USING NETWORK KNOWLEDGE TO IMPROVE WORKLOAD PERFORMANCE IN VIRTUALIZED DATA CENTERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David Erickson
May 2013

David Erickson

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

(Nick McKeown)   Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

(Mendel Rosenblum)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

[illegible]

(Christos Kozyrakis)

Abstract

The scale and expense of modern virtualized data centers motivates running them as efficiently as possible. These data centers are primarily composed of physical machines running virtualization software, with each physical machine hosting many virtual machines (VMs) simultaneously. The performance of the workload running inside a VM is affected not only by other VMs on the same physical machine, but in the case of a workload that uses the network, the location of the VM and other VMs it is communicating with in the network topology, and the utilization of all network links in between. This thesis explores how the performance of workloads running inside VMs can be improved when network traffic and topology data informs the assignment of VMs to physical machines (VM placement).

To answer this question, I built a network control system based on the OpenFlow control protocol, Beacon, and a cluster control system that managed the Xen virtualization layer, resource measurement, experiments, and the VM placement algorithms used to optimize experiments. This system is named *Virtue*, and was used to run and monitor experiments on an 80 server cluster.

The VM placement algorithms evaluated in this thesis were able to improve the median performance of network-heavy, scale-out workloads by over 70% compared to random initial placements in a multi-tenant configuration. Performance improvements of 33% to 129% were observed when running the same experiment and varying both the edge link speeds between 100Mb/s and 750Mb/s, and the core network oversubscription ratio between 16:1 and 1:1.

Acknowledgements

After being accepted to Stanford's Masters program I immediately began looking for professors doing networking and systems related research. I cold emailed Nick indicating my desires in this area, and was thrilled (and somewhat shocked) that he both replied and offered me a research assistant position in his group. In the subsequent years I discovered truly how fortunate I was to have this opportunity. Nick has demonstrated by example how to change the world both as a researcher and entrepreneur. His advice and guidance during my time at Stanford have been invaluable in my personal growth and shaping the way I think and act. My words cannot really do this justice, but thanks Nick for the opportunity of a lifetime.

Along the journey I had the privilege of interacting with many other professors. Thanks to Prof. Guru Parulkar who provided me with excellent guidance and suggestions at many significant times during my PhD, and has always had an open door whenever I wanted to chat. I was also fortunate to work with Prof. Mendel Rosenblum who repeatedly provided great insights by drawing upon his tremendous area expertise. Also thanks to Prof. Christos Kozyrakis for both being on my defense and reading committee and the advice and interactions we had over the years across multiple projects.

Large research projects tend not to occur by a single author in isolation, and mine is no exception. Thanks to Brandon Heller, Jonathan Chu, Shuang Yang, and Ali Yahya for their contributions to Virtue. Thanks to Microsoft for the PhD Fellowship which funded much of this research. And thanks to Google for supporting my research with both equipment and manpower. In particular, thanks to Stephen Stuart, Scott Whyte, Mark Gensheimer, and Matt Smith.

A special thank you to my office*mates* for the last six years: Glen Gibb and Brandon

Heller. I couldn't have asked for a better pair of friends that were always up for idea discussion, technical questions, and keeping each other motivated.

Thanks to the entire past and present McKeown Group, which during my time expanded significantly to include industry collaborators, visiting researchers, and prior McKeown Group graduates. The breadth of knowledge and talent present in these individuals has always amazed me, and I have loved learning about and being involved with the many ongoing research projects and activities, and receiving their feedback on mine: Guido Appenzeller, Neda Beheshti, Martìn Casado, Adam Covington, Saurav Das, Nandita Dukkipati, Jonathan Ellithorpe, Glen Gibb, Natasha Gude, Nikhil Handigol, Brandon Heller, Peyman Kazemian, Masayoshi Kobayashi, John Lockwood, Jianying Luo, Jad Naous, Justin Pettit, Srini Seetharaman, Alireza Sharafat, Rui Zhang-Shen, Rob Sherwood, Dan Talayco, Paul Tarjan, David Underhill, Tatsuya Yabe, KK Yap, Yiannis Yiakoumis, and James Hongyi Zeng.

Thank you to my Mom who chose to make lemonade when life gave her lemons. She provided me with all the love, support, and encouragement anyone could ever hope to have from a parent.

Last, and certainly not least, thank you to my family. My wife Debbie who, without complaint, moved away from family and has sacrificed, supported, and loved me throughout all the ups and downs that came during the PhD process. Your patience and caring have meant the world to me. And to my daughter Aspen, who motivates me to work even harder with her smiles and hugs. Love you both!

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 State of the Art 1

1.2 The Network 2

1.3 Related Work 5

1.3.1 Improve the Network 6

1.3.2 Improve VM placement 6

1.4 Why hasn't this been done? 7

1.5 Research plan 9

1.6 Thesis contributions and structure 10

2 Beacon **13**

2.1 Related Work 15

2.2 Developer Productivity 16

2.2.1 Programming Language 16

2.2.2 Libraries 18

2.2.3 Tool chain 19

2.2.4 API 19

2.3 Modularity 21

2.4 Performance 24

2.4.1 Event Handling 24

2.4.2 Reading OpenFlow Messages 25
2.4.3 Writing OpenFlow Messages 27
2.5 Evaluation 30
2.5.1 Performance 30
2.5.2 Modularity 33
2.5.3 Developer Productivity 36
2.6 Conclusion 37

3 Virtue Platform 39
3.1 Data Center Network Research Cluster 40
3.1.1 Servers 40
3.1.2 Network 41
3.2 Virtue Software 42
3.2.1 Experiment Description File 42
3.2.2 Sun Grid Engine 43
3.2.3 Workload Executor 43
3.2.4 Workloads 44
3.2.5 Virtue Application 44
3.2.6 Optimizers 48
3.2.7 Experiment Overhead 49
3.2.8 Automation 50
3.3 A Day in the Life of an Experiment 50

4 Algorithms 53
4.1 Optimization Metric: Minimax 53
4.1.1 Baseline Placement 55
4.2 Mixed-Integer Model 56
4.3 Practical Algorithms 56
4.3.1 Simulated Annealing 57
4.3.2 Greedy Network 59
4.3.3 Greedy Fill 61

5 Evaluation **63**

5.1 Infrastructure 65

5.2 Web Workload 65

5.2.1 Web Workload, 20 VMs 66

5.2.2 Web Workload, 120 VMs 70

5.3 MapReduce Workload 72

5.3.1 Hadoop, 20 VMs 72

5.3.2 Hadoop, 120 VMs 73

5.4 Combined Web + MapReduce, 120 VMs 74

6 Conclusion **79**

A Controller Evaluation Settings **81**

B EDF **83**

C Model **89**

Bibliography **91**

List of Tables

2.1 Candidate languages and their attributes. 17

4.1 Optimization algorithms and the data they use. 57

5.1 Percentages of a 1Gb/s link used to poll counter, for different numbers of flows and polling intervals. 66

5.2 Median relative performance improvement of *Greedy Fill* vs *Random* by edge link rate and network core oversubscription ratio. 78

List of Figures

1.1 Oversubscription in data centers. 2

1.2 Example of poor VM placement causing network bottlenecks. 3

1.3 Workflow to optimize VM placement. 4

1.4 Web workload performance by algorithm. 9

2.1 Overview of Beacon. 14

2.2 Example of Beacon's Core API. 19

2.3 Beacon Web User Interface: Switch List. 21

2.4 Beacon Web User Interface: Switch Flow-Table List. 22

2.5 Beacon's Service Registry. 23

2.6 Beacon *IOFMessageListener* Pipelines. 24

2.7 Possible I/O Designs. 25

2.8 Cbench Throughput Tests. 32

2.9 Cbench Latency Tests – Single Thread. 34

3.1 View of the Data Center Network Research Cluster (DNRC). 40

3.2 Workflow diagram of *Virtue*. 42

3.3 *Virtue* Web UI: Exploring an experiment. 46

3.4 *Virtue* Web UI: Selecting a subset of the topology to examine. 47

5.1 Subset of testbed topology used in experiments. 64

5.2 Data flow in the Web workload. 66

5.3 Web 20 VMs 40 PMs. 67

5.4 Iterative algorithm improvement. 69

5.5 Web 120 VMs 80 PMs. 70
5.6 Hadoop 20 VMs 40 PMs. 72
5.7 Hadoop 120 VMs 80 PMs. 73
5.8 Multi-Tenant Workload 120 VMs 80 PMs. 75
5.9 How much faster the Web and Hadoop workloads run when the network is not oversubscribed. 77

Chapter 1

Introduction

Modern virtual machines (VMs) arose as a tool to test, manage, and consolidate x86 servers. They have since matured into the building block of the cloud [3, 29], supporting services that can scale rapidly from one to as many VMs as demand requires (scale-out) in both public and private virtualized data centers (VDCs) using multiple different virtualization layers [20, 36, 44]. The size of modern data centers provides a strong motivation to optimize efficiency; Amazon's U.S. East data center reportedly contains over 320,000 servers [2]. Optimization can reduce operating expenses, defer upgrades, free up capacity to sell, and/or automatically improve cloud customers', also called tenants, performance. These are all attractive options, since thousands of tenants may be competing for the same infrastructure resources, and because a large data center costs several hundred million dollars to build and operate.

1.1 State of the Art

Despite cost pressures, current cloud environments do not appear to exploit the full potential of VMs to optimize a single scale-out workload, let alone an entire data center. To our knowledge, with the exception of maintenance events, every major cloud operator places VMs statically, and leaves them in place for their entire lifetimes. A *truly dynamic data center*, one in which virtual machines move frequently and with low overhead, could accomplish goals that such static placement prevents. Goals include minimizing energy

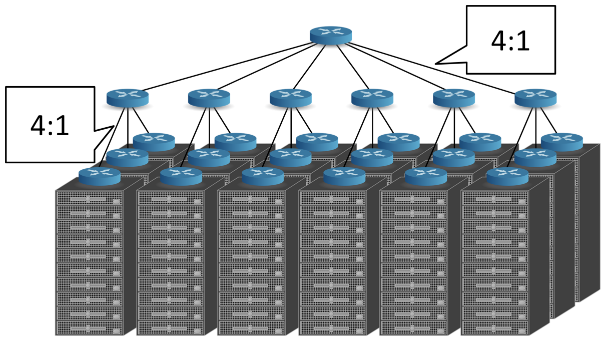


Figure 1.1: Oversubscription in data centers.

consumption (thus saving money) by consolidating VMs onto the minimum number of physical servers possible without violating service level agreements (SLAs) or maximizing workload performance by load balancing tenant workloads evenly across the physical infrastructure. Such goals require ongoing measurement and movement of virtual machines as the workload demand and characteristics change over time, and as tenants add and remove VMs.

Both industry and academia realize the significant efficiency benefits of dynamic VM placement. Optimization tools from VMware [54] and Citrix [11] automate the initial placement of VMs on physical machines (PMs) and can load balance VMs across servers within a resource pool. These and similar products (e.g., [17,21,24]) and research projects (e.g., [71,75,85]) use *local* information from the hypervisor about local resources in their algorithms, such as CPU, RAM, disk, and local network interface traffic counters, and support optimization of roughly rack-sized resource pools (up to 32 hosts). Other shared resources that impact the performance or operation of a VDC are candidates for inclusion into optimization algorithms. Examples include: CPU cache lines, memory subsystem bandwidth (not just capacity), emerging compute devices such as GPUs, and the network.

1.2 The Network

A major resource that is currently invisible to optimization tools is the network. The omission of network knowledge becomes apparent as such systems scale from a single rack, where there is typically no network oversubscription, to the size of a data center, where

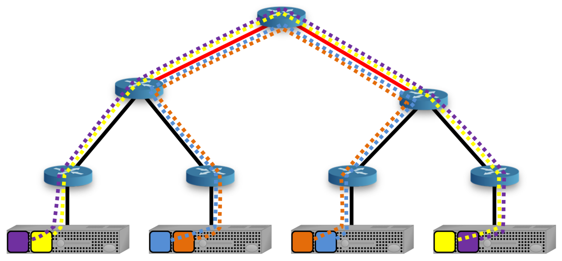


Figure 1.2: Example of poor VM placement causing network bottlenecks.

network oversubscription is common. Data centers are built with one or more tiers of network hierarchy, and as data moves between tiers, there is typically a reduction in available bandwidth, which is called oversubscription. An example of this can be seen in Figure 1.1, which shows an example of a data center with two tiers of network hierarchy above the top of rack (ToR) switches. Between each layer there is a 4:1 reduction of bandwidth. Oversubscription is common because buying, cabling, and powering enough switches to ensure there is no oversubscription (a full bisection bandwidth network, or FBB), can be prohibitively expensive, or simply unnecessary. Thus in data centers with oversubscription, where a virtual machine physically resides determines the quantity of network bandwidth available between it and other VMs in the data center, thus affecting its performance. Figure 1.2 shows an example of a data center with oversubscription. This data center contains four tenants, each with two virtual machines sending significant traffic between them. Each tenant has one VM on the left half of the data center, and one on the right, forcing all traffic through the most oversubscribed links in the data center at the top tier. As a result, network performance for all tenants suffers. While this particular example is a worst-case scenario, it illustrates the importance of incorporating network knowledge when placing VMs.

There are many benefits to performing network-aware VM placement. Network-aware VM placement can increase traffic locality to reduce network bottlenecks and decrease latency, enabling a web service to handle more requests, a MapReduce [49] job to complete in less time, and a search request to return faster, and all three can improve simultaneously. It can also inform consolidation by minimizing the number of network elements needed to meet a desired service level. Improvements in efficiency can benefit both providers and tenants in cloud environments (where the two are decoupled), as well as private data center owner/operators.

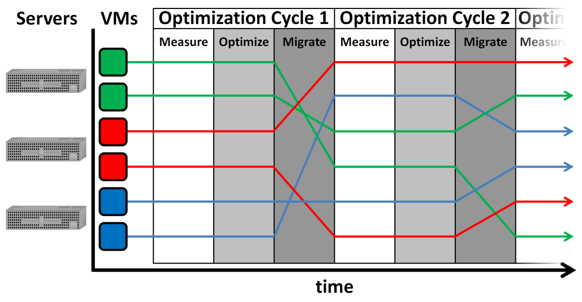


Figure 1.3: Workflow to optimize VM placement.

Today, two commercial VM placement products are widely used in enterprise data centers, but they are not network-aware [11, 54]. I suspected that public clouds could significantly improve tenant workload performance by taking advantage of network knowledge; this belief lead me to the first of two main questions posed in this thesis:

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

That is, with perfect knowledge and an optimal algorithm for placing VMs, how much faster might one or more scale-out workloads run? Most related work does not answer this question, and (a) only considers the efficiency benefits for the operator, but not the improvements in the workload itself [72, 85]; or (b) *does not have complete topology and traffic matrix information when optimizing VM placement* [45, 63, 75, 85].

Figure 1.3 shows the time evolution of an iterative VM placement algorithm, with each iteration consisting of three phases. In the first phase, CPU and network resource use is measured. Next, an optimization algorithm picks a new mapping of VMs to PMs. Finally, the VMs are migrated to their new host(s). The three phases are repeated indefinitely, with the goal of improving workload performance in each cycle. Each phase of the cycle is challenging:

- **Measure.** A measurement period must be picked that is small compared to the rate in which workloads change their network and CPU usage; changes occur during different phases of computation and as workloads come and go. Also, the system must take into account that network usage is elastic — TCP will use more capacity should it be available.

- **Optimize.** Traffic-aware VM placement is NP-hard and does not admit a constant-factor approximation [72].

- **Migrate.** It takes time and places additional load on CPUs and network links when moving VMs. Fortunately both the time and resource overhead is being rapidly improved by other researchers [64, 88], with experimental results showing a VM with 1GB RAM being migrated in about 7 seconds on a 1Gb/s network. My research focuses on understanding the measure and optimize phases which have not been examined by other researchers, and does not include migration data. However, a VDC operator could incorporate the migration cost, as measured in their own infrastructure, into their algorithms as needed.

For the rest of this thesis I will assume an optimization system based on the three-phase optimization cycle. The system performance will critically depend on the algorithm in the "optimize" phase, which leads to this thesis' second question:

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

There are many possible algorithms that could be used to find more optimal VM placements, from heuristics that try to make informed explorations of the solution space, to formalized mixed integer programs. A mixed integer program has the potential to produce an optimal solution, given an indefinitely long solving time. Heuristics can find solutions much faster, but typically at the cost of producing a less than optimal solution.

