Using Network Knowledge to Improve Workload Performance in Virtualized Data Centers

David Erickson, Brandon Heller, Nick McKeown, Mendel Rosenblum
Stanford University

***Abstract*—The scale and expense of modern data centers motivates running them as efficiently as possible. This paper explores how virtualized data center performance can be improved when network traffic and topology data informs VM placement. Our practical heuristics, tested on network-heavy, scale-out workloads in an 80 server cluster, improve overall performance by up to 70% compared to random placement in a multi-tenant configuration.**

I. Introduction

Modern virtual machines (VMs) arose as a tool to test, manage, and consolidate x86 servers. They have since matured into the building block of the cloud, supporting scale-out services in both public and private virtualized data centers (VDCs). The size of modern data centers provides a strong motivation to optimize efficiency; Amazon's US East data center reportedly contains over 320,000 servers [1]. Optimization can reduce operating expenses, defer upgrades, free up capacity to sell, or automatically improve tenants' performance – all attractive options when a large data center costs hundreds of millions of dollars to build and operate.

Despite cost pressures, today's cloud environments do not appear to exploit the full potential of VMs to optimize a single scale-out workload, let alone an entire data center. To our knowledge, with the exception of maintenance events, every major cloud operator places VMs on physical machines (PMs) statically, without network knowledge, leaving them in place for their entire lifetimes. A *dynamic data center* would not just alleviate hot spots, but would use all available data to optimize the workloads running inside, both individually and collectively. For example, including network knowledge in the assignment of VMs to PMs (VM placement) could increase traffic locality to reduce network bottlenecks, enabling a web service to handle more requests, a MapReduce job to complete faster, and both to improve simultaneously. Improvements in efficiency can benefit both providers and tenants in cloud environments (where the two are decoupled), as well as private data center owner/operators.

Today, two commercial VM placement products are widely used in enterprise data centers, but they are not network-aware [2], [18]. We suspected that public clouds could do much better by taking advantage of network knowledge; this belief leads us to the first of two questions posed in this paper:

> *What are the performance benefits of adding network knowledge to VM placement for scale-out workloads?* (1)

That is, with perfect knowledge and an optimal algorithm for placing VMs, how much faster might one or more scale-out workloads run? Most related work does not answer this question, and (a) only considers the efficiency benefits for the operator, but not the improvements in the workload itself [34], [44]; or (b) *does not have complete topology and traffic matrix information when optimizing VM placement* [13], [26], [37], [44].

Figure 1 shows the time evolution of an iterative VM placement algorithm, with each iteration consisting of three phases. In the first phase, we measure CPU and network resource use. Next, an optimization algorithm picks a new mapping of VMs to PMs. Finally, the VMs are migrated to their new host. The three phases are repeated indefinitely, with the goal of improving the optimization metric (usually throughput or completion time) in each cycle. Each phase of the cycle is challenging:

- **Measure.** We need to pick a measurement period that is small compared to the rate in which workloads change their network and CPU usage; changes occur during different phases of computation and as workloads come and go. Also, the system must take into account that network usage is elastic — TCP will use more capacity should it be available.
- **Optimize.** Traffic-aware VM placement is NP-hard and does not admit a constant-factor approximation [34].
- **Migrate.** Moving VMs takes time, consumes resources on both the origin and target hosts, and the network links in-between.

For the rest of the paper, we will assume the system uses the three-phase optimization cycle. This paper focuses on the "measure" and "optimize" phases as these are both the least studied and are necessary to enable an exploration of possible network-aware workload performance improvements. VM migration is further discussed in §VI.

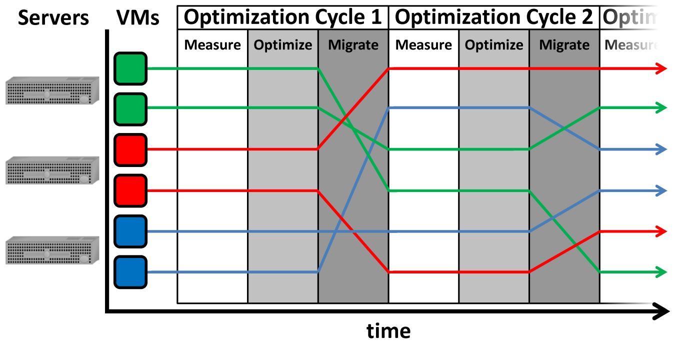


Fig. 1. Workflow to optimize VM placement.

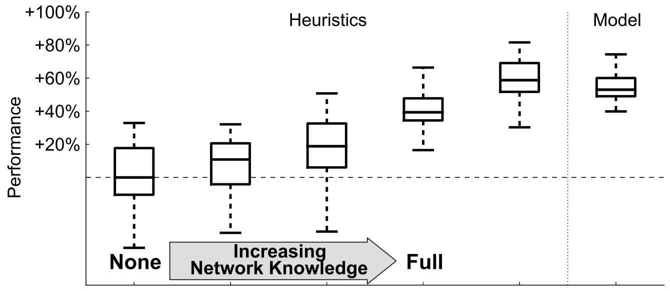


Fig. 2. Performance improvement by algorithm.

The system performance will critically depend on the algorithm in the "optimize" phase, which leads to this paper's second question:

What are practical network-aware VM placement algorithms, and what are their tradeoffs? (2)

Figure 2 gives a sneak preview of our findings (full results are in §V). The graph plots the improvement in performance (in this case, webpage serving throughput for web servers on 40 PMs) for different optimization algorithms. For now we are not concerned with the actual numbers, but want to highlight the trend of improving performance from left to right as the algorithms have access to more network knowledge. On the right, the last three algorithms (two heuristics, and a mixed integer optimization) know the topology and full traffic matrix. Performance improves and becomes less variable, at the cost of gathering more data and using a more complex algorithm. We will see that improvements of 70% are possible, by carefully placing VMs using full network knowledge.

We find it encouraging that these gains occur where they are hardest to achieve; that is, in public data centers (such as EC2 and Rackspace) where workloads are:

- Opaque (i.e., applications provide no hints on how they use resources).
- Heterogeneous (i.e., the VDC hosts many concurrent workloads).

Our contributions are:

Algorithms (§III). We introduce placement algorithms that improve performance by exploiting network and topology knowledge. *Model* is our performance benchmark: it solves a mixed-integer optimization to calculate a near-optimal placement. *Model* does not scale, so we introduce two greedy algorithms and four simulated annealing variations. Some algorithms iteratively improve a starting placement, while others start from scratch.

Prototype (§IV). We designed and built a prototype VDC resource manager, *Virtue*, that can efficiently launch scale-out multi-tenant workloads, measure resource utilizations, feed the results to an external optimizer, and then run an experiment *with a new placement* – entirely unattended. This functionality enables before-and-after comparisons from our baseline random placements to improved placements returned by our optimizers.

