Distributed Virtual Machine Migration for Cloud Data Centre Environments

Gregg Hamilton

Submitted in fulfilment of the requirements for the degree of
Master of Science by Research

School of Computing Science

College of Science and Engineering
University of Glasgow

March 2014

Abstract

Virtualisation of computing resources has been an increasingly common practice in recent years, especially in data centre environments. This has helped in the rise of cloud computing, where data centre operators can over-subscribe their physical servers through the use of virtual machines in order to maximise the return on investment for their infrastructure. Similarly, the network topologies in cloud data centres are also heavily over-subscribed, with the links in the core layers of the network being the most over-subscribed and congested of all, yet also being the most expensive to upgrade. Therefore operators must find alternative, less costly ways to recover their initial investment in the networking infrastructure.

The unconstrained placement of virtual machines in a data centre, and changes in data centre traffic over time, can cause the expensive core links of the network to become heavily congested. In this thesis, S-CORE, a distributed, network-load aware virtual machine migration scheme is presented that is capable of reducing the overall communication cost of a data centre network.

An implementation of S-CORE on the Xen hypervisor is presented and discussed, along with simulations and a testbed evaluation. The results of the evaluation show that S-CORE is capable of operating on a network with traffic comparable to reported data centre traffic characteristics, with minimal impact on the virtual machines for which it monitors network traffic and makes migration decisions on behalf of. The simulation results also show that S-CORE is capable of efficiently and quickly reducing communication across the links at the core layers of the network.

Acknowledgements

I would like to thank my supervisor, Dr. Dimitrios Pezaros, for his continual encouragement, support and guidance throughout my studies. I also thank Dr. Colin Perkins, for helping me gain new insights into my research and acting as my secondary supervisor.

Conducting research can be a lonely experience, so I extend my thanks to all those I shared an office with, those who participated in lively lunchtime discussions, and those who played the occasional game of table tennis. In alphabetical order: Simon Jouet, Magnus Morton, Yashar Moshfeghi, Robbie Simpson, Posco Tso, David White, Kyle White.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of cloud computing has been steadily increasing in recent years for tasks from hosting websites to performing business processing tasks. This has resulted in a great change in the way that data centres are architected and operated on a day-to-day basis. With the costs of setting up and running a data centre requiring a large initial outlay, operators must ensure that they can recoup the expense and maximise the time before they must update their infrastructure with another outlay for expensive hardware.

Traditional ISP networks are typically sparse and mostly over-provisioned along their backbone, as profits for an ISP network come from their ability to provide a desired speed to the end user. However, as cloud data centre operators turn a profit primarily from the computing resources they can provide to customers, operators are inclined to provide as many servers as possible to maximise the number of virtual machines (VMs) they can host on them. The cost for interconnecting all these servers within a data centre to provide a network with capacity great enough to allow all-to-all communication can be prohibitively expensive.

Achieving a sensible cost-to-profit ratio from a data centre is a balancing act, requiring operators to make decisions about the initial network infrastructure to ensure they see a return on their investment. This often results in the use of Clos fat-tree style topologies that are tree-like architectures with link capacities becoming more and more constrained and potentially over-subscribed towards the root of the tree.

Most over-subscribed topologies, such as fat-tree, provide sufficient link capacity for VMs at lower-level links towards the leaf of the tree, such as within racks. However, as data centre traffic operates at short timescales and often has long-term unpredictability, a substantial amount of traffic could be transmitted across over-subscribed network links.

Approaches to deal with link over-subscription in cloud data centre networks often consist of routing schemes that are non-programmable and pseudo-random, or through the migration of VMs to new locations within a data centre to reduce link congestion. Routing solutions

are often statically configured and do not directly target the problem of reducing congested links, while migration solutions are often centrally controlled and can be time consuming to come up with a near optimal solution for a new VM placement scheme.

1.1 Thesis Statement

I assert that a distributed, network-aware VM migration algorithm exploiting network monitoring instrumentation in end-systems can reduce congestion across heavily over-subscribed links under realistic data centre traffic loads, with minimal overhead on the data centre infrastructure. I will demonstrate this by:

- Providing an implementation of a distributed VM migration algorithm that is capable of operating within the bounds of existing data centre network architectures and traffic.
- Enabling a hypervisor to conduct network monitoring for the VMs it hosts, as well as making migration decisions on behalf of the VMs.
- Defining a mechanism able to identify the location of a remote VM within a data centre.
- Evaluating the properties of the algorithm and its implementation over realistic data centre workloads within simulation and testbed environments, showing that it can efficiently reduce network congestion, with minimal operational overhead on the infrastructure on which it runs.

1.2 Motivation

With the pervasive nature of cloud computing in today's data centres, and the related resource over-subscription that comes with it, data centre operators require new techniques to make better use of the limited, but expensive, resources they have. In particular, they have to ensure they make the maximum return possible on their investment in their infrastructure [1].

Studies have concentrated on the efficient placement, consolidation and migration of VMs, but have typically focused on how to maximise only the server-side resources [2, 3]. However, server-side metrics do not account for the resulting traffic dynamics in an over-subscribed network, which can negatively impact the performance of communication between VMs [4, 5].

Experiments in Amazon's EC2 revealed that a marginal 100 msec additional latency resulted in a 1% drop in sales, while Google's revenues dropped by 20% due to a 500 msec increase in

search response time [6]. It is therefore apparent that something needs to be done to improve the performance of the underlying network by reducing the congestion across it while still maintaining the efficiency of server resource usage.

Some VM migration works have considered how to improve overall network performance as the aim of migration schemes [7, 8]. However, such works are concerned with balancing load across the network, rather than actively removing congestion from over-subscribed and expensive links in the network, and often operate in a centralised manner. This leaves a research gap for a distributed VM migration scheme that is able to actively target the links most likely to experience congestion in a network, and iteratively remove traffic causing the congestion to other, less congested and less over-subscribed links.

This thesis presents such a distributed VM migration scheme, aimed at reducing not just the cost to the operator for running the data centre by making more efficient use of resources, but also reducing congestion from core links to lower the overall communication cost in the network.

1.3 Contributions

The contributions of this work are as follows:

- The implementation of a distributed VM migration scheme. Previous studies have focused on centrally-controlled migration algorithms that do not operate on information local to each VM.
- A hypervisor-based network throughput monitoring module that is able to monitor flow-level network throughput for individual VMs running upon it. Existing works typically instrument VMs themselves, or can achieve only aggregate monitoring of overall network throughput for each VM.
- A scheme to identify the physical location of a VM within a network topology, in order to allow for proximity-based weightings in cost calculations. As VMs carry configuration information with them when they migrate, they do not have any location-specific information. The scheme for location discovery here provides a method of identifying VM locations, and proximities, without the need to consult a central placement table.
- An evaluation of the performance that the distributed VM migration scheme should be able to achieve, in terms of migration times, and the impact on the systems on which it runs.

1.4 Publications

The work in this thesis has been presented in the following publication:

- *"Implementing Scalable, Network-Aware Virtual Machine Migration for Cloud Data Centers"*,
  F.P. Tso, G. Hamilton, K. Oikonomou, and D.P. Pezaros,
  in IEEE CLOUD 2013, June 2013

1.5 Outline

The remainder of this thesis is structured as follows:

- Chapter 2 presents an overview existing work on data centre network architectures and their control schemes. There is a discussion of common data centre architectures and the control loop mechanisms used to maintain network performance.
- Chapter 3 provides a description of the distributed migration algorithm upon which this work is based.
- Chapter 4 describes a working implementation of the scheme based on the algorithm described in Chapter 3. The individual components required for the successful implementation of a distributed migration scheme with an existing hypervisor are introduced.
- Chapter 5 details an evaluation of the distributed migration algorithm in both simulation and testbed environments.
- Chapter 6 summarises the findings and contributions of this work, and discusses the potential for expansion into future work.

Chapter 2

Background and Related Work

This chapter presents a background on data centre architectures, and the properties of the traffic that operate over them. Control loops for managing global performance within data centres are then discussed, from routing algorithms to migration systems.

2.1 Data Centre Network Architectures

The backbone of any data centre is its data network. Without this, no machine is able to communicate with any other machine, or the outside world. As data centres are densely packed with servers, the cost of providing a network between all servers is a major initial outlay for operators [1] in terms of networking equipment required.

To limit the outlay required for putting a network infrastructure in place, a compromise often has to be reached between performance and cost, such as over-subscribing the network at its core links.

Due to the highly interconnected nature of data centres, several scalable mesh architectures have been designed to provide networks of high capacity with great fault tolerance. DCell [9] is a scalable and fault-tolerant mesh network that moves all packet routing duties to servers, and relies upon its own routing protocol. BCube [10] is another fault-tolerant mesh network architecture designed for use in sealed shipping containers. As components fail over time, the network within the shipping container exhibits a graceful performance degradation. BCube makes use of commodity switches for packet forwarding, but doesn't yet scale above a single shipping container, making it unsuitable for current data centre environments.

While mesh networks can provide scalable performance bounds as the networks grow, the wiring schemes for mesh networks are often complex, which can make future maintenance and fault-finding a non-trivial task. The high redundancy of links in mesh networks that

happens to allow for good fault tolerance also increases the infrastructure setup cost due to the volume of networking hardware required.

The more commonly used alternative to mesh networks in the data centre are multi-tiered tree networks. The root of the tree, which is the core of the network, has switches or routers that provide a path between any two points within a data centre. From the root, the network branches out to edge, or leaf, interconnects that link individual servers into the network. In a multi-rooted tree, there are often two or more tiers of routers providing several levels of aggregation, or locality, within which shorter paths may be taken, without the need for all packets to pass through the core of the network. Multi-tiered trees are also often multi-rooted trees, providing redundant paths among any two points in the network, while still requiring less wiring and less network hardware than mesh networks.

The most often used architecture in data centres is a slight variation of a multi-tiered tree, known as a fat tree, which is based upon a communication architecture used to interconnect processors for parallel computation [11]. Instead of having links of equal capacity within every layer of the tree, bandwidth capacity is increased as links move away from edges and get closer to the core, or root, of the tree. Having increased capacity as we move towards the core of the tree can ensure that intra-data centre traffic that may have to traverse higher-level links has enough capacity for flows between many servers to occur without significant congestion.

The costs of housing, running and cooling data centres continues to rise, while the cost of commodity hardware, such as consumer-level network routers and switches, continues to drop. Data centre operators have not been blind to this, and have adapted multi-rooted fat tree topologies to make use of cheap, commodity Ethernet switches that can provide equal or better bandwidth performance than hierarchical topologies using expensive high-end switches [12].

A typical configuration for a fat tree network is to provide 1 Gbps links to each server, and 1 Gbps links from each top of rack switch to aggregation switches. Further layers up to the core then provide links of 10 Gbps, increasing capacity for traffic which may have to traverse the core of the network. Amazon is known to use such an architecture [13].

