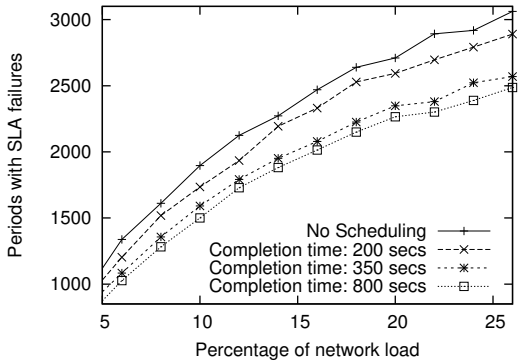


Fig. 5. Evaluating our approach under different time constraints set on migrations.

This experiment shows an important aspect of our work: by limiting the network and disk bandwidth, our policy waits for the "hot" spots (*PM*s) of the infrastructure to cool down before performing the migration. Reducing the acceptable migration time, forces the scheduler to use "hot" *PM*s instead of waiting.

Failing SLAs vs violating time constraints: In this evaluation scenario we reuse the setup of the previous experiment. We have 500 *PM*s, each with a 10 *Gbps* network link. Each *VM* writes into its 50 *GB* virtual disk at a rate of 30 *MB/sec*. We schedule $1,000$ migration tasks, one every 100 seconds, while we assume a Gaussian network traffic.

However, in this experiment, we set the time frame for migration to 350 seconds, we vary the network load, and we report the performance of our policy for three *danger* thresholds, 100%, 150%, 200%. We measure the time constraint violations and the periods where we have stressed network SLAs. The *danger* threshold affects the time at which our scheduler decides that a time constraint is about to be violated and thus the network bandwidth of already saturated *PM*s should be used.

Figure 6 shows how the amount of time constraint violations reduces as we increase *danger*. The reduction of the time constraint violations comes at the expense of

stressing "hot" *PM*s. We expect the administration of the *IaaS*-cloud to set both the *danger* and *warning* thresholds according to the size of the infrastructure, the load it serves, and the requirements in terms of SLA satisfaction and load balancing.

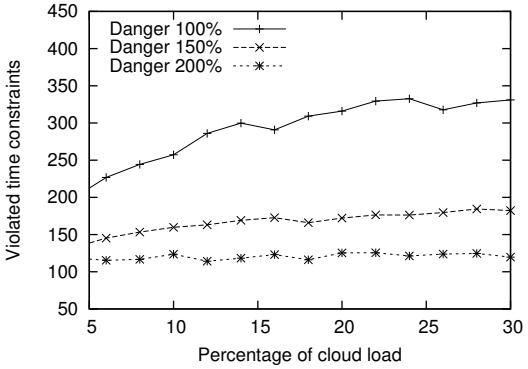


Fig. 6. Time constraint violations as we increase the *danger* threshold property.

Bandwidth shared between two migration tasks: In this experiment, we let two *VM*s of equal size migrate between two *PM*s. As a result, both migration tasks use the same network links and share the same network bandwidth. In total, the bandwidth available to the two *VM*s have is 40 *MB/sec* and each migration should complete in less than $2,500$ seconds. The period of $2,500$ seconds is enough for the two tasks to reach the equilibrium points of resource sharing in this experiment. We expect long migration time constraints to be set for large *VM*s that have to be transferred using limited resources (e.g., nightly load balancing operations).

