2010 39th International Conference on Parallel Processing

Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration

Jason Sonnek, James Greensky, Robert Reutiman and Abhishek Chandra
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{sonnek, greensky, reutiman, chandra}@cs.umn.edu

***Abstract*—Virtualization is being widely used in large-scale computing environments, such as clouds, data centers, and grids, to provide application portability and facilitate resource multiplexing while retaining application isolation. In many existing virtualized platforms, it has been found that the network bandwidth often becomes the bottleneck resource, causing both high network contention and reduced performance for communication and data-intensive applications. In this paper, we present a *decentralized affinity-aware migration* technique that incorporates heterogeneity and dynamism in network topology and job communication patterns to allocate virtual machines on the available physical resources. Our technique monitors network affinity between pairs of VMs and uses a distributed bartering algorithm, coupled with migration, to dynamically adjust VM placement such that communication overhead is minimized. Our experimental results running the Intel MPI benchmark and a scientific application on a 7-node Xen cluster show that we can get up to 42% improvement in the runtime of the application over a no-migration technique, while achieving up to 85% reduction in network communication cost. In addition, our technique is able to adjust to dynamic variations in communication patterns and provides both good performance and low network contention with minimal overhead.**

I. Introduction

The emergence of cloud computing (e.g., Amazon EC2 [1]) has led to a growing interest in deploying a wide variety of applications [2], [3], [4] on shared computing environments. In particular, because of the relative abundance of resources and low cost of resource outsourcing, clouds are highly attractive for compute-intensive applications [5], [6]. The success of clouds has been driven in part by the use of virtualization as their underlying technology. Virtual machines (VMs) provide flexibility and mobility through easy migration, which enables dynamic mapping of VMs to available resources. Virtual machines also provide performance isolation and security that facilitates multiplexing and utilization of shared resources. For these reasons, virtualization has also become popular in other domains such as scientific and high-performance computing [7], [8], [9], [10].

A virtualized computing platform provides an abstraction of a "pool of resources" where different application components (or jobs[1]) can be placed on any resource. This property has been exploited for executing "embarrassingly" parallel or largely independent bag-of-tasks applications [10], [5] in virtual computing environments. However, several compute-intensive applications in the scientific and data analytics domains have intricate patterns of communication and data dependencies, requiring exchanges of large amounts of data and state for carrying out their computation. For such applications, the communication patterns between different components are a key factor that must be considered during resource allocation. However, for most cloud environments, such information is not readily available to the infrastructure provider, and the volume of traffic exchanged between any two VMs is both job-dependent and time-varying, so VMs are largely placed on servers based solely on available capacity.

At the same time, in many existing virtualized platforms, it has been found that the network bandwidth often becomes the bottleneck resource. This is because the physical topology in a large-scale computing platform typically has a hierarchical structure [11]: a given pair of compute nodes may be located on the same rack, may be part of the same cluster sharing a common LAN, or may be on separate clusters communicating through a slow link (e.g., a wide-area link for distributed computing platforms). Because of the high speeds and large number of CPU cores sitting on each rack, cluster, etc., their interconnect switches and links become bottlenecks [12], reducing the capacity of the infrastructure to support more applications. As a result, it is in the interest of the cloud provider to reduce the network overhead as much as possible. This implies that in addition to considering the physical characteristics (CPU speed, memory and storage) of nodes on which virtual machines are placed, the network topology must also be considered in order to increase the efficiency of the platform, and reduce network contention.

The goal of this work is to improve application performance in the presence of data dependencies and communication patterns, while reducing the network communication cost

[1]In the rest of this paper, we will use "job" to mean an independently executable application component encapsulated within a VM.

0190-3918/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPP.2010.30

imposed on the underlying platform. Towards this end, we present *Starling*[2]*: a decentralized affinity-aware migration* technique that incorporates heterogeneity and dynamism in network topology and job communication patterns to allocate virtual machines on the available physical resources. Intuitively, by placing two co-communicating VMs as close to one another as possible within the hierarchy, we can reduce network transfer costs and improve performance. Thus, our technique attempts to place two heavily data-dependent or communicating VMs as close to each other as possible (same node, rack, cluster or local network), in order to reduce traffic over bottleneck network links, while improving the overall performance of the applications. This paper makes the following contributions:

• *Affinity-based virtual machine placement and migration:* While several existing virtual machine placement and migration techniques [11], [13] employ resource usage and load information, they do not consider affinities between VMs in making these decisions. Our technique explicitly incorporates inferred job dependency information along with the underlying network topology information to make placement and migration decisions. In addition, while most existing VM management algorithms focus only on load-balancing or consolidation as their objective, our technique also enables better application performance without adversely impacting these system-level goals.

• *Implicit inference of dynamic job dependencies:* Our technique infers the communication/data dependencies between different jobs of an application by monitoring the network traffic between different pairs of VMs in a non-intrusive manner. It does not require any workflow or data dependency information to be provided explicitly by the application, and can also infer changes in these dependencies during the execution of the application. This technique could be especially valuable in cloud computing or other outsourced computing environments, in which obtaining application profiles is especially challenging since the the application is typically a black box from the perspective of the platform provider.

• *Decentralized control:* Another key feature of our technique is that we have implemented it in a completely distributed manner. We use a *distributed bartering* algorithm in which physical servers independently negotiate affinity-based VM relocations on the basis of local information. The decentralized nature of our technique makes it easier to scale to large-scale systems with thousands of nodes, and would be particularly amenable to use in computing environments distributed across a wide area (such as distributed Grids or clouds). At the same time, such a decentralized approach enables the use of localized and diverse policies for resource allocation, rather than having a single, centralized system-wide policy.