1.3 Related Work

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

1.3.1 Improve the Network

For single workloads that run across thousands of machines, with little or no locality, providing high or full bisection bandwidth [38, 51, 52, 57, 59, 77, 83] and routing flows across multiple paths [39, 74] may be the only way to improve server-to-server bandwidth. However, even with full bisection bandwidth (FBB), VM placement can improve performance. A single modern core can saturate a 10Gb/s link [30]; yet, many servers have multiple cores, which means more network bandwidth is available between VMs on the same PM than between VMs on separate PMs.

1.3.2 Improve VM placement

The closest and most recent related work is Net-Cohort [63], which measures VM CPU and NIC counters and uses them to determine a better VM placement. Net-Cohort infers VM “ensembles” by looking for hierarchical clusters, however, the authors do not describe how VM clusters map to physical machines. The Net-Cohort paper demonstrates application performance improvements between 136-485% after a single optimization stage from a single starting placement. In contrast, I explore the incremental benefits of gradually increasing network knowledge with seven algorithms, rather than one. I consider algorithms that use the directly measured network traffic matrix, rather than inferred values, and I include the network topology. I measure 1,675 before-after placements, rather than one. My experiments also vary link rates and oversubscription ratios.

Optimization tools from VMware [54] and Citrix [11] automatically balance the placement of VMs on PMs within a resource pool. These and similar products (e.g. [17, 21, 24]) only use *local* information from the hypervisor in their algorithms, such as CPU, RAM, disk, and local network traffic counters, and only support load balancing of resource pools containing up to 32 hosts. Unlike commercial tools, my research uses algorithms with full network knowledge, including the traffic matrix and topology, and runs them on up to 80 PMs.

In other research, [72] defines the Traffic-aware VM Placement Problem (TVMPP) and shows it to be NP-hard. The authors propose a cluster-and-cut algorithm to minimize total network traffic. The evaluation uses VMs with minimal network usage: 80% of the measured VMs have network rates below 106 Kb/s and only 4% exceed 1 Mb/s, and does not measure workload performance. My research aims to maximize workload performance and evaluates workloads that saturate network links.

Sandpiper [85] uses a greedy approach to minimize a VM's "volume", the product of its CPU, memory, and network use, and uses it to mitigate resource hotspots. Sandpiper does not consider the traffic matrix and also does not evaluate the resulting workload performance. My research considers the entire traffic matrix and evaluates total system performance.

Mantri [40] uses static rack bandwidth when placing MapReduce tasks to reduce the effects of network congestion, and is able to reduce total job completion time by 21-31%. Other research has explored live VM migration [48, 78, 89], efficient scale-out service provisioning [80], and the interface presented to the VDC manager and users [70].

Systems like Oktopus [43] and SecondNet [58] allocate virtual networks with reservations for large workloads. This method requires applications to know communication patterns in advance and a data center that can accept such requests. In contrast, my solution requires no knowledge from services and discovers all resource usage. Seawall [76] and NetShare [67] share network bandwidth among a network's tenants more consistently, complementing our approach.

1.4 Why hasn't this been done?

Current virtualized data centers do not integrate network knowledge into their VM placement algorithms. There are several possible reasons.

- **The scale of existing optimization tools is too small.** Existing optimization systems only look at single rack-size numbers of physical machines, which are connected to one or more top of rack (ToR) switches, with full bisection bandwidth between all PMs on the switches. Thus, there is little incentive to add network knowledge to such a system.

- **The network is not really a problem.** Existing data centers have more available bandwidth at all network hierarchy levels than their tenants are capable of using.

This is hard to determine since every data center and workload have unique characteristics. In some cases, this assumption may be true, but measurement studies of current cloud providers have shown that there is some oversubscription preventing clients from achieving full 1Gbps throughput [68, 84]. Additionally, there has been a significant increase in the body of research in recent years on how to build and operate full bisection bandwidth networks and deployments of such networks now being installed in major data centers such as Amazon EC2's HPC clusters [1], which would presumably be unnecessary if the networks were not reaching the limits of their capacities.

- **It is difficult to get detailed network information.** Historically switches provided minimal programmatic interfaces for querying information about the state of the switch. For example, typical switches export a SNMP interface that allows a program to poll counters, but other than a vendor-specific console CLI, this is about it. To perform full network-aware optimization requires knowledge of the exact network topology (a list of hosts, switches, and links between them), and set of network routes that packets traverse between hosts. A network route between any two hosts is composed of the rule(s) in the forwarding information base (FIB) at each network switch that match the packets going from one host to another.

  Until recently, querying for such information from existing switches in a standardized way was not possible. The only practical solution to gathering the necessary information would be to hard code the topology and routing table, which is extremely fragile in the presence of upgrades and failures. Fortunately, the recent Software Defined Networking (SDN) effort created open protocols such as OpenFlow, enabling fine-grained control of switches from a logically centralized controller. The controller can then export the necessary network knowledge to a network-aware optimization algorithm.

- **Having the network knowledge makes solutions difficult to find.** Finding a good placement is no easy task; network-aware VM placement is NP-hard and does not admit a constant-factor approximation [72]. Practical complications include variable resource demands that change with time, workloads that run simultaneously, and the

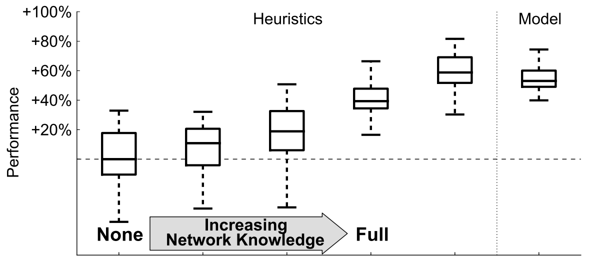


Figure 1.4: Web workload performance by algorithm.

difficulty of inferring the workloads' true resource demands when measuring shared and fully utilized resources (e.g., the elasticity of TCP on shared network links).

Even though it is difficult to integrate network knowledge into VM placement optimization, I have found that it is possible, and this knowledge has improved the performance of networked workloads in a VDC by over 70%.

1.5 Research plan

Finding the answer to what the performance benefits are of adding network knowledge to VM placement for scale-out workloads required a test platform capable of running the necessary experiments. No existing platforms offered the required degree of control and knowledge, so I built a small VDC and a control system to run and monitor experiments that I named *Virtue*. Using *Virtue*, I was able to create a series of before and after optimization experiments to measure the performance affect of network-aware VM placement. The before case must estimate how a specific workload would be placed in a VDC today, measure the workload's performance in this configuration, and then compare it to the performance measured using an optimized VM placement (after case). I later describe details of the platform *Virtue*, the before base case used throughout this work known as *Random*, and multiple algorithms that create optimized VM placements in the after cases, including *Model*, which is used as a rough upper bound, solved using a mixed integer program.

Figure 1.4 gives a sneak preview of my findings (full results are in Chapter 5). The graph plots the improvement in performance (in this case, webpage serving throughput for web servers on 40 PMs) for different optimization algorithms. For now do not be concerned with the actual numbers, but focus on the trend of improving performance from left to right as the algorithms have access to increasing network knowledge. On the right, the last three algorithms (two heuristics, and a mixed integer optimization) know the topology and full traffic matrix. Performance improves and becomes less variable, at the cost of gathering more data and using a more complex algorithm. Later I will show that improvements of 70% are possible, by carefully placing VMs using full network knowledge.

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

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

1.6 Thesis contributions and structure

The main contributions of this thesis are:

- **Experiments.** I evaluate all of the optimization algorithms on an 80-node cluster, with different workloads. I show that *Random* placement gives low performance (throughput and completion time) and significant variance. I show that my optimization algorithms steadily improve performance when given progressively more network knowledge. *My heuristics enable 70% higher throughput than a* Random *algorithm or an algorithm representative of current commercial offerings.*

- **Algorithms.** I introduce VM placement algorithms that improve performance by exploiting network and topology knowledge. As mentioned earlier, *Model* is the performance benchmark: it solves a mixed-integer optimization to calculate a near-optimal placement. *Model* does not scale, so I introduce two greedy algorithms and four simulated annealing variations.

- **Prototype.** I designed and built a prototype VDC resource manager, *Virtue*, that can efficiently launch scale-out multi-tenant workloads, measure resource utilizations, feed the results to an external optimizer, and then run an experiment *with a new placement*, entirely unattended. This system enabled before-and-after comparisons from baseline *Random* placements to improved placements returned by my optimizers. *Virtue* is built on another tool I created, *Beacon*, which controls the network and provides programmatic access to switch internals using the OpenFlow protocol.

In what follows, I will first introduce and describe the platform that was created to efficiently explore answers to the questions posed in this introduction. In particular, I will first describe *Beacon* and then the *Virtue* platform that has been built on top of it. I will next introduce VM placement algorithms and explore how they affect performance across a range of workloads and network configurations. Finally, I will conclude with lessons learned for future, more production-ready VDC optimizers.

Chapter 2

Beacon

Software Defined Networking (SDN) is an approach to designing and operating networks in which the software that controls and configures a network device (control plane) is separated from the hardware that performs the actual packet forwarding (data plane). SDN evolved from research in Professor McKeown's research group [46,47,69]. SDN has many possible uses: centralized network management from a single user interface; improved resource optimization using global rather than local-only network knowledge; separating the edge-to-edge communication concerns from the underlying core network transport mechanisms with network virtualization; network applications with software rather than hardware times to market; and even interoperability between disparate control and data planes using standard protocols. The first implementation of a standard protocol for controlling data planes, OpenFlow, was created at the end of 2007.

OpenFlow's abstraction of a data plane is primarily a flow-table. Each entry in the flow-table contains a set of packet header fields used to match incoming packets, such as a source MAC address or destination IP address. There is also a corresponding list of actions that are taken when a packet matches the entry, common actions include forwarding out one or more ports, or rewriting a header field. Additionally, each flow typically also has counters associated with it. Programming and maintenance of the switch's flow-table, with the exception of flow expiration, is the responsibility of the control plane software.

This chapter presents Beacon, an SDN control platform. As shown in Figure 2.1, Beacon provides a framework for controlling network elements using the OpenFlow protocol,

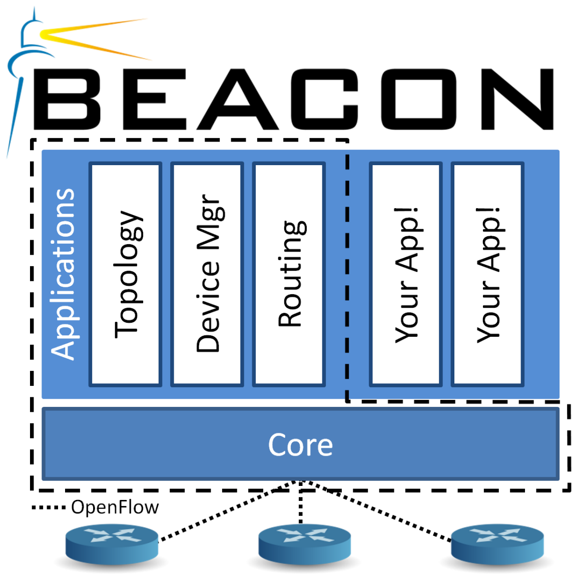


Figure 2.1: Overview of Beacon.

and a set of built-in applications that provide commonly needed control plane functionality. Beacon was first created in early 2010, a time when there were just two open source OpenFlow controllers. One was the basic reference controller that was included with the OpenFlow reference software switch, and only supported operation as a hub or learning switch. The second was NOX [53], created in late 2007–early 2008 by Nicira Networks, which was in widespread use by the OpenFlow community in 2010. Beacon was created to be my research platform, using the best available OpenFlow controller features, and improving in a few key dimensions. Beacon's contributions include:

Developer Productivity. Design and architectural choices that helped developers maximize their time spent productively developing applications on top of Beacon.

Modularity. An example implementation supporting runtime modularization of its components.

Performance. A multithreaded implementation showing linear performance scaling using an OpenFlow controller benchmark.

In what follows I will first present related work in §2.1, then discuss in detail the three main contributions of Beacon: developer productivity in §2.2, modularity in §2.3, and performance in §2.4. Each of the three main contributions will be evaluated in §2.5, and §2.6

will conclude the chapter.

2.1 Related Work

There is a significant body of prior work that led to OpenFlow, and its separation of the control and data planes. I do not attempt to enumerate it in this work, but instead refer the reader to the related work sections in the OpenFlow, Sane, and Ethane papers [46,47,69].

Focusing on just OpenFlow controllers, the first OpenFlow controller platform, NOX [53], was released in early 2008. The core of NOX is written in C++, and applications for NOX can be written in C++ or Python. NOX originally used cooperative threading to process events in a single threaded manner. In 2011 Amin Tootoonchian released a branch of NOX that enables multithreading for its learning switch application.

I began development on Beacon in early 2010; it had limited public availability throughout that year, with its first major public announcement and release in September of 2011.

Many other controllers have been released since Beacon's initial development. They broadly fall into two main categories: open source single-instance controllers, and (so far) commercial closed-source distributed controllers.

The open source controllers are, generally speaking, used for research and development; therefore they tend to be single instance, with varying APIs presented to applications built on their platforms. A major distinction amongst them is the language they are written in.

- *C*. Trema [35], written by NEC, which can also run applications written in Ruby. MUL [23], written by Kulcloud Networks.
- *Haskell.* Nettle [81] and McNettle [82] (currently unreleased).
- *Java*. Maestro [73], released at the end of 2010. Floodlight [15], a fork of Beacon's early 2011 code, released at the beginning of 2012 under the commercial-friendly Apache license.
- *Python*. POX [27], written primarily by Murphy McCauley. RYU [31], written by NTT laboratories.

Distributed controllers are able to distribute their state across multiple running instances, for availability and performance reasons. Some of the public controllers in this space include ONIX [66] from Nicira Networks, Big Network Controller [10] from Big Switch Networks, and ProgrammableFlow [28] from NEC. Of these controllers, ONIX is currently the only one capable of scaling performance by adding additional instances.

2.2 Developer Productivity

There are many ways to create a controller, and the best or right way to maximize the productivity of developers working on a controller or its applications is somewhat subjective. Every developer is different, and techniques that are productive for one developer, could be unproductive when used by another. In this section I address design decisions that I found hindering my productivity as a developer on an existing controller platform, and the resulting choices I made to improve my own productivity with Beacon, and hopefully that of other developers as well. I will present the major decisions made in the following categories: programming language, libraries, editing tool chain, and API.

2.2.1 Programming Language

As a user of NOX for several years, I found that the choice of C++ as its primary programming language was a leading cause of developer time sinks. At the time I experienced 20-30 minute full compilation times, compiler errors that obfuscated the actual cause, and manual memory management leading to time-consuming debugging sessions to resolve SEGFAULTs, memory leaks, etc.

There are approaches that have sought to minimize such problems. Compilation times can be decreased by using an incremental compilation if edits are confined to a peripheral component, but any edits to a shared component can require a near full compile. Also, strict use of techniques such as smart pointers can ameliorate memory errors, if developers make no mistakes. These and other issues inherent to C/C++ all have a cost on developers' time, and are usually tolerated with the belief that C/C++ will deliver the highest application

Language	Managed Memory	High Performance	Cross Platform
C/C++		✓	[1]
C#	✓	✓	[2]
Java	✓	✓	✓
Python	✓		✓
Ruby	✓		✓

Table 2.1: Candidate languages and their attributes.

performance. For some environments this is the right decision; however, for rapid development of an OpenFlow controller that will run on commodity hardware where CPU and RAM are easily (and cost effectively) scaled, I did not believe it to be a necessary choice.

Table 2.1 shows a list of candidate languages considered for Beacon, along with attributes of languages I considered important: managed memory, high performance, and cross platform.

For Beacon I decided to use a language that had automatic memory management to eliminate the majority of memory-related errors. This decision narrowed the language options to those that are virtual machine-based or scripting languages. Languages with automatic memory management also usually have a minimal pre-compilation step, or none, eliminating time wasted waiting for the program to compile. These languages also typically provide excellent error reporting that indicates the exact line(s) that failed to compile/run. Candidate languages ultimately included C#, Java, Python, and Ruby.

High performance is, by definition, a subjective term. However, the lack of true multi-threading in the primary interpreters for Ruby and Python, due to issues such as the Global Interpreter Lock (GIL), eliminated these languages as candidates for Beacon.

The two remaining candidate languages were C# and Java. Upon closer inspection, the Common Language Runtime (CLR), which is Microsoft's interpreter for C#, lacked official support for any platform other than Windows. There is an open source project named Mono [22] that supported other platforms, but it seemed prudent that the primary target

[1]C/C++ applications can be cross platform, but they may require a significant developer effort depending on the complexity of the application, along with stringent requirements for libraries and versions of libraries on the target operating systems. Also, as new operating system versions are released, frequently the applications must be updated to run on them.