In Figure 7, we show the throughput consumed by each task during migration. *VM*-0 starts migrating first and thus it uses the entire bandwidth. After 100 seconds the second task starts causing the bandwidth to be shared evenly between the two *VM*s. After $1,000$ seconds we initiate a process inside *VM*-0 that writes on the disk producing dirty blocks. This causes the *danger* threshold to be surpassed on *VM*-0 and thus it gets a greater portion of the bandwidth. At the same time the I/O is limited and thus the rate of the dirty pages is reduced. From $1,000$ to almost $2,400$ seconds *VM*-0 cycles through the following states: it gets a warning and it reduces the available disk bandwidth, but this is not enough to migrate *VM*-0 within the designated time frame so the *danger* threshold is exceeded. Under danger, *VM*-0 consumes some of the bandwidth of *VM*-1. This causes *VM*-0 to instantly exit the warning and danger "zones" and our policy tries to balance the network usage between the two *VM*s. As the constraints on the disk throughput in *VM*-0 are lifted and we further reduce the available bandwidth, the *warning* and *danger* thresholds are surpassed again. This continuous cycle causes the throughput of *VM*-0 to revolve around the average of 28 *MB/sec* and that of *VM*-1 to revolve around 10 *MB/sec*. Finally, the migration of *VM*-1 completes before that of *VM*-0 and *VM*-0 gets the entire bandwidth.

Impact of prioritizing migration tasks on throughput: The *danger* and *warning* thresholds are used to prioritize migration tasks as they reflect when our resource sharing policy will take action to ensure a timely migration. In

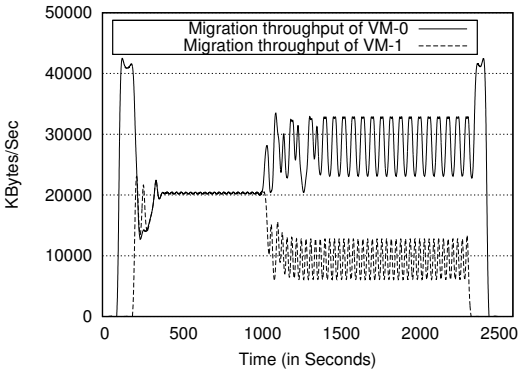


Fig. 7. Throughput of two *VM* migrations sharing the same network links.

this experiment, we have three migration tasks, each one with a different set of *danger* and *warning* thresholds. The *danger* thresholds are 200%, 300%, 400% and the *warning* thresholds are 250%, 350%, 450% respectively for the *VM*s with IDs 0, 1, 2. All migration tasks have the same source and target *PM*s and thus use the same network links and share the same network bandwidth. The migrating *VM*s are idle throughout the migration period.

In Figure 8 we present the network throughput consumed by each migration task over time. All three tasks start in a danger state according to the throughput they have available and their *danger* threshold. In the period of 500 to $1,700$ seconds first *VM*-0, then *VM*-1 and finally *VM*-2 exit the *danger* "zone" leaving more bandwidth available. A time period of equal bandwidth share follows ($1,700$ to $2,200$). After that, the migration tasks finish in the order their thresholds dictate. We do not see the remaining tasks take up the available bandwidth immediately after a migration finishes because as a migration nears its end, it often enters the danger state leading the rest of the migrations to release some of the network resources they occupy. This is because in the third phase of the migration, the network bandwidth consumed is affected by the short block size of the I/O requests.

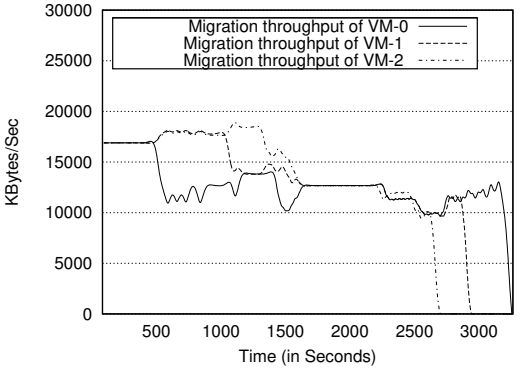


Fig. 8. Prioritize migration requests.

6.3 Cost Driven Management of Migration Tasks

Here, we assume that the infrastructure expresses in financial terms ($) the cost of *a)* missing a specific migration time constraint, *b)* limiting the disk bandwidth of a migrating

VM and *c)* experiencing a lower network bandwidth than the one guaranteed in the corresponding *VM*'s SLA.