We have implemented our affinity-based migration technique on a 7-node Xen cluster. Our experiments with the Intel MPI benchmark suite [14] and a scientific simulation-based application [15] show that we can get up to 42% improvement in the runtime of the application over a no-migration technique, while achieving up to 85% reduction in network communication cost. In addition, our technique is able to adjust to dynamic variations in communication patterns and provides both good performance and low network contention with minimal overhead.

[2]Starlings are communicative birds that tend to flock together in large, close-knit groups.

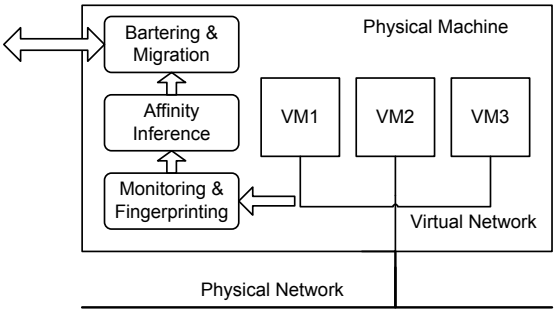

Fig. 1. System Architecture.

II. System Model and Architecture

We consider our system model to consist of a virtualized computing platform, in which physical servers are connected to each other in a hierarchical network topology. For instance, the topology may consist of server clusters connected through a bottleneck link, and servers within a cluster further partitioned into racks, so that the inter-server bandwidth is dependent on whether the servers are on the same rack, within the same cluster or on different clusters. In general, any hierarchical network topology can be considered.

For our application model, we consider an application consisting of multiple computational jobs that may have various data dependencies and communication patterns between them, arising due to reasons we discuss in detail in the next section. These data and communication dependencies in general can be thought of as forming a communication graph. However, we do not require such a communication graph to be explicitly provided or known a priori. We assume that each computational job can be encapsulated within a virtual machine, and the VMs can be placed and/or migrated to any physical server in the system, though the cost of migration would depend on the location of the VM in the network hierarchy.

Figure 1 shows the architecture of our proposed affinity-aware migration algorithm. This algorithm is completely distributed and runs on each node in the system. It consists of the following main components:

- *Traffic Monitoring and Fingerprinting:* Monitors the traffic flowing into/out of each VM hosted on a node, and keeps track of this traffic over time by fingerprinting the traffic volume in a succinct way.
- *Affinity Inference:* Determines the affinities between each VM running on a node and the other VMs that it communicates with. Note that the VM pairs being fingerprinted could be physically located on the same node or across nodes.

- *Bartering and migration:* Based on the affinity inference between different VM pairs, it negotiates a better placement for its VMs (if needed) or responds to such negotiation requests from other nodes. Through such negotiations, it migrates VMs which exchange a large volume of network traffic closer to each other.

III. Traffic Monitoring and Affinity Inference

First, we want to infer the dependencies between different jobs running within the VMs, and in particular the dependency of a job on various physical servers it is interacting with. A compute job may have high communication dependency on a physical server for two main reasons:

- The job may require data which is generated by another job located on the server. This kind of dependence is common in many scientific applications, especially those that require frequent coordination and data exchanges amongst the jobs (i.e., not embarrassingly parallel). Examples include applications parallelized via data partitioning but needing data exchanges, or multi-step simulations that need jobs to synchronize and exchange data/state at each step.
- The job may require data stored on the server for carrying out its computation. This kind of dependence is common in data-intensive applications, which perform computations on large quantities of data. The physical server in this case may be a file server, but may also support executing compute jobs. This paradigm is particularly common in data-intensive computational frameworks such as MapReduce [16] or Hadoop [17], in which block storage on each physical machine is abstracted into a global distributed storage system, and the data in this global store is utilized by computing jobs which are distributed amongst the physical machines.

The primary goal of the monitor is to capture such dependencies between pairs of jobs, as well as between jobs and physical servers. Given this information, we can infer that a set of jobs is co-communicating, or that a job is dependent on data hosted on a certain physical machine, so that they can be placed closer to each other.

The monitoring service observes all incoming and outbound traffic on a physical server in real-time, capturing statistics about traffic source, destination and volume. In our Xen-based [18] implementation, traffic statistics are obtained by dynamically filtering streams obtained from tcpdump running in dom0. Traffic information is represented in the form of a set of $< sourceID, destinationID, volume >$ tuples, where the source ID and destination ID are the IP addresses of VMs, and the volume is the amount of traffic exchanged between them over a time window W. Note that we also record traffic sent directly between VM i and a physical server j to handle the case where applications use data stored in a distributed storage system [16], [17], but do not refer to it explicitly in this discussion for ease of exposition, and rather assume all communication is between pairs of VMs.

To capture dynamic changes in traffic volume between two VMs over time, we maintain volume as an exponential average over its past values:

$$volume[t] = \alpha \cdot volume[t-1] + (1-\alpha) \cdot traf[t],$$

where, α is the averaging constant with a value between 0 and 1, $volume[t]$ and $volume[t-1]$ are the volumes computed for windows at time t and $t-1$ respectively, and $traf[t]$ is the traffic measured at time t.

The *communication fingerprint* CFP_i of a VM i is then defined as a vector of its traffic tuples to all other VMs:

$$CFP_i[j] = volume_{i,j} \forall j \neq i,$$

where, $volume_{i,j}$ is the traffic volume between VMs i and j. Intuitively, the communication fingerprint of a VM specifies its communication patterns and traffic volumes to the other VMs, and corresponds to the data dependencies and communication affinities of the job running inside the VM. Using the communication fingerprint of a VM, we can infer a job's data dependencies, and identify instances in which the job could benefit from relocation to another physical server. In addition, these communication fingerprints can be aggregated in a number of interesting ways to infer the overall network traffic flow, as well as the topology of the network, details of which can be found in a technical report [19].

IV. Affinity-aware Bartering and Migration

In this section, we present our affinity-aware bartering and migration algorithm, that combines the job dependency information with the network topology information to minimize communication overhead. We begin by formulating this placement problem as an optimization problem, and then present a distributed bartering algorithm that dynamically migrates VMs to move the system towards this optimal placement.

