Implementing Scalable, Network-Aware Virtual Machine Migration for Cloud Data Centers

Fung Po Tso*, Gregg Hamilton*, Konstantinos Oikonomou†, and Dimitrios P. Pezaros*
*School of Computing Science, University of Glasgow, G12 8QQ, UK
†Department of Informatics, Ionian University, 49100 Corfu, Greece
Email: {posco.tso, dimitrios.pezaros}@glasgow.ac.uk, okon@ionio.gr, g.hamilton.3@research.gla.ac.uk

***Abstract*—Virtualization has been key to the success of Cloud Computing through the on-demand allocation of shared hardware resources to Virtual Machines (VM)s. However, the network-agnostic placement of VMs over the underlying network topology can itself be a factor of performance degradation by causing congestion at the core layers of the infrastructure where bandwidth is heavily oversubscribed. In this paper, we design and implement S-CORE, a scalable live VM migration scheme to dynamically reallocate VMs to servers while minimizing the overall communication footprint of active traffic flows. We evaluate S-CORE over diverse aggregate load and coordination policies. Our results show that it can achieve up to a 87% communication cost reduction with a limited number of migration rounds, and can be easily accommodated within commodity hardware and hypervisor architectures. The associated memory, CPU, and network overhead are also minimum under typical Cloud Data Center workloads.**

***Index Terms*—Cloud Computing, Data Center, Distributed Virtual Machine Migration, Network Management.**

I. Introduction

Resource virtualization has facilitated the emergence of elastic computing environments where ICT infrastructure can be outsourced and expanded on-demand, saving clients from large up-front and maintenance costs. At the same time, for Cloud computing providers, the significant capital outlay for setting up a Cloud site makes return-on-investment through maximization of resource usage efficiency crucial [1]. Considerable research effort has recently focused on the underlying Data Center (DC) infrastructure design [2][3] as well as into system resource allocation mechanisms [4].

A number of studies have concentrated on the efficient placement, consolidation and migration of Virtual Machines (VMs) to maximize server-side resources, such as CPU, RAM and network I/O [5][6]. At the same time, it has been repeatedly reported that virtualization itself can create network congestion which evolves as the major performance bottleneck for DCs [2][7]. The use of server-side metrics alone, takes no account of the resulting traffic dynamics in an already over-subscribed network [8][9]. Network-aware VM placement studies to date typically only consider the problem of initial placement, ignoring subsequent changes in traffic demands [4][10]. The limited body of work that has addressed network-aware VM migration, has either relied on complex centrally-controlled optimization algorithms [11][12], or penalized bandwidth in pursuit of fault tolerance without considering the network cost of migration [13].

Experiments over Amazon's EC2 revealed that a marginal 100 msec additional latency resulted in 1% drop in sales, while Google's revenues dropped by 20% due to a 500 msec increase in search response time [14]. It is therefore becoming apparent that any resource virtualization and live VM migration scheme will need to incorporate mechanisms to reduce the resulting network cost, while optimising server resource usage.

In this paper, we present the implementation and experimental evaluation of *S-CORE*, a scalable VM migration scheme that minimizes the overall communication cost of the DC topology while adhering to server-side resource capacity limits. By assigning distinct link weights at the different layers of the DC infrastructure and taking into account the amount of data traffic routed over these links, a function of the network-wide communication cost is defined that can then be minimized in terms of the contributing pairwise aggregate VM traffic load. *S-CORE* adopts a distributed approach based on information available locally at each VM to inform migration decisions, rather than using in-network or global statistics, a property that makes it scalable and realistically implementable over large-scale DC infrastructures. It iteratively localizes pairwise VM traffic to lower-layer links where bandwidth is not as over-subscribed as it is in the core, and where interconnection switches are cheaper to upgrade [15]. To the best of our knowledge, *S-CORE* is the first scheme to incorporate a distributed migration solution with multiple distinct policies. Our implementation is an easy plug-in module to the Xen [16] hypervisor and is completely transparent and backwards compatible to the hosted VMs. Our results show that *S-CORE* can significantly reduce traffic over the high-cost links at the core of the topology that are shown to experience congestion, even when lower communication layers are under-utilized [7][8]. *S-CORE* can achieve an overall communication cost reduction of as high as 87% (i.e., only deviating by 13% from optimal allocation), as this is approximated by centralized algorithms that assume global traffic knowledge but are prohibitively expensive to implement in practice. In addition, our implementation shows that *S-CORE* is light-weight and inexpensive to operate as part of the Xen hypervisor. With typical DC traffic loads, S-CORE only incurs a 0.01% CPU utilization and 187KB memory footprint.

The remainder of this paper is structured as follows. Section II presents *S-CORE* and its token policies. Section III describes the design and implementation of *S-CORE* within the

Xen hypervisor. Section IV presents an extensive evaluation of S-CORE under diverse migration policies and demonstrates the significant performance gains achieved while keeping overhead low. Section V discusses related work, and Section VI concludes the paper.