[2]No official support from Microsoft for platforms other than Windows. Mono, an open source interpreter, supports running C# (and other .NET languages) on a number of non-Windows platforms.

operating systems for Beacon should be fully supported by the language maintainer(s), and I expected Linux to be one of these targets. Oracle, the primary maintainer of the reference Java Virtual Machines, officially supported all of the major operating systems.

This process of elimination left Java as the main candidate. Additionally, many other successful and high-performance applications written using Java such as Cassandra [4], Hadoop [5], Lucene [6], ZooKeeper [9], and Tomcat [8] reinforced Java as a good language choice for Beacon.

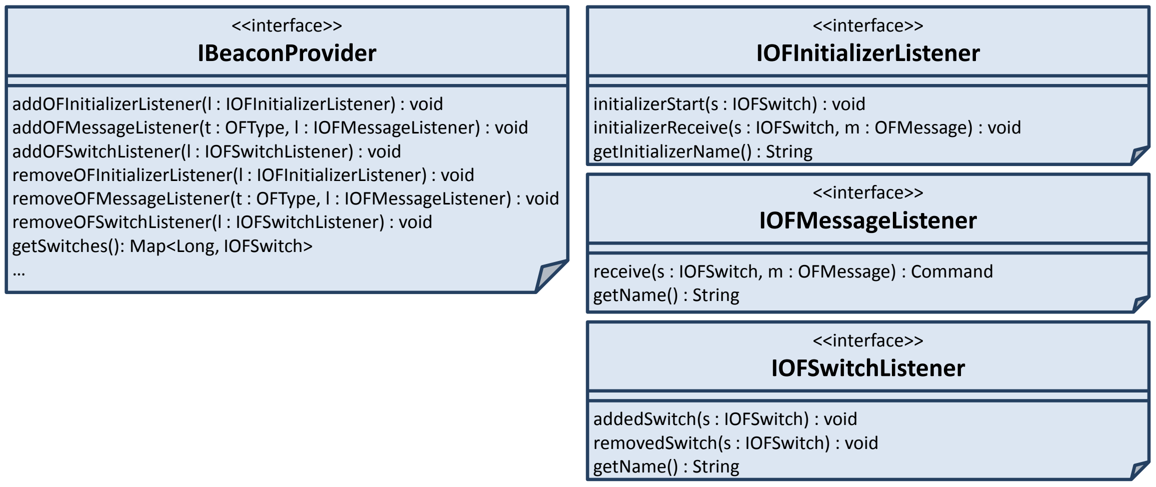
2.2.2 Libraries

Another way applications can improve productivity is through code reuse. In particular, if an existing third-party library can save the developer from writing similar code or speed up development in some way, or both, then it should be considered for inclusion in the application. Beacon uses a number of libraries; the three most significant are Spring [32], Jetty [19], and Equinox [14], each of which in turn depends on other libraries.

Although Spring provides a wide variety of functionality, Beacon uses two main components: the Inversion of Control (IoC) container, and the web framework. A common task in Java is to create instances of objects and then to "wire" them together by assigning one as a property of the other. This often requires purpose-built factory classes, which is tedious. IoC frameworks allow developers to list either in an XML file or as Java annotations, which objects to create, how they are wired up, and provide methods to easily retrieve these pre-wired instances. This saves significant developer time. Beacon uses Spring's IoC framework for wiring within and between bundles, and is explained further in §2.3. Spring's web framework is used to map web (and REST) requests to simple method calls, and auto conversion of request and response data types.

Jetty is an open source enterprise web server. It is optionally included with Beacon, and works with Spring to serve the web UI and the REST interfaces to Beacon.

Equinox is an implementation of the OSGi specification, and its use in Beacon is explained further in §2.3.



<<interface>>
IBeaconProvider
addOFInitializerListener(l : IOFInitializerListener) : void
addOFMessageListener(t : OFType, l : IOFMessageListener) : void
addOFSwitchListener(l : IOFSwitchListener) : void
removeOFInitializerListener(l : IOFInitializerListener) : void
removeOFMessageListener(t : OFType, l : IOFMessageListener) : void
removeOFSwitchListener(l : IOFSwitchListener) : void
getSwitches(): Map<Long, IOFSwitch>
...
<<interface>>
IOFInitializerListener
initializerStart(s : IOFSwitch) : void
initializerReceive(s : IOFSwitch, m : OFMessage) : void
getInitializerName() : String
<<interface>>
IOFMessageListener
receive(s : IOFSwitch, m : OFMessage) : Command
getName() : String
<<interface>>
IOFSwitchListener
addedSwitch(s : IOFSwitch) : void
removedSwitch(s : IOFSwitch) : void
getName() : String

Figure 2.2: Example of Beacon's Core API.

2.2.3 Tool chain

Beacon makes use of the best available open source tool chains. For developers comfortable with editing code in a graphical environment, the Eclipse [13] editor is the recommended choice. Using Eclipse, developers can edit code, run Beacon, debug it on the fly, execute unit tests, and export executables to be used by others. Eclipse itself is built using Equinox, and has significant built-in support for OSGi and modularity.

For developers who prefer editing code using console text editors, Beacon can also be built using Maven [7], which allows developers to compile, run, execute unit tests, and build Beacon executables, all from the command line, on any platform supported by Java.

2.2.4 API

The Application Programming Interface (API) is designed to be both simple and powerful to help developers interact with Beacon, and to impose effectively no restrictions; developers are free to use any available Java constructs, such as threads, timers, sockets, etc. OpenFlow is inherently an event-based protocol; therefore the API for interacting with OpenFlow switches is based around events. Beacon uses the observer pattern to enable listeners to register to receive particular events.

Beacon includes the OpenFlowJ library for working with OpenFlow messages. OpenFlowJ is an object-oriented implementation of the OpenFlow 1.0 specification, which itself is released as C structures, representing the layout of the messages on the wire. OpenFlowJ contains code to deserialize messages coming off the wire into objects, and to serialize and write message objects to the wire.

An example of the core API used in Beacon can be seen in Figure 2.2. Beacon provides an implementation of the *IBeaconProvider* interface, allowing listeners to register to be notified when switches are added or removed (*IOFSwitchListener*), to perform switch initialization (*IOFInitializerListener*), and to receive specific types of OpenFlow messages arriving from connected OpenFlow switches (*IOFMessageListener*).

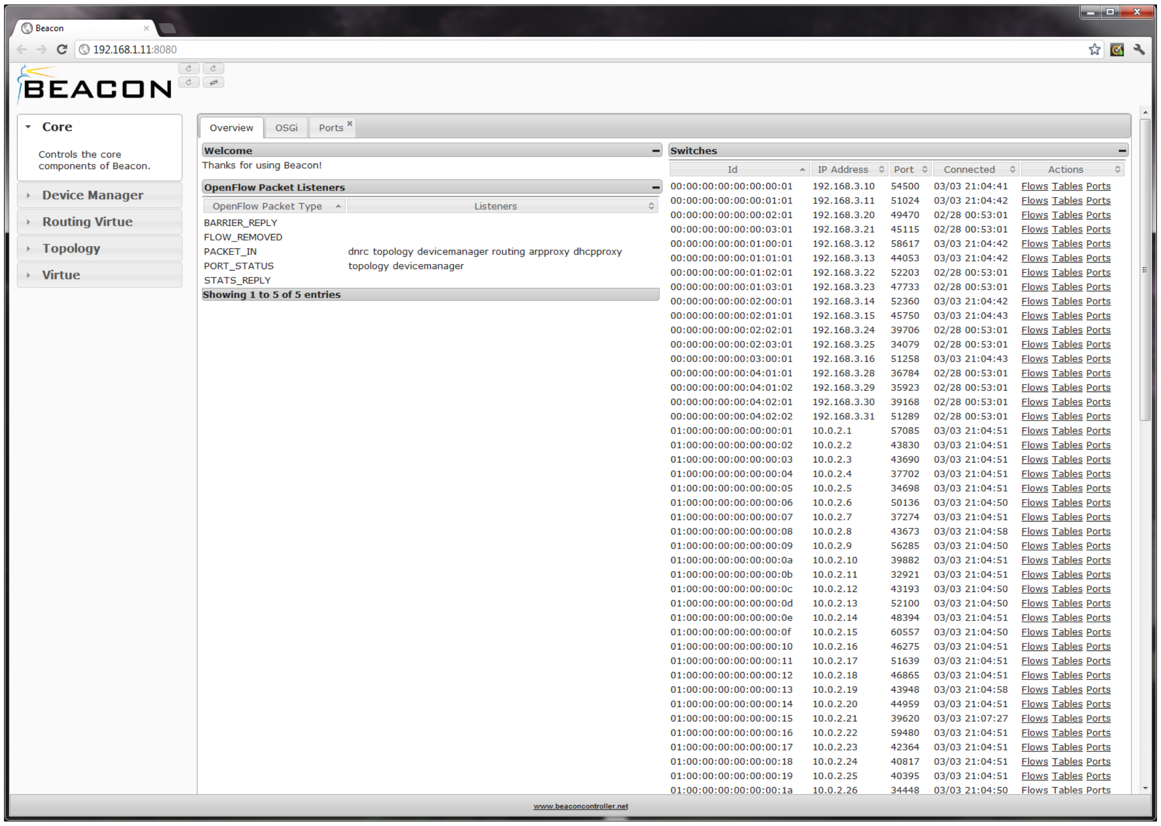
In addition to the core, Beacon includes reference applications that build upon the core, and add additional API.

Device Manager. This application tracks devices seen in the network. It tracks each device's addresses (Ethernet and IP), the last seen date, and what switch and port it was seen on. Device Manager provides an interface (*IDeviceManager*) to search for known devices, and the ability to register to receive events when new devices are added, moved, updated, or removed.

Topology. The Topology application discovers links between connected OpenFlow switches. Its interface (*ITopology*) enables the retrieval of a list of such links, and event registration to be notified when links are added or removed.

Routing. This application provides the shortest path layer two routing between two devices in the network. This application also exports the *IRoutingEngine* interface, allowing interchangeable routing engine implementations. The included implementation uses the all-pairs shortest path [50] computation method. Computing the routes uses both the Topology and Device Manager applications.

Web. The Web application provides a skeleton website for Beacon, which contains a pluggable JavaScript-based accordion on the left side, which when clicked shows a selection of tabs in the center viewport. The Web application provides the *IWebManageable* interface, enabling users of the interface to add their own accordion elements, and a set of tabs (and their corresponding Web URLs) to be displayed when clicked. Figure 2.3 shows



Welcome

Thanks for using Beacon!

OpenFlow Packet Listeners

OpenFlow Packet Type	Listeners
BARRIER_REPLY	
FLOW_REMOVED	
PACKET_IN	dnrc topology devicemanager routing arpproxy dhcpproxy
PORT_STATUS	topology devicemanager
STATS_REPLY	

Showing 1 to 5 of 5 entries

Switches

Id	IP Address	Port	Connected	Actions
00:00:00:00:00:00:00:01	192.168.3.10	54500	03/03 21:04:41	Flows Tables Ports
00:00:00:00:00:00:01:01	192.168.3.11	51024	03/03 21:04:42	Flows Tables Ports
00:00:00:00:00:00:02:01	192.168.3.20	49470	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:00:03:01	192.168.3.21	45115	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:01:00:01	192.168.3.12	58617	03/03 21:04:42	Flows Tables Ports
00:00:00:00:00:01:01:01	192.168.3.13	44053	03/03 21:04:42	Flows Tables Ports
00:00:00:00:00:01:02:01	192.168.3.22	52203	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:01:03:01	192.168.3.23	47733	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:02:00:01	192.168.3.14	52360	03/03 21:04:42	Flows Tables Ports
00:00:00:00:00:02:01:01	192.168.3.15	45750	03/03 21:04:43	Flows Tables Ports
00:00:00:00:00:02:02:01	192.168.3.24	39706	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:02:03:01	192.168.3.25	34079	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:03:00:01	192.168.3.16	51258	03/03 21:04:43	Flows Tables Ports
00:00:00:00:00:04:01:01	192.168.3.28	36784	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:04:01:02	192.168.3.29	35923	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:04:02:01	192.168.3.30	39168	02/28 00:53:01	Flows Tables Ports
00:00:00:00:00:04:02:02	192.168.3.31	51289	02/28 00:53:01	Flows Tables Ports
01:00:00:00:00:00:00:01	10.0.2.1	57085	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:02	10.0.2.2	43830	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:03	10.0.2.3	43690	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:04	10.0.2.4	37702	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:05	10.0.2.5	34698	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:06	10.0.2.6	50136	03/03 21:04:50	Flows Tables Ports
01:00:00:00:00:00:00:07	10.0.2.7	37274	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:08	10.0.2.8	43673	03/03 21:04:58	Flows Tables Ports
01:00:00:00:00:00:00:09	10.0.2.9	56285	03/03 21:04:50	Flows Tables Ports
01:00:00:00:00:00:00:0a	10.0.2.10	39882	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:0b	10.0.2.11	32921	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:0c	10.0.2.12	43193	03/03 21:04:50	Flows Tables Ports
01:00:00:00:00:00:00:0d	10.0.2.13	52100	03/03 21:04:50	Flows Tables Ports
01:00:00:00:00:00:00:0e	10.0.2.14	48394	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:0f	10.0.2.15	60557	03/03 21:04:50	Flows Tables Ports
01:00:00:00:00:00:00:10	10.0.2.16	46275	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:11	10.0.2.17	51639	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:12	10.0.2.18	46865	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:13	10.0.2.19	43948	03/03 21:04:58	Flows Tables Ports
01:00:00:00:00:00:00:14	10.0.2.20	44959	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:15	10.0.2.21	39620	03/03 21:07:27	Flows Tables Ports
01:00:00:00:00:00:00:16	10.0.2.22	59480	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:17	10.0.2.23	42364	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:18	10.0.2.24	40817	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:19	10.0.2.25	40395	03/03 21:04:51	Flows Tables Ports
01:00:00:00:00:00:00:1a	10.0.2.26	34448	03/03 21:04:50	Flows Tables Ports

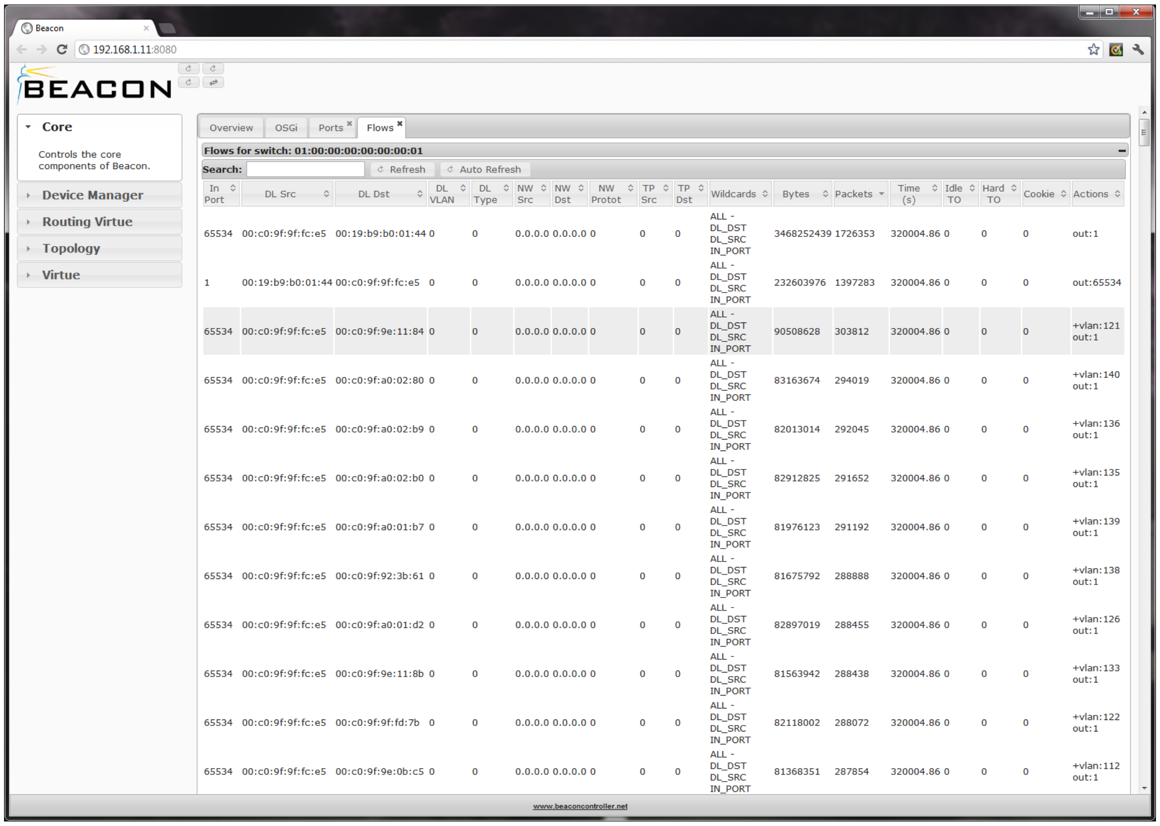
Figure 2.3: Beacon Web User Interface: Switch List.

a screenshot of the Web interface showing a list of all currently connected switches. Figure 2.4 shows a list of all the flows in the flow-table for one of the connected switches.