We simulate the migration of $2,000$ *VM*s, hosted across $1,000$ *PM*s. The *period* is 60 seconds and a migration starts every 100 seconds. Each *VM* owns a disk of 50 GB and all migrating *VM*s are uniformly distributed across *PM*s. We have a three-level tree network. The lowest level consists of 100 switches, each one serving 10 *PM*s. These switches are networked using 10 switches of the next level and, finally, there is a single root switch. Each switch assumes that a bandwidth of B_m is available for the total migration traffic. B_m is fixed at 1.6 GBps. For each such switch we compute a saturation ratio $S_r = B_a/B_m$, where B_a is the actual bandwidth consumed by the migration tasks routed through that switch. The saturation ratio S_r is computed in every period and is used to specify the coefficient C_{net} of Eq. 4. As the cost of using a network switch is higher when its saturation ratio is high, we let C_{net} be proportional to the sum of the saturation ratios of the switches across the path of the disk transfer. Also, the *NetCost* of Eq. 4 is assumed to increase linearly with the used bandwidth (degree p is 1). The disk cost coefficient C_{disk} of Eq. 3 is chosen randomly in each period based on a uniform distribution. The lower bound of the distribution is fixed at $\$0.5$. The upper bound varies. We use one of the following values for the upper bound, in each experiment: $\$5$, $\$60$, $\$125$, $\$250$, and $\$500$. As in the case of the network cost, the *DiskCost* of Eq. 3 grows linearly with bandwidth use, i.e., q is set to 1. A violated time constraint of a migration incurs a cost (*penalty*) for the cloud provider. Moreover, when a time constraint is violated we compute a new deadline based on the remaining disk size. Similar to the disk cost coefficient, parameter *penalty* is randomly selected using a uniform distribution, with $\$0.5$ as the lower bound and an upper bound with one of the following values: $\$5$, $\$60$, $\$125$, $\$250$ and $\$500$. In fact, in each experiment, we use the same upper bound for the distributions of *penalty* and C_{disk}. C_{risk} of Eq. 5 is proportional to $penalty * e^{-timetodeadline}$. The factor $e^{-timetodeadline}$ increases the value of C_{risk}, as we approach the end of the corresponding migration's time-window.

We compare the cost of serving the $2,000$ migration tasks using the two *Broker* policies: *a)* The *priority-based* resource management with the *danger* threshold set to 100%, 150% and 200%. The *warning* threshold is set to be 50% higher than the *danger* (e.g., for *danger* 150% the *warning* becomes 200%). *b)* The *cost-driven* resource management, denoted by CostAware. The total cost for the $2,000$ migrations, incurred by the violated migration time-constraints, the restrictions in the migrating *VM*s disk rate, and the network congestion, appears on the y-axis, in Fig. 9. The upper bound for the distributions of *penalty* and C_{disk} appears on the x-axis. As Fig. 9 depicts, there is an, at least, 2x improvement for any upper bound on *penalty* and C_{disk}, when using CostAware *Brokers* over *priority-based Brokers*; which are unaware of the actual cost of violated time-constraints and network/disk rate restrictions. In Figs. 10a and 10b, we zoom-in to the total cost plotted in Fig. 9, for CostAware and Danger 150%, respectively. In particular, Figs. 10a and 10b show how much of the total cost is due to network congestion (Network-Cost), disk-rate restrictions (Disk-Cost), and missed time constraints on migrations (Time-Missed-Cost). We plot one stack for each upper bound value of *penalty* and C_{disk} (x-axis). Fig. 10a shows that the effectiveness of CostAware is achieved through the restriction of the dirty rate for migrating *VM*s with costly time constraint violations. Based on the values of the cost formula parameters used in this experiment, stressing the network or violating certain migration time constraints, are less cost-effective options than restricting certain *VM*s' disk bandwidth. The CostAware *Brokers* are able to detect the most cost-effective options and give a substantial improvement over Danger 100%, Danger 150%, or Danger 200%.