Experiments (§V). We evaluate all of our optimization algorithms on an 80-node cluster, with different workloads. We show that random placement gives low performance (throughput and completion time) and significant variance. We show that our optimization algorithms steadily improve performance when given progressively more network knowledge. Our heuristics give up to 70% higher throughput than a *Random* algorithm or an algorithm representative of current commercial offerings.

II. Related Work

When trying to optimize a VDC that runs network-heavy, scale-out workloads, we have two high-level choices: improve the network, or improve VM placement.

Improve the network. For single workloads that run across thousands of machines with little or no locality, providing high or full bisection bandwidth [7], [16], [17], [22], [39], [42] and routing flows across multiple paths [8], [36] may be the only way to improve server-to-server bandwidth. However, even with full bisection bandwidth (FBB), VM placement can improve performance. A single modern core can saturate a 10Gb/s link [5]; yet, many servers have multiple cores, which means more network bandwidth is available between VMs on the same PM than between VMs on separate PMs.

Improving VM placement. The closest and most recent related work is Net-Cohort [26], which measures VM CPU and NIC counters and uses them to determine a better VM placement. Net-Cohort infers VM "ensembles" by looking for hierarchical clusters, however, the authors do not describe how VM clusters map to physical machines. The Net-Cohort paper demonstrates benefits in application-level metrics after a single optimization stage from a single starting placement. In contrast, we explore the incremental benefits of gradually increasing network knowledge with seven algorithms, rather than one. We consider algorithms that use the directly measured network traffic matrix, rather than inferred values, and the network topology. We measure 1,675 before-after placements, rather than one. Our experiments also vary link rates and oversubscription ratios.

Optimization tools from VMware [18] and Citrix [2] automatically balance the placement of VMs on PMs within a resource pool. These products only use *local* information from the hypervisor, such as CPU, RAM, disk, and local network traffic counters, and only support load balancing of resource pools containing up to 32 hosts. Unlike commercial tools, *Virtue* uses algorithms with full network knowledge, including the traffic matrix and topology, and runs them on up to 80 PMs.

Meng [34] defines the Traffic-aware VM Placement Problem (TVMPP) and shows it to be NP-hard. The authors propose a cluster-and-cut algorithm to minimize total network traffic. The evaluation uses VMs with minimal network usage: 80% of the measured VMs have network rates below 106 Kb/s and only 4% exceed 1 Mb/s. *Virtue* aims to maximize workload performance and evaluates workloads that saturate network links.

Sandpiper [44] uses a greedy approach to minimize a VM's "volume", the product of its CPU, memory, and network use; like other systems [37], Sandpiper does not consider the traffic matrix or evaluate the resulting workload performance (like [33], [35]). *Virtue* considers the full traffic matrix and evaluates total system performance.

Mantri [9] uses static rack bandwidth when placing MapReduce tasks to reduce effects from network congestion. Other research explores live VM migration [14], [40], [46], efficient scale-out service provisioning [41], and the interface presented to the VDC manager and users [32].

Systems like Oktopus [12] and SecondNet [21] reserve network bandwidth, but require *a priori* demand knowledge. In contrast, *Virtue* requires no service knowledge, and discovers all resource usage. Seawall [38] and NetShare [30] share network bandwidth among a network's tenants more consistently, complementing our approach.

III. ALGORITHMS

It is difficult to know exactly what algorithms public cloud providers use today to place VMs, since these algorithms are viewed as a competitive advantage and are therefore kept hidden. To our knowledge, none leverage knowledge of the network topology or the traffic matrix. In this section, we show algorithms that do leverage network information.

A. Optimization Metric: Minimax

All of our algorithms place VMs by minimizing the utilization of the maximally utilized resource in the data center, which for our experiments is either a PM's CPU or a network link. Other resources such as disk or memory could also be included, but this work focuses on the incremental benefit of adding network knowledge to today's CPU-only algorithms.

Minimizing the maximum utilization, a heuristic often referred to as Minimax, is commonly used for load-balancing. Minimax is simple to understand and implement, performs well in practice, and has previously been used to allocate tasks onto networked infrastructures [11]. The main shortcoming of Minimax is that it will not try to improve performance if the utilization of the maximally utilized resource cannot be decreased. For example, if two VMs are communicating at full line rate but cannot be placed on the same PM, the optimizer cannot reduce the 100% utilization of the links between them, and it will not even attempt to optimize any of the resources. This problem is likely in scale-out, multi-tenant data centers where it is likely that at least one link is fully utilized. Nonetheless, as we will see, Minimax is a good starting point to demonstrate performance improvements using network-aware algorithms, and it is not hard to find algorithms that go beyond just considering the most utilized resource.

B. Random Placement

The *Random* algorithm generates baseline placements to compare our other algorithms against. *Random* distributes VMs at random across all PMs; some PMs host several VMs, while others host none. If the algorithm picks an oversubscribed PM (in our evaluations, each PM is allowed to host at most four VMs), it randomly picks a new PM. Because it does not consider resource usage, *Random* represents a lower bound on performance; dynamic placement optimization algorithms that are aware of resource usage should fare better. *Random also represents the performance we can expect in a large VDC with a static placement policy, which has become fragmented over time as tenants add and remove VMs.*

TABLE I. OPTIMIZATION ALGORITHMS AND THE DATA THEY USE.

	CPU	VM NIC	T. Matrix	Topo
Random				
SA C	✓			
SA CN	✓	✓		
SA CNTM	✓	✓	✓	
SA All	✓	✓	✓	✓
Greedy Net		✓	✓	✓
Greedy Fill	✓	✓	✓	✓
Model	✓	✓	✓	✓

C. Mixed-Integer Model

If *Random* is our lower bound, then *Model* is an approximate upper bound to compare our algorithms against. The *Model* algorithm uses system resource data (CPU speeds, network topology, link capacities, and routing tables) and measured workload data (CPU utilization and traffic matrix). It solves a mixed-integer optimization to find a new VM placement that minimizes the maximally used CPU or network link. *Model* is described as a multi-commodity flow problem with added CPU and VM placement constraints. *Model* works for any network topology, and our implementation is based on the commercial CPLEX solver that guarantees an optimization metric within a fixed fraction (the *gap*) of its believed optimal value. However, *Model* must explore an enormous solution space, and it may take a long, variable time to return a solution. As a result, we will see that it is only practical as a benchmark for systems with 20 or fewer PMs.[1] The poor scaling properties of *Model* motivate more practical algorithms.