II. System Definition and Description

A typical reference network architecture for DCs is a layered tree with multiple redundant links [17][2][3]. We defined network links that connect ToR and servers as 1-*level links*, those between ToR and aggregation switches as 2-*level links*, etc.

TABLE I: List of Notations.

Notation	*Description*
$\mathbb{V}$	Set of all VMs in the DC
$\mathbb{V}_u$	Set of VMs that communicate with VM u
$\mathcal{A}$	Allocation of VMs to servers
$\mathcal{A}_{opt}$	Optimal allocation
$\mathcal{A}_{u\to\hat{x}}$	New allocation after migration $u \to \hat{x}$
$\ell^{\mathcal{A}}(u,v)$	Communication level between VMs u and VM v
c_i	Link weight for a i-level link
$\lambda(u,v)$	Traffic load between VM u and VM v per time unit
$C^{\mathcal{A}}(u)$	Communication cost for VM u for allocation $\mathcal{A}$
$C^{\mathcal{A}}$	Overall communication cost for allocation $\mathcal{A}$
$u \to \hat{x}$	Migration of VM u to a new server $\hat{x}$
c_m	Migration cost

Moving up the network hierarchy, operators face steep technical and financial challenges in sustaining high bandwidth, and the over-subscription ratio increases sharply along the path. When a packet enters the network, it incurs a communication cost (of consuming network bandwidth), which increases when moving up the hierarchy, i.e., $c_1 < c_2 < c_3 < c_4$). We formalize the problem of communication cost reduction and the concepts of allocation, communication level, and link weights, with important notations listed in Table I.

The *overall communication cost* for all VM communications over the DC is defined as the aggregate traffic, $\lambda(u,v)$, for all communicating VM pairs and all communication levels, $\ell^{\mathcal{A}}(u,v)$, multiplied by their corresponding link weight c_i.

$$C^{\mathcal{A}} = \sum_{\forall u\in\mathbb{V}} \sum_{\forall v\in\mathbb{V}_u} \lambda(u,v) \sum_{i=1}^{\ell^{\mathcal{A}}(u,v)} \mathsf{c}_i. \qquad (1)$$

Let $\mathcal{A}_{opt}$ denote an *optimal allocation*, such that $C^{\mathcal{A}_{opt}} \leq C^{\mathcal{A}}$, for any possible $\mathcal{A}$. It is shown in [18] that this problem is of high complexity and specifically NP-complete, therefore there exists no possible polynomial time solution for centralized optimization. Even if there was however, the centralized approach would require global knowledge of traffic dynamics which is prohibitively expensive to obtain in a highly dynamic environment like a DC.

This calls for a scalable and efficient alternative, and thus we have formulated the following *S-CORE distributed migration policy* for virtual machines: A VM u migrates from a server x to another server $\hat{x}$, provided that Equation 2 is satisfied, i.e., given the observed amount of aggregate traffic, a VM u individually tests the candidate servers (for new placement) and migrates only when the benefit outweighs the migration cost c_m. We refer interested readers to [18] in which we have formulated and proved the S-CORE scheme.

$$2 \sum_{\forall z\in\mathbb{V}_u} \lambda(z,u) \left(\sum_{i=1}^{\ell^{\mathcal{A}}(z,u)} \mathsf{c}_i - \sum_{i=1}^{\ell^{\mathcal{A}_{u\to\hat{x}}}(z,u)} \mathsf{c}_i \right) > c_m, \qquad (2)$$

***Token Policies**:* As S-CORE operates in a distributed manner, VMs must know when they are allowed to migrate. We achieve this through the passing of a *token* containing information for all VMs that consists of a VM ID and an associated communication level value.

We have defined four token policies:

- *Round-robin token policy* passes the token among VMs based on their IDs in an ascending order, assuming each VM has a unique and totally ordered identifier. The basic round-robin policy may not be efficient in all cases, such as when the token is passed on to a VM that will not migrate, wasting one iteration.
- Centralised *global token policy* builds and distributes the token based on the highest pairwise communication cost reduction. Such a policy requires computing and sorting communication cost centrally and is thus not scalable.
- *Distributed token policy* prioritizes VMs for whom network communication passes through the highest-layers in the network. Links are most costly at this level and it is therefore reasonable to assume that migration is likely to take place.
- *Load-aware* is a variant of the distributed token policy, which considers the aggregate network load (incoming and outgoing) for each VM. VMs at the same communication level, but with higher aggregate load, receive the token first.

Migration decision is a task which requires monitoring aggregate traffic for a certain period of time to capture a reliable traffic demand. Token passing is therefore a relatively infrequent activity, operating at an interval from a few seconds up to several hours.

III. Design and Implementation

In this section we discuss a a real-world implementation of the S-CORE migration system, highlighting the rationale as well as the operational and design details of the individual components.