A. Optimal Affinity-Aware Placement

To achieve an optimal placement which minimizes the network communication overhead between different VMs, our goal can be specified as the following optimization problem:

$$Minimize \sum_{i,j} CC_{i,j}$$

$$s.t. |VM_m| \leq C_m \forall \text{ physical servers } m.$$

Here, $CC_{i,j}$ is the network communication cost (defined below) between two VMs i and j, and C_m is the maximum number of VMs that can be hosted on the server m. The value of C_m would depend on the total server capacity and VM resource usage requirements (in terms of CPU, memory, disk, etc.). Intuitively, the formulation above says that the VMs should be placed on the physical servers in a manner that minimizes the inter-VM communication cost, while meeting the server capacity constraints.

The *network communication cost* $CC_{i,j}$ can be defined as the time it takes for two VMs i and j to communicate and exchange data with each other. This cost can be thought of as

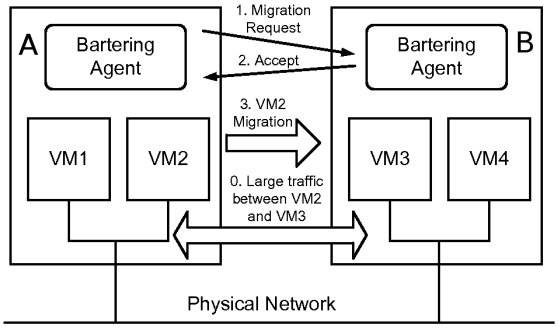

Fig. 2. Distributed Bartering Algorithm: In this scenario, VM2 and VM3 have a high traffic volume, leading to (1) the bartering agent on A sending a migration request to B, (2) acceptance of request because of available capacity, and (3) migration of VM2 to B.

the component of an application's total runtime, corresponding to the network communication between jobs encapsulated within VMs i and j. The communication cost is a function of the traffic volume $volume_{i,j}$ between the two VMs and the bandwidth $BW_{i,j}$ available between them, which depends on the physical servers that the two VMs are running on. The traffic volume can be computed using the communication footprints described in Section III, while the bandwidth can be determined from the network topology information[3]. We compute the communication cost as:

$$CC_{i,j} = \frac{volume_{i,j}}{BW_{i,j}}, \quad (1)$$

so that two VMs placed close to each other (on the same server, same rack, etc.) would have a smaller cost for the same amount of traffic compared to two VMs located far away (on different servers, different racks, etc.). Furthermore, besides improving an application's performance, minimizing the communication cost will also reduce the network overhead of the underlying infrastructure by moving traffic from bottleneck links to high bandwidth links.

B. Distributed Bartering Algorithm

The optimization problem described above is an instance of the graph partitioning problem, which is known to be NP-complete [20]. As a result, heuristics are typically employed to solve such problems. However, many existing heuristics are centralized and assume prior information about the job dependencies, which are also assumed to be static. To minimize overhead and increase the scalability of our solution, we propose a *distributed bartering algorithm* that allows VMs to negotiate placement on a physical server that is closer to the data they require. This algorithm avoids bottlenecks and central points of failure associated with a centralized solution, and can also adapt more quickly to dynamic local changes in job dependency patterns and network topology. In addition, such a decentralized approach enables the use of different policies by different agents based on their location as well as application-specific requirements of the VMs they are managing.

Each physical server runs a *bartering agent* which analyzes network traffic fingerprints and topology information, and determines when a job could benefit from relocation. When the total traffic between a given virtual machine and physical server exceeds some threshold, the bartering agent negotiates a relocation with a desired server and initiates a migration if successful. Next, we describe how relocation negotiations are carried out between different servers and how the migration eventually takes place.

1) Negotiating a Migration: Figure 2 illustrates the bartering algorithm, which works as follows. The bartering agent on a physical server (e.g., server A in Figure 2) periodically checks the communication footprint CFP_i for each VM hosted on that server to determine if the VM's inter-server traffic (traffic being sent to another physical machine, such as server B in Figure 2) exceeds intra-server traffic (traffic being sent to other VMs hosted on the same physical server). If this is the case (e.g., assume VM2's traffic to B is higher than its intra-server traffic), and the total volume of traffic exceeds migration thresholds (computed as discussed below), the bartering agent (on A) will attempt to negotiate a new home by sending a migration request to the physical server (B) receiving the most traffic from the VM. In the following, we will use Figure 2 to refer to the requesting server (A), the target server (B) and the VM to be migrated (VM2).

Upon receipt of the migration request, the bartering agent on B will take one of the following actions:

1) If B has capacity available to host VM2, it will send an 'accept' response to A. In this case, the bartering agent on A will initiate live migration of VM2 to B.
2) If B does not have capacity available, it will return a list of *swap candidates*: a subset of its hosted VMs suitable for possible swapping with VM2. If the agent on A finds a desirable swap candidate (say, VM4), then the swap will be carried out with VM2 migrating to B and the selected swap candidate (VM4) migrating to A. The choice of swap candidates and the swapping mechanism is discussed in more detail below.
3) If there are no suitable swap candidates, then B returns a list of *neighbors*: nodes which are nearest to it in terms of network bandwidth/latency. In this case, the bartering agent on A will contact each of the neighbors in turn by recursively initiating the relocation process. To limit the number of attempts, each node is contacted at most once. Also, the bartering agent will only migrate a VM to a neighbor if the neighbor has significantly (e.g., factor of 10) higher bandwidth to the desired server (B) than the original host (A). If none of the neighbors which meet the migration criteria can host the VM, the server will give up the migration attempt.