D. Practical Algorithms

The practical algorithms we consider next give performance numbers that generally fall between *Random* and *Model*. We examine algorithms with varying degrees of network knowledge, as this knowledge could be difficult to gather, or might simply be unavailable. We want to understand how performance improves as algorithms become more network-aware. Table I categorizes the algorithms by the information they use, including VM CPU utilization and PM CPU capacity, VM NIC traffic counters and PM NIC link capacity, the VM-to-VM traffic matrix, and the full network topology, including link capacities and routing information. We do not consider RAM so we can focus on the performance improvements when we add network knowledge over and above today's CPU-based optimization.

1) Simulated Annealing: Four of our practical algorithms are based on simulated annealing. Simulated Annealing (*SA*) is a probabilistic optimization heuristic that strives to find the global optimum in a search space that may have many local minima. Initially, *SA* algorithms take random steps with high probability in an effort to escape local minima. As time progresses, *SA* algorithms become more conservative and less likely to take steps that do not immediately improve the optimization metric [29].

We implemented four variants of simulated annealing with different energy functions. Below, we describe each variant

[1]The evaluations in this paper all run on tree-like networks, so we could probably accelerate *Model* by exploiting this knowledge.

Algorithm 1 Greedy Fill

```
1: $unassign(vms)$
2: $sortByTrafficSumAsc(vms)$
3: for VM vm in vms do
4:     $minUtil \leftarrow MAX\_VALUE$
5:     for PM pm in pms do
6:         $util \leftarrow MaxUtil(vmonpm)$
7:         if $util < minUtil$ then
8:             $minUtil \leftarrow util$
9:             $minPM \leftarrow pm$
10:        end if
11:    end for
12:    $assign(vm, pm)$
13: end for
```

along with the information about the infrastructure that each one requires:

- ***SA C*** - Uses CPU utilization reported by VMs and the maximum CPU capacity of the PMs. *This is a simplified version of what appears to be used by VMware's DRS today* [18], [19]. The energy function returns the estimated maximally utilized PM CPU.
- ***SA CN*** - *SA C* plus limited network awareness: the VM NIC's rate over time, and the maximum capacity of the physical NIC on the machine. The energy function returns:

$$\max(\text{max PM CPU utilization}, \text{max PM NIC utilization})$$

- ***SA CNTM*** - *SA CN* plus traffic awareness: the measured traffic matrix between VMs. Improves upon *SA CN*'s PM NIC utilization estimate by removing intra-PM VM traffic from the estimate of traffic crossing the PM's NIC. The energy function is the same as that of *SA CN*.
- ***SA All*** - *SA CNTM* plus full network topology, link capacities, and routing. The energy function returns:

$$\max(\text{max PM CPU utilization}, \text{max network link utilization})$$

2) Greedy Network: The *Greedy Net* algorithm uses traffic matrix, topology, and routing data, but no CPU data. *Greedy Net* repeatedly finds the most congested link, then moves VMs sending traffic across this link to other PMs when a move will decrease the maximum link utilization. It stops when no move will decrease the maximum link utilization.

3) Greedy Fill: The *Greedy Fill* algorithm uses the same information as *SA All* and is shown in Algorithm 1. It starts with a clean slate, first unassigning all VMs, then sorting them in ascending order based on the sum of their average network rate. *Greedy Fill* assigns each VM to a PM, at each step computing the maximum CPU or network link utilization for each possible VM to PM assignment, then picking the PM that minimizes the maximum utilization. This method represents an interesting class of algorithms and shows whether a heuristic can do better if it rips up the existing VM placement and starts over.

If our algorithm assigned VMs to PMs starting with the heaviest network users first, it would likely run into trouble later. Consider the case where one VM is a heavy network user but only because it talks at a modest rate to many other VMs. The heavy VM will be placed first, and the VMs it communicates with may be placed much later by the algorithm. Later, when *Greedy Fill* finally gets to placing the communicating VMs, there may be no slots available on PMs near the heavy VM from a network perspective, thus putting additional load on the core network. Therefore, we take the counterintuitive step of assigning VMs in *ascending* network-use order.

As we see later, our results show that *Greedy Fill* performs surprisingly well. If it becomes cheap to move VMs [14], [40], [46], this algorithm could become very attractive.

IV. Testbed

To understand how each algorithm performs in practice, we built a system to run relatively large, controlled experiments. Our software, *Virtue*, lets us configure, deploy, measure, and optimize real and synthetic workloads at scale, while varying network properties and VM placement. *Virtue*'s software comprises 41,000 lines of custom Java, Bash scripts, and GUI interface code. *Virtue* runs customized OS and switch firmware images on bare metal server and switch hardware; something we could not do on shared testbeds such as Emulab [24]. We partnered with Google to create the Data Center Network Research Cluster (DNRC), which provides full bare-metal access.

A. Servers

The DNRC contains 80 Google production servers that first ran in 2004; each has two sockets with hyper-threaded Intel Netburst-based Xeon CPUs running at 2.8GHz with 8GB of DDR RAM. All servers, 20 in a rack, run the XenServer 5.6FP1 hypervisor with paravirtualized Linux VMs. Each XenServer runs Open vSwitch (OVS), a fast, externally controllable software switch, which measures VM-to-VM traffic inside and between PMs.

B. Network

The DNRC network is a full-bisection-bandwidth three-layer $k = 4$ fat tree [7], built from Pronto 3240 48 $\times$ 1Gb/s + 4$\times$10Gb/s switches. In addition to two 10Gb/s data uplinks, each top-of-rack switch has a 1Gb/s control port used for NFS and terminal access. Each switch runs Indigo firmware; using OpenFlow [31], it configures routes, monitors link utilization, and modifies network bandwidth. Configurable rate limiters can adjust the oversubscription ratio (16:1 to FBB) and edge link speeds (100Mb/s to 1Gb/s), helping to explore how network provisioning levels affect workload performance.

We measure the VM-VM traffic matrix by periodically polling flow counters in OVS, one per IP source-destination pair. Operators of a real system may want to measure the traffic matrix frequently, to adapt quickly to changing workloads. Measuring traffic matrices is widely thought to be expensive, but not in this environment, where reading a counter takes 88 bytes on the wire.

Table II shows network bandwidth consumed by polling traffic counters, as a fraction of a 1Gb/s link, for varying

TABLE II. PERCENTAGES OF A 1GB/S LINK USED TO POLL COUNTER, FOR DIFFERENT NUMBERS OF FLOWS AND POLLING INTERVALS.