2.3 Modularity

Another major goal of Beacon was to enable modularity. Modularity is defined at multiple layers. By using interface code abstractions, Beacon enables different concrete implementations of these interfaces to be created and used. This is considered compile time modularity. Additional areas of modularity are at startup, and runtime.

Startup modularity is a feature in most OpenFlow controllers. In Beacon it allows users to choose which applications they want to run at startup. Amongst OpenFlow controllers,



Flows for switch: 01:00:00:00:00:00:00:01

In Port	DL Src	DL Dst	DL VLAN	DL Type	NW Src	NW Dst	NW Protot	TP Src	TP Dst	Wildcards	Bytes	Packets	Time (s)	Idle TO	Hard TO	Cookie	Actions
65534	00:c0:9f:9f:fc:e5	00:19:b9:b0:01:44	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	3468252439	1726353	320004.86	0	0	0	out:1
1	00:19:b9:b0:01:44	00:c0:9f:9f:fc:e5	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	232603976	1397283	320004.86	0	0	0	out:65534
65534	00:c0:9f:9f:fc:e5	00:c0:9f:9e:11:84	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	90508628	303812	320004.86	0	0	0	+vlan:121 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:a0:02:80	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	83163674	294019	320004.86	0	0	0	+vlan:140 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:a0:02:b9	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	82013014	292045	320004.86	0	0	0	+vlan:136 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:a0:02:b0	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	82912825	291652	320004.86	0	0	0	+vlan:135 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:a0:01:b7	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	81976123	291192	320004.86	0	0	0	+vlan:139 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:92:3b:61	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	81675792	288888	320004.86	0	0	0	+vlan:138 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:a0:01:d2	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	82897019	288455	320004.86	0	0	0	+vlan:126 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:9e:11:8b	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	81563942	288438	320004.86	0	0	0	+vlan:133 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:9f:fd:7b	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	82118002	288072	320004.86	0	0	0	+vlan:122 out:1
65534	00:c0:9f:9f:fc:e5	00:c0:9f:9e:0b:c5	0	0	0.0.0.0	0.0.0.0	0	0	0	ALL - DL_DST DL_SRC IN_PORT	81368351	287854	320004.86	0	0	0	+vlan:112 out:1

Figure 2.4: Beacon Web User Interface: Switch Flow-Table List.

runtime modularity is a feature unique to Beacon. It enables users to add, remove, or replace (upgrade) code at runtime, without needing to shut down the entire Beacon process. This enables new ways for developers to interact with deployed Beacon instances. Possible use cases include: creating and installing an application to temporarily improve debug information gathering; rolling out bug fixes or enhancements to applications running in Beacon; installing new applications at runtime from a "app store"; or quarantining and disabling misbehaving applications.

Beacon's modularity is provided by the OSGi framework. OSGi is a specification defining bundles, JAR (archive) files containing classes and/or other file resources, and adding mandatory metadata. Bundle metadata must specify the bundle's identification, version, any dependencies on other bundles or code packages, and a list of code packages exported for other bundles to consume. Developers are free to determine how their application is

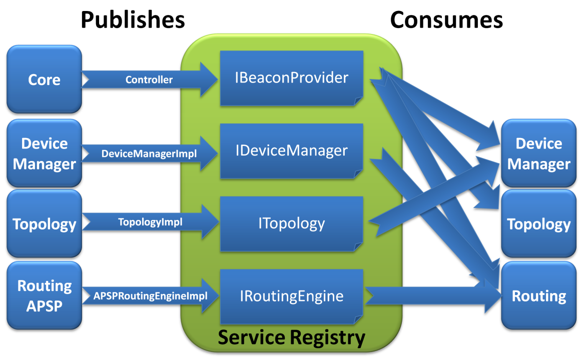
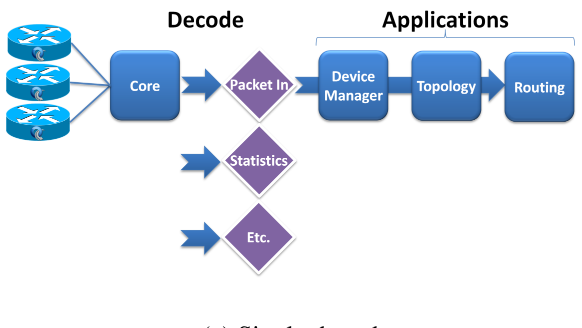


Figure 2.5: Beacon's Service Registry.

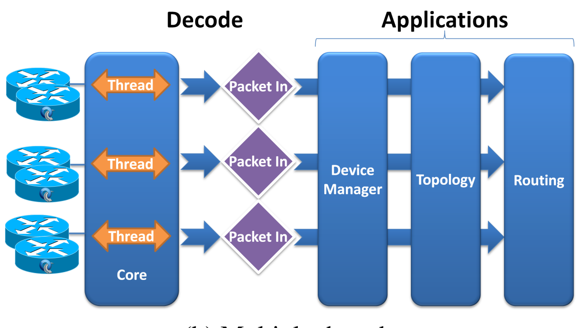
modularized. For example, a bundle may contain multiple applications, just one application, or single application may be spread across multiple bundles. These decisions will usually be made based on the degree of modularity an application has, if pieces of it are candidates for swapping at start or runtime.

Beacon uses an implementation of the OSGi specification named Equinox. Equinox enables Beacon to have modularity at both start time and runtime, allowing bundles to be shut down and removed, restarted, replaced, upgraded, or added, all without shutting down Beacon's core. Modularity does come at a cost to the developer's time; there is effort involved in maintaining the metadata associated with bundles, and ensuring dependencies are correctly exported and consumed. Tools for interacting and maintaining OSGi metadata have been steadily improving, and there is reason to believe in the future that developer burden will be significantly reduced.

A component of OSGi specification is the service registry, where the registry acts as a broker for services to register themselves, and consumers to come and retrieve an instance of a particular service that meets their needs. Beacon heavily uses this model, where service providers export many of the service interfaces mentioned in §2.2.4, and consumers request and receive implementations of such services. Figure 2.5 shows an example of service implementations exported by Beacon's included applications, the interfaces the services expose, and consumers of each service. The service lifecycle happens dynamically: services can come and go based on what bundles are currently installed and running.



(a) Single thread



(b) Multiple threads

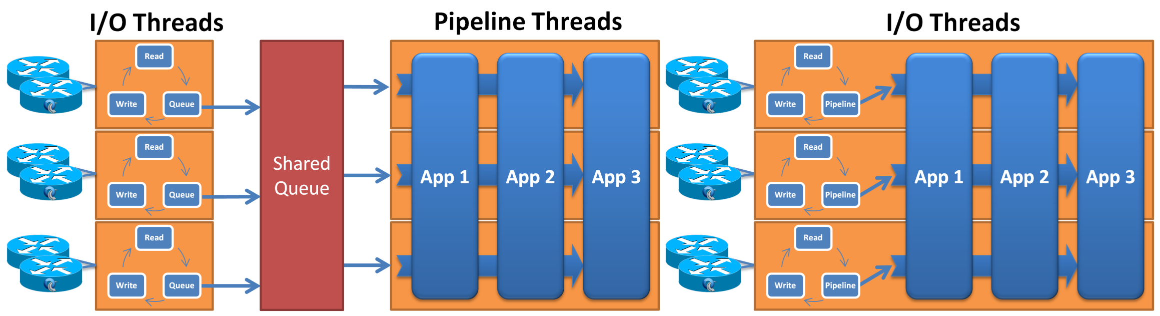
Figure 2.6: Beacon *IOFMessageListener* Pipelines.

2.4 Performance

Performance, in the context of an OpenFlow controller, generally refers to the number of *Packet In* events a controller can process and respond to per second, using a low-overhead application such as a Learning Switch. Another measure can be the average time the controller takes to process such an event. In this section I will describe how Beacon processes messages coming from OpenFlow switches, and how it achieves its high level of performance.

2.4.1 Event Handling

As mentioned in §2.2.4, applications implementing the *IOFMessageListener* interface can register with the *IBeaconProvider* service to receive specific type(s) of OpenFlow messages arriving from connected OpenFlow switches. Registered listeners create a serial processing pipeline for each OpenFlow message type. The ordering of listeners within each pipeline is configurable, and at each step in the pipeline, the listener can choose whether to allow the event to propagate to the next listener in the pipeline or to cease propagation. An example pipeline can be seen in Figure 2.6a. In this pipeline there are three applications registered to receive *Packet In* messages: Device Manager, Topology, and Routing.



(a) Shared Queue (b) Run-to-Completion

Figure 2.7: Possible I/O Designs.

2.4.2 Reading OpenFlow Messages

One of the performance goals with Beacon was to be the first open source OpenFlow controller capable of processing multiple OpenFlow events in parallel. There is a large design space on how to accomplish this, each point having different consequences for programmers, performance, and fairness. There are two major designs considered in this chapter for reading and processing OpenFlow messages, each with its own pros and cons, and implementation details.

- **Shared Queue**. This design can be seen in Figure 2.7a, and contains two sets of threads. The first, I/O threads, reads and deserializes OpenFlow messages from switches, where each switch is assigned to a specific I/O thread, then queues read messages into a shared queue. Each thread is also responsible for writing any queued, outgoing OpenFlow messages for the switches assigned to it.

  The second set of threads, pipeline threads, dequeue OpenFlow messages from the shared queue, and runs each of them one at a time through the *IOFMessageListener* pipeline corresponding to the type of message.

  This is efficient from a pipeline thread point of view, in that anytime there are messages in the queue, all pipeline threads can be busy processing them. However, this design also necessitates a lock on the shared queue, and the corresponding lock contention by both the I/O threads and the pipeline threads. The programming model for this design also precludes some optimizations. Messages from the same switch could be processed by different pipeline threads simultaneously, requiring *IOFMessageListeners* to lock shared data structures on even switch-local data.

Variants of this design could have more than one queue, perhaps one per switch, or even more than one per switch, each with different priorities. This could enables pipeline threads to provide more fairness of message processing between switches, via round-robin servicing of the queues. However, in the case that a small fraction of the connected switches is very busy, each pipeline thread could waste significant time polling empty queues from other switches.

- **Run-to-completion**. Figure 2.7b shows this design. It is a simplified version of the shared queue design, where there is a single pool of I/O threads, each behaving similarly to the shared queue design. However, instead of queueing the message to be processed by a pipeline thread, it directly runs each read message through the *IOFMessageListener* pipeline for the specific message type itself.

  A key difference from the shared queue is that this design does not require locks anywhere on the read path (not counting any locks the applications may contain), because the same thread that deserializes the message also runs it through the pipeline. It also provides the following guarantee to *IOFMessageListeners*: for any given switch, only one thread will ever be processing messages from that switch at a time. This enables lock-free operation for switch-local data structures, such as those in Beacon's Learning Switch application.

  This design is not without its own cons: it is not work-conserving. If half of the connected switches are busy, and all of the busy switches happen to be assigned to only half of the I/O threads, then the remaining I/O threads are idle. In a deployment configured to have as many I/O threads as CPU cores, this would leave CPU cores idle. One solution to this problem would be to monitor and load balance the assignment of switches to I/O threads. Or, alternatively, create a new I/O thread per switch and let the system scheduler manage thread multiplexing across available CPU cores.

  Further, fairness between switches assigned to the same I/O thread is not guaranteed. There is a fundamental tradeoff of throughput versus fairness and latency. For example, how many messages should an I/O thread read from a switch, decode, and run through the pipeline, before moving to the next switch? All available messages? Only some? The best choice for performance is to make as few system calls as possible, and process all available data from a switch in a batch at once, to also maximize the chances of data being in cache. This is simultaneously the worst choice for fairness, because a single busy switch can delay the processing of messages from all other switches assigned to the same thread. For maximum fairness, a thread should process a single message from each switch in round-robin fashion.

In the simplest actual implementation of this design, all available data from a switch are read, deserialized, and processed before moving on to the next switch. When there are no remaining data left to process, the thread waits for further data to be available from switches.

Other possible implementation choices include: reading all available data from each switch into buffers, then deserializing and processing one message from each switch in a round-robin fashion, until all data have been processed, then waiting for further data; or instead of processing a single message per switch, process up to a configurable max number of messages before moving to the next switch, to attempt to bound the maximum queueing delay that messages from switches on the same thread could experience.

Beacon uses the run-to-completion design, and supports a configurable number of I/O threads. OpenFlow switches upon connection are assigned in a round-robin fashion to I/O threads, and remain statically assigned to the thread until they disconnect. Each I/O thread takes the simple approach of processing all available data from each switch before moving to the next.

2.4.3 Writing OpenFlow Messages

The way that messages are written to switches also affects performance. Beacon is multithreaded; therefore writes can occur from multiple threads simultaneously and methods must be synchronized to prevent race conditions. Synchronization constructs are omitted from the pseudocode in this section due to a lack of space, but can be viewed in the actual Beacon source code.

Algorithm 1 Initial Switch Methods

```
1: function WRITE(OFMessage msg)
2:     buf.append(msg)
3:     flush()
4: end function
5:
6: function FLUSH
7:     socket.write(buf)
8: end function
```

Algorithm 2 Revised Switch Flush Method

```
1: function FLUSH
2:     if !written && !needSelect then
3:         socket.write(buf)
4:         written $\leftarrow$ true
5:         if buf.remaining() $>$ 0 then
6:             needSelect $\leftarrow$ true
7:         end if
8:     end if
9: end function
```

The first simple design for how applications write to OpenFlow switches can be seen in Algorithm 1. Applications call the write method, which would serialize and append the message to a switch-specific buffer, then immediately write it to the socket.

Performance with this basic design was not at the desired level, and performance tracing showed that the Java JVM was issuing a system kernel write for every corresponding application write, with no intermediate batching. In a busy system, the time spent in user-kernel transitions was significant.

A revised design fixes this problem. The first part of this design is shown in Algorithm 2, containing a modified flush method that now contains two boolean flags: written and needSelect. The written flag is set to true when a socket write occurs, which prevents subsequent writes from occurring until the flag has been set back to false. The needSelect flag is set to true when there are unwritten data after a socket write, indicating the outgoing TCP buffer was full, and the remaining data needs to be written in the future when there is space in the outgoing TCP buffer.

Algorithm 3 Revised I/O Loop

```
 1: function IOLOOP
 2:     while true do
 3:         for all Switch sw : switches do
 4:             sw.written ← false
 5:             sw.flush()
 6:             if sw.needSelect then
 7:                 sw.selectKey.addOp(WRITE)
 8:             end if
 9:         end for
10:         readySwitches = select(switches)
11:         for all Switch sw : readySwitches do
12:             if sw.selectKey.readable then
13:                 readAndProcessMessages(sw)
14:             end if
15:             if sw.selectKey.writeable then
16:                 sw.needSelect ← false
17:                 sw.flush()
18:             end if
19:         end for
20:     end while
21: end function
```

The second part of the design can be seen as modifications to the I/O loop shown in Algorithm 3. In particular, lines 3–9, and 15–18 have been added to support this design. Lines 3–9 ensure that each switch will write any outgoing data once per I/O loop when the outgoing TCP buffers are not full. Lines 15–18 ensure that when the outgoing data exceeds the available outgoing TCP buffer space, writes are only performed when space has opened in the TCP buffers, which is detected using the system select function call.

The end result is that for lightly loaded systems, messages are either written immediately or once per I/O loop. For systems under heavy load where the outgoing TCP buffers are frequently full, writes only occur when there is space to put the data in the outgoing TCP buffers, and a natural batching effect occurs between I/O loops, decreasing the write calls, and user-kernel overhead.

2.5 Evaluation

Beacon has now been in use for three years, since 2010. This section will evaluate the results of Beacon's three primary contributions, and also include a retrospective discussion of how well the design points have held up over this period of time. The evaluation will present them in reverse order of their introduction: performance, modularity, and developer productivity.