Tree-like networks are typically over-subscribed from ratios of 1:2.5 to 1:8 [12], which can result in serious congestion hotspots in core links. VL2 [14] has been developed in order to achieve uniform traffic distribution and avoid traffic hotspots by scaling out the network. Rather than make use of hierarchical trees, VL2 advocates scaling the network out horizontally, providing more interconnects between aggregate routers, and more routes for packets to traverse. A traffic study in [14] found data centre traffic patterns to change quickly and be highly unpredictable. In order to fully utilise their architecture with those findings, they made use of valiant load balancing, which makes use of the increased number of available

paths through the network by having switches randomly forward new flows across symmetric paths.

While some data centre architecture works attempt to expand upon existing network topology designs, PortLand [15] attempts to improve existing fat tree-style topologies. PortLand is a forwarding and routing protocol designed to make the operation and management of a dynamic network, such as a cloud data centre network, where VMs may be continually joining and leaving the network, more straightforward. It consists of a central store of network configuration information and location discovery, as well as the ability to migrate a VM transparently without breaking connectivity to the rest of the hosts within the network. The transparent VM migration is achieved by forcing switches to invalidate routing paths and update hosts communicating with that VM, and forwarding packets already in transit to the new location of the migrated VM. PortLand merely adapts existing architectures to provide a plug-and-play infrastructure, rather than attempting to improve performance in any serious way. This is revealed through the evaluation, which measured the number of ARP messages required for communication with the central network manager component as the number of hosts grows, rather than evaluating the protocol under varying application traffic loads.

Multi-rooted tree architectures are currently the most used architecture for data centre networks but they do have problems with high over-subscription ratios. While studies such as VL2 have further adapted multi-rooted tree architectures, they still do not completely overcome the over-subscription issue, requiring other, more targeted action to be taken.

2.2 Data Centre Traffic Characteristics

Several data centre traffic studies have been produced. As part of the VL2 work, a study of a 1,500 server cluster was performed over two months [14]. The findings of the traffic study were that 99% of flows were smaller than 100 MB, but with 90% of the data being transferred in flows between 100MB and 1GB. The break at 100 MB is down to the file system storing files in 100 MB-sized chunks. In terms of flows, the average machine has around 10 concurrent flows for 50% of the time, but will have more than 80 concurrent flows at least 5% of the time, with rarely more than 100 concurrent flows. The ratio of traffic within the data centre to traffic outside the data centre is 4:1. In terms of traffic predictability, they take a snapshot of the traffic matrix every 100s, finding that the traffic pattern changes constantly, with no periodicity to help in predictions of future traffic. To summarise, the VL2 study reveals that the majority of flows consist of short, bursty traffic, with the majority of data carried in less than 1% of the flows, and most machines have around 10 flows for 50% of the time, and the traffic changes rapidly and is unpredictable by nature.

Other studies reinforce the fact that data centre traffic is bursty and unpredictable [16, 17].

Kandula et al. [16] performed a study into the properties of traffic on a cluster of 1,500 machines running *MapReduce* [18]. Their findings on communication patterns show that the probability of pairs of servers within a rack exchanging no traffic is 89% and 99.5% for server pairs in different racks. A server within a rack will also either talk to almost all other servers within a rack, or fewer than 25%, and will either not talk to any server outside the rack, or talk to 1-10% of them. In terms of actual numbers, the median communication for a server is two servers within a rack and four servers outside its rack. In terms of congestion, 86% of links experience congestion lasting over 10 seconds, and 15% experience congestion lasting over 100 seconds, with 90% of congestion events lasting between 1 to 2 seconds. Flow duration is less than 10 seconds for 80% of flows, with 0.1% of flows lasting for more than 200 seconds, and most data is transferred in flows lasting up to 25 seconds, rather than in the long-lived flows. Overall, Kandula et al. have revealed that very few machines in the data centre actually communicate, the traffic changes quite quickly with many short-lived flows, and even flow inter-arrivals are bursty.

A study of SNMP data from 19 production data centres has also been undertaken [17]. The findings are that, in tree-like topologies, the core links are the most heavily loaded, with edge links (within racks) being the least loaded. The average packet size is around 850 bytes, with peaks around 40 bytes and 1500 bytes, and 40% of links are unused, with the actual set of links continuously changing. The observation is also made that packets arrive in a bursty ON/OFF fashion, which is consistent with the general findings of other studies revealing bursty and unpredictable traffic loads [14, 16].

A more in-depth study of traffic properties has been provided in [19]. SNMP statistics from 10 data centres were used. The results of the study are that many data centres (both private and university) have a diverse range of applications transmitting data across the network, such as LDAP, HTTP, MapReduce and other custom applications. For private data centres, the flow inter-arrival times are less than 1 ms for 80% of flows, with 80% of the flows also smaller than 10KB and 80% also lasting less than 11 seconds (with the majority of bytes in the top 10% of large flows). Packet sizes are also grouped around either 200 bytes and 1400 bytes and packet arrivals exhibited ON/OFF behaviour, with the core of the network having the most congested links, 25% of which are congested at any time, similar to the findings in [17]. With regard to communication patterns, 75% of traffic is found to be confined within racks.

The data centre traffic studies discussed in this section have all revealed that the majority of data centre traffic is composed of short flows lasting only a few seconds, with flow inter-arrival times of less than 1 ms for the majority of flows, and packets with bursty inter-arrival rates. The core links of the network are the most congested in data centres, even although 75% of traffic is kept within racks. All these facts can be summarised to conclude that data centre traffic changes rapidly and is bursty and unpredictable by nature, with highly

congested core links.

2.3 Traffic Engineering for Data Centres

In order to alleviate some of the congestion that can occur with highly unpredictable intra-data centre traffic several control loop schemes have been devised. The majority of control loops available nowadays are for scheduling the routing of individual flows to avoid, or limit, congested paths.

Multi-rooted tree architectures provide at least two identical paths of equal cost between any two points in the network. To take advantage of this redundancy *Equal-Cost Multi-Path (ECMP)* routing [20] was developed. In ECMP, a hash is taken over packet header fields that identify a flow, and this hash is used by routers to determine the next hop a packet should take. By splitting a network and using a hash as a key to routing, different hashes will be assigned to different paths, limiting the number of flows sharing a path. A benefit of the hashing scheme is that TCP flows will not be disrupted or re-routed during their lifetime. However, ECMP only splits by flow hashes, and does not take into account the size of flows. Therefore, two or more large flows could end up causing congestion on a single path. Similarly, hashing collisions can occur, which can result in two large flows sharing the same path.

Valiant Load Balancing (VLB), used in VL2 [14], is a similar scheme to ECMP. However, rather than computing a hash on a flow, flows are bounced off randomly assigned intermediate routers. While the approach may more easily balance flows, as it uses pseudo-random flow assignments rather than hash-based assignments, it is not any more intuitive than ECMP. By not targeting the problem of unpredictable traffic, and merely randomising the paths for flows, link congestion can still occur.

While the works discussed above make unintuitive decisions about routing flows in the data centre, there has been a move towards works that dynamically adapt to the actual traffic characteristics.

Hedera [21] is a flow scheduling system designed to provide high bisection bandwidth on fat tree networks. Built upon PortLand and ECMP, it uses adaptive scheduling to identify large flows that have been in existence for some length of time. After identifying large flows, it uses simulated annealing to schedule paths for flows to achieve close-to-optimal bisection bandwidth. Their evaluations found that a simple first-fit approach for assigning large flows beat ECMP, and their simulated annealing approach beat both ECMP and the first-fit approach. However, as they did not have access to data centre traffic traces, they evaluated their algorithms upon synthetic traffic patterns designed to stress the network, rather than attempting to generate synthetic traffic patterns using reported data centre traffic characteristics.

MicroTE [22] makes use of short-term predictability to schedule flows for data centres. ECMP and Hedera both achieve 15-20% below the optimal routing on a canonical tree topology, with VL2 being 20% below optimal with real data centre traces [22]. While studies have shown data centre traffic to be bursty and unpredictable at periods of 150 seconds or more [16, 17], the authors of MicroTE state that 60% of top of rack to top of rack traffic is predictable on the short timescales of between 1.6 and 2.6 seconds, on average, in cloud data centres. The cause of this is said to be during the *reduce* step in MapReduce, when clients transfer the results of calculations back to a master node in a different rack. MicroTE is implemented using the *OpenFlow* [23] protocol that is based on a centralised controller for all switches within a network. When a new flow arrives at a switch, it checks its flow table for a rule. If no rule exists for that flow, it contacts a single central OpenFlow controller that then installs the appropriate rule in a switch. In MicroTE, servers send their average traffic matrix to the central OpenFlow controller at a periodicity of 2 seconds, where aggregate top of rack to top of rack matrices are calculated. Predictable traffic flows (flows whose average and instantaneous traffic are within 20% of each other) are then packed onto paths and the remaining unpredictable flows are placed using a weighted form of ECMP, based upon remaining bandwidth on available paths after predictable flows have been assigned. By re-running the data centre traces, it is shown that MicroTE achieves slightly better performance than ECMP for predictable traffic. However, for traffic that is unpredictable, MicroTE actually performs worse than ECMP. An evaluation of the scalability of MicroTE reveals that the network overhead for control and flow installation messages are 4MB and 50MB, respectively, for a data centre of 10,000 servers, and new network paths can be computed and installed in under 1 second. While MicroTE does rely on some predictability, it provides minimal improvement over ECMP and can provide poorer flow scheduling than ECMP when there is no predictability, and also has a greater network overhead than ECMP, making it unsuitable for data centres where traffic really is unpredictable and not based upon MapReduce operations.

Another flow scheduler is *DeTail* [24]. It tackles variable packet latency and long flow completion time tails in data centres for deadlines in serving web pages. Link-layer-flow-control (LLFC) is the primary mechanism used to allow switches to monitor their buffer occupancy and inform switches preceding it on a path, using Ethernet pause frames, to delay packet transmissions to reduce packet losses and retransmissions that result in longer flow completion times. Individual packets are routed through lightly loaded ports in switches using packet-level adaptive load balancing (ALB). As TCP interprets packet reordering as packet loss, reordering buffers are implemented at end-hosts. Finally, DeTail uses flow priorities for deadline-sensitive flows by employing *drain byte* counters for each egress queue. Simulations and testbed experiments show that DeTail is able to achieve shorter flow completion times than flow control and priority queues alone under a variety of data centre workloads, such as bursty and mixed traffic. Unlike ECMP and VLB, DeTail adapts to traffic in the net-

work and schedules individual packets based on congestion, rather than performing unbalanced pseudo-random scheduling. However, DeTail pushes extra logic to both the switches and end-hosts, rather than tackling the problem of placement of hosts within the network infrastructure to achieve efficient communication.

The works above have discussed control loops in data centre networks that are focused on traffic manipulation, typically through flow scheduling mechanisms. However, there are ways to engineer and control data centre networks other than by manipulating traffic alone. The following sections discuss VM migration, and how it can be used by data centre operators to improve the performance and efficiency of their networks.