	Polling Interval			
Flows	20s	10s	5s	1s
100	0.0003%	0.0007%	0.0014%	0.007%
1000	0.0035%	0.0070%	0.0140%	0.070%
10,000	0.0352%	0.0704%	0.1408%	0.704%
100,000	0.3520%	0.7040%	1.4080%	7.04%

numbers of flow entries and polling intervals. The number of flow entries in an OVS instance is determined by the number of VMs per PM, along with the number of other communicating VMs. For example, a VDC with 100 VMs per PM, where each VM communicates bidirectionally with 50 other VMs, needs 10,000 flow entries. *Polling all counters every second uses less than 1% of the link capacity*, and future VDCs with more VMs per PM will have faster network links that reduce this fraction.

C. Virtue

Virtue configures and monitors experiments for a VDC. Similar in scope to many data center resource managers [3], [4], [20], [25], *Virtue* gives low-level configurability (network bandwidths and routes) and handles batches of queued experiments, rather than a continuous workload. *Virtue*'s input is the Experiment Description File (EDF), a full description of experiment hardware (PMs, switches, links) and software (VMs, workloads, routes). Given an EDF, *Virtue* coordinates with XenServers, Open vSwitch virtual switches, Indigo physical switches, and the network control plane, *Beacon* [15], to set up an experiment. *Virtue* deploys all VMs in the workload to PMs and creates a MySQL database entry for the experiment. Once each VM is ready, *Virtue* starts the experiment, begins gathering resource statistics and self-reported workload performance metrics, and provides a dashboard to monitor the workload.

D. Optimizers

We implemented *Model* on CPLEX and set it to solve to within 10% of the estimated optimal solution within a 3 hour runtime limit. We found 3 hours to be a good cutoff; experiments taking longer to find a solution typically took dramatically longer, up to multiple days. Due to the difficulty of the problem and observed runtime scaling, we only evaluated *Model* on our smaller-sized experiments. Optimizers for the remaining algorithms are written in Java and are single threaded. Each algorithm receives as input the EDF from a completed experiment run and all its recorded resource measurements. After optimization, each algorithm outputs a new EDF with modified VM placement and routing. This workflow enables *Virtue* to measure before-and-after performance comparisons for different algorithms, or iterate on a single experiment to further improve optimization results. To accelerate the optimization process, we ran our optimizers in Amazon's EC2 cloud. CPLEX solving a single EDF uses all threads on a High-CPU Extra Large instance, and we used up to 50 such instances to optimize all experiments with different random workload placements, simultaneously. The Java-based optimizers use a Cluster Compute Eight Extra Large Instance, optimizing 25 placements in parallel across the instance's 32 exposed processing cores.

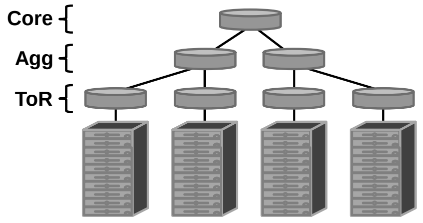


Fig. 3. Subset of testbed topology used in experiments.

V. EVALUATION

The experiments in this section compare multiple workloads, at multiple scales, using our VDC optimization algorithms.

Methodology. For each workload and scale, we generate a set of *Random* VM placements. Experiments with 120 VMs on 80 PMs have 25 initial placements, and experiments with 20 VMs on 40 PMs have 50 due to the decreased setup and teardown time needed for this smaller scale. Each combination of workload, scale, and placement runs on the DNRC for 3 minutes [2] or until the workload terminates. During each run, *Virtue* records resource utilization and performance metrics. After all placements for a particular workload and scale have been run, their data is fed into each optimization algorithm, creating new sets of VM placements, which are then run on the DNRC to determine the relative performance differences.

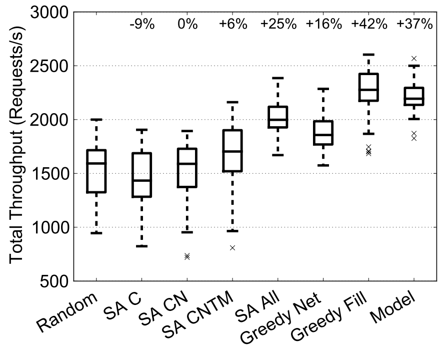
Setup. Figure 3 shows the DNRC subset used in our experiments. 80-PM experiments used four racks of 20 PMs per rack; 40-PM experiments used 10 PMs per rack. RAM is statically allocated for each VM and not included in the optimization algorithms. The total bandwidth oversubscription from PM to PM through the core is set to 16:1, broken down as 4:1 between the top of rack (ToR) switch and the aggregation (AGG) switch, and a further 4:1 between the AGG and core switch. We picked a 16:1 total oversubscription to be well below reported total oversubscription levels of 240:1, and 5:1 to 20:1 from ToR to AGG [17]. We limit edge network links to 100Mb/s to better match the ratio between same-server and different-server VM-to-VM bandwidth (our machines are a few years old). For comparison, we run experiments with other link rates and oversubscription levels.

Graphs. We use boxplots to show the range of measured performance numbers for different placements. The boxes cover inter-quartile ranges (IQRs), the median is marked within each box, the whiskers extend to the furthest point (1.5 times the IQR), and any additional outlier points beyond the whisker's range are marked with an x. Each algorithm's median value improvement relative to *Random* is listed at the top of each boxplot. To highlight important results, we switch to a question/answer format.

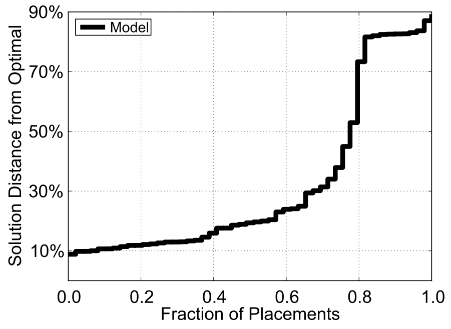
A. Web Workload

To emulate the workload of a multi-tier website, we created a client + 2-tier web workload. The workload has three groups of VMs : (1) clients that continuously send requests to web tier VMs, 20 in parallel; (2) web VMs that listen for incoming

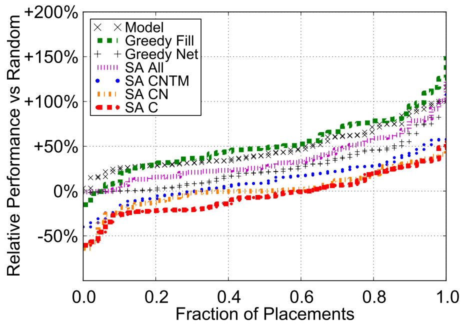
[2]The experiment duration was long enough to ensure the performance of the JVM and total workload stabilized. A shorter runtime on throughput focused workloads while accurately measuring performance should be possible.



(a) Box plots of measured throughputs for each algorithm



(b) CDF of Model solution distance from optimal.