To avoid unnecessary migrations and oscillations due to transient communication patterns, each bartering agent uses an *inertia factor* > 1, so that a migration is attempted

[3]Network topology inference is beyond the scope of this work, and we assume that the topology of the network is known.

only if the inter-server traffic exceeds the intra-server traffic by inertia factor. To avoid deadlocks (where two machines simultaneously send requests to each other and wait for each other's response), as well as to avoid redundant migrations arising due to race conditions (e.g., VMs 2 and 3 being migrated to each other's servers simultaneously), a bartering agent only allows a single ongoing relocation transaction at any point of time at its server. The requesting agent selects a random wait-time based on an exponential backoff before retrying a request that has been denied due to an ongoing transaction. The agents also maintain timers to timeout long-standing requests that have not received a response.

2) VM Swapping: As described above for Case 2 in the bartering algorithm, in some cases, a VM migration may be desirable even if the destination machine is full. For instance, considering the setup shown in Figure 2, assume that each server can fit only 2 VMs each. In this case, to enable migration of VM2 to machine B, the bartering algorithm allows *VM swapping*, where the bartering agents on the two machines swap VMs to honor the resource limitations at each server. This swapping is done only if the swap will result in a better placement of the VMs in terms of their communication affinities.

There are two decisions involved in the swapping process:

- *Selecting swap candidates:* When a bartering agent on a full server receives a migration request (e.g., machine B from A for VM2), it needs to select a set of local VMs for possible swapping if they exist. There are two kinds of VMs that are possible swap candidates in this case:

  Type1 : A VM that will benefit by moving to A from B. This could be a VM whose traffic to machine A is higher than its intra-machine traffic and may have tried migrating to A unsuccessfully in the past. A could also be in the neighbor list of such a VM's desired server, so that it may benefit from moving to A anyway.

  Type2 : An isolated VM, i.e., one which does not communicate with any other VMs (whose inter-VM communication to *any* VM in the system is below an *isolation threshold*). Intuitively, such a VM would not be affected irrespective of where it is placed.

  Each agent maintains a list of such swap candidates running on its machine, and sends the appropriate ones depending on the requesting server.
- *Making a swap decision:* Once the requesting server receives the list of swap candidates, it needs to select one of them. If there are multiple swap candidates, then the requesting server will give higher preference to type 1 candidates, picking the one with the maximum differential in inter-vs.-intra-machine traffic. In this case, it will also ensure that the preferred swap candidate in fact will end up with higher intra-machine traffic after the swap. This check is needed to ensure an overall benefit from the swap. For instance, in the example above, VM2 may be communicating a lot with both VM3 and VM4. In that case, swapping VM2 and VM4 would not make any difference in the overall traffic while incurring the unnecessary overhead of migration. A type 2 candidate is selected only if no type 1 candidate exists. In addition, since migration is not free, we also use a *swap inertia parameter* to weigh the cost of swapping, so that we avoid swapping when it results in only minor benefit.

The swap is done by triggering each migration in succession, so that temporarily one of the machines will be overloaded, however, this enables synchronized swaps with each VM in a consistent state between any migrations. In some cases, the use of temporary servers may be needed to avoid overloading the servers participating in the swap [13], but we do not consider this scenario in our implementation.

3) Computing Migration Thresholds: As mentioned above, to ensure that the application will benefit from a VM migration, we must ensure that the performance benefit due to the relocation outweighs the cost of migration. This can be accomplished by computing the migration cost (in time) for a VM from one server to another, and comparing it to the communication cost savings that would be achieved due to the relocation in terms of reduced application runtime.

A VM's migration cost is typically dependent on its memory size as well as the bandwidth of the link over which migration has to take place. Thus, for a given VM and target physical server, we can compute a migration cost estimate proportional to the VM's memory size and inversely proportional to the bandwidth of the link connecting the host physical server to the destination server. Using this estimate, we can determine the *migration threshold*: the traffic volume a VM must be exchanging with the destination server within each measurement window in order to benefit from relocation to that server. Intuitively, this migration threshold will be higher for bigger VMs and over slow, bottleneck links, preventing costly migrations until they provide a large enough benefit. Note that the actual downtime of a VM could be reduced further by techniques such as pre-copying of inactive memory pages [21]. At the same time, migration over WAN without shared storage may require copying of local storage as well [22], resulting in much higher migration cost and thus much larger migration thresholds (proportional to storage size).

V. Evaluation

A. Experimental Setup

We conducted our experiments on a 7-node cluster, where each node is a 2xdual-core 2800 MHz AMD Opteron Processor 2220 with 4 GB RAM and 250 GB disk space, and the nodes are connected via Gigabit Ethernet. Each node runs Xen 3.2.1, with the Dom0 and the DomU's running Debian Etch Linux kernel 2.6.18-6-xen-amd64 for one set of experiments (Intel MPI benchmarks) and running Ubuntu 8.04 Linux kernel 2.6.24-24-xen for the second set of experiments (Cube Application), due to library dependencies. Due to space constraints, we present a subset of results, and more detailed results can be found in a technical report [19].

1) Benchmarks and Application: We have used the following benchmarks/applications for our evaluation:

- The *Intel MPI benchmark suite* [14] consists of multiple MPI-based benchmark programs that can test different communication patterns. Examples of such benchmarks include *Ping-pong* which sets up pairwise communication between sets of processes, *Exchange* which sets up a chain of processes doing two-way communication, *Scatter-gather* which carries out scattering and gathering of data among a set of randomly placed processes, *All-to-all* in which each process sends and receives data from all other processes, and *Broadcast* where a root process broadcasts data to all other processes.
- *Cube MHD Jet* (Cube) [15] is an astrophysics application that performs numerical magnetohydrodynamic (MHD) simulations: it simulates a jet of plasma travelling through a magnetic field in three-dimensional space. It consists of multiple computational processes communicating intermediate state with each other. The application can be configured to communicate in arbitrary grid patterns depending on how many nodes are available.