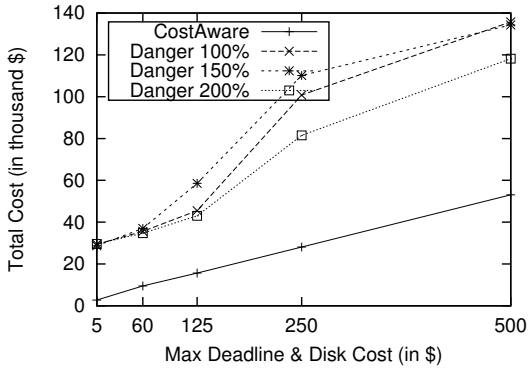
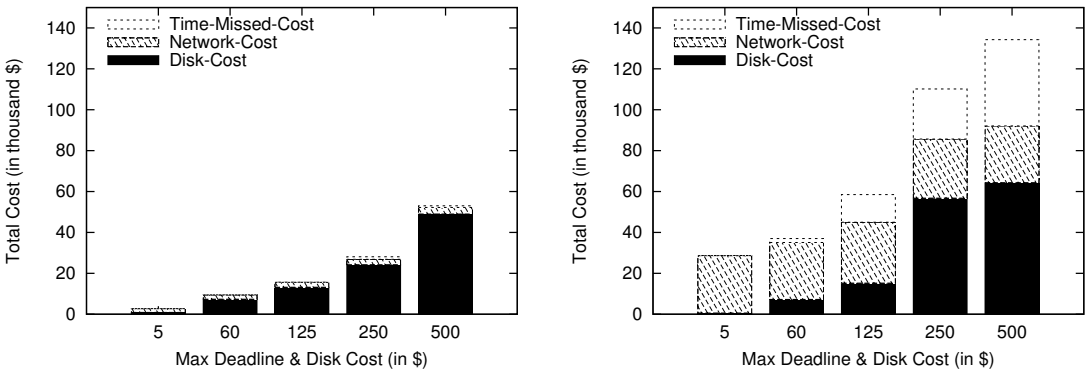


Fig. 9. Cost as we increase the disk and time constraint penalty.

7 Related Work

Live OS migration was shown [3] to have the potential to reduce downtime to tens of milliseconds by first copying RAM content and only afterwards the remaining *VM* state including virtual devices (e.g., CPU, network). *VM*-Monitors (*VMM*s) [12], [13] are not concerned with migrating the *VM* persistence layer (i.e., the "virtual disk(s)"). In this direction, the cloud design may actually assist and its administration can explicitly replicate disk images at the block level [27], [28]. For small to medium-size clouds, designers and administrators can resort to SANs or deploy distributed file systems [15] (*DFS*s). The incremental transfer of virtual systems to different locations presents an alternative that avoids scalability issues often hindering *DFS*s. However, transferring large amounts of data is not always possible [29]. In [7], [8], live *VM* migration in WANs is advocated as a way to relieve overloaded *PM*s. A workload-driven migration of virtual storage is discussed in [30]. *VMware ESX 5.0* [9] is capable of live migrating *VM*s including the persistence layer using *IO Mirroring*; here, all write operations are performed concurrently in both source and target *PM*s during a *VM* migration.

Modelling and predicting the performance of *VM*s during a migration operation, is essential in maintaining the SLA agreed between the cloud provider and the users. The migration impact on *VM*'s performance is examined in [26], where modelling of the migrating behaviour allows accurate predictions in 90% of the cases. While [26] focuses on the hypervisor's performance, our work tries to timely migrate the storage resources as well. Dynamic resource provisioning has been employed in [31] to reduce the rate of SLA violations. Reducing SLA violations and enhancing quality-of-service is the goal in [32]; a multi tier hybrid storage is assumed in that context. Compared to [31], [32]



(a) CostAware resource management.

(b) Cost of the priority based policy.

Fig. 10. Evaluation of the financial cost under the two *Broker* policies.

our work assumes the existence of a resource allocation target and works towards reaching it.