(c) Relative CDF.

Fig. 4. Web 20 VMs 40 PMs

requests, make 10 sequential Memcached-like requests to random VMs in the Memcached tier, then return a response to the client; and (3) the Memcached tier, which receives requests from the web tier. The client sends a 700-800B request to the web tier and gets a 33.66KB response from the web tier. These numbers correspond to the top websites' average request/response size for an HTML page [6]. The web tier's Memcached requests and response sizes are the same as those measured on Facebook's Memcached infrastructure [10]. The optimal performance we have measured from this workload comes at a 1:1:1.333 VM ratio.

1) Web Workload, 20 VMs: Our first experiment is a 20-VM web workload (6 clients, 6 web, 8 Memcached) run on a pool of 40 PMs. The total network oversubscription is 16:1 and so we expect to see a big variation in performance depending on which rack a VM is placed.

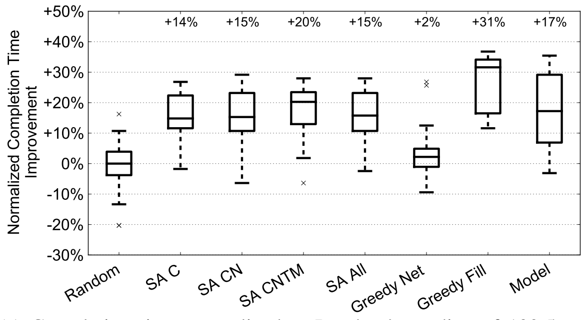
Does the web workload's performance vary across *Random* placements? *Yes, by over 2× between the worst and best performing placements.* Figure 4a plots the performance distribution of each algorithm, along with the set of initial random placements. This workload shows significant variation in performance for the *Random* placements because it is oblivious to CPU and network usage. In fact, the best *Random* placement yields twice the throughput of the worst, despite using the same number of VMs.

Does a network-aware algorithm accelerate a single web workload? *Yes. Network-aware algorithms improve performance by up to 42%; network-oblivious optimization (*SA C*), reduces median performance by 9%.* Comparing *Random* to our optimization algorithms in Figure 4a, we see a steady upward progression, where more network knowledge leads to higher total throughput. Surprisingly, simulated annealing with CPU data (*SA C*), which is similar to today's network-oblivious optimization products, performs slightly worse than random. *SA C* evenly spreads CPU load across machines, precluding it from creating network-beneficial placements containing multiple VMs on a single PM, which random can create. *SA CN* adds NIC traffic counters and performs similarly to random; its improvement relative to *SA C* suggests that for this workload, where plenty of cores are available, spreading the network load is more important than spreading the CPU load. Adding the traffic matrix in *SA CNTM* yields another small improvement, as highly-communicating VMs can then be co-scheduled to reduce network load. *SA All* adds topology knowledge to further improve performance by helping the optimizer differentiate between placements that look great for each edge NIC but stress the core network links. *Greedy Net*, which has no CPU info, performs well, indicating the network dependence of this workload. *Greedy Fill*, which has CPU data, performs the best, even outperforming *Model*.

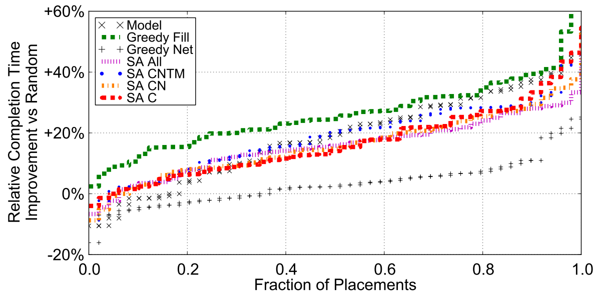
Why does *Greedy Fill* outperform *Model* for this workload? *Model runs for a maximum of 3 hours, bringing only 6% of placements within 10% of the believed optimal solution (gap).* Figure 4b plots a CDF of the optimality of the placements created by *Model*, where optimality is a numerical value reported by CPLEX representing its believed distance from the optimal solution. Few solutions are actually solved to within our configured 10% gap; however, the majority are solved to less than 25%. This tells us two things: *Model* would take much longer than our 3 hour cut off to solve all mappings to within the 10% gap; more importantly, a large fraction of placements (those with a gap significantly higher than 10%) are not likely to yield peak performance.

Does network-aware optimization ever harm performance? *Of the 200 placements generated by fully network-aware algorithms, only 3 reduced performance.* Figure 4c examines whether an algorithm always improves performance, or whether it occasionally hurts performance. On this and subsequent *relative* graphs, each point on a CDF corresponds to the relative performance difference between an algorithm's chosen placement and the random placement from which it received measurement data: for Web, requests/s after/before; for Hadoop, runtime before/after. In this Web workload, algorithms with full topology knowledge do not harm performance, with the exception of 3 placements for *Greedy Fill*. For these algorithms, performance is good, with median improvements from 1.2× to 1.5×, and a maximum improvement of up to 2.5×. However, algorithms with no topology knowledge cannot prevent oversubscribing the core. Within this group, *SA CNTM* has the most information and does the best, improving performance in around 70% of the cases and harming performance for the remaining.

Do multiple optimization cycles improve performance? *Yes, but with diminishing returns.* Our algorithms operate without specific workload knowledge, so they must make placement decisions by measuring the infrastructure. If we knew exactly how a workload would perform absent such bottlenecks, we could make better placement decisions. To understand how our algorithms could perform with better demand knowledge, we ran four iterations of both *Model* and



(a) Completion time normalized to Random's median of 139.5 seconds.



(b) Relative CDF.

Fig. 5. Hadoop 20 VMs 40 PMs

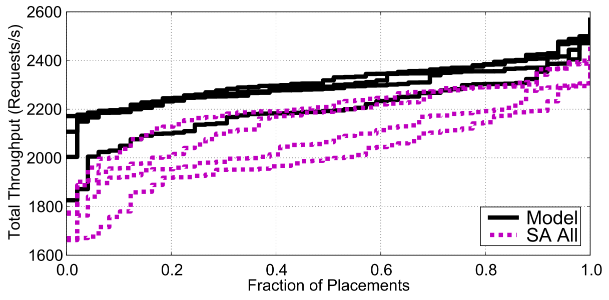


Fig. 6. Iterative algorithm improvement.

SA All, starting with measurements from a random placement. Each successive iteration should remove bottlenecks, bringing traffic matrix measurements closer to the workload's actual demand and bringing performance closer to its peak. Figure 6 shows a CDF comparing the performance of each iteration. We see that *SA All* starts at the bottom line and performs better at each iteration. Between the first and fourth (bottom and top) iterations, there is a 10% median performance improvement. *Model* shows similar behavior between its first and second iterations, finding a 5% median improvement, but the subsequent two show little difference, indicating that *Model* is unable to further improve performance due to its limited view of the world.