2.4 Virtual Machine Migration

Given the need for data centre operators to recover the cost of the initial outlay for the hardware in their infrastructures, it is in their interests to try and maximise the use of the resources they hold.

To meet the need to recover outlay costs, hardware virtualisation has become commonplace in data centres. Offerings such as VMware's vSphere [25] and the Xen hypervisor [26] provide hardware virtualisation support, allowing many operating systems to run on a single server, in the form of a *virtual machine (VM)*. Hypervisors and VMs operate on the basis of *transparency*. A hypervisor abstracts away from the bare hardware, and is a proxy through which VMs access physical resources. However, the VMs themselves, which contain an operating system image and other image-specific applications and data, should not have to be aware that they are running on a virtualised platform, namely the hypervisor. Similarly, with many VMs sharing a server, the VMs should not be aware of other VMs sharing the same resources.

Xen, the most common open source hypervisor, operates on a concept of domains. Domains are logically isolated areas in which operating systems or VMs may run. The main, and always-present, domain is *dom0*. dom0 is the Xen control domain, and an operating system, such as Ubuntu Linux [27], runs in this domain, allowing control of the underlying Xen hypervisor and direct access to the physical hardware. In addition to dom0, new guest domains, referred to as *domU* can be started. Each domU can host a guest operating system, and the guest operating system need not know that it is running upon a virtualised platform. All calls to the hardware, such as network access, from a domU guest must pass through dom0.

As dom0 controls hardware access for all domU guests, it must provide a means for sharing access to networking hardware. This is achieved through the use of a network bridge, either via a standard Linux virtual bridge, or via a more advanced bridge such as the Open vSwitch [28] virtual switch. Open vSwitch provides a compatibility mode for standard Linux

virtual bridges, allowing it to be used as a drop-in replacement for use with Xen. With a virtual bridge in place in Xen's dom0, packets from hosts running in domU domains can then traverse the bridge, allowing communication between VMs on the same server, or communication with hosts outside the hypervisor.

With the solutions above, instead of running services on a 1:1 ratio with servers, data centre operators can instead run many services, or VMs, on a single server, increasing the utilisation of the servers. With many-core processors now the norm, running many VMs on a server can make better use of CPU resources, so that, rather than running a set of services that may not be optimisable for parallelised operations, many VMs and other diverse and logically separated services can be run on a single server.

In a modern data centre running VMs, it can be the case that, over time, as more VMs are instantiated in the data centre, the varying workloads can cause competition for both server and network resources. A potential solution to this is VM live migration [29]. Migration allows the moving of servers around the data centre, essentially shuffling the placement of the VMs, and can be informed by an external process or algorithm to better balance the use of resources in a data centre for diverse workloads [2].

VM migration involves moving the memory state of the VM from one physical server to another. To copy the memory of the VM requires stopping execution of the VM and reinitialising execution once the migration is complete. However, live migration improves the situation by performing a "pre-copy" phase, where it starts to copy the memory pages of the VM to a new destination without halting the VM itself [29]. This allows the VM to continue execution and limit the downtime during migration. The memory state is iteratively copied, and any memory pages modified, or "dirtied", during the copying are then re-copied. This process repeats until all the memory pages have been copied, at which point the VM is halted and any remaining state copied to and reinitialised on the new server. If memory is dirtied at a high rate, requiring large amounts of re-copying, the VM will be halted and copied in a "stop-and-copy" phase.

The remaining sections of this chapter will focus on various aspects of VM migration, including models of migration, and a variety of VM migration algorithms, identifying their benefits and shortcomings.

2.4.1 Models of Virtual Machine Migration

While VM migration can be used to better balance VMs across the available physical resources of a data centre [2], VM migration does incur its own overhead on the data centre network, which cannot be ignored.

It has been shown that VM downtime during migration can be noticeable and can negatively impact service level agreements (SLAs) [30]. The setup used in the aforementioned work was a testbed running the Apache web server, with varying SLAs attached to various tasks such as the responsiveness of a website home page, or the availability of user login functionality. The testbed was evaluated using a workload generator and a single VM migration, with the finding that it is possible to have a 3 second downtime for such a workload. This result reveals that migration can have a definite impact on the availability of a VM, and migration is a task whose impact, in addition to the benefit gain after migration, must be taken into consideration.

As VM migration carries its own cost in terms of data transferred across the network and the downtime of a VM itself, a method for considering the impact is to generate models for VM migration. [31] shows that the two most important factors in VM migration are link bandwidth and page dirty rate of the VM memory. It derives two models for migration: an *average* page dirty rate and *history-based* page dirty rate. The models were evaluated against a variety of workloads including CPU-bound and web server workloads, with the finding that their models are accurate in 90% of actual migrations. This shows that migration impact can be successfully predicted in the majority of cases, and models of VM migration have been used in studies of migration algorithms [3, 7].

2.5 System Control Using Virtual Machine Migration

VM migration has typically been used to improve system-side performance, such as CPU availability and RAM capacity, or minimising the risk of SLA violations, by performing migrations to balance workloads throughout data centres [2, 3, 32, 33].

SandPiper [2] monitors system-side metrics including CPU utilisation and memory occupancy to determine if the resources of a server or individual VM or application are becoming overloaded and require VMs to be migrated. SandPiper also considers network I/O in its monitoring metrics, but this can only be used to greedily improve network I/O for the VM itself, rather than improving performance across the whole of the network, or reducing the cost of communication across the network. Mistral [32] attempts to optimise VM migration as a combinatorial optimisation problem, considering power usage for servers and other metrics related to the cost of migration itself but it does not attempt to improve the performance of the data centre network infrastructure. A compliment to VM migration is, if servers are under-utilised, making better use of the server resources available by increasing the system resources available to VMs using the `min`, `max` and `shares` parameters available in many hypervisors [34] to increase the share of CPU and memory resources available to the VMs.

A wide area of concern for which VM migration is seen as a solution is maintaining SLAs

and avoiding any SLA violations [33, 35], or avoiding SLA violations during migration [3]. Such works make use of workload predictability [33] and migration models to achieve their goals [3]. Workload forecasting has also been used to consolidate VMs onto servers while still ensuring SLAs are met [36, 37].

However, these works again make no attempt to improve the performance of the underlying network, which is the fundamental backbone for efficient communication among networked workloads.

2.6 Network Control Using Virtual Machine Migration

The works discussed above in Section 2.5 make no attempt to target improving the performance of the core of the network through VM migration. This section will identify works that specifically address the problem of maintaining or improving network performance.

Studies have attempted to use VM placement to improve the overall data centre network cost matrix [38, 39]. VM placement is the task of initially placing a VM within the data centre, and is a one time task. Migration can be formulated as an iterative initial placement problem, which is the situation in [39]. However, initial placement does not consider the previous state of the data centre, so formulating migration as iterative placement can cause large amounts of re-arranging, or shuffling, of VMs in the data centre, which can greatly increase VM downtime and have a negative impact on the network, due to the large number of VMs being moved.

Network-aware migration work has considered how to migrate VMs such that network switches can be powered down, increasing locality and network performance, while reducing energy costs [40]. However, the work can potentially penalise network performance for the sake of reducing energy costs if many more VMs are instantiated and can't be placed near to their communicating neighbours due to networking equipment being powered down.

Remedy [7] is an OpenFlow-based controller that migrates VMs depending upon bandwidth utilisation statistics collected from intermediate switches to reduce network hotspots and balance network usage. However, Remedy is geared towards load-balancing across the data centre network, rather than routing traffic over lower level, and lower cost links in the network to improve pairwise locality for VMs.

Starling [8] is a distributed network migration system designed to reduce network communication cost between pairs of VMs and makes use of a migration threshold to ensure costly migrations with little benefit to outweigh the disruption of migration are not carried out. Starling makes use of local monitoring at VMs to achieve its distributed nature. It can achieve up to an 85% reduction in network communication cost, although the evaluation has a strong

focus on evaluating running time for applications, rather than assessing the improvement in network cost. While Starling is novel and aims to improve network performance, it does not make use of network topology information, such as hops between VMs, to identify traffic passing over expensive, over-subscribed network paths, so cannot actively target the act of reducing communication cost from high layer, heavily congested links..

2.7 Discussion

In this chapter I have introduced data centre network architectures and various network control mechanisms. I discussed how resource virtualisation, through VM migration, is now commonplace in data centres, and how VM migration can be used to improve system-side performance for VMs, or how load can be better balanced across the network through strategic VM migration.

However, all the VM migration works in this chapter have not addressed the fundamental problem of actively targeting and removing congestion from over-subscribed core links within data centre networks. The remainder of this thesis will attempt to address this problem by presenting a VM migration scheme for distributed migration to reduce the overall communication cost in the network, through a discussion of the implementation of the scheme and simulation and testbed evaluations of the scheme.

Chapter 3

The S-CORE Algorithm

As has been identified in Chapter 2, existing VM migration algorithms do not actively consider the layout of the underlying network when making migration decisions, nor do they actively attempt to reduce traffic on the most over-subscribed network links.

This chapter summarises a distributed VM migration algorithm, *S-CORE*, which considers the cost of traffic travelling over various layers in a data centre topology where each layer can have an increasing link cost towards increasingly over-subscribed links at the the core of the network. The aim of the algorithm in S-CORE is to iteratively reduce pairwise communication costs between VMs by removing traffic from the most costly links.

The theoretical basis behind S-CORE has previously been presented in [41] and the full theoretical formulation and proof behind S-CORE can be found in [42], which can be referenced for the full details of the algorithm. A summary of the important aspects of the S-CORE algorithm, required for the work presented in this thesis, is discussed here.

3.1 A Virtual Machine Migration Algorithm

Modern data centre network architectures are multi-layered trees with multiple redundant links [4, 12]. An illustration of such a tree is provided in Figure 3.1, from [43].

Let $\mathbb{V}$ be the set of VMs in a data centre, hosted by the set of all servers $\mathbb{S}$, such that every VM $u \in \mathbb{V}$ and every server $\hat{x} \in \mathbb{S}$. Each VM u in the data centre is unique and it is assigned a unique identifier $\texttt{ID}_u$.

Each VM is hosted by a particular server and let $\mathcal{A}$ denote an *allocation* of VMs to servers within the data centre. Let $\hat{\sigma}^{\mathcal{A}}(u)$ be the server that hosts VM u for allocation $\mathcal{A}$, $u \in \mathbb{V}$ and $\hat{\sigma}^{\mathcal{A}}(u) \in \mathbb{S}$. Let $\mathbb{V}_u$ denote the set of VMs that exchange data with VM u.