In our experiments, we place the processes for each benchmark/application within VMs which are assigned to physical machines. The goal is to have the VMs be placed in a communication-optimal manner.

2) Network Topology Setup: We create a logical network hierarchy on top of 6 nodes in our cluster. Our hierarchy consists of two logical clusters each consisting of 3 machines. We used the Linux token bucket packet scheduler to rate control the traffic going between different sets of machines. Another node in our physical cluster is set up as an NFS server to export the file system for the cluster, which is needed for migration purposes. Since the benchmarks and applications used in our experiments are designed to be run on clusters with ∼1000 nodes connected over 1-10Gbps links, while we used only 24 cores, we had to throttle the intra-cluster and inter-cluster network bandwidth to 25 and 5 Mbps respectively in order to show meaningful network contention. At the same time, to achieve realistic migration times expected in real platforms, the inter-machine bandwidth throttling for migration purposes was set to 500Mbps intra-cluster and 100Mbps inter-cluster respectively.

3) Comparison Algorithms: In our experiments, we compared the following placement/migration algorithms:

- *No migration:* The VMs are run as initially assigned.
- *Affinity-based migration:* Here, a VM is migrated closer to other VMs with which it has higher affinity using the distributed bartering algorithm described in Section IV-B.
- *Best/Optimal placement:* Here, the VMs are pre-placed in an optimal configuration in terms of minimizing the network overhead, based on their communication pattern, and no migration takes place during the execution. The Intel MPI benchmarks select the communication pattern at runtime when they are started, and hence, it was not possible to determine the communication pattern beforehand. For these benchmarks, we ran trials with multiple "near-optimal" (closely-placed) configurations and selected the best among all runs.

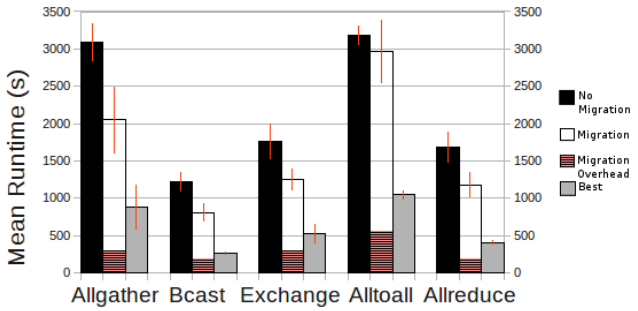

Fig. 3. Runtimes for MPI benchmarks - static configuration

4) Metrics: We use the following metrics to compare the performance of the different placement/migration algorithms:

- *Application runtime:* This metric provides a measure of the application performance.
- *Total network communication cost:* This metric is defined as the sum of the cumulative network communication cost between all VM pairs in the system, using the definition of communication cost ($CC_{i,j}$ for VMs i and j) from Equation 1 (Section IV). We assume $BW_{i,j}$ for two VMs running on the same server to be ∞ as there is no network traffic in this case (VMs communicate through memory).

All results are based on averages taken over multiple trials and all graphs show 95% confidence intervals. For our affinity-based migration algorithm, we used the following monitoring and migration parameters: a monitoring window W of 20 seconds, an exponential averaging constant α= 0.125 for computing the traffic volume, an inertia factor of 1.2 and a swap inertia factor of 1 for the migration and swapping decisions, respectively.

B. Benefit for Static Configurations

In our first set of experiments, we show the benefit of using affinity-based migration when the communication pattern of the applications remains fixed.

1) Intel MPI Benchmarks: We first ran different benchmarks from the MPI benchmark suite to see the impact on the application runtime. For each benchmark, we used 6 processes each running within one VM, and each physical server was assigned one VM chosen at random. The benchmarks were run under several different configurations matching the above initial placement methodology, with and without affinity-based migration. For migration purposes, each server was limited to 3 VMs as its limit. We also compared these results to "best" configurations, where 2 servers in the same cluster were manually assigned 3 VMs each to minimize the network communication cost.

Figure 3 shows the average runtime results for these experiments. As seen in the figure, the benchmarks show about