A. Implementation Environment

Our main development platform is Xen [16] with Ubuntu 12.04 as dom0 (domain zero, the initial domain started by Xen on boot). The management interface we used with Xen is *xm* [19], which is written in Python and communicates with Xen to perform tasks such as VM instantiation, migration

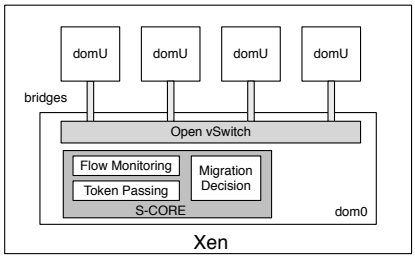


Fig. 1: S-CORE Architecture

and probing for details of individual VMs. To allow for easy communication with the functions of xm required for S-CORE, and given the distributed and periodic nature of the algorithm where a hypervisor must make a migration decision only when a co-located VM receives the token, we have implemented S-CORE in Python.

In order to enable network communication between co-located VMs on a server, as well as between VMs and the outside world, a network bridge is created in dom0 through which the network traffic to and from all VMs on a physical host passes. While the basic Linux bridge utilities offer limited capabilities and do not allow access to individual flow statistics, Open vSwitch [20] can be used as a drop-in replacement with bridge compatibility enabled. Open vSwitch provides flow-level access and manipulation to enable flow-level monitoring at the hypervisor level for all local VMs, rather than on a per VM basis.

B. VM vs Hypervisor

While conceptually S-CORE relies on VMs passing the token amongst themselves and making their own migration decisions, in practice this is unsuitable for a number of reasons. The operational paradigm in system virtualization is that the hosts being virtualized should not be aware that they are in fact running within a Virtual Machine. Consequently, the presence of a hypervisor should also be transparent, not requiring VMs to communicate directly with the underlying hypervisor.

This eliminates the possibility of a VM itself deciding to migrate, since migration is a facility provided by the hypervisor. Enabling such an ability would violate the VM transparency and would require it to directly interact with the hypervisor to allow migration. In S-CORE, we decided to implement our solution within dom0 of the Xen hypervisor itself. The modular architecture of S-CORE is shown in Fig. 1. The benefit of having such a modular architecture is not only to keep transparency intact but also to make the system easily upgradable.

C. Flow Monitoring

In order to enable an accurate measure of the aggregate throughput between communicating VMs some form of flow statistics gathering is required. However, Open vSwitch only maintains flows for as long as they are active and discards any inactive flows after 5 seconds, hindering the accumulation of any long-term history. To overcome this limitation, we have implemented our own flow table for storing flow-level statistics. For the purposes of S-CORE, the flow table must support the following operations: *Fast addition of new flows*; *Updating existing flows*; *Retrieval of a subset of flows, by IP address*; *Access to the number of bytes transmitted per flow*; *Access to flow duration, for calculation of throughput*.

The flow table will be periodically updated through polling Open vSwitch for datapath statistics allowing for the storage of flows for as long as it is required. Flows will be stored from when they start up till a migration decision is made for a VM. As the most frequent operation on S-CORE's flow table is the addition of new flows or the updating of existing flow counters, we require the ability to easily add new flows, and to also perform quick lookup and update of existing flows. To achieve this, we use a hash table structure to store flow data. Each entry stores the MAC address associated with a particular source IP and a hash table for quick lookup of the destination flow data, using the source IP address as a key. The destination hash table is keyed by destination IP, and stores protocol type, source and destination ports, the number of bytes transmitted in that flow, and a timestamp of when the flow was started. Open vSwitch identifies *datapaths* as a flow in a single direction, so bidirectional flows are composed of two individual datapaths. To address this, two data structures are used, with the second storing destination IP addresses as the main key.

D. Token Passing

S-CORE is a distributed migration system, requiring the use of a token passed between VMs in order to allow the localized migration decisions to take place. When a token addressed to a VM is received, the concerned VM needs to evaluate the overall communication cost between itself and all neighbors it communicates with. It must then evaluate if it can achieve a lower overall communication cost by migrating to a different physical host. If a lower communication cost is achievable and the destination host has available resources, then migration should take place. We have used the IPv4 address of a VM as the 32-bit VM ID carried in each token. As all VMs must have a unique IP address, this provides a unique identifier simplifying the token passing process, as the token can be sent directly to the IP address of the next VM.

To efficiently pack the token for network transmission, it is stored and transmitted as a block of 32-bit unsigned integers. Similarly, for the distributed token policy that requires an additional highest communication level entry, we specify an 8-bit value that follows the VM ID.

Since our implementation stores IP addresses as VM IDs and passes the token to each IP address in turn, a question arises: How does the dom0 acquire the token? Instead of running a token listening server on each VM, a token listening server runs on a known port in dom0 of each hypervisor. For the token server to receive the token, a NAT redirect is installed in dom0's *iptables*, redirecting messages for a particular port to dom0 itself. When dom0 holds the token for a VM it hosts, it is then able to conduct the migration decision process on behalf of the VM, before forwarding the token along.

E. Xen Wrapper

It is possible to retrieve the MAC addresses of VMs using the xm tools as the *xm* management interface for Xen (or rather, *xend*, the control daemon that xm communicates with) does not store information about the IP addresses of each running VM. The xm toolkit is itself written in Python, which allowed us to create our own Python wrappers around most of the functions concerned with listing VMs, retrieving network details of a VM, and migrating a VM.