The data centre network topology dictates the switching and routing algorithms employed in the data centre, and the topology in Figure 3.1 is used for the purposes of illustrating the

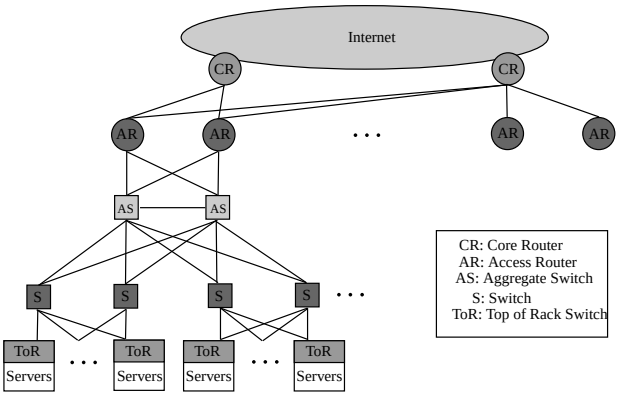

Figure 3.1: A typical network architecture for data centres.

algorithm presented here. However, the S-CORE algorithm is applicable to any data centre topology, so the algorithm presented here is not specific to any one topology.

As shown in Figure 3.1, the topology has multiple layers, or levels, which network communication can travel over. At the highest and most over-subscribed level are a set of core routers. At the level below are access routers, which interconnect the core router and aggregate switches from the level below. The aggregate switches are, in turn, connected to switches which then connect to the top of rack switches.

Network links that connect top of rack switches to switches below the aggregate switches will be referred to hereafter as 1-*level links*, and those between the switches and aggregate switches as 2-*level links*, and so on.

Table 3.1: List of notations for S-CORE.

Notation	*Description*
$\mathbb{V}$	Set of all VMs in the data centre
$\mathbb{V}_u$	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

Due to the cost of the equipment, the capacity at different levels as we progress up the tree is typically increasingly over-subscribed, with the greatest over-subscription at the highest

levels of the tree.

When a packet traverses the network between two VMs, it will incur a communication cost (in terms of resource usage, which is the shared bandwidth of the network), which will increase as the packet travels through many different levels of the topology hierarchy, due to varying over-subscription ratios [12]. When moving up from the lowest levels to highest levels of the hierarchy, the communication cost, c_i, will increase, i.e., $c_1 < c_2 < c_3 < c_4$. The value of link weights can be determined by data centre operators by taking into account their operational costs for setting up the different layers of their topology (i.e., more expensive networking hardware at the core of the network than at the edges), or by using other factors such as the over-subscription ratio of the different levels in their network hierarchy.

The problem of communication cost reduction and the concepts of VM allocation, communication level, and link weights, with important notations are formalised and listed in Table 3.1.

The *overall communication cost* for all VM communications over the data centre 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.

$$C^{\mathcal{A}} = \sum_{\forall u \in \mathbb{V}} \sum_{\forall v \in \mathbb{V}_u} \lambda(u, v) \sum_{i=1}^{\ell^{\mathcal{A}}(u,v)} \mathsf{c}_i. \tag{3.1}$$

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 [42] that this problem is of high complexity and is NP-complete, so there exists no possible polynomial time solution for centralised optimisation. Even if there was, the centralised approach would require global knowledge of traffic dynamics which can be prohibitively expensive to obtain in a highly dynamic and large scale environment like a data centre.

This calls for a scalable and efficient alternative, and thus we have formulated the following *S-CORE distributed migration policy* for VMs: A VM u migrates from a server x to another server $\hat{x}$, provided that Equation 3.2 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 any cost incurred during migration c_m. As noted above, [42] can be referred to for the full definition and proof of the S-CORE scheme.

$$2 \sum_{\forall z \in \mathbb{V}_u} \lambda(z, u) \left(\sum_{i=1}^{\ell^{\mathcal{A}}(z,u)} \mathsf{c}_i - \sum_{i=1}^{\ell^{\mathcal{A}_{u \to \hat{x}}}(z,u)} \mathsf{c}_i \right) > c_m, \tag{3.2}$$

Chapter 4

Implementation of a Distributed Virutal Machine Migration Algorithm

Given the S-CORE algorithm presented in Chapter 3, a realisation of this algorithm must be developed in order to evaluate its real-world performance and to overcome any implementation issues not covered by the theoretical algorithm, such as determining the location between two VMs in a data centre.

This chapter describes an implementation of the S-CORE VM migration system, incorporating the S-CORE algorithm, and addresses the rationale behind the implementation choices, as well as addressing the practical problems posed by today's data centres on how such a distributed algorithm may successfully and efficiently operate.

4.1 Token Policies

One of the main tasks in any VM migration algorithm is the order in which to migrate VMs. As S-CORE operates in a distributed manner and is not controlled through a central mechanism, VMs must explicitly know when they are allowed to migrate. This is achieved in this implementation through the use of a token that is passed among VMs. A basic token contains slots, with each slot containing a VM ID and a communication level for each VM. The structure of the basic token is shown in Figure 4.1. In order to make use of the token policy, each VM in a data centre is assumed to have a unique identifier, which is already the

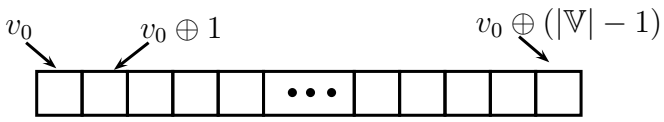

Figure 4.1: The token message structure.

case in data centres (and computer networks in general) as systems would not be uniquely reachable otherwise.

When a VM holds the token, it is able to make a migration decision for itself based upon its current communication cost and the communication cost if it migrates elsewhere. It may also update communication cost values for other VMs it communicates with. For example, if VM u holds the token, then $\mathtt{l}_u$ can update its own token entry, as VM u is aware of its own highest communication level $\ell_u^{\mathcal{A}}$. VM u is also aware of the communication level of those VMs it communicates with (i.e., $v \in \mathbb{V}_u$), and can update $\mathtt{l}_v$ if the new communication cost, $\ell^{\mathcal{A}}(u, v)$, is higher than the existing recorded communication cost in the token. After deciding whether to migrate, the VM holding the token can then pass it to the next VM listed in the token, dependent upon the token passing policy in place.

Given the generality of S-CORE, token policies can be based on a number of heuristics, and can even be calculated using metrics that are gathered centrally or in a distributed manner. The token can also be extended to provide extra information within each slot, such as the cost of migration itself for a particular VM.

This section discusses only the operational details of each token passing policy, and not necessarily the details of initial token construction for each policy.

Four token policies were implemented for this work:

- Round-robin
- Global
- Distributed
- Load-aware

The *round-robin* token policy is a simplistic policy wherein a token is constructed and it is passed from VM to VM in strict token slot order (i.e., the order in which the token was constructed, which could be ordered by VM ID). This policy may not be the most efficient, as it cannot skip passing the token to VMs that will not be migrated, resulting in migration iterations potentially being wasted.

The centralised *global* token policy gathers communication statistics over a time period and centrally computes communication costs and a migration order dependent on the greatest pairwise communication cost reductions for VM pairs. This can be potentially costly in terms of the time required to perform a central migration optimisation calculation on a data centre consisting of tens or hundreds of thousands of VMs, where communication cost data may go stale quickly. It also has the potential to greatly impact the performance of all VMs in the data centre as the data is transmitted to and gathered in a central location, which is not a

desirable trait for an algorithm meant to improve the network performance or communication efficiency of VMs.

The *distributed* token policy does not require a centrally calculated token. Instead, it starts passing the token among VMs for whom network communication passes through the highest-layer links in the network. As the highest layer core links are the most costly, and most over-subscribed, it is a reasonable assumption to make that migration at this level is most likely to take place over communication at lower levels, as there are higher gains to be achieved by migrating VMs away from using high-level links.

The highest communication level for each VM is initialised to zero. The token starts by being passed to a VM with communication passing through the highest layer and with the lowest VM ID of all VMs communicating over that layer (which can be achieved through a leader election algorithm in which VMs participate, but is not discussed here). This VM updates the token with its own communication cost, and also updates the communication cost of any neighbouring VM, if required. After making its own migration decision, the token is passed to the next VM communicating at that layer in the data centre. If no other VM is available at that layer, or no other VM at that layer remains that still has to make a migration decision, the token is passed to a VM communicating across the next-highest layer. When all VMs and layers have been exhausted, the policy restarts from the beginning, with the VM communicating at the highest layer with the lowest VM ID.

The details of the distributed token policy are presented in Algorithm 1.

A feature of the distributed token policy is the ability for a VM to determine communication costs for VMs it communicates with. This is discussed in detail in Section 4.2.5.

The final token passing policy to be introduced is the load-aware token policy. It is a variant of the distributed token policy and considers aggregate network load of incoming and outgoing traffic for each VM. Unlike the distributed token policy, which passes the token to VMs at the same communication level in VM ID order, the load-aware policy passes the token among VMs at the same communication level, starting with to the VM with the highest aggregate load in that level first. This requires a small number of comparisons before the token is passed to the VM with the next-highest aggregate load (or the VM with the highest aggregate load in the next communication level). However, as migration is expected to be most likely to happen at the higher layers, and the greatest cost reductions can be expected from migrations at higher layers at the core of the network, this could allow for a more efficient migration phase, and allow the state of all VM placements in the data centre to reach close-to-optimal sooner than in the distributed token policy. Unlike the global token policy, which requires central aggregation of statistics and a central calculation, which can be costly and unscalable, the load-aware token policy is able to make use of statistics available locally at VMs.

Algorithm 1 Distributed Token Policy

$\texttt{cl} \leftarrow \texttt{l}_u$ ▷ $\texttt{cl}$ maintains the current value of $\texttt{l}_u$
$found \leftarrow$ FALSE ▷ Flag regarding next VM

for $\forall v \in \mathbb{V}_u$ **do** ▷ Update VMs connected to u
  if $\texttt{l}_v < \ell^{\mathcal{A}}(u, v)$ **then**
    $\texttt{l}_v \leftarrow \ell^{\mathcal{A}}(u, v)$
  end if
end for

$z \leftarrow u \oplus 1$ ▷ Pick the next VM after u
while $\texttt{cl} \geq 0$ && $!found$ **do**
  while $\texttt{l}_z \neq \texttt{cl}$ **do**
    $z \leftarrow z \oplus 1$ ▷ Pick the following VM
  end while
  if $\texttt{l}_z \leftarrow \texttt{cl}$ **then**
    $found \leftarrow$ TRUE ▷ Next node is found
  else ▷ Next node is not found at this level
    $\texttt{cl} \leftarrow \texttt{cl} - 1$ ▷ Go to a lower level
    $z \leftarrow v_0$ ▷ and start from the beginning
  end if
end while

if $!found$ **then** ▷ No unchecked VMs are left
  Pick VM $z : \min_{\texttt{ID}_x}\{\forall x \in \mathbb{V} : \texttt{l}_x = \max_{\forall v \in \mathbb{V}}(\texttt{l}_v)\}$
end if
Send $token$ to VM z

The four token policies discussed above have been implemented as part of the S-CORE migration system, the implementation details of which are now introduced.