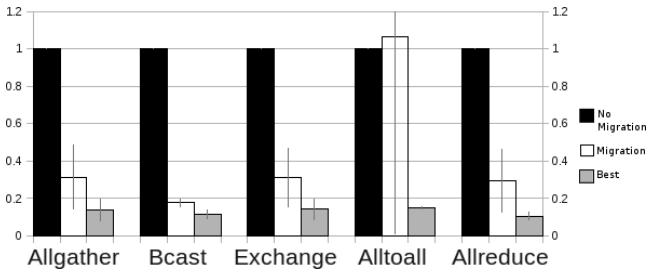

Fig. 4. Normalized communication cost for MPI benchmarks - static configuration

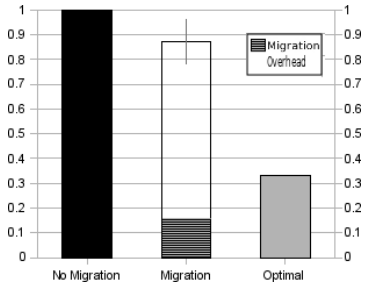

Fig. 5. Normalized Runtimes for Cube application

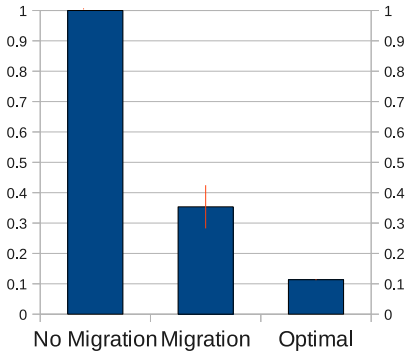

Fig. 6. Normalized communication cost for Cube application

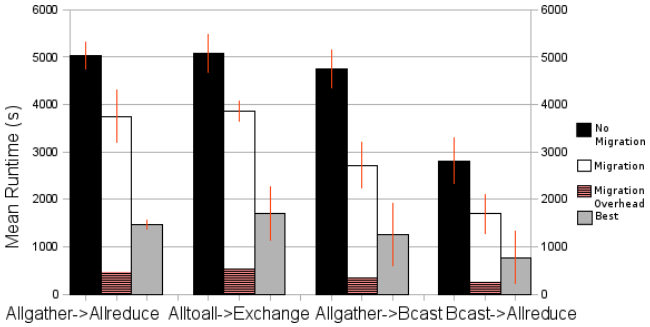

Fig. 7. Runtimes for MPI benchmarks - dynamic configuration

6.7-33.9% average reduction in runtime, compared to No Migration, when using our algorithm. However, the gains in the Alltoall benchmark are not statistically significant. This may be attributed to the unusual communication pattern created by this benchmark, in which all processes (VMs) exchange data with all other processes, resulting in many migrations before Starling reaches a steady-state. In a real platform, we'd expect smaller cliques of co-communicating VMs; if Alltoall is excluded, the average reduction is 28.9-33.9%. We also see that the Best runtimes are on an average 32-43% of Migration runtimes. There are two main reasons for this gap. First of all, the Migration runtimes include the migration time which is shown as the shaded bar within the migration bars in the graph. This migration time forms 14-23% of the total runtime. Secondly, since our affinity-based algorithm is completely distributed and uses only local information, it sometimes settles into a non-optimal configuration, explaining the gap in its performance from Best.

Next, we look at the network communication cost of the different algorithms to understand how much network overhead savings they provide. Figure 4 shows the communication cost for these algorithms normalized with respect to that of No Migration. As shown in the figure, we see that the normalized communication cost of the affinity-based migration algorithm is 0.18-0.32 compared to 0.11-0.15 for Best on average. This shows that our migration algorithm is able to substantially reduce the network communication cost. Again, the confidence interval for Alltoall benchmark was very large, indicating a lot of variation because of the extreme nature of its communication pattern.

2) Cube MHD Jet Application: In the next set of experiments, we ran the Cube application with 12 VMs, and a limit of 4 VMs per machine. The application was set up to communicate in a 4x3 grid pattern, and was run for 5000 steps of the simulation. For the initial configuration, 2 VMs were placed at random on each of the 6 physical machines, and the application was run with migration enabled and also with migration disabled. Here, based on the application's communication pattern, the optimal configuration had 4 VMs per physical machine on 3 machines within the same cluster.

Figures 5 and 6 show the runtime and communication cost results for the application normalized by those of the No Migration algorithm. We see an average improvement of 13% in runtime and a normalized communication cost of 0.35 w.r.t. No Migration. This is in comparison to a 77% improvement in runtime and a 0.11 normalized communication cost for Optimal.

C. Benefit for Dynamic Configurations

We now show the benefit of using affinity-based migration when the communication pattern of the applications changes over time. In this set of experiments, we chained multiple Intel MPI benchmarks to execute in succession to emulate changing communication patterns. In this case, each of the benchmarks was started in the same configurations as before, however, the communication pattern between the VMs changed on switching from one benchmark to the next. This was done to

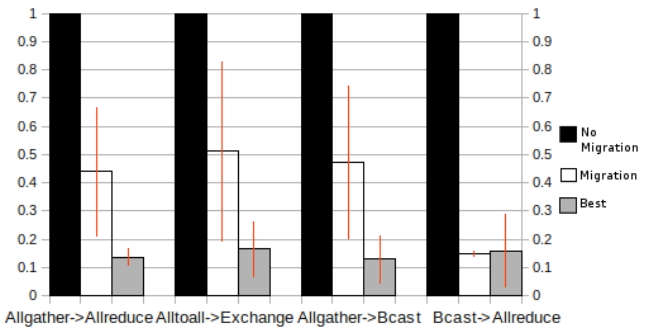

Fig. 8. Normalized communication cost for MPI benchmarks - dynamic configuration

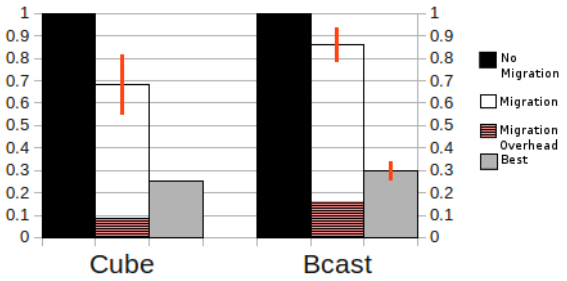

Fig. 9. Runtimes for multiple concurrent benchmarks

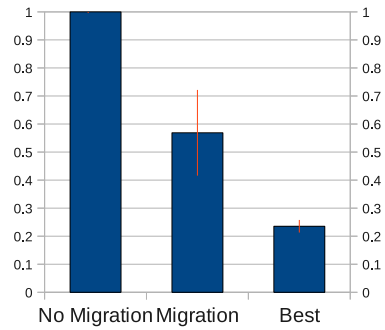

Fig. 10. Normalized communication cost for multiple concurrent benchmarks

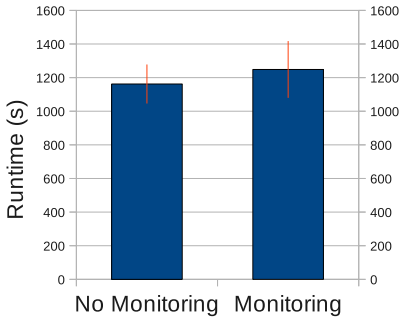

Fig. 11. Monitoring overhead

demonstrate how each of the algorithms responds to changes in application behavior. In this case, for No Migration, each VM remains in its initial location, while for the Migration case, the VMs are migrated based on the communication patterns observed in the system (thus re-migrating some of the VMs after the switch to a different benchmark). For the Best case, the VMs were started on a favorable placement based on the first benchmark, and were kept at those locations throughout the execution. However, we ran multiple such configurations, and picked the best overall execution.