Given that IP addresses are passed in the token, and xm can retrieve the MAC addresses of individual VMs, how can these be mapped to each other to identify a particular VM that should be migrated? As mentioned in Section III-C, the flow table also stores a MAC address with each IP address. This allows dom0 to do a lookup for the MAC address associated with the IP address in the token it has received, and then make calls to xm to find the particular VM that matches that MAC address, and perform a migration, if necessary.

F. Migration Decision

1) Aggregate Throughput Calculation: When dom0 receives the token for a co-located VM, the first step is to calculate the aggregate load between that VM and all the neighbors it communicates with. This is achieved by looking up S-CORE's flow table for the source and destination flows associated with that IP address, and calculating the total number of bytes transmitted. As each flow stores a timestamp of when it was started, these timestamps can be used to deduce the length of time for which the flow statistics have been gathered since last being cleared, allowing calculation of the aggregate throughput in the form of bytes-per-second.

2) Location Identification: Once the aggregate throughput to each communicating neighbor has been calculated, the communication cost must be evaluated. In real terms, the communication cost can be derived from the number of hops between a VM and any neighbor that it is communicating with. This could be achieved by a network diagnostics tool such as, e.g., *traceroute*, but layer 2 switches would not show up as hops in this case. Another alternative would be a lookup service listing the cost for any VM to communicate with another. However, VMs carry their IP addresses when they migrate, which renders this method unusable in a Data Center with a dynamically changing VM allocation.

On the contrary, the physical servers and the hypervisors running on them, do not move around. This makes a reliable lookup service possible, and is the option chosen for S-CORE. As we store a flow table of the IP addresses each VM communicates with, we can probe neighboring VMs to find out the IP address of their dom0. Similar to the token passing method, we can send a custom *location request* packet to the IP address of each communicating VM. A NAT redirect in dom0 of each hypervisor will then capture this packet and pass it to dom0, which can send a *location response* containing dom0's static address back to the VM.

With that information, the dom0 currently holding the token can make a lookup into a precomputed location cost mapping with its own IP address and the IP address of each underlying dom0 of communicating VMs. The location cost for each VM is then combined with each aggregate throughput value to produce an overall communication cost for each neighboring VM, as well as a total cost for its current allocation.

3) Migration Location Identification: Since we now have the IP addresses of each hypervisor, after probing for the communication cost, we can order neighboring VMs from highest to lowest communication levels and probe each server to see if it is able to host the current VM. A *capacity request* packet is sent to the hypervisor of the neighboring VM with the highest communication cost, which responds with a *capacity response* packet, detailing how many more VMs it is able to host, and the amount of RAM it has available (to account for VMs with heterogeneous RAM requirements).

If the hypervisor has the capacity to host the additional VM, the dom0 holding the token will then calculate the overall communication cost for the VM if it were to migrate to that hypervisor; it will migrate there if the communication cost is reduced, and not migrate otherwise. If the hypervisor hosting the neighboring VM with the highest communication cost does not have the capacity to host the VM for which migration has been instructed, the hypervisor of the VM with the next-highest communication cost will be subsequently considered. This operation is repeated until a hypervisor with available capacity is found, the overall communication cost of moving to that hypervisor is reduced, and a migration is conducted. If no suitable hypervisor is found, the algorithm terminates and the token is passed on to the next host, i.e., the next hypervisor.

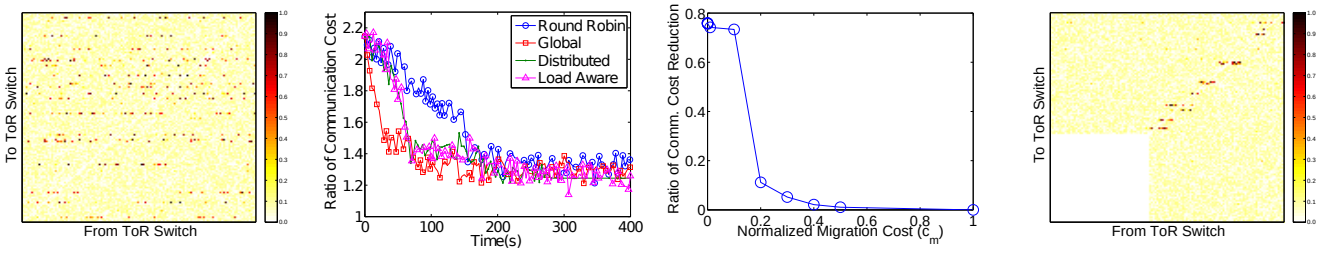
IV. Evaluation

We have implemented S-CORE over experimental testbed and simulated environments in order to evaluate its operational feasibility and overhead, and the algorithm's scale properties, respectively. This section discusses the results of an extensive evaluation and highlights the communication cost reduction achieved, as well as the scheme's instrumentation and system footprint.

A. Simulation Setup

We have simulated S-CORE's communication cost reduction with the different token policies over a layered DC topology, using the **ns-3** network simulator [21].