4.2 Implementation Setup

This section discusses the implementation-specific details and problems that had to be overcome to provide a real-world implementation for the S-CORE migration scheme.

S-CORE was implemented on top of the Xen [44] hypervisor, running Ubuntu 12.04 as dom0 (domain zero, the control domain started by Xen on boot, and which controls all guest domain VMs). In order to communicate with and control Xen, we used the *xm* [45] management interface. xm is written in Python and controls all VM duties within Xen, such as VM instantiation, migration and probing of VM information and state.

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, S-CORE has been implemented in Python.

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 [28] can be used as a drop-in replacement with Linux 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.

4.2.1 Implementation in VM vs Hypervisor

The S-CORE algorithm expects VMs to pass the token amongst themselves and make their own migration decisions. However, this is unsuitable in practice. In system virtualisation, the general paradigm is that virtualised hosts should not be aware that they are running within a VM. Therefore, the hypervisor should be transparent to the VM, and a VM should have no direct control communication to the hypervisor on which it runs.

As the hypervisor initiates and performs VM migration, this creates a problem if a VM itself, which is not aware of the hypervisor, wishes to migrate. Enabling or adding any such ability to VMs would violate the fundamental transparency between VMs and their hypervisor.

Since the hypervisor can monitor all network traffic to and from each VM it hosts, the above problem was avoided by implementing S-CORE within dom0 of the Xen hypervisor itself,

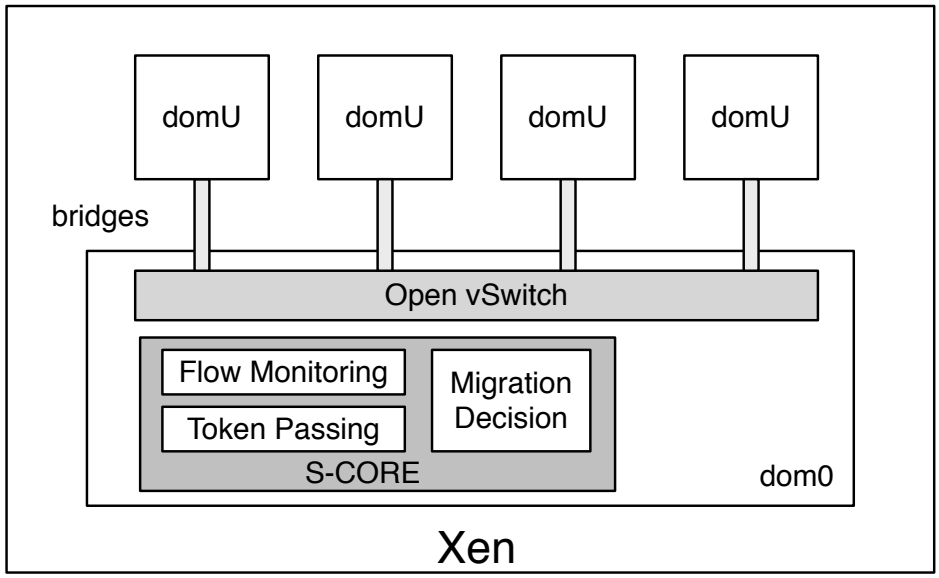

Figure 4.2: The S-CORE architecture.

instead of within VMs. The modular architecture of S-CORE is shown in Figure 4.2. The benefit of such a modular architecture is not only to keep transparency intact but also to make the system easily upgradable.

As the hypervisor is transparent to resident VMs, VMs cannot directly instantiate or conduct their own migration. By opting to run S-CORE within dom0, taking a migration decision at the level of dom0 greatly simplifies the implementation of the algorithm. New communication channels between the VM and hypervisor do not have to be implemented to allow VMs to initiate migration, so the transparent view that VMs have of the platform they are running on does not need to be broken.

While some works do follow the path of removing the aspect of transparency from the VM [2], there are other benefits to be gained by not instrumenting VMs themselves. With many public cloud providers in existence enabling users to run their own VMs, instrumenting the VMs becomes impossible, unless providers wish to supply users with their own restricted VM images. While there are also many private cloud data centres running VMs, it can still be difficult to instrument VMs themselves, as many different corporate teams may work within the same data centre and have different requirements, causing them to result to rolling their own VM images.

4.2.2 Flow Monitoring

To enable an accurate measure of the aggregate throughput between communicating VMs some form of continual gathering of statistics for flows is required. Open vSwitch provides per-flow statistics, however, it 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, a custom flow table for storing flow-level statistics was developed. 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 accumulation of flow statistics for as long as is required. Flows are stored from when they start until a migration decision is made for a VM. As the most frequent operations 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 lookups and updates 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. Each 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 of our flow table data structures are used, with the second storing destination IP addresses as the main key, allowing fast and easy lookup of bidirectional flows.

Storing both the source and destination flow data structures allows quick addition of, quick retrieval of, and quick updating of flows by both source and destination addresses. In particular, it allows quick and easy retrieval of flows by IP address, so that all flows belonging to a VM can be retrieved, and the aggregate of those flows to other VMs can be calculated.

4.2.3 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. As discussed in Section 4.1, many token policies can be defined, which determine how the token is passed between VMs, and how each VM must decide whether or not it should migrate. We have implemented four token passing policies, and the underlying details of how the token passing mechanism is implemented in S-CORE will be discussed in this section.

As a recap, the token passing policies implemented are:

- Round-robin
- Global
- Distributed
- Load-aware

When a VM receives a token for which its VM ID is the next entry, 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 (or hypervisor). If a lower communication cost is achievable and the destination host has available resources, then migration should take place.

As each host in a network must be accessible via a unique IP address, the IPv4 address of a VM has been used as a 32-bit VM ID carried within each token slot. Using the IPv4 address as the VM index also provides the benefit of allowing the token to be sent directly next to the VM ID of the next VM, rather than having to perform some form of ID-to-address mapping.

To efficiently pack the token for network transmission, it is stored and transmitted as a block of slots of 32-bit and 8-bit unsigned integers, where the 8-bit unsigned integer holds the communication level for a VM.

Since the implementation stores IP addresses as VM IDs and passes the token to each IP address in turn, a question arises: How does the dom0 of the hypervisor 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 by accessing the flow table for the VM's flows and performing the cost reduction calculation, before forwarding the token along.

Failure recovery when the token is lost (due to a process or communication failure) can be addressed by an algorithm such as the classic Gallager, Humblet and Spira distributed leader election algorithm [46] wherein a minimum-weight spanning tree with a single leader is constructed using only the local knowledge initially available at nodes.

4.2.4 Xen Wrapper

While it may seem straightfoward for dom0 to migrate a VM after receiving the token containing the VM's IP address and deciding that the VM should be migrated, the process is not as simple as that.

It is only 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 the creation of 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 discussed in Section 4.2.2, the flow table also stores a MAC address alongside 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.

If there is no entry in the flow table that maps the IP address to a MAC address, this means there has been no communication from that particular VM to any other VM, and therefore there is no benefit for that VM to gain from being migrated to any other location.

4.2.5 Migration Decision

With Xen's dom0 now able to monitor the flows for all VMs, and able to receive and send the token on behalf of a VM, and map the ID in the token to a particular VM, it must also be able to make a migration decision for the given VM. This section details the final components that make up the actual migration decision process.

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 neighbours 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.

Communication Cost

Once the aggregate throughput to each communicating neighbour has been calculated, the communication cost must be evaluated. The communication cost is the number of links over

which packets from the VM must traverse to communicate with another VM, with each level of links in the topology having an increasing cost value.

In real terms, the communication cost can be derived from the number of hops between a VM and any neighbour that it is communicating with. This could be achieved by a network diagnostics tool such as *traceroute*, but layer 2 switches would not show up as hops in this case. Another alternative would be a a VM ID lookup service listing the cost for any VM to communicate with another. However, VM IP addresses are used as their VM ID, and VMs carry their IP addresses when they migrate, which renders this method unusable in a data centre with a dynamically changing VM allocation.

On the contrary, the physical servers and the hypervisors running on them, do not move around within the data centre (unless for some form of maintenance). This makes a reliable lookup service based on the addresses of physical servers possible, and is the option chosen for S-CORE. As a flow table of the IP addresses each VM communicates with is stored, neighbouring VMs can be probed 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 pre-computed 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 neighbouring VM, as well as a total cost for its current allocation.

With a total cost for its allocation, the VM can then consider if it would be beneficial to migrate to another location, if a such a suitable location is available.

Migration Target

When deciding if migration would be a beneficial move for a VM, hypervisors suitable to move the VM to must be identified. As our algorithm is distributed, and we do not store a central list of all the hypervisors (or the underlying servers), we must take an alternative approach to identifying potential hypervisors as the destination of a migrated VM. An intuitive way to consider this is that, logically, the biggest cost reduction gains could be achieved by moving a VM to the same hypervisor as the VM that it has the highest communication cost with.

As the IP addresses of each hypervisor can be determined, and the communication cost then probed, neighbouring VMs can be ordered from highest to lowest communication levels and

each hypervisor probed to see if it has sufficient server resources to host the current VM. A custom *capacity request* packet is sent to the hypervisor of the neighbouring VM with the highest communication cost, which responds with a custom *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).

The capacity request and response packets are text-based communications of the form:

- `hypervisor_capacity_request`
- `hypervisor_capacity_response <avail_doms> <avail_mem>`

Due to the limited types of data required in the request/response packet pair for capacity information, use of text-based exchanges is sufficient. However, were the request/response packets to be extended further (e.g., to include average CPU usage), a *type-length-value* (TLV) encoding could be adopted.

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 neighbouring VM with the highest communication cost does not have the capacity to host the VM for which migration is being considered, 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, this step terminates and the token is passed on to the next VM ID for the next iteration of the migration process.

This completes the discussion of the modules that make up the implementation of S-CORE. Both simulations of S-CORE's performance and the performance of the implementation presented here, and its impact, on a testbed setup are presented in Chapter 5.

Chapter 5

Evaluation

S-CORE has been evaluated using both a simulation setup and a testbed environment. The purpose of these evaluations are to determine the feasibility of S-CORE in a real-world environment by assessing its scalability properties and its overhead. This chapter presents simulation results of the S-CORE algorithm to show its communication cost reduction properties and testbed results to show the performance and overhead of the implementation presented in this thesis.

5.1 Simulations

S-CORE's communication cost reduction with has been evaluated across the four different token policies over a layered data centre topology, using the *ns-3* network simulator [47].

The simulated network topology is comprised of 2560 hosts (128 top of rack switches, 20 hosts per rack), which can sufficiently capture the hierarchical link over-subscription ratio at aggregate and core links found in data centres [4]. In order to model a typical data centre server environment, each host can accommodate at most 16 VMs, assuming 2 VMs per core, with each occupying 1GB of RAM. Increasing a VM's resource requirements is equivalent to combining, for example, two or more VMs' resources into one.