B. MapReduce Workload

MapReduce is one of the most frequently run applications in data centers, typified by Amazon's Elastic MapReduce service based on its own version of Hadoop. It is also very different from a web workload. MapReduce performance tends to be dominated by the CPU, with network usage happening between phases in short bursts of all-to-all communication. To represent MapReduce workloads, we use the Hadoop TeraSort benchmark which utilizes a set of VMs to sort a large data set. In each experiment, we picked a data set size so that the benchmark would complete in approximately three minutes – the time of the measurement phase in our optimization cycle — including setup and teardown time. Because MapReduce tends to be CPU or disk limited, we expect to see only a small runtime improvement when moving to network-aware algorithms.

1) Hadoop, 20 VMs: Our first Hadoop experiment runs TeraSort on 20VMs to sort 2GB of data. Each VM stores the data it is working on locally, on the PM's hard drive.

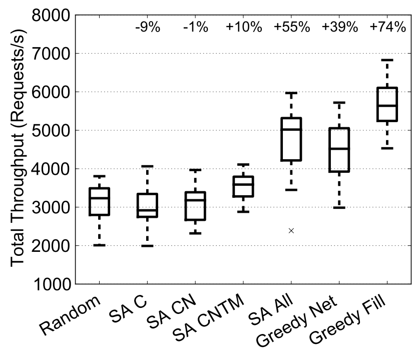
Does a network-aware algorithm help Hadoop finish sooner? *Yes, but at most 17% faster than if did not use network knowledge.* Figure 5a shows a box plot of the time it took TeraSort to complete, for each optimization algorithm, compared to *Random*'s median completion time (139.5 seconds). TeraSort is primarily CPU-limited, so there was little improvement when moving from *SA C* to network-aware algorithms. *Greedy Net* performs no better than *Random* because it measures low network utilization (and does not have access to CPU usage), so it naively places several VMs on the same PM. *Greedy Fill* performs the best because the optimal placement is one VM per PM across the first two racks, which comes as a consequence of our strategy to break ties left-to-right when choosing where to place a VM among otherwise identical PMs. *Model* has trouble finding near-optimal solutions in the time allowed, and so its performance varies widely. Figure 5b shows the good news: all heuristics other than *Greedy Net* improve more than 90% of the placements.

C. Combined Web + MapReduce, 120 VMs

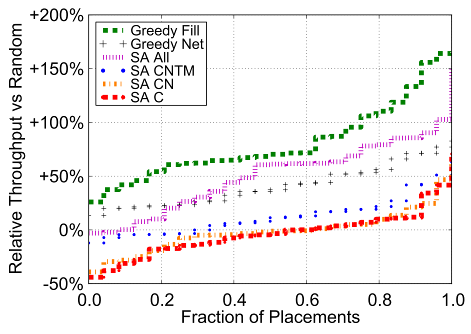
To explore how our optimization algorithms might work in a multi-tenant VDC, we run a mix of workloads totaling 120 VMs on 80 PMs. 30 of the VMs run Hadoop TeraSort (sorting 2GB of data) and 90 VMs run 11 different web services (5 with 3 VMs, 3 with 7 VMs, and 1 each with 10, 14, and 30 VMs).

Does a network-aware algorithm accelerate total web tenant throughput? Without *network knowledge,* SA C *reduced performance by 9% compared to* Random, *whereas optimizing with full network knowledge improved performance by 74%, demonstrating the importance of network-aware algorithms.* Figure 7a plots the combined throughput of the web workloads for different algorithms. The fourth box from the left shows that having NIC and traffic matrix knowledge (*SA CNTM*) makes the workload 10% faster. If the algorithms have full network knowledge, the workload runs 39% faster with *Greedy Net*, 55% faster with *SA All*, and 74% faster with *Greedy Fill*. These three algorithms consistently improve average performance; Figure 7b shows that they improve upon *Random* in all but 3 cases (96% of the time).

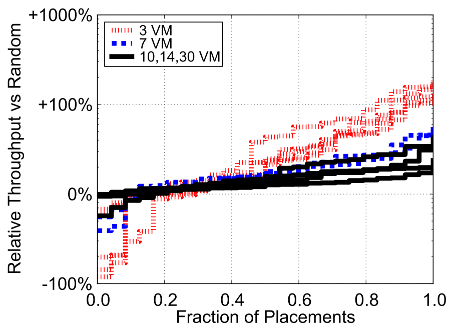
How do individual tenants fare? *In total, 85% of tenant-placement pairs improve with* SA All. Figure 7c plots the per-tenant results using *SA All*. Big workloads are almost all better off with *SA All* than *Random* (with 10% worse off). Small workloads are affected much more (up and down) with 81% doing better, and 19% worse.



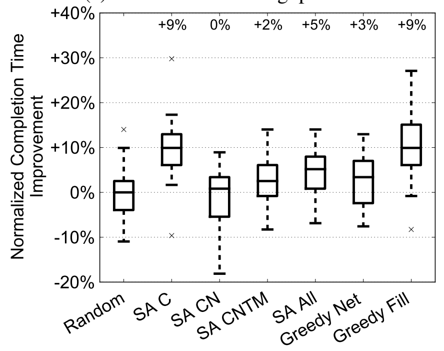
(a) Web Tenant Throughput Sum.



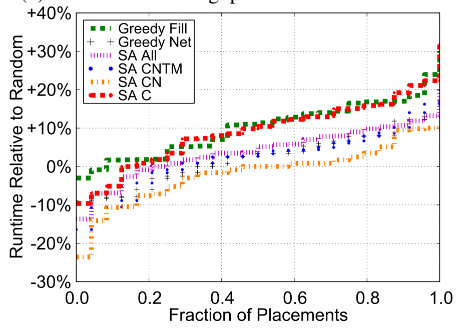
(b) Web Tenant Throughput Sum Relative CDF.



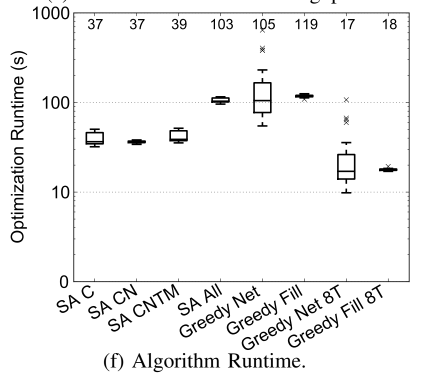
(c) *SA All* relative client throughput.



(d) Hadoop Tenant completion time normalized to Random's median of 122 seconds.