Autonomic and load-balancing systems are often employed to schedule *VM* migrations. In such systems [33]–[37] the performance of the cloud is continuously monitored and proper action is taken to handle bottlenecks and resource shortages. Our approach can work in tandem with such autonomic systems and can extend their effectiveness. Time-warranties are typically used to enhance the quality of service offered by real-time systems [38], [39]. I/O throttling during *VM* migration enables for time and performance warranties and ultimately for migration completion. *VMware ESX 5.0 IO Mirroring* [9] reduces the transfer rate to the slowest medium of the source or target *PM* and guarantees migration convergence.

MigrateFS complements the above approaches by providing both disk I/O and network throttling while synchronizing on demand virtual disk images across *PM*s. The feasibility of on-demand disk synchronization is shown in [7], [8], [40] where *VM*s are live migrated over WANs and AZs. Similarly to *DFS*s, *MigrateFS* provides a *POSIX* API, yet, it differs in that data (i.e., *VM* disk images) are not available to all nodes; rather, they can be pair-wise synchronized on-demand. This synchronization places no limits on scaling and so, our *MigrateFS*-based approach becomes an attractive choice for live-migration in large share-nothing clouds. Also, as there is no single point of failure in *MigrateFS*, we introduce no discernible performance bottlenecks. Our work presents a comprehensive, scalable and distributed *VM* migration approach that is predominantly weaved around time-constraints. The above features set our work apart from the related efforts [7]–[9].

8 Conclusions

Migrating *VM*s in live fashion is of key importance to *IaaS*-clouds as it helps accomplish major operational and administrative objectives including effective load-sharing and improved utilization of physical machinery. The movement of *VM*s over the network inevitably consumes significant cloud resources, thus such tasks should be scheduled during periods of low load. In this work, we focus on emerging highly-scalable share-nothing cloud installations and employ on-demand virtual disk synchronization across *PM*s to attain live migration under explicit time-constraints. Our approach is empowered by the combined use of a network of *Brokers* and the *MigrateFS* file system. *MigrateFS* effectively synchronizes disk images between physical computing systems, while the *Brokers* manage the resources of the share-nothing cloud elements. The joint objective of the two components is to offer a scheme that gracefully deals with time-constrained *VM* migration requests and at the same time, does not deplete cloud resources.

The resource management policies we developed apply on both clouds with uniform *SLA*s across *VM*s and clouds with widely varying *SLA*s. Our lightweight priority-based policy adjusts the network and virtual disk bandwidth in time using a simple, yet effective, protocol. In cases where the cloud provider needs to differentiate among the operation of different *VM*s, our cost-driven policy offers a general model to capture different costs and intelligently adjust the network and disk rates.

Our prototype experimentation demonstrates the I/O performance gains compared to network storage solutions, and the significantly reducing SLA violations due to heavy network traffic. Moreover, the extension of the cost-driven policy offers a 2x improvement in the cases where it applies. In the future, we plan to examine statistics-driven *VM* migration scheduling algorithms in settings consisting of interoperating *IaaS*-clouds and SDNs and investigate how our approach can be applied to *PaaS*-clouds so that real-time service migration is realized. We also intend to extend the SLA monitoring mechanism of our approach to take corrective action not only when the network is stressed but also when disk I/O spikes [41]. Finally, we shall investigate how our approach fares in conjunction with a resource–reallocation algorithm capable of determining *VM*s requiring migration so that cloud efficiency rates are further improved.

Acknowledgments

A preliminary version of the paper appeared in [42]. This work has been partially supported by *i-Marine* and *Sucre* EU FP7 projects as well as ERC Starting Grant # 279237.