Simulation Environment: The simulated topology is comprised of 2560 hosts (128 ToR switches, 20 hosts per rack) which can sufficiently capture hierarchical link oversubscription ratio at aggregate and core links found in the DCs [2]. In order to model a typical DC server environment, each host can accommodate at most 16 VMs assuming 2 VMs per core, each occupying 1GB of RAM. Increasing a VM's resource requirements is equivalent to combining, for example, two or more VMs' resources into one. We set $\mathtt{c}_1 = e^0$, $\mathtt{c}_2 = e^1$, $\mathtt{c}_3 = e^3$ and $\mathtt{c}_4 = e^5$ for each link cost. VM migration carries its own cost in terms of network bandwidth for moving a VM's memory contents and VM downtime. We set our migration



(a) *(b)* *(c)* *(d)*

Fig. 2: (a) Normalized traffic matrix between Top-of-Rack switches; (b) Communication cost reduction with dynamic DC traffic flows; (c) Ratio of communication cost reduction under various migration cost threshold settings with the distributed token policy. (d) Normalized traffic matrix between Top-of-Rack switches after 5th iteration;

overhead cost c_m to zero to allow for a fair comparison among the centralized approach and S-CORE.

DC Traffic Pattern: We have also built a DC traffic generator to test S-CORE under realistic DC load, as these have been reported in a number of DC measurement studies [22][2][23][7]. The sample of a 10s Traffic Matrix (TM) of all ToR switches is given in Fig. 2a, which exhibits identical TM properties with those unveiled in [23]. As a benchmark, the centralized optimal values are approximated using the Genetic Algorithm (GA).

B. Simulation Results

The results in Fig. 2b show that, despite the dynamic instantiation of new traffic flows (i.e., small spikes along the curves), S-CORE can still adapt and converge quickly to approximation of optimal network-wide VM allocations calculated by the genetic algorithm, which is computed using the TM given in Fig. 2a for all scenarios. We note that this optimal approximation is only used for reference here and should vary over time due to fluctuating traffic dynamics. In all four scenarios, the *global* token policy constantly exhibits best performance in terms of communication cost reduction speed and proximity to the optimal cost. However, it requires global knowledge of the traffic dynamics and can therefore be prohibitively expensive to implement in practice, even under a distributed migration algorithm. The basic *round-robin* policy exhibits the slowest cost reduction and largest deviation from the approximate optimal amongst all four token passing policies. The less expensive *distributed* and *load-aware* token passing policies produce highly comparable performance to the global one. All token policies converge and stabilize when the VM distribution considerably reduces the overall communication cost.

To reflect the fact that c_m is usually non-zero due to VM migration overhead, we ran simulations with different c_m threshold values. Fig.2c shows that if we increase c_m to 10% of overall communication cost, a pronounced communication cost reduction can still be seen. The ratio communication cost reduction plunge sharply if we further increase c_m to 20% and more. This phenomenon demonstrates that S-CORE will work well by setting reasonable c_m accordingly. Fig. 2d depicts that after VMs migrate, the number of ToR hotspots is significantly reduced. Even though there are still ToR hotspots, these ToRs are in close physical proximity, which means that inter-ToR traffic flows remain within the lower levels of the topology hierarchy. An obvious advantage of the locality property of S-CORE is that these idle servers can be powered down to reduce the energy consumption of the DC, addressing the aims of studies on partial shutdown of servers or network elements [6][25].

VM stability is crucial for dynamic VM migration algorithms as unstable VM migrations (i.e., oscillations) can themselves potentially have a big impact on the network and servers. Whilst no dynamic algorithm can completely eliminate the possibility of VM oscillations, we argue that S-CORE can minimize short-term oscillations for two reasons. First, S-CORE uses the average rate of data exchanged between VM pairs over a certain time window, which can be set suitably long to capture the dynamism of the environment while not responding to instantaneous traffic bursts. Second, VMs do not migrate arbitrarily nor do they measure individual flow arrivals and completion. Rather, they only consider migration periodically, when they receive the migration token, and their computation is based on aggregate traffic load over that period. Therefore, the short-term effects of sudden arrivals of mice flows are canceled out when averaged over one iteration of the algorithm.

C. Testbed Setup

We have used Intel's P4 3GHz servers with 2GB RAM running Xen hypervisor ver. 4.1 with Ubuntu server 12.04 as dom0. VMs are of ubuntu 10.04 with 196MB RAM allocated. In the experiments, initially we started two VMs on each server. Each VM hosts a HTTP server as well as an *iperf* server and client. We have also set-up a Network File System (NFS) server, since live migration requires VM images to reside on shared storage through which only transferring of memory state is needed while keeping the actual file system intact.