(e) Hadoop Tenant Runtime Relative CDF.



(f) Algorithm Runtime.

Fig. 7. Multi-Tenant Workload 120 VMs 80 PMs

Does a network-aware algorithm help the Hadoop tenant complete faster? *Yes, but marginally, because our Hadoop workload is CPU limited.* Figures 7d and 7e show the runtime of the 30 Hadoop VMs. *SA C* and *Greedy Fill* make Hadoop finish 9% sooner, because it is sensitive to being co-scheduled with other Hadoop processes. *SA C* minimizes the maximum CPU, so a single Hadoop VM is assigned to each PM, and *Greedy Fill* places a Hadoop VM on each of the first 30 PMs because their average network use is below any of the Web workload VMs.

How long do our optimization algorithms take to run? *One thread takes 37-119s, but eight threads only needs 17-18s.* Figure 7f compares the runtimes of the (lightly optimized) algorithms. *SA C*, *SA CN*, and *SA CNTM* take about the same time to run (37-39s). *SA All*, *Greedy Net*, and *Greedy Fill* take longer when finding the maximally utilized link, increasing median runtimes to 103-109s. *Greedy Net*'s runtime varies, because it iteratively moves VMs and their traffic, and may evaluate and move the same VM multiple times. Finally, *Greedy Fill*'s runtime is nearly constant because for each VM it examines all possible PM placements exactly once.

Greedy Net and *Greedy Fill* are easy to multi-thread by examining all possible PM locations for a VM in parallel. We tried both algorithms with 8 threads, labeled in Figure 7f as *Greedy Net 8T* and *Greedy Fill 8T*. The median runtime of *Greedy Net* decreased from 105s to 17s (617% faster) and *Greedy Fill* from 119s to 18s (661% faster).

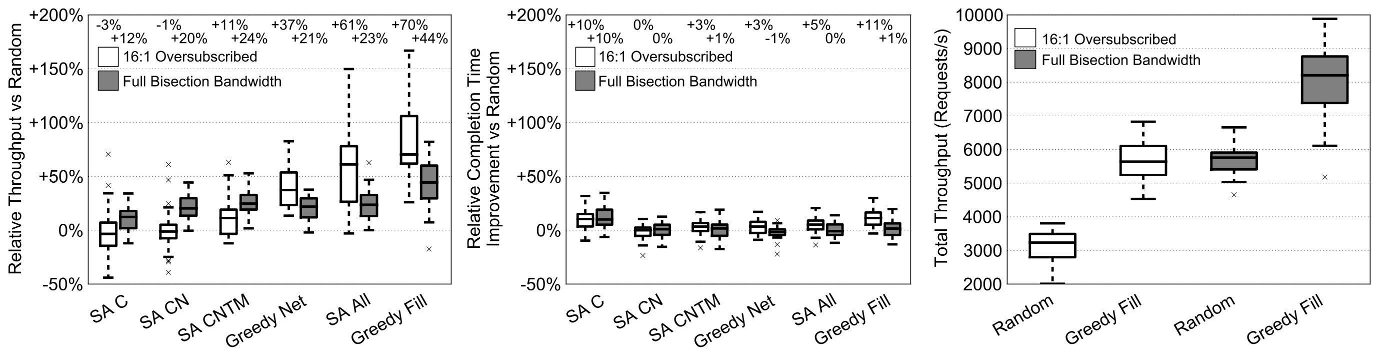
What if the core network is FBB? *The web workloads get 20-44% faster, while Hadoop is unaffected.* Figure 8 shows how much faster the Web and Hadoop workloads run when we stop oversubscribing the network. Each plot shows two sets of results: 16:1 oversubscription and no oversubscription. Figure 8a shows the surprising result that even when the network fabric is not a bottleneck, web workloads run faster when optimizing with network knowledge; median performance improves by 20% to 44%. The *SA* algorithms all improve by about the same amount because when the network is no longer a bottleneck, they all focus on minimizing the maximum utilization of PM NICs. *Greedy Fill* does much better; its technique of unassigning all VMs then assigning them one by one enables it to reach final states that *SA* does not. As expected, Figure 8b shows that CPU-limited Hadoop is unaffected by network oversubscription.

TABLE III. MEDIAN RELATIVE PERFORMANCE IMPROVEMENT OF *Greedy Fill* VS *Random* BY EDGE LINK RATE AND NETWORK CORE OVERSUBSCRIPTION RATIO.

	16:1	4:1	FBB
100Mb/s	129%	69%	58%
250Mb/s	70%	38%	41%
500Mb/s	57%	33%	-[3]
750Mb/s	37%	34%	-[3]

As network links get faster, will our algorithms still improve performance? *We see performance improvements of 33-129% for each tested link speed/network oversubscription combination.* All the results presented so far were for a network with 100Mb/s links at the edge. If we make the links faster, keeping everything else constant, we should expect *Random* performance to get better as the links get faster, because the network is less of a bottleneck, leaving less headroom for our algorithms to improve. Indeed, Table III shows this to be the case. Consider the first column for which the network link increases from 100Mb/s to 750Mb/s while keeping the core oversubscription constant at 16:1. Our best algorithm, *Greedy Fill*, still improves upon *Random* but the improvement drops

[3]Our testbed's software did not yet support this combination of edge link speed and core network bandwidth; we expect to support it soon.



(a) Web. (b) Hadoop. (c) Web.

Fig. 8. How much faster the web and Hadoop workloads run when the number is not oversubscribed.

from 129% to 37%[4]. In every combination of link-speed and oversubscription *Greedy Fill* improves performance by at least 33%, suggesting that a network-aware algorithm can improve performance in a wide variety of VDC settings.

VI. Discussion

For the optimization approach discussed in this paper to be a success, it must operate faster than significant traffic matrix or server CPU load changes. In addition, the performance improvement must be large enough to make up for performance lost during the migration phase. The time to complete an optimization cycle is the sum of the time to measure the traffic matrix, run the optimization algorithm, and actualize the improved VM placement by migrating VMs to their new locations. Our evaluation showed that the measurement phase completes in under one second and an optimization algorithm can run in the low tens of seconds. The final component is the VM migration time, which we discuss next.

VM Migration. The time to migrate a VM is primarily a function of the size of the VM's resident memory, the frequency of VM memory changes, and the network bandwidth available between the origin and target hosts. These values will all vary depending on the execution environment, however we can examine some data points for estimation.