References

[1] C. Weng, M. Li, Z. Wang, and X. Lu, "Automatic Performance Tuning for the Virtualized Cluster System," in *Proc. of the 29th IEEE*

Int. Conf. on Distributed Computing Systems, Montreal, Canada, June 2009.
[2] VMware, "VMware DRS - Dynamic Scheduling of System Resources," www.vmware.com/products/drs/overview.html, Oct. 2009.
[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proc. of the 2nd Symposium on Networked Systems Design & Implementation*, Boston, MA, May 2005.
[4] Z. Liu, W. Qu, W. Liu, and K. Li, "Xen Live Migration with Slowdown Scheduling Algorithm," in *Proc. of the 2010 Int. Conf. on Parallel and Distributed Computing, Applications and Technologies*, ser. PDCAT '10, Wuhan, China, Dec. 2010, pp. 215–221.
[5] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation," in *Proc. of the 1st Int. Conf. on Cloud Computing (CloudCom'09)*, Beijing, China, Dec. 2009.
[6] Y. Luo, B. Zhang, X. Wang, Z. Wang, Y. Sun, and H. Chen, "Live and Incremental Whole-System Migration of Virtual Machines Using Block-Bitmap," in *Proc. of IEEE Int. Conf. on Cluster Computing*, Tsukuba, Japan, September 2008.
[7] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live Wide-Area Migration of Virtual Machines Including Local Persistent State," in *In VEE '07: Proc. of the 3rd Int. Conf. on Virtual Execution Environments*, San Diego, CA, June 2007.
[8] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, "CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines," *SIGPLAN Not.*, pp. 121–132, March 2011.
[9] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai, "The Design and Evolution of Live Storage Migration in VMware ESX," in *Proc. of the 2011 USENIX Annual Technical Conference*, Portland, OR, 2011.
[10] D. Josephsen, *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ: Prentice Hall PTR, 2007.
[11] VMware, "vSphere," www.vmware.com/products, May 2012.
[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP*, New York, NY, 2003, pp. 164–177.
[13] Kernel Based Virtual Machine, "www.linux-kvm.org," May 2012.
[14] Red Hat, "Global File System," http://www.redhat.com/gfs/, May 2012.
[15] ——, "GlusterFS," www.gluster.org, May 2012.
[16] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments," in *40th Int. Conf. on Parallel Processing*, Taipei, Taiwan, Sept. 2011.
[17] P. Pradeep, H. Kai-Yuan, S. K. G., Z. Xiaoyun, U. Mustafa, W. Zhikui, S. Sharad, and M. Arif, "Automated Control of Multiple Virtualized Resources," in *Proc. of the 4th ACM European Conf. on Computer Systems*, Nuremberg, Germany, 2009.
[18] OpenStack, "www.openstack.org," May 2012.
[19] OpenNebula, "www.opennebula.org," May 2012.
[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. of the ACM SIGCOMM Conf.*, Seattle, WA, August 2008.
[21] V. Mann, A. Vishnoi, A. Iyer, and P. Bhattacharya, "VMPatrol: Dynamic and automated QoS for virtual machine migrations," in *8th Int. Conf on Network and Service Management (CNSM)*, Las Vegas, NV, USA, Oct. 2012, pp. 174–178.
[22] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, "Remedy: Network-aware Steady State VM Management for Data Centers," in *11th Int. IFIP TC 6 Conference on Networking*, Prague, Czech Republic, May 2012, pp. 190–204.
[23] R. Jain, *The Art of Computer Systems Performance Evaluation: Techniques for Experimental Design, Measurement, Simulation and Modeling*. New York, NY: John Wiley & Sons, 1991.
[24] FUSE, "Filesystem in Userspace," fuse.sourceforge.net, May 2012.
[25] R. Coker, "Bonnie++ file system benchmark," www.coker.com.au/bonnie++, May 2012.
[26] S. Akoush, , R. Sohan, A. Rice, A. W. Moore, , and A. Hopper, "Predicting the Performance of Virtual Machine Migration," in *Proc of 2010 IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Miami, FL, Aug. 2010.
[27] "Disrtibuted Replicated Block Device," www.drbd.org, Sept. 2012.
[28] D. T. Meyer and B. Cully, "Block Mason," in *Proc. of the First Workshop on I/O Virtualization*, ser. WIOV '08, San Diego, CA, Dec. 2008.