A single VM is modelled as a socket application which communicates with one or more other VMs in the network. Similar to actual virtualisation, each server has a VM hypervisor network application to manage a collection of VMs, supporting migration into and away from each server.

Links costs are set as $\mathsf{c}_1 = e^0$, $\mathsf{c}_2 = e^1$, $\mathsf{c}_3 = e^3$ and $\mathsf{c}_4 = e^5$ for each layer in the topology hierarchy. VM migration carries its own cost in terms of network bandwidth for moving a VM's memory contents and VM downtime, which can negatively affect other VMs communicating across the network. To account for this in the simulations, a migration overhead

cost, c_m, is introduced. The migration overhead cost is initially set to zero to allow for a fair comparison among the centralised approach and S-CORE. However, since a data centre operator may wish to limit the number of migrations a VM undertakes over a temporal interval, different values associated with the cost of migration can be used. For example, an operator may wish to limit the number of migrations within a time period, to limit the overall negative impact on its network. Simulation results for various values of c_m which are presented later in Section 5.1.3.

5.1.1 Traffic Generation

A data centre traffic generator to test S-CORE under realistic data centre-style loads was also used in the simulations, as data centre traffic characteristics have been reported in a number of measurement studies [4, 16, 48, 49].

The traffic generator maintains 10 active flows, on average, per VM [4, 49]. Most flows (90%) are smaller than 10 KB, modelling metadata communication or queries, and the remaining 10% of flows have a mode of 128MB (a common chunk size of MapReduce jobs) [16, 4, 50]. Among all generated flows, 80% of the flows stay within the rack whereas 20% of them leave the rack [16, 49]. The traffic generator models only 20% of top of rack switches as hotspots because this is the case in real data centres, and even the hotspot top of rack switches end up exchanging much of their data with only a few other top of rack switches [48, 16, 49].

The sample of a 10s traffic matrix of all top of rack switches is given in Figure 5.1, which exhibits identical traffic matrix properties with those unveiled in [48].

As can be seen in Figure 5.1, the traffic matrix in data centres is indeed sparse and only a handful of top of rack switches become hotspots. However, a significant fraction of traffic amongst hotspots has to be routed over upper layers in the topology hierarchy, resulting in episodes of congestion and high communication cost.

5.1.2 Global Optimal Values

To have a baseline against which to compare the performance of S-CORE, an optimal value of the placement of VMs within the data centre, based on communication cost, is required. However, minimising the overall communication cost for the topology is an NP-complete problem, and an exhaustive search across all permutations would be prohibitively time consuming. For example, assuming communication within a rack has zero cost, with $16 \times 20 = 320$ VMs per rack, a centralised algorithm would need to explore at least $\binom{40,960}{320}$ combinations.

As a benchmark, the centralised optimal values are instead approximated using a genetic algorithm. The genetic algorithm starts with a population consisting of $1,000$ individuals representing densely-packed VM distributions in a data centre, and stops when there is no significant improvement in communication cost reduction ($< 1\%$) in 10 consecutive generations. The crossover operator has been implemented using edge assembly crossover (EAX) to generate a new child from two parent distributions and the replacement of individuals is based on tournament selection. Mutation happens by swapping a random number of VMs between racks.

Execution time over a medium-load simulation setup is almost 12 hours using a system with 8GB RAM and a 2.66GHz quad-core CPU.

5.1.3 Simulation Results

The results in Figure 5.2 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 traffic matrix given in Figure 5.1 for all scenarios. 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 stabilise when the VM distribution considerably reduces the overall communication cost.

To reflect the fact that c_m is often non-zero due to VM migration overhead on the network, simulations were run with different c_m threshold values.

Figure 5.3 shows that if c_m is increased to 10% of the overall communication cost, a pronounced communication cost reduction can still be seen. The ratio of communication cost reduction plunges sharply if c_m is further increased to 20% or more. This phenomenon demonstrates that S-CORE will work well by setting reasonable c_m values according to the policies of a particular data centre operator. For example, if an operator wishes to limit migration to ensure that the impact upon other tenants in its data centre, they may choose to increase the migration cost.

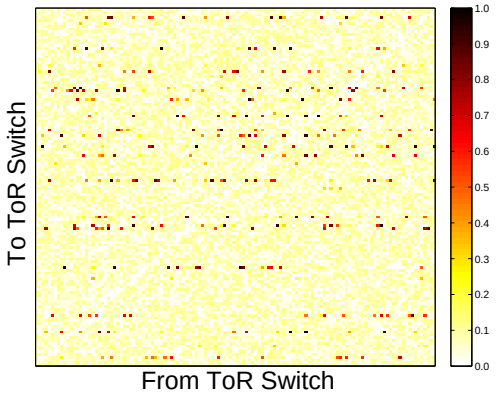

Figure 5.1: Normalised traffic matrix between top-of-rack switches.

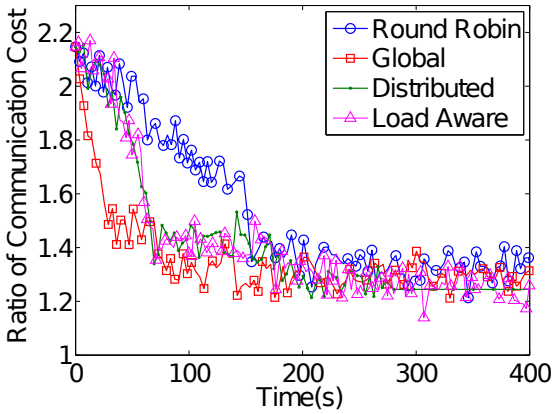

Figure 5.2: Communication cost reduction with data centre flows.

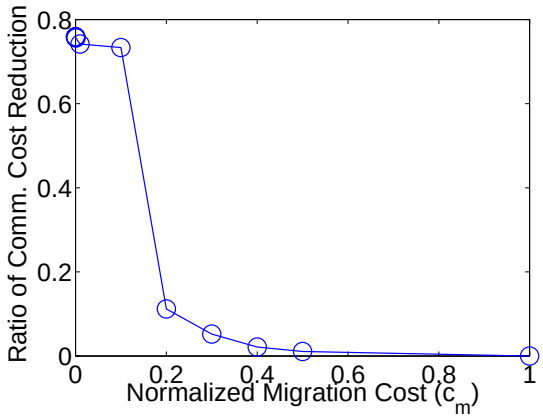

Figure 5.3: Ratio of communication cost reduction with the distributed token policy.

Figure 5.4 reveals that after VMs migrate, the number of top of rack hotspots is significantly reduced. Even though there are still top of rack hotspots, these top of rack switches are in close physical proximity, which means that inter-top of rack 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 data centre, addressing the aims of studies on partial shutdown of servers or network elements [3, 40].

5.1.4 VM stability

VM stability is crucial for dynamic VM migration algorithms as unstable VM migrations can themselves potentially have a big impact on the network and servers. VM placement instability can occur due to oscillations, where VMs will periodically jump between two placements in the expectation of gaining some improvement from the new placement, while gaining no real long-term improvement as it will later decide to revert to its previous placement or move elsewhere, incurring migration overhead in terms of VM downtime and migration cost.

Whilst no dynamic algorithm can completely eliminate the possibility of VM oscillations, S-CORE can minimise short-term oscillations due to 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 to derive a migration decision is based on aggregate traffic load over that period. Therefore, the short-term effects of sudden arrivals of small flows are cancelled out when averaged over one iteration of the algorithm.

5.2 Testbed Evaluation

5.2.1 Testbed Setup

To test the implementation of S-CORE, rather than just its theoretical properties as were tested above, an evaluation was performed on a testbed environment.

The testbed, as shown in Figure 5.5, consisted of six 8-port gigabit switches and four servers interconnected by 1 Gbps links. The switches are arranged to form a typical 3-tier data centre topology. The servers in the testbed consisted of Intel P4 3GHz servers each with 2GB RAM running Xen 4.1 with Ubuntu Server 12.04 operating as dom0. The domU guest VMs are Ubuntu 10.04 images, with 196MB RAM allocated for each guest VM. The experiments

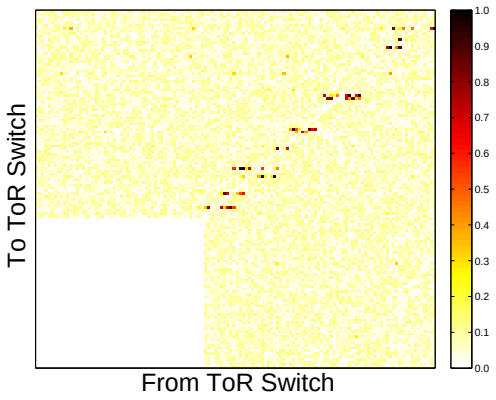

Figure 5.4: Normalised traffic matrix between top-of-rack switches after 5 iterations.

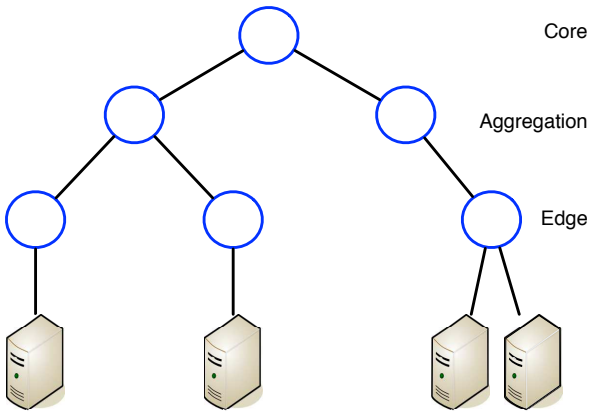

Figure 5.5: Testbed topology.

started with two VMs initially located on each server. Each VM hosts a HTTP server as well as an *iperf* [51] server and client.

Live migration requires that VM images reside on shared storage, rather on hypervisor servers themselves, to allow for migration among different hypervisors to take place. A Network File System (NFS) server was set up to host VM images.

While the testbed is limited in terms of scalability up to data centre-sized topologies for full testbed experiments, the distributed nature of S-CORE allows the evaluation of module components and their scalability in isolation, or the evaluation of properties such as VM live migration times that should not be negatively impacted by the size of the testbed.

5.2.2 Module Evaluation

The implementation of S-CORE is built into several modules, addressing the various aspects of the distributed operation of S-CORE. As with any such modules that run on end-hosts, it is important to assess the impact that these modules have on system resources, in terms of memory and processor time consumption. It is also important to know how the distributed performance of the modules impacts the network as a whole. This section aims to address the system-side performance of S-CORE's modules.

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 hypervisor resources, thus leaving the majority of resources available to the actual VMs. While a number of studies have revealed that server and network resources are mostly under-utilised, the aim is to stress test the 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 only 10 active flows per VM [4, 16, 48, 49], so 1 million flows are used merely to stress the implementation.

Two different sets of flows were defined: 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 originate from the same source IP address (type 2).