D. Module Evaluation

Given that S-CORE modules run within the hypervisor rather than in the VMs themselves, it is imperative that S-CORE can suitably monitor and perform migration decisions for all the VMs a hypervisor hosts while consuming minimum

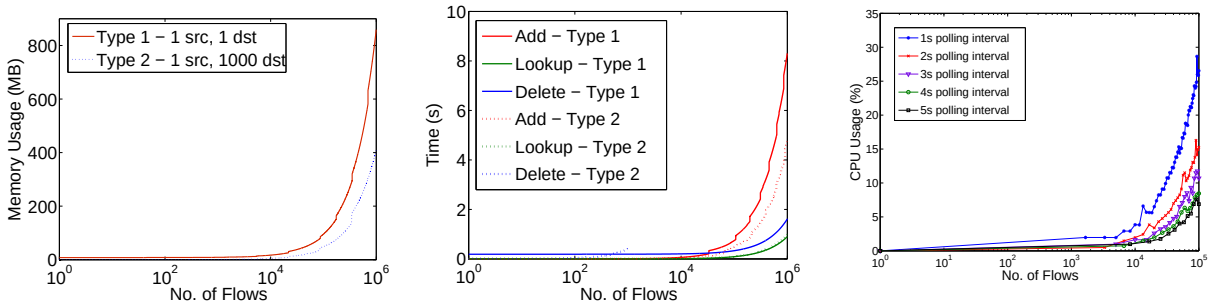


Fig. 3: (a) Flow table memory usage; (b) Flow table operation times for up to 1 million unique flows; (c) CPU utilization when updating flow table at varying polling intervals. All with log-scale at the x-axis.

hypervisor resources, thus leaving most resources available to the actual VMs. While a number of studies have revealed that server and network resources are mostly under-utilized, we aim to stress-test our implementation to ensure that S-CORE will not misbehave in worst case scenarios.

The first main module in S-CORE is the flow table, which stores TCP and UDP flow data for the VMs running on the hypervisor. It implements the requirements of adding new flows, updating the number of bytes transferred for existing flows, retrieving flow data, and clearing old flows. In order to stress-test the resource consumption of adding flows to the flow table, experiments were conducted where up to 1 million flows were generated and added to the table, even though a realistic typical load is 10 active flows per VM [22][2][23][7]. We defined two different sets of flows: The first set is 1 million flows with all source IP addresses being unique (type 1). This results in a new entry being created at the root of the flow table for each flow. The other set is 1 million unique flows, where groups of 1000 flows share the same source IP address (type 2).

As shown in Fig. 3a, the size of the flow table scales sub-linearly. With 10,000 flows, the flow table has a memory footprint of only 4MB and 16MB for type 2 and type 1 flows, respectively; with 100,000 flows, the corresponding footprint is 46MB and 91MB. The substantially different memory usage values are down to the structure of the flow table. As there will be a limited number of VMs running on a server, the flow table stores entries grouped by IP address at the root of the hash table. When 1 million flows with unique source addresses are used, this results in 1 million entries being added to the root of the underlying data structure. However, when there are 1 million flows with 1000 overlapping source IP addresses, only 1000 entries are added to the root of the dictionary, while the remaining flow data is added to the nested structures pointed to by these 1000 entries.

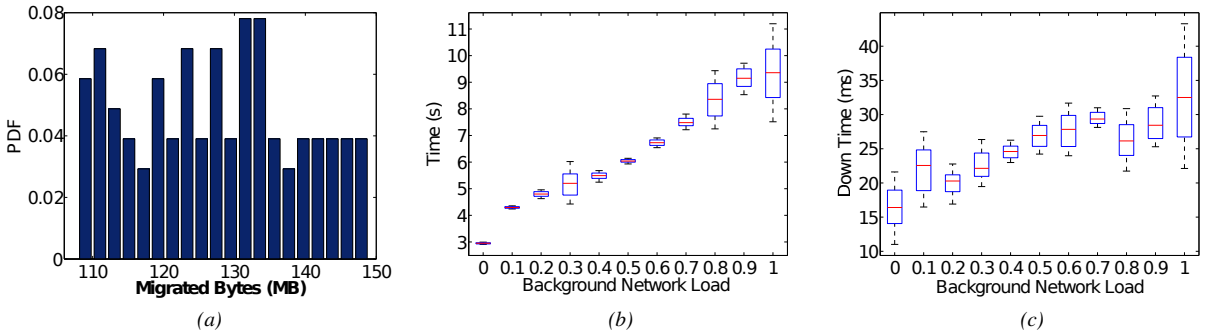
However, a number of studies have reported that the total number of concurrently active flows between VMs is much more contained: in a production cluster of 1,500 servers, the median number of active correspondents for a server are two other servers within its rack and four servers outside the rack. A busy server can talk to all servers in its rack or 1-10% outside the rack [22]. At the same time, in a large-scale cloud DC, the number of concurrent flows going in and out of a machine is still almost never more than 100 [2]. With a more realistic scenario where every virtual server concurrently sends or receives 10 flows, with 100 in the worst case, we anticipate that actual memory consumption of the flow table will be between 24.75 KB - 186.47 KB for a hypervisor hosting 16 VMs.

To understand the time taken to perform the different operations on the flow table, we have measured the time to add, lookup and delete flows, summing the times over the number of flows, for the same sets of flows. Fig. 3b shows the time to perform various flow table operations with differing numbers of flows in a single operation. From Fig. 3b we can see that flow addition, lookup and deletion operations all require less time on a flow table with a type 2 flow set. Nevertheless, addition, lookup and deletion operations will not need more than 100ms for a realistic DC production workload of 100 concurrent flows.