2.5.1 Performance

Beacon is intended to be a platform on which other applications are built; therefore the performance of its OpenFlow message handling capability is relevant to developers selecting a platform to develop on. The main tool currently in use to synthetically benchmark OpenFlow controllers is Cbench.

Cbench simulates a configurable number of OpenFlow switches, each sending a stream of *Packet In* messages to the controller undergoing the test. Cbench is intended to test hub/learning switch behavior, and can vary the source MAC address in the *Packet In* messages by a configurable range to stress the controller's table lookup code. In the following benchmarks Cbench is configured to run 13 tests per controller/thread combination, each lasting 10 seconds. Each test's total responses received are averaged to produce a responses-per-second result, then the last 10 tests' results[3] are averaged to produce the final result shown in the following graphs.

The tests are intended to be as repeatable as possible; therefore they have been run on Amazon's Elastic Computer Cloud (EC2) using a Cluster Compute Eight Extra Large (cc2.8xlarge) instance, which is a virtual machine (VM) with 16 physical (32 including hyper-threading) cores from 2 x Intel Xeon E5-2670 processors, 60.5GB of RAM, running Ubuntu 11.10 based on a StarCluster-modified VM image (ami-4583572c). This instance and software configuration is publicly available for rent by Amazon.

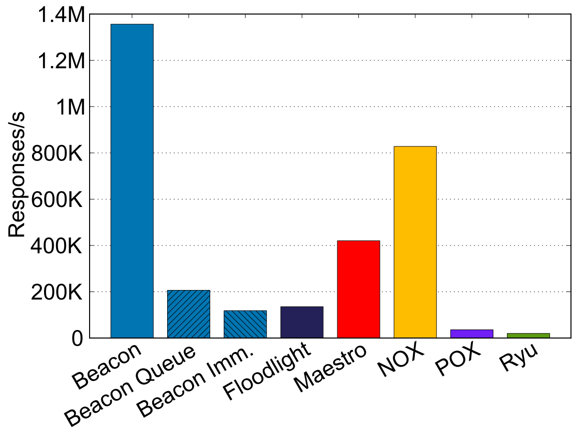
Cbench is run locally, connecting to the controller under test using loopback, and given

[3]The first three tests are considered warmups to provide time for the VM-based languages to perform any adaptive optimization and caches to warm up, and their results are not counted.

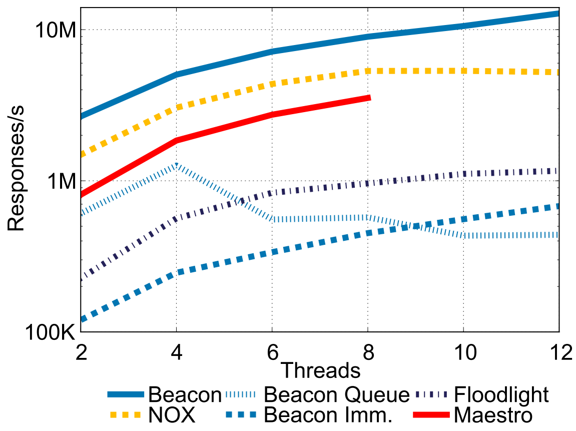
a dedicated CPU core. Running locally was chosen because the traffic between the controller and Cbench can exceed the capacity of the instance's 10Gb NIC, and there is currently no way to assign more than one 10Gb NIC to the instance.

In an effort to be inclusive, as many open source controllers are included in these tests as possible at the time of testing (December 2012), except in cases where the controller did not work correctly with Cbench, and communication with the author did not result in a functioning controller. The following controllers are used in the benchmarks performed in this section: Beacon, Floodlight, Maestro, NOX, POX, and Ryu. Two modified versions of Beacon are included in the benchmarks: "Beacon Queue" and "Beacon Immediate". Beacon Queue is configured to use the shared queue multithreaded design, and the optimized message writing design. Beacon Immediate uses the run-to-completion message reading design, but writes messages to the socket immediately, using the original simple design. Both are presented to demonstrate the significant performance differences between the original and optimized designs. Exact settings used for each controller are listed in Appendix A.

Cbench operates in one of two modes, throughput or latency. In throughput mode, each of 64 emulated switches constantly sends as many *Packet In* messages as possible to the controller, ensuring that the TCP buffer is always full, and that the controller always has work to be done. Figure 2.8 shows the results of running Cbench in throughput mode. First, figure 2.8a runs each controller using a single thread. In this test both Cbench and the controller are each bound to a distinct single physical core on the same processor. Beacon provides the highest single-threaded throughput at 1.35 million responses per second, followed by NOX with 828,000, Maestro with 420,000, Beacon Queue with 206,000, Floodlight with 135,000, and Beacon Immediate with 118,000. Beacon Queue performs much slower than Beacon on one CPU core because the OS must schedule both the I/O and pipeline threads using just one CPU core. Beacon Immediate attempts to write every outgoing message immediately to the TCP socket, requiring a kernel call and its associated overhead for each, reducing performance below Beacon Queue. The large performance difference between Beacon and Floodlight, despite being based on very similar code, is Floodlight's use of the Netty framework for its I/O loop, versus Beacon's custom I/O loop. Beacon's Learning Switch application also uses a custom hash map implementation based



(a) Single thread



(b) Multi-threaded scaling

Figure 2.8: Cbench Throughput Tests.

on Hopscotch Hashing [60], which is significantly more memory efficient than Java's built-in HashMap. Both Python-based controllers run significantly slower, POX serving 35,000 responses per second and Ryu with 20,000.

Figure 2.8b evaluates the scaling properties of the multi-threaded controllers as they scale from using two to 12 total threads, as well as associated CPU cores. In this test, one instance of Cbench is used for tests with four or fewer threads, and a second instance of Cbench is launched for tests with six or more threads. A second Cbench is needed because a single instance is unable to to saturate Beacon when running six or more threads. Beacon scales nearly linearly from two to 12 threads, processing 12.8 million *Packet In* messages per second with 12 threads. Adding another thread in Beacon adds both I/O and pipeline processing, and the Learning Switch application is lock free, so there is no additional lock contention when adding threads. NOX scales nearly linearly from two to eight threads, handling 5.3 million *Packet In* messages per second at eight threads. However, after eight threads, performance decreases. This is because the process now spans two CPU sockets, and cache coherency protocols (and their associated overhead) are needed to maintain synchronization primitives used by NOX. In this benchmark threads one through seven are on the first physical socket, and eight through 12 are on the second socket. Maestro scales linearly to its maximum of 8 threads handling 3.5M *Packet In* messages/s. Beacon Queue has a much slower absolute performance than Beacon, but scales well to four cores, then decreases dramatically due to the same overheads that NOX experiences. Beacon Queue introduces lock contention both at the queue between the I/O and pipeline threads, and additionally on the MAC address to port map for each switch, because this design allows for more than one thread at a time to be processing OpenFlow messages from the same switch. Floodlight scales at the same rate from two through six threads, then slows down from eight through 12; however, it still slowly gains performance. Beacon Immediate shows the slowest initial absolute performance amongst the multi-threaded controllers, but manages to scale linearly, beating Beacon Queue in performance when using 10 and 12 threads because it does not have the locking overhead that Beacon Queue does.

In Cbench's latency mode, each emulated switch sends a single packet to the controller, waits until it receives either a *Packet Out* and/or *Flow Mod* message, then repeats the process as quickly as possible. This test has been configured with a single emulated switch, and because only one message is in flight at any given time, the total number of responses received at the end of the time period can be used to compute the average time it took the controller to process each. The results are shown in figure 2.9. Beacon has the lowest average latency at 24.7 μs, Beacon Queue at 38.7 us, followed by Floodlight, Maestro, NOX, and Ryu, each between 40 and 60 μs. POX is the outlier in this test taking 145 μs on average to process and reply to a *Packet In* message. The extra scheduling time needed when shuffling messages between the I/O thread, queue, and pipeline thread is apparent in the latency difference between Beacon and Beacon Queue. Beacon Immediate is omitted from this graph because Beacon's write behavior and performance is equivalent to Beacon Immediate when there is at most a single outstanding message per switch, which is the case when testing with Cbench in latency mode.

2.5.2 Modularity

Beacon was built with the goal of being highly modular, its functionality being cleanly separated into distinct modules which can be easily added, replaced, extended, or removed. Beacon enables this to happen not only at compile and launch, but also at runtime. I asked two users of Beacon who had each used it in a significant research project to provide me with feedback both good and bad about their experience, and if they used Beacon's runtime

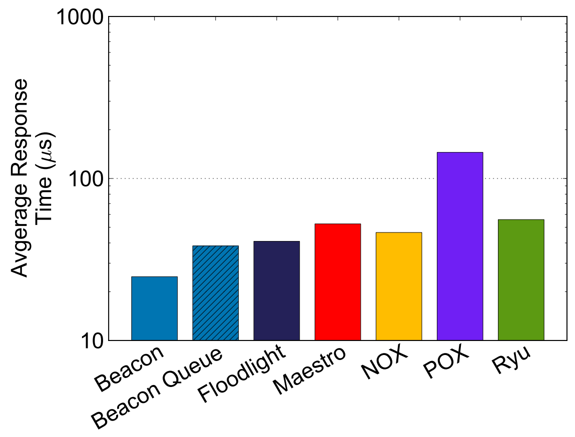


Figure 2.9: Cbench Latency Tests – Single Thread.

modularity capability, whether or not it met their needs.

The first user, Volkan Yazici, reported using Beacon for a distributed OpenFlow controller [87], a multicast media network, and an access controlled Wi-Fi network. He commented, "the simple yet effective design and self-explanatory codebase are the major drives for our preference of Beacon," and, "...(although the) OSGi bundling scheme has its own learning curve, it greatly simplifies the modularization of the whole framework. Further, in the context of our distributed controller implementation, OSGi was the main enabler for us to restart the necessary components of the controller to provide fault-tolerance."

The second user and author of FlowScale [16], Ali Khalfan, responded via email with the following:

> Things I liked:
>
> Its object oriented approach made it really easy to extend the code and build a new application on top of it.
>
> I also really liked the modular approach of Beacon. Using OSGI had great advantages for me in that I was able to easily strip all of the parts I didn't need from Beacon (e.g. web interface, or the learning switch) and just keep the controller itself. The loose coupling of the application was really impressive. In removing some of Beacon's functions, I didn't even have to look at the code or change it.

> The OSGI interface was really useful in my case. Different modules were developed based on the needs of FlowScale. For example, we had a RESTful API interface in its own module. We also had other modules that ran in a continuous loop that we in charge of collecting statistics from the OpenFlow switches and monitoring and updated the flows on different time intervals. Using OSGI, I was able to shutdown those modules, uninstall them, update the code, and then deploy them, while the main application was still running. It didn't always work in a smooth fashion, but it was very helpful at times. Also, if some of the functions were buggy and required a restart, I could just restart their bundles without having to restart the entire application again. It was also a lot easier to troubleshoot a problem since I can isolate the problem to a specific OSGi bundle.
>
> Things I don't like:
>
> One problem with Beacon is that the complexity of OSGI bundles make it really difficult to deploy. Developing the code was easy, but getting it to run took a long time. In addition, the learning curve was really steep and could be really frustrating. A major amount of my time was spent in actually learning how the OSGI framework worked (I'm still learning). But once you get the hang of it you could appreciate its different functions. It was also difficult to get support for troubleshooting OSGI issues.

For my own research, Beacon has been used as a major part of the *Virtue* research platform (see §3) for over two years. Beacon's modular architecture enabled me to create multiple *Virtue*-specific applications (see §3.2.5) and insert them into the pipeline without having to modify existing code. Not all modifications were able to be directly plugged in, Device Manager required many direct modifications to enable hosts with multiple locations. An internally more modular design of Device Manager may have enabled it to be extended rather than directly modified. During development, the runtime modularity functionality of Beacon was primarily used locally to reload modified bundles without restarting a running Beacon instance. This did save time that would have been spent stopping and starting Beacon to test changes, which adds up over the duration of development.

In summary, users have reported that Beacon's modular design decisions were useful during their own development, but at the same time have stated that the addition of OSGi and runtime modularity has come at a sometimes-steep learning curve, and impacted ease of use. I believe there are multiple ways to decrease this developer impact in both the development tools and in Beacon's code. To some extent the tools for working with OSGi are still relatively new (despite OSGi's age), and as has been noted, somewhat developer unfriendly. Tool improvements to help identify common metadata and launch errors would help significantly. Beacon could also add additional self-diagnostics to identify runtime problems such as missing dependencies or services and where possible suggest fixes.

2.5.3 Developer Productivity

Two ways to evaluate how productive developers are using Beacon is first, by the number of individuals exposed to the ideas, design decisions, and uses of Beacon, and second, the size and success of the posterity of Beacon, defined as the projects that use Beacon or its posterity directly (its code, etc), or indirectly by making the same or similar design decisions after having been exposed to it. To explore each of these metrics I will present analytics gathered for the first, and an exploration of the posterity (projects and extensions) of Beacon for the second.

Three different types of analytics were tracked since mid-2011: downloads of Beacon source code and binary packages, page views of Beacon content, and views of YouTube video tutorials. Tracking the Beacon source code downloads is unfortunately incomplete; the primary method used to distribute Beacon to users (as listed in all tutorials) has been to download it using the Git version control system. Git does not provide a built-in way to track checkouts of code, so the numbers presented for source downloads are of a subset of total downloads, which end users performed using download links on the Beacon website. As of February 2013 Beacon has the following analytics:

- 1686 Beacon downloads (subset, see above)
- 90,719 Beacon content page views
- 5,729 Beacon YouTube tutorial views

Beacon has been used in projects such as FlowScale [16] and research to design architectures for distributed controllers [86]. Floodlight, a fork of Beacon's code created by Big Switch Networks, has been very successful in the open source community, receiving over 6,000 downloads as of August 2012. It is also used as the base for the company's commercial Big Network Controller.

More recently industry leaders such as Big Switch Networks, Brocade, Cisco, Citrix, Ericsson, IBM, Juniper, Microsoft, and Redhat came together to create the OpenDaylight [25] project, who's goal it is to create a community-led, industry-supported open source SDN platform. Two initial controllers have been submitted to OpenDaylight, both (at least initially) based on Beacon. One is an open source version of Big Switch Network's Big Network Controller renamed to the OpenDaylight SDN Controller Platform, built on Floodlight which was forked from Beacon. The second, the OpenDaylight Controller was submitted by Cisco, and has significant code (key interfaces from core and topology, and the packet library) and design-level (Java, modularity using OSGi, Maven, Spring) similarities that indicates it originally began as Beacon, or borrowed code and ideas from Beacon. The name of the project further indicates the evolution, from Beacon $\rightarrow$ Floodlight $\rightarrow$ OpenDaylight.

2.6 Conclusion

With Beacon, I initially set out to create the ideal platform for my own research, using what I judged to be the best available software engineering techniques to improve upon existing OpenFlow controllers in three areas: developer productivity, modularity, and performance.

I improved developer productivity by selecting a language, Java, that provides automatic memory management, nearly instantaneous compilation time, and easy identification of compilation errors. Further, I selected libraries to reduce the amount of new code needing to be created, the best available development tool chains, and I created an API with as little surface area as possible, but a high degree of control for developers.

Beacon uses OSGi to enable a previously unseen degree of modularity in OpenFlow controllers. Beacon is modularized into individual components that can each be added, removed, or upgraded at runtime. This enables new use cases such as online app-store functionality, adding code to running instances to improve troubleshooting, etc.

The initial performance goal for Beacon was to be the first multi-threaded OpenFlow controller with linear scaling. It achieved this, and in the process became the fastest publicly available single instance controller, according to the Cbench performance test.

Beyond just my research, Beacon has been fortunate to have thousands of people view its material and use it. Beacon has been used in many research projects, spawned an extremely successful fork, Floodlight, which is itself used in commercial controller products, and continues to influence forthcoming projects such as OpenDaylight.

Chapter 3

Virtue Platform

To understand how VM placement optimization algorithms perform in practice, I built a system that I named *Virtue*, to run relatively large, controlled experiments. *Virtue* contains a software control plane that controls a real data center cluster composed of servers and both hardware and software network switches. Users of *Virtue* create a file describing an experiment's details such as: number and placement of virtual machines, network link speeds and routes, and workload details. The user provides this file to *Virtue* which, then runs the experiment while measuring resource utilization and workload performance. A VM placement optimization algorithm uses these measurements to output an optimized experiment with an improved VM placement, which *Virtue* then runs again to determine before and after optimization performance.