To test the memory consumption of adding flows to the flow table, 1 million flows from each of the two flowsets described above were generated and the size of the table was iteratively

measured after each flow was added, by reading the *vmSize* parameter for the process from within Linux. The results of this can be found in Figure 5.6.

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 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 spread across 1000 source IP addresses, only 1000 entries are added to the root of the table, with the remaining flow data added to nested structures associated with these 1000 source 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 [16]. At the same time, in a large-scale cloud data centre, the number of concurrent flows going in and out of a machine is still almost never more than 100 [4]. With a more realistic scenario where every virtual server concurrently sends or receives 10 flows, with 100 in the worst case, it is anticipated 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, the time taken to add, lookup and delete flows has been measured, summing the times over the number of flows, for the same sets of flows. Figure 5.7 shows the time to perform various flow table operations with differing numbers of flows in a single operation. From Figure 5.7 it can be seen that flow addition, lookup and deletion operations all require less time on a flow table with a type 2 flow set (i.e., few VMs on a hypervisor, each with many flows). Nevertheless, addition, lookup and deletion operations should not need more than 100ms for a realistic data centre production workload of 100 concurrent flows.

The flow operation times reveal that all the flow operations scale sub-linearly. For inserting 100, 1000, 10,000 and 100,000 flows, the times are 0s, 0s, 0.001s, 0.07s and 0.94s, respectively. The 0s values are a result of the granularity of Python's time functions.

As flows are only periodically updated, these times have little effect on the overall running of the system. Further, assuming 16 VMs each with 20 individual flows, as above, it would take only a fraction of a second to add or update that many flows. Moreover, flow lookup and deletion are only performed when a VM migration is being considered, and only 1 source IP address will be retrieved for those operations, as only the flows to and from a single VM are

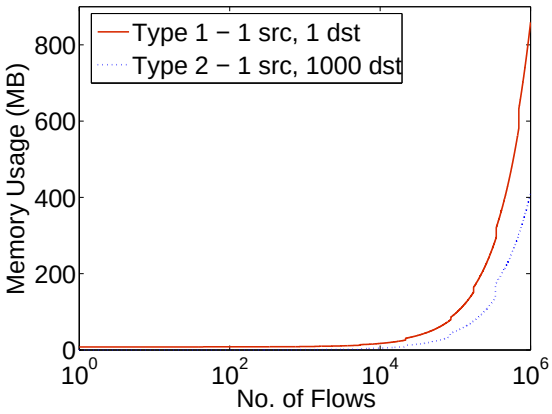

Figure 5.6: Flow table memory usage.

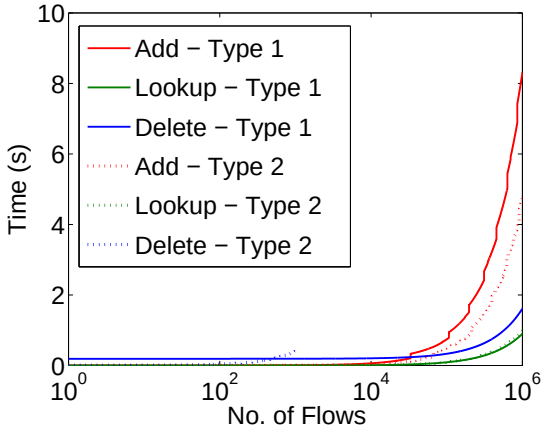

Figure 5.7: Flow table operation times for up to 1 million unique flows.

considered.

While memory usage and time taken are useful metrics in measuring footprint on the hypervisor's dom0, they reveal little about what effect they may have on the actual operation of dom0. Memory can be provisioned for in advance, and time taken does not tell us how it is likely to affect the processing capacity of the physical server on which the hypervisor is running.

In order to evaluate the run-time impact on the processing capability of the physical servers, the CPU usage of flow table operations in the normal background running state of the flow table was measured. 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. The CPU clock time for adding each flow was measured and calculated as a percentage of CPU utilisation, as shown in Figure 5.8. 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 utilisation. In the best case for 10,000 flows added or updated each time, CPU utilisation was only around 1% at a polling rate of 5 seconds, while the worst case CPU utilisation 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.

When a dom0 holds the token for a particular VM, it must retrieve the flows for that VM, calculate the aggregate throughput of the flows to each neighbouring VM, retrieve the cost of the links between them, and derive an overall communication cost for each neighbouring VM, and an overall allocation cost for the placement of the given VM.

To evaluate the impact of both flow table lookup size and the location lookup cost, an experiment was conducted in which the number of VMs (and hence, the number of flows) that the VM under consideration for migration communicated with was varied. The results revealed that for a VM with 10 communicating neighbours, the runtime is only 0.32s. The runtime linearly increments to 2.97s and 30.54s for a VM with 100 and 1000 neighbours, respectively. This reveals that, for a VM with a reasonable amount of communicating neighbours (< 100), the runtime of the communication cost calculation lookup is negligible.

5.2.3 Network Impact

Similar to other data centre management schemes, S-CORE will inevitably impose control overhead on the network. An improperly designed control scheme may overwhelm the network with additional load due to control packets, but how much overhead will S-CORE create?

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 data centre. A typical production data centre has 100,000-500,000 servers, in which case the token size will merely be between 500KB – 2.5MB.

Live migration requires VM images to reside on shared storage (e.g., using the network file system (NFS)). As the actual file system is on the shared storage, and mounted on the servers, only the VM's memory state will be copied from one server to another over the network. However, the copying of the memory state from a source server to a destination server can also be attributed as a network overhead.

During the memory migration, in particular the iterative pre-copy stage [29], the hypervisor copies all memory pages from the source to destination server. If some memory pages change (become "dirty") during this process, they will be re-copied until some pre-defined threshold has been reached, at which point the VM will be stopped and all remaining memory pages copied without risk of dirtying. 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. Figure 5.9 shows the probability density function (PDF) of the number of migrated bytes for each VM migration captured in the 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 a minimal Ubuntu 10.4 VM image and a few lightweight test services running inside, i.e., a HTTP server and SSH server, the VM memory sizes 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 data centre 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 can have about 800MB memory usage [29], which is still an affordable network overhead for an infrequent migration schedule, such as once every few days, or even every few hours, in line with our iterative, distributed token policy. In addition, as S-CORE migration is intended to lower overall communication cost, 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 data centre operators.

5.2.4 Impact of Network Load on Migration

When a VM is being migrated over a highly utilised path, the migration time could be increased which could, in turn, increase the downtime and potentially violate any service level agreements (SLAs) between the data centre operator and the VM tenant.

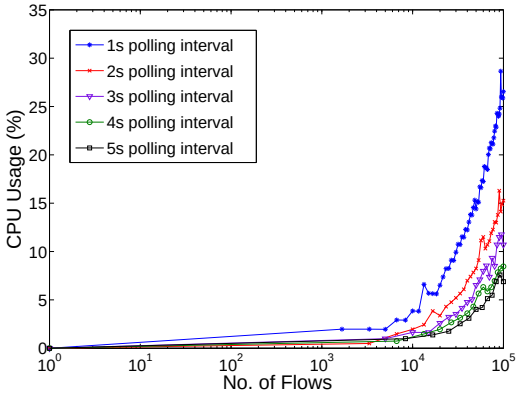

Figure 5.8: CPU utilisation when updating flow table at varying polling intervals.

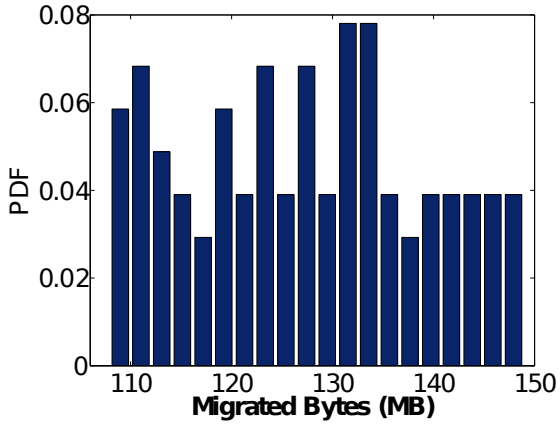

Figure 5.9: PDF of migrated bytes per migration.

To determine how VM migration could be impacted by network load, an experiment was run in which two servers (i.e., dom0 of two hypervisors) generated a constant bit-rate UDP stream as background traffic while migrating a VM from one server to the other. The migration packets were captured with *tcpdump* [52] and the total migration time was calculated by taking the time difference between the first and the last packets received. As migration time does not equate directly to downtime for live migration, as part of the memory image is copied while the VM is still running, further measurements had to be taken to determine the downtime of the VM. The downtime of a VM was determined by probing the migrating VM with high precision ping, *fping* [53], with the interval between pings set to 1ms.

Figure 5.10 and Figure 5.11 illustrate the distribution of VM migration time and downtime, respectively, for the migrated bytes shown in Figure 5.9 under varying background traffic on their local links. As depicted from Figure 5.10, 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 utilised links ($\geq$70% of link capacity utilised) due to transferring the large spread of migrated memory size, as shown in Figure 5.9, 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 data centre environments, the server downtime 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 [29] of the live migration process where a VM on a server is stopped, and its CPU state and any remaining memory pages are then transferred to another server. As shown in Figure 5.11, downtime 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 utilisation does have some impact on VM migration time and migration downtime, 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 having been migrated in the previous pre-copy stage [29]), and the total actual downtime of the VM (which is what data centre tenants care about), is minimal.

5.3 Discussion

This chapter has presented an evaluation of the S-CORE migration system through both simulations and a testbed environment. The results of the evaluations have shown that the

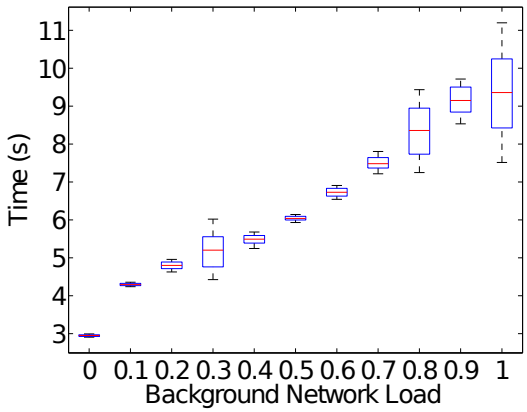

Figure 5.10: Virtual machine migration time.

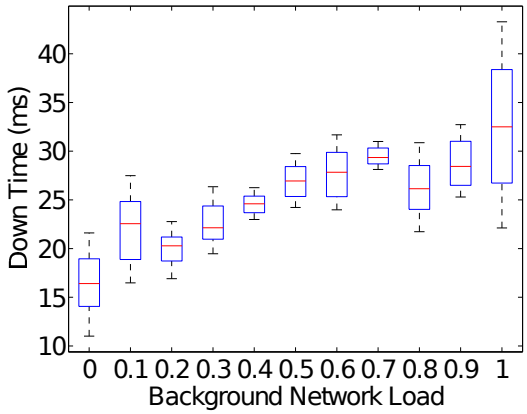

Figure 5.11: Downtime under various network load conditions.

theoretical basis of S-CORE's migration algorithm can successfully reduce the overall communication cost for VMs in a data centre across a variety of token passing policies, with the distributed load-aware token passing policy having the greatest improvement in overall communication cost.