In order to evaluate the run-time impact on the processing capability of the physical servers, we have measured the CPU usage of the flow table in its normal background running state. The experiment consisted of running a separate thread maintaining the flow table which periodically updates itself with new flow information from Open vSwitch, adding an increasing number of new flows each time. This was varied over update periods from 1 to 5 seconds. We have measured the CPU clock time for adding each flow and calculated a percentage of CPU utilization, as shown in Fig. 3c. It is evident that the performance impact for adding up to 10,000 flows is negligible for any polling interval accounting for less than 5% CPU utilization. In the best case for 10,000 flows added or updated each time, CPU utilization was only around 1% at a polling rate of 5 seconds, while the worst case CPU utilization was 3.6% at a polling rate of 1 second. For a more realistic load of 1,000 flows, the best and worst cases are 0.002% and 0.01%, respectively.

E. Network Overhead

Similar to other DC management schemes, S-CORE will inevitably impose control overhead on the network. An im-



(a) *(b)* *(c)*

Fig. 4: (a) PDF (probability density function) of migrated bytes per migration; (b) Virtual machine migration time and (b) down time under various link load condition. Background Network Load is the ratio of 1Gb/s CBR.

properly designed control scheme may overwhelm the network with additional –control– load, but *how much overhead will S-CORE create?* First, S-CORE uses a token, which is exchanged between VMs and consists of a 32-bit ID and an 8-bit communication level for each VM to facilitate and control synchronous VM migration. The size of the token is therefore proportional to the total number of VMs in the DC. A typical production DC has 100,000-500,000 servers, in which case the token size will merely be between 500KB - 2.5MB.

Migration of memory state is an overhead too (as actual file system stays in the NFS server). During the memory migration, in particular the iterative pre-copy stage [26], the hypervisor copies all memory pages from source to destination. If some memory pages change (become "dirty") during this process, they will be re-copied. Therefore, the actual amount of data being copied over the network is largely dictated by the page dirty rate since higher page dirty rates result in more data being transferred over the network. Fig. 4a shows the PDF of the number of migrated bytes for each VM migration captured in our experiments. The spread appears flat and wide due to the highly varying memory dirty rate at the time when a VM is being migrated. However, with minimal installation of a ubuntu 10.4 VM image and a few lightweight test services running inside, e.g., a HTTP server and a SSH server, the VM memory size to migrate are all below 150MB. The mean and standard deviation of migrated bytes are 127MB and 11MB respectively. However, given the link capacity in today's Cloud DC networks, this additional control load is negligible (1-second's worth of transmission time over a 1 Gb/s link). Even a typical highly loaded commercial web server has about 800MB memory usage [26], which is still a completely affordable network overhead for an infrequent migration schedule in line with our token policy. In addition, the network overhead of performing a one-off or infrequent migration for such a service may result in a lower overall communication cost in the long term, which is beneficial to DC operators.

F. Impact of Link Load

Sometimes, transient traffic bursts may lead to busier links. *Will migration over busier links worsen the machine down-time?* To answer this question, we set up an experiment in which two servers, i.e., dom0, generated a constant bit rate UDP stream as cross-traffic while migrating a VM from one to the other. We then captured the migrated packets with *tcpdump* to determine the total migration time by comparing the time difference between the first and the last packets received. We determine the downtime of a VM by probing the migrating VM with high precision ping (*fping*) with ping interval set to 1ms.

Fig. 4b and Fig. 4c illustrate the distribution of VM migration time and down-time, respectively, for the migrated bytes shown in Fig. 4a under varying background traffic on their local links. As depicted from Fig. 4b, the mean total migration time increases from 2.94s for no background traffic to 4.29s with 100Mb/s of background traffic. With a background traffic load between 100Mb/s and 1Gb/s, migration time increases sub-linearly from 4.29s to 9.34s. Migration time shows a larger spread for highly utilized links ($\geq$70% of link capacity utilized) due to transferring the large spread of migrated memory size, as shown in Fig. 4a over the limited amount of available link capacity. In particular, TCP's congestion control may be triggered in some cases, causing a long tail in migration completion time.

Most importantly, in the DC environment, the server down time is more often measured by the period of time that the VM is unable to service user requests. This happens in the stop-and-copy stage [26] of the live migration process where a VM on a server is suspended, and its CPU state and any remaining inconsistent memory pages are then transferred to another server. As shown in Fig. 4c, down-time is an order of magnitude smaller than the migration time and only increases mildly from 16.38ms to 32.63ms with increased background traffic on the link. This implies that while higher link utilization does have some impact on VM down-time, this does not cause significant service disruption as the amount of data transferred during this stage is often minimal and can be finished quickly over the network (most data being migrated in the previous pre-copy stage [26]).