Understanding the performance of a particular workload and network configuration requires many different initial VM placements, resulting in the need to run thousands of experiments using *Virtue*. A significant amount of effort went into ensuring that *Virtue* required as little human "babysitting" as possible. As a result, a user can queue up many experiments to run unattended, including optimization and the running of the optimized results.

In this chapter, I will first describe the hardware platform used to run *Virtue*, followed by a detailed description of the components that comprise the *Virtue* software platform, followed by "a day in the life of an experiment", and a full walkthrough of what happens during the execution of an experiment.

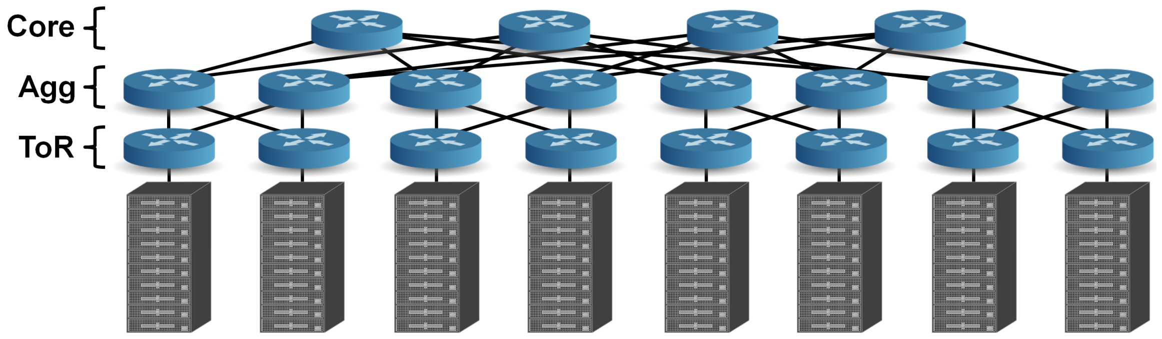


Figure 3.1: View of the Data Center Network Research Cluster (DNRC).

3.1 Data Center Network Research Cluster

Virtue runs customized OS and switch firmware images, which requires access to bare-metal server and switch hardware. Therefore, *Virtue* could not run on shared testbeds such as Emulab [61] or existing VDCs such as EC2. I was fortunate to be able to partner with Google to create the Data Center Network Research Cluster (DNRC), which provides full bare-metal access. The DNRC contains both servers and network switches.

3.1.1 Servers

As shown in Figure 3.1, the DNRC contains 160 Google production servers. Two versions of server hardware were used in the experiments:

1. Dual-socket hyper-threaded Intel Netburst-based Xeon CPUs running at 2.8GHz with 8GB of DDR RAM. These machines began production use within Google in 2004.

2. Dual-socket AMD Opteron 8214 HE running at 2210Mhz with 8GB of DDR2 RAM. These machines began production use within Google in 2006.

The first version of hardware was used for the bulk of the experiments in the evaluation, and the second version was used in the most recent experiments. All servers, 20 in a rack, run XenServer 5.6FP1 (hardware version 1) or 6.0.2 (hardware version 2) hypervisor with paravirtualized Linux VMs. XenServer and VMware ESXi were both considered for use as the hypervisor software in these experiments, ultimately XenServer was selected because the source code was openly available in case modifications were needed. Each XenServer runs Open vSwitch (OVS), a fast software switch with an OpenFlow [69] interface for monitoring and control. OVS measures and controls VM-to-VM traffic inside and between PMs.

3.1.2 Network

When designing the network for *Virtue*'s experiments a few needs were identified:

- The VM to VM traffic matrix, updated frequently.
- The network topology, and how routes between hosts map to it.
- A programmatic way to vary the network's bandwidth.

These needs could have been addressed using a combination of end-host modifications to measure and report traffic, a hard-coded network topology and routing table, standard switches using equal-cost multi-path routing to utilize path diversity, and switch control via SNMP or scripted CLI commands. Instead I chose to meet these needs by designing a network where each network element could be controlled with the OpenFlow control protocol, enabling easier and logically centralized switch programming and counter access, and no need for end-host modification.

The hardware in the DNRC network is a full-bisection-bandwidth three-layer $k = 4$ fat tree [38] built from Pronto 3240 48×1Gb/s $+ 4 \times 10$Gb/s switches. We selected a fat tree design because it was able to achieve full-bisection bandwidth for the full 160 hosts. In addition to two 10Gb/s data uplinks, each top-of-rack switch has a 1Gb/s control port used for NFS and Internet access. Each physical switch runs Indigo firmware [18] containing the OpenFlow agent. OpenFlow [69] is used to configures routes, monitor link utilization, and control network bandwidth via queues. Configurable rate limiters can adjust the oversubscription ratio (16:1 to FBB) and edge link speeds (100Mb/s to 1Gb/s), enabling me to explore how network provisioning levels affect workload performance.

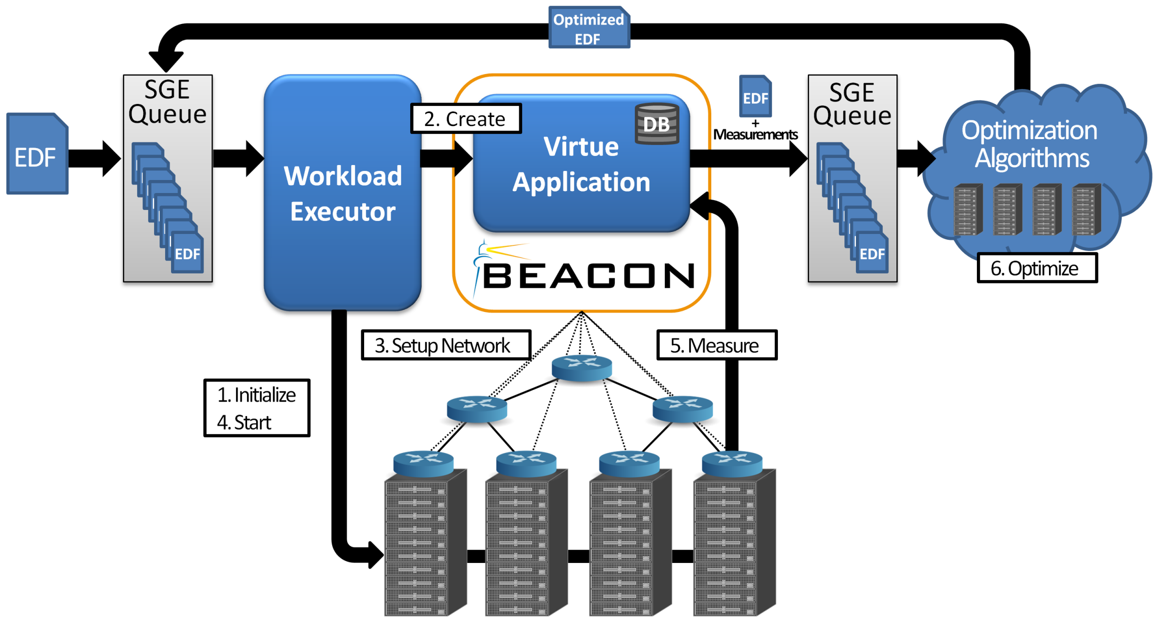


Figure 3.2: Workflow diagram of *Virtue*.

3.2 Virtue Software

Virtue's software is composed of 41,000 lines of custom Java, Bash scripts, and GUI interface code. *Virtue* configures and monitors experiments for a VDC, similar in scope to many data center resource managers [12, 26, 56, 62]. *Virtue* also gives low-level configurability (network bandwidths and routes) and handles batches of queued short-running experiments, rather than a continuous workload. In this section, I describe the functions of the various components that comprise *Virtue*, as seen in Figure 3.2.

3.2.1 Experiment Description File

Individual experiments to be run on *Virtue* are precisely described in a JSON-encoded Experiment Description File (EDF). Each EDF contains the following data:

- Network switches
- Network links, including switch endpoints and link speeds

- Network routes, including the full set of links to traverse for every communicating VM pair
- Physical Machines, including CPU count and CPU speed in cycles per second
- Virtual Machines, including the PM to which each VM is assigned, and the workload that each VM should run

An example EDF for a small experiment can be seen in Appendix B. This small experiment contains two PMs, three VMs, one physical switch, two software switches, and the necessary links and routes to interconnect the VMs.

Having an EDF for each experiment enables reproducible experiments by allowing users to re-run the described experiment using *Virtue*. EDFs also make it easy to run variations of experiments simply by copying the file, making the desired changes, and then running it with *Virtue*.

3.2.2 Sun Grid Engine

Two points in *Virtue*'s workflow require experiment queues: prior to running an experiment on the DNRC hardware and prior to optimizing an experiment. In the first case, only one experiment can run at a time on the dedicated DNRC hardware. It is necessary to queue in the latter case because the number of instances of optimization allowed to run in parallel are limited to ensure that each optimization algorithm has as consistent a completion time as possible.

I used the Sun Grid Engine (SGE) [34] to provide the job execution framework for the two queues in *Virtue*. EDFs and scripts that are used to run them with *Virtue* are submitted as jobs to SGE, which runs the submitted scripts when a job execution slot is available. SGE also provides easy tools to monitor the list of queued, running, and completed jobs.

3.2.3 Workload Executor

The Workload Executor receives an EDF as input and is responsible for configuring the physical infrastructure to match the experiment. These duties include cloning, configuring, and starting virtual machines, copying and initializing the experiment's workload(s) on each virtual machine, starting the workload(s) on all virtual machines simultaneously, and finally cleanup and teardown of the workload(s) and virtual machines.

Additionally, the Workload Executor is responsible for notifying the *Virtue* Application to create a new experiment with the provided EDF once the workload is initialized (but prior to starting it) and also to notify the *Virtue* Application when the experiment has been completed. Both of these notifications are further described in §3.2.5.

3.2.4 Workloads

Virtue can execute real workloads, such as Hadoop and synthetic workloads. Early development of *Virtue* included a VDC simulator that was intended to test different sizes or characteristics of VDCs that could not be evaluated using the DNRC. Only synthetic workloads could be run on the simulator, which motivated the creation of a workload API to allow synthetic workloads to run on both the DNRC, and in the simulator. This API contains functionality to consume CPU cycles, open listening sockets, connect to remote sockets, read and write from those sockets, create timers, and report application level statistics to *Virtue*. Functioning backend implementations of the workload API were created for the DNRC and simulator; however, the simulator is not evaluated in this thesis and is considered a work in progress.

3.2.5 Virtue Application

The Virtue Application runs inside Beacon and extends Beacon in a number of ways:

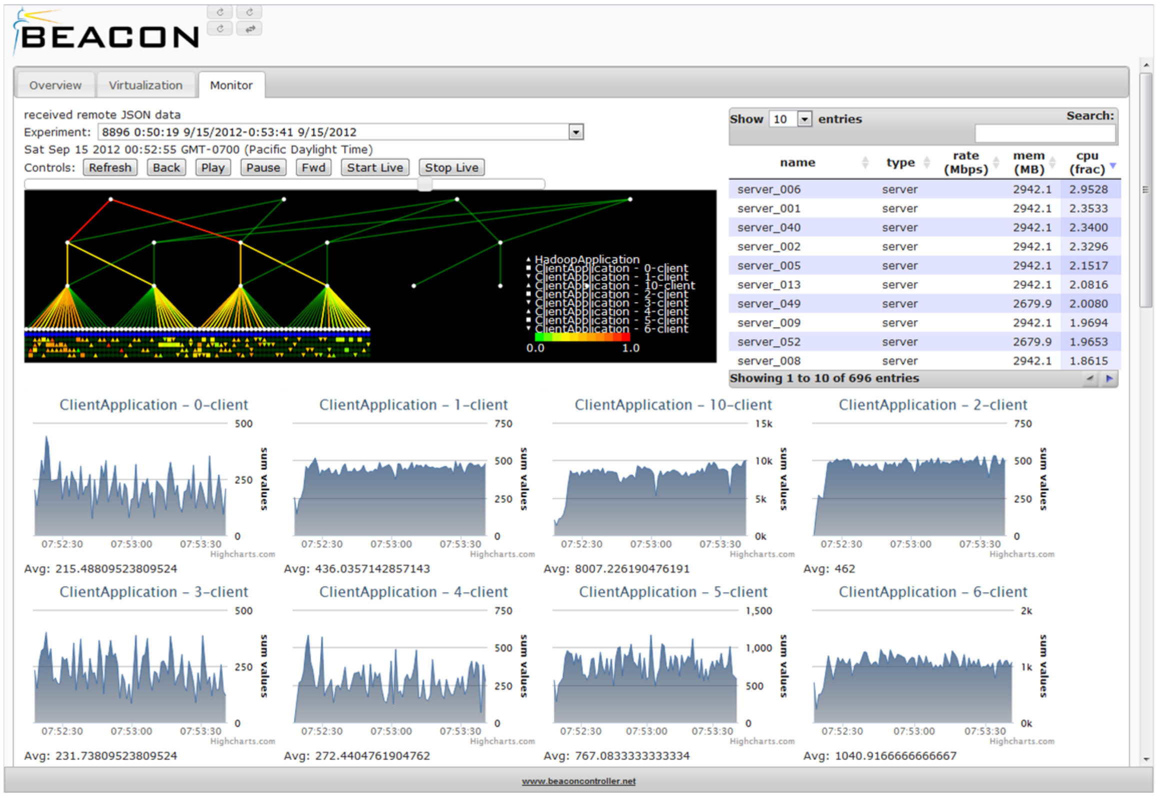
- *Devices with multiple network locations.* Each ToR switch in the DNRC also connects to the management switch, which is not controlled by OpenFlow. Connected to this management switch are control servers and the Internet gateway. Traffic from any of these servers or the gateway to a VM will return through the ToR switch closest to the destination VM, causing Beacon to normally believe that the server or gateway was changing its source location between ToR switches, when, in fact, it just has multiple possible paths.

- *Broadcast traffic*. Beacon does not include an application implementing spanning tree functionality to prevent broadcast traffic loops. Broadcasting traffic in the DNRC would cause broadcast traffic loops; therefore, *Virtue* takes the simple approach of converting broadcast DHCP packets to unicast to the known DHCP server, converting broadcast ARP packets to unicast packets if the target device is known, and otherwise dropping all broadcast traffic.

- *Routing*. Although our experiments use only a tree subset of the actual fat tree network topology, *Virtue* supports the routing of traffic using any of the up to four distinct paths between PMs. The initial routing implementation used OpenFlow's standard reactive routing, where end to end flow entries would be installed between source and destination switches when the first packet in the flow reached the controller. There were multiple problems with this implementation. First, the delay of setting up each flow on demand negatively affected the performance of the workload, both at the start of the experiment and during the experiment because VMs would talk to different VMs over time, allowing flows to timeout and need to be created again when the same pair of VMs eventually communicated again. Second, the physical switches have slow embedded CPUs, and would livelock when receiving a high rate of flow installation messages, increasing the delay, and occasionally even rebooting.

  To solve these problems I created an alternative implementation that installed all needed flows before an experiment began, and did not modify any entries in the physical switches. This implementation assigned four VLAN IDs to each PM, and inserted static flow entries into each physical switch at connect time, set to match on *only* the VLAN ID, and forward packets out the port toward the PM for that particular VLAN ID. Traffic exiting a VM is tagged with one of the four VLAN IDs assigned to the destination VM's PM. When the packet arrives at the destination PM, OVS strips the VLAN ID, and delivers the packet to the destination VM. This implementation solved the problems of inconsistent workload performance due to delays in terms of adding and removing flow entries during experiment execution.