The testbed evaluation showed the performance of S-CORE's various modules and their impact on the systems they execute on, as well assessing the impact on migration caused by varying network traffic loads. The results from this showed that, for normal data centre traffic loads and VM distributions, the time, memory usage, and CPU usage, for adding, updating and deleting flows were minimal for a migration system such as S-CORE that operates periodically, as were the times for performing migration calculations. In assessing migration downtime under varying traffic loads, it was also found that downtimes can be minimised to the order of milliseconds, with all migration downtimes in the evaluation being less than 1 second.

Chapter 6

Conclusions

Virtualisation has been an increasingly popular mechanism in recent years to make better use of powerful hardware resources. In particular, VMs have paved the way for cloud computing, where operators can reap the benefits of over-subscribing hardware resources including servers and networking resources. With data centres holding tens of thousands, or even hundreds of thousands, of intercommunicating VMs, pairwise VM traffic can cause a large degree of congestion, especially in the highly over-subscribed core links of the network.

Chapter 3 summarised an existing distributed migration algorithm, S-CORE, designed to reduce network communication from costly core links at the high layers of the network topology, to less costly lower layers.

In this thesis I have discussed an implementation of the S-CORE migration scheme, and presented an extensive evaluation of S-CORE through both simulations and testbed experiments.

The remainder of this chapter will revisit my thesis statement, and how this thesis has addressed it, along with the contributions made in this thesis and future work than can be undertaken to extend the work presented.

6.1 Thesis Statement

The thesis statement is repeated here for reference:

I assert that a distributed, network-aware VM migration algorithm exploiting network monitoring instrumentation in end-systems can reduce congestion across heavily over-subscribed links under realistic data centre traffic loads, with minimal overhead on the data centre infrastructure. I will demonstrate this by:

- Providing an implementation of a distributed VM migration algorithm that is capable of operating within the bounds of existing data centre network architectures and traffic.
- Enabling a hypervisor to conduct network monitoring for the VMs it hosts, as well as making migration decisions on behalf of the VMs.
- Defining a mechanism able to identify the location of a remote VM within a data centre.
- Evaluating the properties of the algorithm and its implementation over realistic data centre workloads within simulation and testbed environments, showing that it can efficiently reduce network congestion, with minimal operational overhead on the infrastructure on which it runs.

To show that I have addressed the statement above, I will summarise the work undertaken in this thesis, and the results of that work.

I implemented the distributed S-CORE migration system on top of the Xen hypervisor in the Python programming language, and have shown its performance impact is negligable on a testbed setup, under data centre traffic characteristics reported in other studies.

The S-CORE modules were implemented within the control domain (dom0) of the Xen hypervisor, and are able to perform network monitoring and migration decision duties on behalf of the VMs hosted, partially through the use of packet interception to allow the hypervisor to receive control packets sent to VMs.

S-CORE is able to successfully identify the location of a remote VM by using the IP addresses of VMs as VM IDs, and having the underlying hypervisor of a VM capture a location request packet sent to a VM it hosts, with the hypervisor responding with the address of the physical server. As physical servers do not move, a static mapping of the topology can be created in advance, and this can be consulted when a location request is made.

Finally, the S-CORE scheme has been evaluated in simulations and on a testbed environment, using data centre traffic characteristics, with the results showing that S-CORE can monitor such traffic with minimal impact on the CPU or memory of the physical servers. S-CORE is also able to make timely migration decisions and can greatly improve network performance in a scalable fashion using its many token passing policies, in particular its load-aware token passing policy.

6.2 Future Work

This thesis has presented a distributed VM migration policy that is able to remove congestion from the over-subscribed core links of data centre networks through pairwise migration.

There are several future paths that this work can follow, which are discussed in this section.

6.2.1 Incorporation of System-Side Metrics

There are many works that focus solely on system-side metrics [2, 34] or network-based metrics [7, 8], but few do both.

Currently, S-CORE only checks that the destination server for migration has a slot available for a VM and has sufficient memory capacity to host the source VM under consideration for migration. However, S-CORE could be extended to balance system-side resources so that, say, two competing VM workloads are not placed on the same server, if possible.

This could be formulated as a combinatorial optimisation problem that considers the communication cost reduction as well as system-side workload type (i.e., CPU usage, memory requirements). To simplify the problem, it could instead be weighted so that communication cost reduction is prioritised and that VMs within the same rack are then balanced based upon system-side resource requirements.

6.2.2 Using History to Forecast Future Migration Decisions

Some attempts have been made at using workload forecasting to aid in migration decisions [36, 37]. Using history can help make stable VM placements, and reduce the risk of oscillations in the migration process, where a VM may repeatedly jump between two servers.

While S-CORE currently uses network throughput metrics as a form of history for making informed migration decisions, it does not store history about the migrations that have taken place. Therefore, there is the possibility that a VM could return to a previous placement location during a migration phase. Storing migration history could help mitigate any such risk, however unlikely.

6.2.3 Implementation in a Lower-Level Programming Language

Xen's management interface, *xm* is implemented in Python. S-CORE's modules were implemented in Python for the reasons of ease of communication with the xm interface and the periodic operation of the modules. The the evaluation in Chapter 5 has shown that the impact on servers and on S-CORE's performance with this implementation is minimal.

However, should the benefits of a lower-level programming language such as C be desired, there are tools out there that allow for translating, or compiling, the Python modules into

C code, which could save performing a full rewrite of the modules in another language. PyPy [54] and Cython [55] are two such tools.

6.3 Summary & Conclusions

This thesis has presented the implementation of a distributed VM migration scheme known as *S-CORE*. S-CORE is capable of performing local monitoring of VM network traffic using modules written to interact with the Xen hypervisor. The modules are able to operate within the hypervisor to make migration decisions on behalf of VMs, maintaining transparency for VMs from the platform on which they are running.

The S-CORE scheme is able to iteratively reduce congestion from heavily over-subscribed core links in the network and reduce the overall communication cost across the network, unlike existing migration works, through its distributed migration algorithm and token passing policies. Simulations and testbed experiments have shown that the implementation of S-CORE is capable of operating under typical data centre traffic loads in reasonable timescales with minimal impact on the servers it operates on, or on the VMs it shares a server with.

Bibliography

[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, December 2008.

[2] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI '07)*, April 2007.

[3] A. Verma, P. Ahuja, and A. Neogi, "pMapper: power and migration cost aware application placement in virtualized systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*, December 2008, pp. 243–264.

[4] 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.

[5] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proceedings of the 29th conference on Information communications (INFOCOM '10)*, March 2010, pp. 1163–1171.

[6] 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 international conference on Knowledge discovery and data mining (KDD '07)*, August 2007, pp. 959–967.

[7] 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 *NETWORKING 2012*, ser. Lecture Notes in Computer Science, 2012, vol. 7289, pp. 190–204.

[8] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-

aware migration," in *Parallel Processing (ICPP), 2010 39th International Conference on*, September 2010, pp. 228 –237.

[9] 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 *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, August 2008, pp. 75–86.

[10] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, August 2009, pp. 63–74.

[11] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, no. 10, pp. 892–901, October 1985.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 63–74, August 2008.

[13] C. Raiciu, M. Ionescu, and D. Niculescu, "Opening up black box networks with CloudTalk," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing (HotCloud'12)*, June 2012, pp. 6–6.

[14] 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 *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, August 2009, pp. 51–62.

[15] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, August.

[16] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference (IMC '09)*, September 2009, pp. 202–208.

[17] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, January 2010.

[18] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[19] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10)*, November 2010, pp. 267–280.

[20] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2992.txt

[21] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI '10)*, April 2010, pp. 19–19.

[22] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies (CoNEXT '11)*, 2011, pp. 8:1–8:12.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, March 2008.

[24] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: reducing the flow completion time tail in datacenter networks," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, August 2012, pp. 139–150.

[25] "VMware vSphere," accessed: 12 September 2012. [Online]. Available: http://www.vmware.com/uk/products/datacenter-virtualization/vsphere/overview.html

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, October 2003, pp. 164–177.

[27] "Ubuntu," accessed: 29 October 2012. [Online]. Available: http://www.ubuntu.com/

[28] "Open vSwitch," accessed: 26 November 2012. [Online]. Available: http://openvswitch.org/

[29] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI '05)*, October 2005, pp. 273–286.

[30] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *Cloud Computing*, ser. Lecture Notes in Computer Science, 2009, vol. 5931, pp. 254–265.

[31] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper, "Predicting the performance of virtual machine migration," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, August 2010, pp. 37 –46.

[32] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, June 2010, pp. 62 –73.

[33] A. Stage and T. Setzer, "Network-aware migration control and scheduling of differentiated virtual machine workloads," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD '09)*, May.

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

[35] D. Breitgand and A. Epstein, "SLA-aware placement of multi-virtual machine elastic services in compute clouds," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, May 2011, pp. 161 –168.

[36] S. Mehta and A. Neogi, "Recon: A tool to recommend dynamic server consolidation in multi-cluster data centers," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, April 2008, pp. 363 –370.

[37] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing SLA violations," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, May 2007, pp. 119 –128.

[38] 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 *Services Computing (SCC), 2011 IEEE International Conference on*, July 2011, pp. 72 –79.

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

[40] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman, "VMFlow: Leveraging vm mobility to reduce network power costs in data centers," in *NETWORKING 2011*, ser. Lecture Notes in Computer Science, 2011, vol. 6640, pp. 198–211.

[41] F. P. Tso, G. Hamilton, K. Oikonomou, and D. P. Pezaros, "Implementing scalable, network-aware virtual machine migration for cloud data centers," in *Cloud Computing (CLOUD), 2013 IEEE 6th International Conference on*, June 2013, pp. 557–564.

[42] 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.

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

[44] "Xen hypervisor," accessed: 6 November 2012. [Online]. Available: http://xen.org/

[45] "Xen management user interface," accessed: 28 November 2012. [Online]. Available: http://wiki.xen.org/wiki/XM/

[46] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 66–77, January 1983.

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

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

[49] 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.

[50] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user MapReduce clusters," EECS Department, University of California, Berkeley, Tech. Rep., April 2009.

[51] "Iperf," accessed: 18 October 2012. [Online]. Available: http://iperf.sourceforge.net/

[52] "tcpdump," accessed: 18 October 2012. [Online]. Available: http://www.tcpdump.org/

[53] "fping," accessed: 9 January 2013. [Online]. Available: http://fping.sourceforge.net/

[54] "PyPy," accessed: 17 January 2013. [Online]. Available: http://pypy.org/

[55] "Cython," accessed: 17 January 2013. [Online]. Available: http://www.cython.org/