Figures 7 and 8 show the runtimes and the communication cost respectively for these dynamic configurations. As seen in the figures, affinity-based migration improves performance while reducing the network cost (an average of 25-42% for runtime and normalized communication cost of 0.15-0.51). The Best case runtime is 39-46% of the migration runtime and it has a normalized communication cost of 0.13-0.17. These results show the benefit of dynamic inference of affinity in the face of changing communication patterns.

D. Multiple Concurrent Benchmarks

In the next set of experiments, we examine the behavior of the migration algorithm in the presence of multiple concurrently running applications, when the platform is already completely provisioned. The goal here is to show that even if the system is completely full (in terms of its CPU and memory resources), there can still be benefit in terms of reducing network contention and improving application performance even further. These experiments also show the benefit of VM swapping explicitly as migrations could not take place here without swapping being enabled. Here, each physical machine was set to a cap of 2 VMs. 1 VM running the Cube benchmark and 1 VM running the Bcast benchmark was randomly placed on each physical machine. The algorithm was able to swap the VMs so that each physical machine ended up with two VMs running the same benchmark. Figures 9 and 10 show the normalized runtime and communication cost respectively for this scenario with similar results as before.

E. Migration and Monitoring Overhead

The migration time overhead has been shown in the earlier results - it ranges from 8-23% in most cases. In terms of the network overhead of migration, it depends on the RAM size of the VMs. For our experiments, we set the RAM size to be 632 MB per VM, and we saw an average migration overhead of 620,906 packets for Etch VM images, and 616,519 packets for Hardy VM Images.

Figure 11 shows the monitoring overhead by comparing the runtime of the Broadcast benchmark with and without monitoring enabled (here we do not carry out any migrations). As seen in the figure, there is no statistically significant difference in the two runtimes, thus showing minimal impact of monitoring on the application performance.

VI. Related Work

Affinity-aware job placement: The VM placement problem presented in this paper is an instance of the graph partitioning problem which is known to be NP-complete [20], and several heuristics (e.g., [23], [24]) have been developed to solve the general graph partitioning problem. There has also been lot of work in partitioning jobs among processors for the purpose of minimizing communication overhead [25], [26], [27]. These approaches have been largely centralized and assume that the job dependencies are static and are known a priori. Our goal, on the other hand, is to perform dynamic job allocation in a completely distributed manner through VM migration and placement.

There has been recent work on topology-aware application mapping and load balancing for scientific applications [28], [29]. However, our work differs in some key aspects because it is geared towards virtualized clusters and cloud environments, as opposed to supercomputing systems targeted in this work. Thus, while this work assumes knowledge about application structure for its mapping, we rely on non-intrusive monitoring-based inference of this information. Secondly, we use a decentralized approach targeted towards a heterogeneous and hierarchical network topology, as compared to a centralized algorithm suitable for a homogeneous and tightly-coupled network topology considered in this work. Another recent work [30] aims at providing location-aware cluster management for cloud environments for data-intensive applications. The goals of this work are similar to ours, though we have also considered communication-intensive applications, and focus on inferring dynamic communication patterns in addition to file and data dependencies.

Load balancing: There is a large body of work in load balancing in distributed systems [31], [32], [33], and recent work in virtual machine migration, placement and load balancing [13], [11]. Most of these approaches mainly consider the local load (e.g., CPU load) on the processors in making their load balancing decisions, while our work also exploits the communication affinities between VMs to achieve better placement and migration. MOSIX [34] is a Linux-based cluster computing system that also achieves runtime load balancing, while our focus is on virtualized environments. MOSIX has a centralized communication-aware algorithm [35] while we use a decentralized algorithm that can be useful for larger systems.

Distributed resource allocation and scheduling: Several resource allocation techniques [36], [37], [38], [39] for high-performance, cluster, and Grid computing have focused on the discovery and scheduling of resources that match the capacity requirements (e.g., CPU, memory, etc.) of compute jobs. Most of these techniques rely on application specification of job resource requirements. Recent work [10] has proposed using a control-theoretic feedback algorithm to dynamically change the resource allocation in a virtualized computing platform. Our work is complementary to some of these techniques as it incorporates network affinity and data dependencies between computation jobs as an additional criterion for doing the resource allocation. In addition, through VM-level observations, our technique performs dynamic and non-intrusive inference of the communication dependencies, and does not rely on explicit application-level specification of these dependencies.

Virtual machine performance optimization: Several mechanisms have been proposed for improving the networking performance of virtual machines. Xen and co. [40] extends co-scheduling [41] to improve the performance of communicating VMs. Xenloop [42] improves the communication performance between collocated Xen VMs through an inter-VM shared memory channel that bypasses the virtualized network interface. Our work can utilize some of these mechanisms to improve the communication performance among collocated VMs, and in fact, it attempts to find more opportunities for such optimizations by moving heavily communicating VMs closer to each other. At the same time, our technique provides a more general approach for minimizing network communication cost over a large distributed system.

Our work is similar in nature to recent work [43], [44] that have considered memory sharing affinities between VMs for load-balancing and consolidation, except that we are focusing on network bandwidth instead of memory usage. In fact, we believe such affinities should be considered across multiple resources to provide a unified framework for optimal VM placement and migration [45].

VII. Conclusions

As virtualization gains in popularity for large computing environments, management of VMs is becoming an important problem. The efficiency of the platform as well as the performance of applications running in the platform are critically dependent on the characteristics of the applications and the topology of the infrastructure. In particular, in many existing virtualized platforms, it has been found that the network bandwidth often becomes the bottleneck resource due to the hierarchical topology of the underlying network, causing both high network contention and reduced performance for communication and data-intensive applications.