  Virtue's routing engine is also responsible for setting up the static flow entries in OVS at the start of each experiment, as specified in the EDF, and also for teardown after


BEACON
Overview
Virtualization
Monitor
Refresh
Back
Play
Pause
Fwd
Start Live
Stop Live
HadoopApplication
ClientApplication - 0-client
ClientApplication - 1-client
ClientApplication - 10-client
ClientApplication - 2-client
ClientApplication - 3-client
ClientApplication - 4-client
ClientApplication - 5-client
ClientApplication - 6-client
0.0
1.0
sum values
07:52:30
07:53:00
07:53:30
Highcharts.com
www.beaconcontroller.net

received remote JSON data
Experiment: 8896 0:50:19 9/15/2012-0:53:41 9/15/2012
Sat Sep 15 2012 00:52:55 GMT-0700 (Pacific Daylight Time)
Controls:

Show 10 entries Search:

name	type	rate (Mbps)	mem (MB)	cpu (frac)
server_006	server		2942.1	2.9528
server_001	server		2942.1	2.3533
server_040	server		2942.1	2.3400
server_002	server		2942.1	2.3296
server_005	server		2942.1	2.1517
server_013	server		2942.1	2.0816
server_049	server		2679.9	2.0080
server_009	server		2942.1	1.9694
server_052	server		2679.9	1.9653
server_008	server		2942.1	1.8615

Showing 1 to 10 of 696 entries

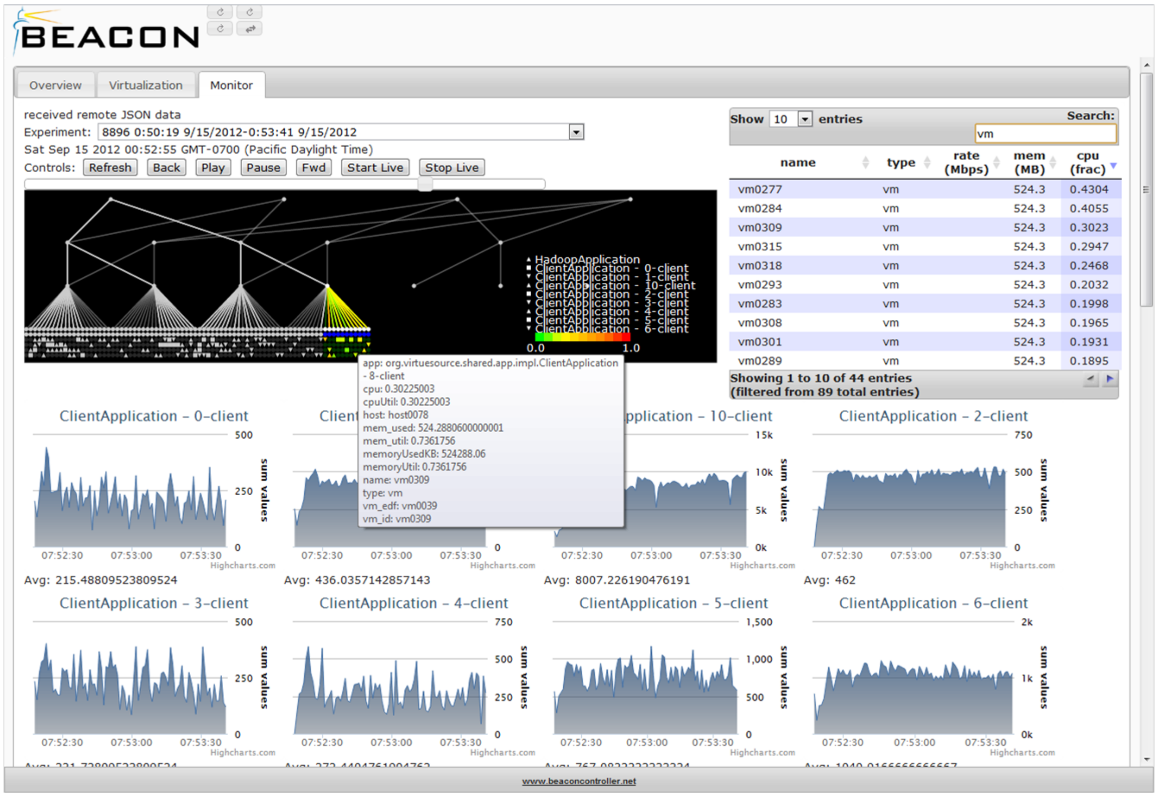
ClientApplication – 0–client: Avg: 215.48809523809524
ClientApplication – 1–client: Avg: 436.0357142857143
ClientApplication – 10–client: Avg: 8007.226190476191
ClientApplication – 2–client: Avg: 462
ClientApplication – 3–client: Avg: 231.73809523809524
ClientApplication – 4–client: Avg: 272.4404761904762
ClientApplication – 5–client: Avg: 767.0833333333334
ClientApplication – 6–client: Avg: 1040.9166666666667

Figure 3.3: *Virtue* Web UI: Exploring an experiment.

an experiment has been completed.

- *Experiment tracking and measurement*. *Virtue* tracks the start and stop time for each experiment, as well as all aspects of the experiment as it runs, to enable post-mortem analysis. *Virtue* tracks the state of the servers using Xen's API, snapshots the listed VMs and PMs, and tracks their CPU and RAM consumption at five-second intervals. On the network side, *Virtue* uses Beacon's API to snapshot the state of switches and links, and polls and stores the port counters and the flow statistics for all flows created for each experiment. Additionally, *Virtue* exposes REST interfaces for applications to self-report statistics such as completion time or requests per second and stores these in its database.

- *Web User Interface*. *Virtue* includes a Web user interface to explore what happened during an experiment. Figure 3.3 shows a screenshot from the Web user interface


BEACON
Overview
Virtualization
Monitor
HadoopApplication
ClientApplication - 0-client
ClientApplication - 1-client
ClientApplication - 10-client
ClientApplication - 2-client
ClientApplication - 3-client
ClientApplication - 4-client
ClientApplication - 5-client
ClientApplication - 6-client
ClientApplication
0.0
1.0
ClientApplication – 0–client
ClientApplication – 10–client
ClientApplication – 2–client
ClientApplication – 3–client
ClientApplication – 4–client
ClientApplication – 5–client
ClientApplication – 6–client
sum values
07:52:30
07:53:00
07:53:30
Highcharts.com
www.beaconcontroller.net

received remote JSON data
Experiment: 8896 0:50:19 9/15/2012-0:53:41 9/15/2012
Sat Sep 15 2012 00:52:55 GMT-0700 (Pacific Daylight Time)
Controls: Refresh Back Play Pause Fwd Start Live Stop Live

Show 10 entries Search: vm

name	type	rate (Mbps)	mem (MB)	cpu (frac)
vm0277	vm		524.3	0.4304
vm0284	vm		524.3	0.4055
vm0309	vm		524.3	0.3023
vm0315	vm		524.3	0.2947
vm0318	vm		524.3	0.2468
vm0293	vm		524.3	0.2032
vm0283	vm		524.3	0.1998
vm0308	vm		524.3	0.1965
vm0301	vm		524.3	0.1931
vm0289	vm		524.3	0.1895

Showing 1 to 10 of 44 entries
(filtered from 89 total entries)

```
app: org.virtuesource.shared.app.impl.ClientApplication
- 8-client
cpu: 0.30225003
cpuUtil: 0.30225003
host: host0078
mem_used: 524.2880600000001
mem_util: 0.7361756
memoryUsedKB: 524288.06
memoryUtil: 0.7361756
name: vm0309
type: vm
vm_edf: vm0039
vm_id: vm0309
```

Avg: 215.48809523809524
Avg: 436.0357142857143
Avg: 8007.226190476191
Avg: 462

Figure 3.4: *Virtue* Web UI: Selecting a subset of the topology to examine.

when viewing an experiment. The top section of the UI displays the start and end times of the experiment and contains controls that allow the user to explore the experiment by changing the current time to different time points within the experiment.

The black area just below the top controls shows a view of the network topology of the experiment, the servers, and virtual machines. Colored lines indicate the utilization of the link at that point in time and the colors of the VMs indicate the CPU utilization. VMs have different markers to indicate the workloads being run on them.

The top right area contains a filterable and sortable list of resource utilizations at the selected point in time. This list can either be searched in the text box or the user can select a range of resources in the topology view, as seen in Figure 3.4, and this list will show only the corresponding resources.

At the bottom of the UI is a set of graphs; these graphs show the self-reported measurements from the running workloads during the course of the experiment. Each graph can be drilled down further, to see exactly what happened during fine grained periods of time.

Having this UI was critical in understanding what was happening in experiments. In particular, often for a given set of experiments that varied only by initial VM placement, there would be a few with particularly low performance either before or after optimization. This UI enabled me to visually examine which VMs had which workloads, and quickly see where the bottleneck(s) were by looking for red network links or VMs, indicating they were fully or nearly fully utilized. This information could usually then be used to step backwards and understand why the algorithm chose that particular VM placement.

The UI also helped debug problems. I noticed when running experiments overnight that usually one of the experiments would experience degraded average performance, but similar configurations did not. I used the UI to track down the time and characteristics of the performance anomalies that were occurring, which correlated with the time that a default cron job was running on all VMs. After disabling the cron job, performance returned to the expected levels.

3.2.6 Optimizers

Individual optimization algorithms are discussed in detail in §4; the *Model* algorithm runs using the CPLEX solver, whereas the remaining algorithms are single or multithreaded Java programs. Each optimizer is provided with the EDF which describes the experiment to optimize, as well as an averaged set of CPU and network measurements gathered during the experiment's execution. The optimizer produces a new EDF as output, containing an improved VM placement with the goal of improving resource utilization and workload performance.

Virtue's experiments run and evaluate many different placements per workload, so it is important that many optimizations can run in parallel and on fast hardware. To achieve this goal, I configured clusters of machines on EC2 using StarCluster [33]. Each cluster had a

SGE queue for EDF files waiting to be optimized.

Two types of clusters were used. For CPLEX optimizations, which could take hours or even days to run, one full High-CPU Extra Large EC2 VM was allocated per optimization, with up to 50 running at a time. The Java optimizers, which usually completed in at most a few tens of minutes, were run on a single Cluster Compute Eight Extra Large EC2 VM. This VM had 32 exposed processing cores and would run up to 25 single-threaded optimizations at a time, fewer for multithreaded optimization.

3.2.7 Experiment Overhead

Virtue was built to be able to run thousands of experiments, with the intention that throughput-measuring workloads would run for roughly three minutes, and those measuring completion time would run to completion (typically still a single-digit number of minutes). In between each experiment there is necessary setup and teardown time used for tasks such as installing and coordinating the launch of the workload across all VMs, and correspondingly stopping the workload and returning the cluster to its state prior to running the experiment. This overhead between experiments becomes important to minimize when there are thousands of experiments to run.

The initial implementation of Virtue would delete all VMs after an experiment, then clone and launch all VMs required for the next experiment. This added approximately ten minutes of overhead in between each experiment, primarily due to the single XenServer pool master which coordinated all operations, but ran very slowly and could only run a few in parallel, despite the actual PMs being capable of running much faster. XenServer further required that the pool master have the same hardware as the pool members, precluding us from using a much faster server as the pool master and eliminating the bottleneck.

An optimized design reused the same VMs for each experiment (of the same size), but would migrate them in between each experiment to ensure they were running on the PM specified by the experiment. This decreased the per-experiment overhead to just a few minutes, which was better but not sufficiently small given the quantity of experiments.

The final optimization was to instantiate enough VMs (four) on every PM such that any workload could be mapped onto already running VMs. Thus the overhead between each experiment was reduced to installing and starting the workload, and shutting it down. This reduced experiment overhead to a few tens of seconds.

3.2.8 Automation

Once the per-experiment overhead was minimized, the focus shifted to how to manage the execution of thousands of experiments. Initially each experiment required a human to run, monitor, and progress it through each stage in the workload pipeline. This initial process was not scalable, the only tenable solution being to build it to run in as automated a fashion as possible, requiring human intervention only when absolutely necessary. A suite of Bash scripts were written to manage the lifecycle of each experiment amongst the various components, however, outside of exceptional conditions, users need only interact with a single script for queueing experiments for execution. This single queue script allows a user to queue up a set of experiments to be run and/or optimized with a specified VM placement algorithm and then optionally run the optimized experiments to determine the before-after performance difference.

After *Virtue* has completed all the requested actions for the queued experiments, or in the case of an exceptional condition during execution, a cell phone text message notification will alert the user with the status.

3.3 A Day in the Life of an Experiment

The overarching goal of *Virtue* is to enable the rapid testing of workloads under different configurations. In the evaluation, for a particular workload and network configuration, either 25 or 50 variations of initial VM placement are tested, corresponding to 25 or 50 EDF files. Each of these is run through the workflow seen in Figure 3.2.

The specific sequence of events that occur when running an experiment using *Virtue* is the following:

- The user writes a short Java [1] application to generate EDF(s) for a particular workload

[1]The EDF is a text-based JSON file that could be generated with a program in any language or even by hand, but *Virtue* provides library code in Java to make creating the generator as easy as possible.

and system/network configuration.

- The user runs the Java application, generating EDF files that vary the initial VM placement in each.
- The user runs a script which creates a deployable workload and queues the EDFs to run on the DNRC.
- The following occurs for each queued EDF:
  - The Workload Executor creates any missing VMs and deploys and initializes the workload to all VMs in the experiment (#1).
  - The Workload Executor passes the EDF to the Virtue Application, which creates an experiment in the database (#2) and configures the network according to the EDF, including link speeds and static routes (#3).
  - The Workload Executor then starts the workload on all VMs in the experiment (#4).
  - The Virtue Application records CPU and network measurements, and application-specific performance metrics while the experiment is running (#5).
  - Once the workload completes, the Workload Executor triggers *Virtue* to end the experiment in the database, and tear down the network routes.
  - (Optional) The EDF plus resource measurements are then queued for optimization with one of the algorithms running on EC2.
- (Optional) The optimized EDFs are then queued to be run on the DNRC, producing before and after optimization performance measurements.
- The user is notified that their experiments have completed.

The design and implementation of *Virtue* has been successful in enabling me to run before and after performance comparisons, across a wide variety of configurations. To date, *Virtue* has been used to run more than 10,000 experiments. The next chapter contains a discussion of the different VM placement optimization algorithms that have been evaluated using *Virtue*.

Chapter 4

Algorithms

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

This section first presents one possible optimization metric used by algorithms to improve performance, followed by the selection of an algorithm that creates baseline VM placements to compare more optimized placements against. Next, this section describes a mixed-integer model of the system used to find the believed optimal VM placement given its knowledge. And finally a discussion of multiple heuristics that were explored as practical alternatives to solving the mixed-integer model.

4.1 Optimization Metric: Minimax

There are multiple possible options for the metric that is optimized by VM placement algorithms. Each metric I explored tends to make different tradeoffs, and it is unclear that there exists a "best" metric. A common problem shared by most metrics is how to mix the utilization values from different types of resources. For example, is having a CPU that is at 90% utilization as "bad" as having a link at 90% utilization? And what if the link was a different capacity, say 1Gb/s vs 10Gb/s? For this work I assume that equal utilizations of different resource types are equivalently "bad", but it would be possible for a VDC operator to selectively scale resource type utilization or even individual resources to boost their priority in the metric computation.

The resources included in the optimization metric are restricted to CPU and network links. There are other resources that could be included, however the goal of this work is to understand the effects of adding network knowledge to CPU knowledge, which is the main resource used by commercial products today.

One example of the early metrics I explored was minimizing the average utilization of CPUs and network links. This looks positive from a network perspective, as it will tend to coalesce communicating VMs close together, decreasing the path lengths that data must travel. Unfortunately however this metric does not work for CPUs, because moving VMs to different PMs does not decrease the total CPU use, nor the available CPU capacity. The end result is VMs packed very tightly together, but CPUs that are highly utilized, which harms performance. A possible fix for this problem is to use the square of the utilization, which would penalize resources quadratically as they approached 100% utilization. Unfortunately this metric was computationally infeasible when used in the *Model* algorithm, and not explored further.

An alternative to averaging all resource utilizations would be to instead examine just a single resource. Minimizing the maximum utilization, a heuristic often referred to as Minimax, is commonly used for load-balancing. Minimax is simple to understand and implement, performs well in practice, and has previously been used to allocate tasks onto networked infrastructures [42]. The main shortcoming of Minimax is that it will not try to improve performance if the utilization of the maximally utilized resource cannot be decreased. For example, if two VMs are communicating at full line rate but cannot be placed on the same PM, the optimizer cannot reduce the 100% utilization of the links between them, and it will not attempt to optimize any other resources. This problem can happen in scale-out, multi-tenant data centers where it is likely that at least one link is fully utilized. Nonetheless, as will be seen, Minimax is a good starting point to demonstrate performance improvements using network-aware algorithms. All the algorithms described in this chapter use Minimax as the optimization metric.

4.1.1 Baseline Placement

When selecting the VM placement to be used as the baseline for comparison it is important to understand the environment it represents. For example, enterprise VDCs are likely to know the behavior of their VMs ahead of time, and place them on PMs accordingly. Further, load balancing products from VMware and Citrix that use CPU knowledge can be used at the scale of a rack in such environments. A baseline for this environment should take both of these factors into consideration. On the other hand, a cloud providers do not know the behavior of tenant VMs before launching them, they only move VMs at runtime in the case of server maintenance, and their algorithm for choosing the PM to launch a VM on could include current resource utilization and/or fault tolerance restrictions. Because cloud providers keep VMs primarily statically assigned to their initial PM, over time as tenants add and remove VMs (causing fragmentation), and as resource use changes, the actual placement (with respect to resource use) will tend towards random assignment.