[29] J. Taheria, A. Y. Zomayaa, H. J. Siegelb, and Z. Taric, "Pareto frontier for job execution and data transfer time in hybrid clouds," *Future Generation Computer Systems*, vol. 37, pp. 321–334, 2014.
[30] J. Zheng, T. Ng, and K. Sripanidkulchai, "Workload-Aware Live Storage Migration for Clouds," in *Proc. of the 7th ACM Int. Conf. on Virtual Execution Environments*, Newport Beach, CA, 2011.
[31] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic Placement of Virtual Machines for Managing SLA Violations," in *Proc. of 10th IFIP/IEEE Symp. Integrated Management*, Munich, Germany, May 2007.
[32] J. Tai, B. Sheng, Y. Yao, and N. Mi, "Live Data Migration For Reducing SLA Violations In Multi-tiered Storage Systems," in *Proc. of 2014 Cloud Engineering Int. Conf.*, Boston, MA, March 2014.
[33] A. Danak and S. Mannor, "Resource Allocation with Supply Adjustment in Distributed Computing Systems," in *Proc. of the 30th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, Genoa, Italy, June 2010.
[34] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis, "Nefeli: Hint-based Execution of Workloads in Clouds ," in *Proc. of 30th IEEE Int. Conf. Distributed Computing Sytems (ICDCS)*, Genoa, Italy, June 2010.
[35] R. Calheiros, R. Ranjan, and R. Buyya, "Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments," in *Proc. of 2011 Parallel Processing Int. Conf.*, Taipei, Taiwan, Sept 2011, pp. 295–304.
[36] M. Menzel, R. Ranjan, L. Wang, S. U. Khan, and J. Chen, "CloudGenius: A Hybrid Decision Support Method for Automating the Migration of Web Application Clusters to Public Clouds." *IEEE Trans. Computers*, vol. 64, no. 5, pp. 1336–1348, 2015.
[37] M. Jayasinghea, Z. Taria, P. Zeephongsekulb, and A. Y. Zomayac, "Task assignment in multiple server farms using preemptive migration and flow control," *Journal of Parallel and Distributed Computing*, vol. 71, no. 12, p. 1608–1621, 2011.
[38] S. Biyabani, J. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real–Time Scheduling," in *Proc. of the Real–Time Systems Symposium*, Huntsville, AL, Dec. 1988.
[39] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," in *Proc. of the IEEE ICMCS*, Florence, Italy, June 1999, pp. 400–405.
[40] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon, "Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads," in *Proc of the ACM Symposium on Cloud Computing 2016 (SoCC'16)*, Santa Clara, CA, Oct. 2016.
[41] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Monitoring Elastically Adaptive Multi-Cloud Services," *IEEE Trans. on Cloud Computing*, 2015, DOI: 10.1109/TCC.2015.2511760.
[42] K. Tsakalozos, V. Verroios, M. Roussopoulos, and A. Delis, "Time-Constrained Live VM Migration in Share-Nothing IaaS-Clouds," in *7th IEEE Int. Conf. on Cloud Computing (CLOUD 2014)*, Alaska, USA, June 2014.

Konstantinos Tsakalozos is a Member of the Technical Staff at *Canonical* Ltd. and has been a a Software Engineer with *Microsoft* in London, United Kingdom. His research interests are in cloud computing, distributed systems and software engineering. He holds a PhD in Computer Science from the University of Athens.

Vasilis Verroios is a doctoral candidate in Computer Science at Stanford University in Stanford, CA. His interests are in data management, machine learning, distributed systems and crowd-sourcing. He holds both MS and BS degrees in Computer Science from the University of Athens.

Mema Roussopoulos is an Associate Professor of Computer Science at Univ. of Athens in Athens, Greece. Her research interests are in networking and distributed systems. She received her PhD in Computer Science from Stanford University.

Alex Delis is a Professor of Computer Science at Univ. of Athens in Athens, Greece. His interests are in cloud computing, management of data and distributed systems. He holds a PhD in Computer Science from the Univ. of Maryland at College Park.