V. Related Work

VM live migration [26] is typically employed to improve server-side performance in terms of physical resources and power consumption [5]. Consolidation concerns the grouping

of VMs on servers via migration, often in order to reduce power usage [6] [27] or even to limit thermal dissipation in the DC [28]. A network-aware variation of this is VMFlow [25], which consolidates VMs to reduce the number of network switches that must be powered on, while satisfying greater network demands than consolidation focusing solely on server-side metrics.

While the above studies have mainly addressed server-side performance issues, there are several studies that, similar to our own, attempt to specifically target the problem of network performance-based migration [4][11][10][29][12][13]. Works on initial VM placement attempt to minimize the overall DC network cost matrix [4] or combine requirements placed on multiple resources into a constraint problem [10]. However, as they address initial placement, they do not adequately tackle the problem of VM migration in a running Cloud DC nor do they take the current state or the variability of DC traffic dynamics into consideration.

VI. Conclusion

In this paper, we have presented the software design and implementation of *S-CORE*, a scalable VM live migration scheme to dynamically reduce the network-wide communication cost of Cloud Data Center topologies. Through distributed VM migration, S-CORE alleviates congestion from the core layers of the Data Center, that can otherwise become the bottleneck and the main factor of performance degradation of the overall infrastructure. We have evaluated S-CORE under four different prioritization policies and demonstrated that we can achieve up to 87% communication cost reduction compared to an approximate optimal approach with a very limited number of VM migrations. We have provided a real implementation of S-CORE for the Xen hypervisor that incurs minimal memory, CPU, and network footprint under typical DC workloads, while keeping the migration downtime very low even in the presence of considerable cross-traffic.

Overall, we have argued that optimizing resource usage in Cloud Data Centers through measurement-based, network-aware VM migration is feasible over existing DC network topologies, server configurations and hypervisor architectures, and can prove a powerful mechanism for providers to significantly increase the usable capacity of their infrastructures even at the onset of highly varying traffic dynamics.

References

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, Dec. 2008.

[2] 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 *Proc. ACM SIGCOMM'09*, 2009, pp. 51–62.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM'08*, 2008, pp. 63–74.

[4] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM'10*, Mar. 2010, pp. 1–9.

[5] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *USENIX NSDI'07*, 2007.

[6] A. Verma, P. Ahuja, and A. Neogi, "pMapper: power and migration cost aware application placement in virtualized systems," in *Proc. ACM/IFIP/USENIX Int. Conf. on Middleware*, 2008, pp. 243–264.

[7] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM Internet Measurement Conf. (IMC'10)*, 2010, pp. 267–280.

[8] G. Wang and T. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proc. IEEE INFOCOM'10*, Mar. 2010, pp. 1–9.

[9] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proc. ACM SIGCOMM Internet Measurement Conf. (IMC'10)*, 2010, pp. 1–14.

[10] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible, "Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement," in *IEEE Int. Conf. on Services Computing (SCC'11)*, July 2011, pp. 72–79.

[11] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, "Remedy: Network-aware steady state VM management for data centers," in *Proc. IFIP TC 6 Networking Conf.*, ser. LNCS, 2012, vol. 7289, pp. 190–204.

[12] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera, "A stable network-aware VM placement for cloud systems," *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGRID '12)*, pp. 498–506, 2012.

[13] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," 2012, pp. 431–442.

[14] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD*. ACM, 2007, pp. 959–967.

[15] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM'11*, 2011, pp. 266–277.

[16] "Xen hypervisor." [Online]. Available: http://xen.org/

[17] Cisco, "Data center: Load balancing data center services," 2004.

[18] F. P. Tso, K. Oikonomou, E. Kavvadia, G. Hamilton, and D. P. Pezaros, "S-core: Scalable communication cost reduction in data center environments," School of Computing Science, University of Glasgow, Tech. Rep. TR-2013-338, 2013. [Online]. Available: http://www.dcs.gla.ac.uk/publications/PAPERS/9397/migration_techreport.pdf

[19] "Xen management user interface." [Online]. Available: http://wiki.xen.org/wiki/XM/

[20] "Open vswitch." [Online]. Available: http://openvswitch.org/

[21] "The ns-3 network simulator." [Online]. Available: http://www.nsnam.org/

[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. ACM SIGCOMM Internet Measurement Conference (IMC'09)*, 2009, pp. 202–208.

[23] S. Kandula, J. Padhye, and P. Bahi, "Flyways to de-congest data center networks." Proc. ACM HotNets, 2009.

[24] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[25] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman, "VMFlow: Leveraging VM mobility to reduce network power costs in data centers," in *Proc. IFIP TC 6 Networking Conf.*, ser. LNCS, 2011, vol. 6640, pp. 198–211.

[26] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," ser. NSDI'05. USENIX Association, 2005, pp. 273–286.

[27] M. Cardosa, M. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server environments," in *IFIP/IEEE Int. Symp. on Integrated Network Management (IM'09)*, June 2009, pp. 327–334.

[28] J. Xu and J. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *IEEE/ACM GreenCom'10*, Dec. 2010, pp. 179–188.

[29] A. Stage and T. Setzer, "Network-aware migration control and scheduling of differentiated virtual machine workloads," in *Proc. ICSE CLOUD'09*, 2009, pp. 9–14.