In our cluster using a 1Gbps link, it takes approximately 21 seconds to migrate a quiescent VM with 1GB of RAM. Separate work has demonstrated that compression can reduce migration time down to seven seconds [45] and RDMA on a 10Gbps network can reduce it further to two seconds [27]. Simultaneous migrations and the congestion of the network by workload traffic will also affect the migration time. Operators could choose to prioritize migration traffic to quickly move VMs and reduce congestion, to prioritize workload traffic to minimize workload performance disruption, or let both traffic types have an equal bandwidth share. The equal or prioritized cases should be able to complete the migration phase in tens of seconds. Temporary network links could be used to further decrease migration phase time [23], [43]. Including the migration phase time range brings the the overall optimization cycle time to the range of tens of seconds to single digit minutes. Next we discuss if this optimization cycle time is sufficiently short enough to be applicable in today's data centers.

Applicability. Researchers have measured a 1500 server production data center cluster running MapReduce and found that the traffic matrix changes frequently at both 10 and 100 second time windows [28]. This matches our expectation of the communication patterns of MapReduce and its distributed data store. Our evaluation showed that *Virtue* is unable to significantly improve such workloads due to the CPU-dependent nature of the workload. There is also the practical infeasibility of migrating VMs which use a local disk containing many gigabytes or terabytes of data, common for a MapReduce workload.

Our evaluation indicates that *Virtue* works best when there are many smaller clusters of communicating VMs that can be consolidated closer together from a network standpoint to increase the effective network bandwidth between them. Intuition suggests that traffic matrices could be more stable in such an environment because there are fewer possible pairs of communicating nodes within each cluster, allowing more time for an optimization cycle or for taking advantage of improved performance from a completed cycle. Unfortunately we are unaware of any measurement studies characterizing traffic matrix stability in environments that may have similar characteristics (such as EC2 or Rackspace) that could be used to validate *Virtue*'s applicability there.

VII. Conclusion

We built Virtue because we felt that prior traffic-aware VM placement work lacked sufficient realism to convince industry that the approach (1) leads to worthwhile efficiency gains and (2) can actually be implemented in practice. This problem is compounded by a lack of public VDC traces needed to run meaningful simulations.

Our approach was to build a custom data center cluster and run everything ourselves. This meant setting up, running, and operating a multi-rack testbed with custom control software, combined with thousands of hours of machine time to run workloads on the testbed and optimization algorithms in the cloud. Every performance number we report comes from measurement; none of them are from simulation or estimation. The payoff is more-complete answers to the original motivating questions: (1) measured performance improvements up to 70% across a range of network configurations; (2) which come from adding full network knowledge, including the traffic matrix, to multiple VM placement algorithms.

Perhaps the biggest takeaway from our results is that VDC operators could save upgrade costs by using network-aware

[4]All the results in this table were measured after DNRC was upgraded to faster CPUs (the network is the same). New CPUs are 2×dual-core AMD Opteron 8214 HE running at 2210Mhz, introduced in 2006.

placement algorithms rather than upgrading to an expensive FBB network. Figure 8c highlights the performance implications of these choices for our multi-tenant web workload, using the 16:1 oversubscribed and FBB network configurations, optimized using *Greedy Fill*. Even on a 16:1 oversubscribed network, *optimization improves performance such that it nearly matches the unoptimized workload placement running on a FBB network.* If, however, you do upgrade to FBB, the same techniques can further improve the workload's performance.

In conclusion, we believe that *Virtue* represents a promising new line of research for enabling VDC operators to use network knowledge to increase workload performance on their infrastructure.

ACKNOWLEDGEMENTS

This work was supported by a Microsoft PhD Fellowship. Thanks to Jonathan Chu, Shuang Yang, and Ali Yahya for their contributions.

REFERENCES

[1] Amazon data center size. http://goo.gl/ZRH2X.

[2] Citrix xenserver workload balancing, 6.0 user guide. http://support.citrix.com/article/CTX130429.

[3] Cloudstack. http://www.cloudstack.org/.

[4] Openstack: Open source software for building private and public clouds. http://www.openstack.org/.

[5] The rise of soft switching part ii: Soft switching is awesome. http://goo.gl/3Hx0g.

[6] Web metrics: Size and number of resources. https://developers.google.com/speed/articles/web-metrics.

[7] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM CCR*, volume 38, 2008.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, San Jose, CA, April 2010.

[9] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[11] G. Attiya and Y. Hamam. Optimal allocation of tasks onto networked heterogeneous computers using minimax. In *INOC'03*, 2003.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, 2011.

[13] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[15] D. Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.

[16] N. Farrington, G. Porter, S. Radhakrishnan, H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM CCR*, volume 40, 2010.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[18] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware*, 2012.

[19] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad. Cloud-scale resource management: Challenges and techniques. 2011.

[20] P. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, 2010.

[21] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[22] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM CCR*, volume 38, 2008.

[23] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM*, 2011.

[24] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX ATC*, 2008.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[26] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *ICAC*, 2012.

[27] W. Huang, Q. Gao, J. Liu, and D. Panda. High performance virtual machine migration with rdma over modern interconnects. In *ICCC*, 2007.

[28] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.

[29] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[30] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. Netshare: Virtualizing data center networks across services. *UCSD, Tech. Rep. CS2010-0957*, 2010.

[31] N. McKeown, T. Anderson, and H. Balakrishnan. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 38(2), April 2008.

[32] M. Mcnett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An extensible framework for managing clusters of virtual machines. In *USENIX LISA*, 2007.

[33] X. Meng, C. Isci, J. Kephart, L. Zhang, and E. Bouillet. Efficient resource provisioning in compute clouds via VM multiplexing. In *ICAC*, 2010.

[34] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[35] A. Rai, R. Bhagwan, and S. Guha. Generalized resource allocation for the cloud. In *SOCC*, 2012.

[36] C. Raiciu, S. Barre, C. Pluntke, and A. Greenhalgh. Improving datacenter performance and robustness with multipath tcp. *SIGCOMM CCR*, 41(4), 2011.

[37] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *ICAC*, 2009.

[38] A. Shieh, S. Kandula, and A. Greenberg. Sharing the data center network. In *USENIX NSDI*, 2011.

[39] A. Singla, C. Hong, L. Popa, and P. Godfrey. Jellyfish: Networking data centers, randomly. *HotCloud, June*, 2011.

[40] A. Stage and T. Setzer. Network-aware migration control and scheduling of differentiated virtual machine workloads. In *CLOUD*, 2009.

[41] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. Dejavu: Accelerating resource allocation in virtualized environments. In *ASPLOS*, 2012.

[42] G. Wang, D. Andersen, M. Kaminsky, and K. Papagiannaki. c-through: Part-time optics in data centers. In *SIGCOMM CCR*, volume 40, 2010.

[43] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan. c-through: Part-time optics in data centers. In *ACM SIGCOMM*, 2010.

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

[45] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *IEEE CLUSTER*, 2010.

[46] M. Zhao and R. J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC*, 2007.