In this paper, we presented a decentralized affinity-aware migration technique that incorporates heterogeneity and dynamism in network topology and job communication patterns to allocate virtual machines on the available physical resources. Our technique monitors network affinity between pairs of VMs and uses a distributed bartering algorithm coupled with migration to dynamically adjust VM placement such that communication overhead is minimized. Our experimental results running the Intel MPI benchmark and a scientific application on a 7-node Xen cluster showed that we can get up to 42% improvement in the runtime of the application over a no-migration technique, while achieving up to 85% reduction in network communication cost. In addition, our technique was able to adjust to dynamic variations in communication patterns and provides both good performance and low network contention with minimal overhead.

ACKNOWLEDGMENT

This work was supported in part by NSF Grant CNS-0643505.

REFERENCES

[1] "Amazon Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2/.
[2] "Zmanda case study: Amazon web services," http://aws.amazon.com/solutions/case-studies/zmanda/.
[3] "Lotuslive inotes," https://www.lotuslive.com/en/services/inotes.
[4] "Nasdaq market replay," http://www.infoq.com/articles/nasdaq-case-study-air-and-s3?
[5] "Amazon Web Services: Case Studies," http://aws.amazon.com/solutions/case-studies/.
[6] "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce/.
[7] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen, "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, 2006.
[8] R. J. Figueiredo, P. Dinda, and J. Fortes, "A Case for Grid Computing on Virtual Machines," in *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Apr. 2003.
[9] K. Keahey, K. Doering, and I. Foster, "From Sandbox to Playground: Dynamic Virtual Environments in the Grid," in *Proceedings of the 5th International Workshop in Grid Computing (Grid 2004)*, Nov. 2004.
[10] S.-M. Park and M. Humphrey, "Feedback-controlled resource sharing for predictable eScience," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
[11] A. Singh, M. Korupolu, and D. Mohapatra, "Server-Storage Virtualization: Integration and Load Balancing in Data Centers," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS, UC Berkeley, Tech. Rep. EECS-2009-28, Feb. 2009.
[13] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.
[14] "Intel MPI Benchmarks," http://www.intel.com/cd/software/products/asmo-na/eng/cluster/219847.htm.
[15] "Computational Astrophysics at University of Minnesota," http://www.astro.umn.edu/groups/compastro/?q=node/1.
[16] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.
[17] "Apache Hadoop," http://hadoop.apache.org/core/.
[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of Symposium on Operating Systems Principles*, 2003.
[19] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration," CSE, University of Minnesota, Tech. Rep. TR09-030, Dec. 2009.
[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[21] C. Clark, K. Fraser, , S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of NSDI*, May 2005.
[22] A. F. Rob Bradford, Evangelos Kotsovinos and H. Schioeberg, "Live Wide-Area Migration of Virtual Machines Including Local Persistent State," in *Proceedings of VEE'07*, Jun. 2007.
[23] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, vol. 49, p. 291307, Feb. 1970.
[24] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.
[25] S. H. Bokhari, "Partitioning Problems in Parallel, Pipeline, and Distributed Computing," *IEEE Transactions on Computers*, vol. 37, no. 1, pp. 48–57, Jan. 1988.
[26] H. Stone and S. Bokhari, "Control of Distributed Processes," *IEEE Computer*, vol. 11, no. 7, pp. 97–106, Jul. 1978.
[27] V. M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, Nov. 1988.
[28] A. Bhatele and L. V. Kale, "Application-specific Topology-aware Mapping for Three Dimensional Topologies," in *Workshop on Large-Scale Parallel Processing (IPDPS)*, 2008.
[29] A. Bhatele, L. V. Kale, and S. Kumar, "Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications," in *23rd ACM International Conference on Supercomputing*, 2009.
[30] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. Lopez, M. Stroucken, and G. R. Ganger, "Tashi: Location-aware Cluster Management," in *First Workshop on Automated Control for Datacenters and Clouds (ACDC'09)*, 2009.
[31] Y.-C. Chow and W. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Transactions on Computers*, vol. 28, no. 5, pp. 354–361, May 1979.
[32] T. Chou and J. Abraham, "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 401–412, Jul. 1982.
[33] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. 12, no. 5, pp. 662–675, May 1986.
[34] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Journal of Future Generation Computer Systems*, vol. 13, no. 4-5, pp. 361–372, Mar. 1998.
[35] A. Keren and A. Barak., "Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster," *IEEE Tran. Parallel and Distributed Systems*, vol. 14, no. 1, pp. 39–50, Jan. 2003.
[36] "Platform LSF," http://www.platform.com/Products/platform-lsf.
[37] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation (1999)," in *Proceedings of the International Workshop on Quality of Service (IWQoS)*, 1999.
[38] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," in *HPDC'98*, Jul. 1998.
[39] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, "Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet," in *Proceedings of the IEEE Fourth International Conference on Peer-to-Peer Systems*, 2004.
[40] S. Govindam, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms," in *Proceedings of the 3rd Intl. Conference on Virtual Execution Environments*, 2007.
[41] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, Oct. 1982.
[42] J. Wang, K.-L. Wright, and K. Gopalan, "XenLoop: a transparent high performance inter-vm network loopback," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC'08)*, 2008, pp. 109–118.
[43] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," in *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, 2008.
[44] T. Wood, G. Tarasuk-Levin, P.Shenoy, P. Desnoyers, E. Cecchet, and M. Corner, "Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers," in *Proceedings of the 5th ACM Intl. Conference on Virtual Execution Environments*, 2009.
[45] J. Sonnek and A. Chandra, "Virtual Putty: Reshaping the Physical Footprint of Virtual Machines," in *Workshop on Hot Topics in Cloud Computing (HotCloud'09)*, Jun. 2009.