This work is primarily concerned with the harder case of running cloud VDCs where VM behavior is unknown ahead of time. Therefore the baseline was chosen to reflect the cloud environment, where actual placement tends towards random. The *Random* algorithm generates baseline placements that distribute VMs at random across all PMs; some PMs host several VMs, while others host none. If the algorithm picks an oversubscribed PM (in the evaluation each PM is allowed to host at most four VMs), it randomly picks a new PM. Because it does not consider resource usage, *Random* represents a lower bound on performance; dynamic placement optimization algorithms that are aware of resource usage should fare better.

One placement is insufficient to be representative of the range of possible placements a cloud VDC could experience, therefore the *Random* algorithm is used to generate a range of initial placements that together constitute the baseline performance of a particular workload.

4.2 Mixed-Integer Model

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

4.3 Practical Algorithms

The practical algorithms considered next give performance numbers that generally fall between *Random* and *Model*. I present algorithms with varying degrees of network knowledge, as this knowledge could be difficult to gather, or might simply be unavailable. One of the major goals of this thesis is to understand how performance improves as algorithms become more network-aware. Table 4.1 categorizes the algorithms by the information they use, including VM CPU utilization and PM CPU capacity, VM NIC traffic counters and PM NIC link capacity, the VM-to-VM traffic matrix, and the full network topology, including link capacities and routing information. RAM and other resources are not considered by the algorithms presented in this chapter so that the evaluation can focus on the performance improvements when network knowledge is added to today's CPU-based optimization.

There are many practical algorithms that could be used to optimize VM placement, but it is impractical to explore all of them. This thesis selected a few different choices to

[1]The evaluations in this thesis all run on tree-like networks, therefore *Model* could probably be accelerated by exploiting this knowledge.

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

Table 4.1: Optimization algorithms and the data they use.

initially explore this space.

The first choice was to try an oft-used greedy algorithm. This thesis examines two different greedy algorithms, one named *Greedy Net* starts with the existing placement and tries to incrementally improve it by using only network knowledge. A second greedy algorithm, *Greedy Fill*, explores what happens when it isn't limited by an initial sub-optimal starting location. *Greedy Fill* uses full CPU and network knowledge.

Greedy is one way of choosing the next state, and there are a variety of other heuristics that each have distinct methods of choosing the next state to explore. *SA* is one such heuristic which was successfully used to map network flows onto network links [39], which is also needed in this work to determine network link utilization. *SA* was selected to represent the non-greedy class of heuristics, and is examined with multiple different sets of input data to better understand how valuable each piece of data is to final workload performance.

4.3.1 Simulated Annealing

Four of the practical algorithms presented here are based on simulated annealing. Simulated Annealing (*SA*) is a probabilistic optimization heuristic that strives to find the global optimum in a search space that may have many local minima. Initially, *SA* algorithms take random steps with high probability in an effort to escape local minima. As time progresses, *SA* algorithms become more conservative and less likely to take steps that do not immediately improve the optimization metric [65]. The parameters include:

- The temperature T, which represents the runtime allotted for the algorithm. Each iteration decrements T by one unit, and the algorithm terminates when the temperature reaches zero. T is 1000 for smaller experiments and 5000 for larger ones.
- An initial state s in the search space S is a full placement of VMs to available PMs. I use a baseline VM placement generated by *Random*.
- A state transition function that generates a neighboring state s' from the current state s. The state transition function generates s' by choosing, with equal probability, between: (1) swapping two randomly selected VMs; and (2) moving a VM from one PM to another. The latter ensures that the algorithm can still find neighbors whenever a swap is not possible.
- An energy function, $E(s_i)$, that computes the "energy" level of state s_i, which for my purposes is an estimate of the maximum utilization of VM placement s_i. The goal of the algorithm is to find the state in the search space with the lowest energy.
- An acceptance probability function, $P(e, e', t, T)$ that determines, in each iteration, the probability of transitioning to a neighboring state s'. P is itself parameterized by the energy of the current state, $e := E(s)$; the energy of the neighboring state, $e' := E(s')$, the current temperature, t, and the original starting temperature, T.

The acceptance function I designed for all four of my *SA* algorithms is shown in Algorithm 4. If the energy level of the neighboring state is less (better) than that of the current state (i.e., $e' < e$), P returns 1.0. Otherwise, P returns a probability that is relatively high initially (while the temperature t is close to T) but tends toward zero as t decreases. This leads to the behavior I described before—that is, a pronounced tendency to randomly explore that decreases with time.[2]

I implemented four variants of simulated annealing with different energy functions. Below, I describe each variant along with the information about the infrastructure that each one requires:

- ***SA C*** - Uses CPU utilization reported by VMs and the maximum CPU capacity of the PMs. The energy function returns the estimated maximally utilized PM CPU. *This*

[2]This accept function differs slightly from the canonical *SA* algorithm to better align with the small energy value range (0-1) in my *SA* algorithms.

Algorithm 4 Simulated Annealing

```
1: function $P(e, e', t, T)$
2:     if $e' < e$ then return 1.0
3:     else
4:         return $exp^{-4.6*(e'/e)^2*((T-t)/T)}$
5:     end if
6: end function
```

is a simplified version of what appears to be used by VMware's DRS today [54,55]. *SA C* can be used as an alternative baseline (instead of *Random*) in comparisons to network-aware algorithms, representing the rack-scale CPU-aware optimization baseline used in smaller enterprise data centers today.

- ***SA CN*** - *SA C* plus limited network awareness: the VM NIC's rate over time, and the maximum capacity of the physical NIC on the machine. The energy function returns:

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

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

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

4.3.2 Greedy Network

In contrast to *SA*'s more complex exploration of the VM placement state space, the next two presented algorithms use a more simple greedy approach for state exploration. The first

Algorithm 5 Greedy Network

```
1: function GreedyNet(edf)
2:     repeat
3:         $l \leftarrow MaxUtilizedLink(edf)$
4:         $curUtil \leftarrow l.util$
5:         $moved \leftarrow false$
6:         for NetworkFlow f in l do
7:             vmCandidates[f.src] += f.rate
8:             vmCandidates[f.dst] += f.rate
9:         end for
10:        $SortDesc(vmCandidates)$
11:        $minUtil \leftarrow curUtil$
12:        for VM vm in vmCandidates do
13:            for PM pm in pms do
14:                $util \leftarrow MaxUtil(edf$ : vm on pm)
15:                if $util < minUtil$ then
16:                    $minUtil \leftarrow util$
17:                    $minPM \leftarrow pm$
18:                end if
19:            end for
20:            if $minUtil < curUtil$ then
21:                $MoveVM(edf, vm, minPM)$
22:                $moved \leftarrow true$
23:                $curUtil \leftarrow minUtil$
24:                break
25:            end if
26:        end for
27:    until $moved == false$
28:    return edf
29: end function
```

algorithm, *Greedy Net*, is described in Algorithm 5 and uses traffic matrix, topology, and routing data. *Greedy Net* does not use CPU data, to highlight the affect of only the network on workload performance. *Greedy Net* works by repeatedly finding the most congested link, then moves VMs sending traffic across this link to other PMs when a move will decrease the maximum link utilization. It stops when no move will decrease the maximum link utilization.

Algorithm 6 Greedy Fill

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

4.3.3 Greedy Fill

The *Greedy Net* algorithm is hamstrung by the starting state — if the initial placement is really bad, *Greedy Net* can get stuck in local minima and never break out to find a good placement. To study this effect, I also created the *Greedy Fill* algorithm, which creates its own initial state every time it runs.

The *Greedy Fill* algorithm uses the same information as *SA All* and is shown in Algorithm 6. It starts with a clean initial state, by first unassigning all VMs from their PMs, then sorting the VMs in ascending order based on the sum of their average network rate. *Greedy Fill* assigns each VM to a PM, at each step computing the maximum CPU or network link utilization for each possible VM to PM assignment, then picking the PM that minimizes the maximum utilization. This method represents an interesting class of algorithms and shows whether a heuristic can do better if it rips up the existing VM placement and starts over.

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

As I will show later, the results show that *Greedy Fill* performs surprisingly well. If it becomes cheap to move VMs [48, 78, 89], this algorithm could become very attractive.

Chapter 5

Evaluation

The experiments in this section compare multiple workloads, at multiple scales, using the VDC optimization algorithms presented in §4.

Methodology. For each workload and scale, a set of *Random* VM placements is generated. Experiments with 120 VMs on 80 PMs have 25 initial placements, and experiments with 20 VMs on 40 PMs have 50 due to the decreased setup and teardown time needed for this smaller scale. Each combination of workload, scale, and placement runs on the DNRC for 3 minutes [1] or until the workload terminates. During each run, *Virtue* records resource utilization and performance metrics. After all placements for a particular workload and scale have been run, their data is fed into each optimization algorithm, creating new sets of VM placements, which are then run on the DNRC to determine the relative performance differences.

Setup. Figure 5.1 shows the subset of the DNRC used in these experiments. 80-PM experiments use four racks of 20 PMs per rack; 40-PM experiments use 10 PMs per rack. 512MB of RAM is statically allocated for each VM and not included in the optimization algorithms. 8GB virtual disks, one per VM, containing the operating system and experiment software are stored on an NFS server. The switch connected to the NFS server has a connection directly to each ToR switch. Experiments demanding non-trivial disk use have access to 20GB virtual disks residing on the PM's local hard disk.

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

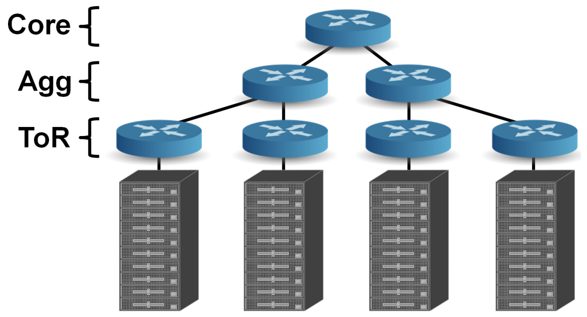


Figure 5.1: Subset of testbed topology used in experiments.

The total network bandwidth oversubscription from PM to PM through the core is configured to 16:1, broken down as 4:1 between the ToR switch and the aggregation (AGG) switch, and a further 4:1 between the AGG and core switch. A 16:1 total network oversubscription was selected for this evaluation which is well below reported total network oversubscription levels in the literature of 240:1, and 5:1 to 20:1 from ToR to AGG [52]. Edge network links are limited to 100Mb/s to better match the ratio between same-server and different-server VM-to-VM bandwidth (the machines in this evaluation are a few years old).

Actual VDCs will have different oversubscription levels within their networks, and the degree to which they are oversubscribed has a major impact on both networked workload performance, and the ability of a network-aware VM placement optimizer to improve performance. In a highly oversubscribed network it is critical to place communicating VMs as close as possible, to limit the number of oversubscribed links that their traffic must traverse. Given an initial network-naive VM placement, a network-aware optimizer should have plenty of opportunity to improve the performance of workloads in an oversubscribed network. As the degree of oversubscription decreases, core network locality becomes less of an issue, limiting the maximum possible performance improvement that VM placement algorithms can achieve. To test this, experiments are run with other link rates and oversubscription levels.

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

I first evaluate the overhead of gathering the traffic matrix, followed by evaluations of a Web workload, then a Hadoop workload, then the two in combination.

5.1 Infrastructure

The VM-VM traffic matrix is measured by periodically polling flow counters in OVS, one per IP source-destination pair. Operators of a real system may want to measure the traffic matrix frequently, to adapt quickly to changing workloads. Measuring traffic matrices is widely thought to be expensive, but not in this environment. OpenFlow enables querying for many flow counters at once, and each flow counter requires approximately 88 bytes of application-level network traffic between the switch and OpenFlow controller.

What percentage of the network is used for polling and creating traffic matrix? *Less than 1% of the bandwidth on edge network links for an expected configuration in today's VDCs.* Table 5.1 shows network bandwidth consumed by polling traffic counters, as a fraction of a 1Gb/s link, for varying numbers of flow entries and polling intervals. The number of flow entries in an OVS instance depends on the number of VMs per PM and the number of other communicating VMs. For example, a VDC with 100 VMs per PM, where each VM communicates bidirectionally with 50 other VMs, needs 10,000 flow entries. *Polling all counters every second uses less than 1% of the link capacity.* Future VDCs with more VMs per PM will have faster network links, further reducing the fraction of network bandwidth consumed by polling flow counters.

5.2 Web Workload

To emulate the workload of a multi-tier website, I created a client + 2-tier Web workload built using Virtue's synthetic workload API. The workload has three groups of VMs shown in Figure 5.2: (1) clients that continuously send requests to Web tier VMs, 20 in parallel; (2) Web VMs that listen for incoming requests, make 10 sequential Memcached-like requests

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

Table 5.1: Percentages of a 1Gb/s link used to poll counter, for different numbers of flows and polling intervals.

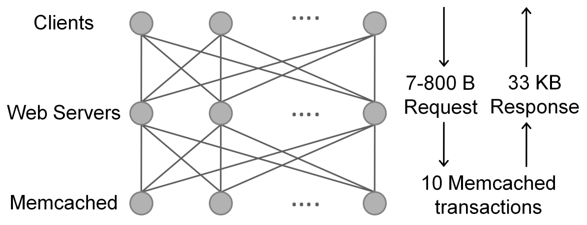


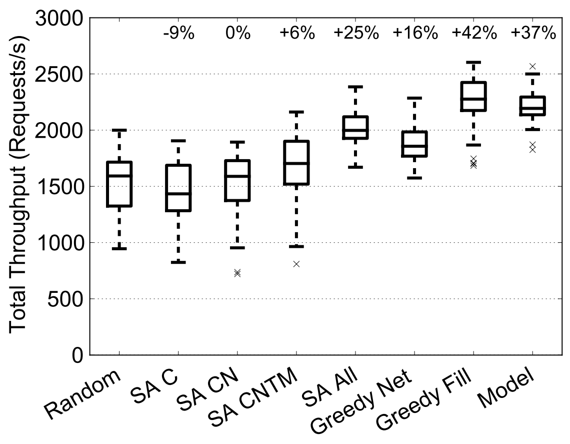
Figure 5.2: Data flow in the Web workload.

to random VMs in the Memcached tier, then return a response to the client; and (3) the Memcached tier, which receives requests from the Web tier. The client sends a 700-800B request to the Web tier and gets a 33.66KB response from the Web tier. These numbers correspond to the top websites' average request/response size for an HTML page [37]. The Web tier's Memcached requests and response sizes are the same as those measured on Facebook's Memcached infrastructure [41]. This workload has been measured to perform optimally when the ratio of client:Web:Memcached VMs is 1:1:1.333.

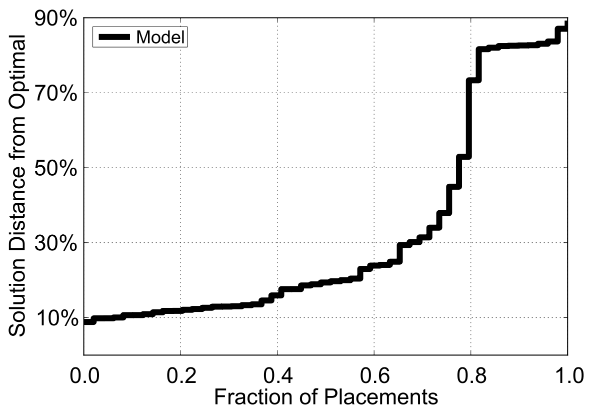
5.2.1 Web Workload, 20 VMs

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

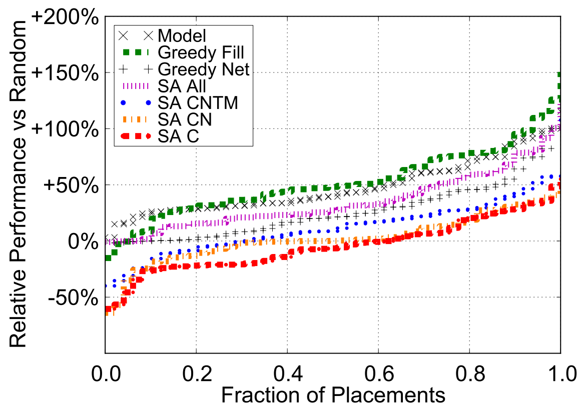
Does the Web workload's performance vary across *Random* placements? *Yes, by over 2× between the worst and best performing placements.* Figure 5.3a plots the performance distribution of each algorithm, along with the set of initial *Random* placements. This workload shows significant variation in performance for the *Random* placements because



(a) Box plots of measured throughputs for each algorithm



(b) CDF of Model solution distance from optimal.



(c) Relative CDF.

Figure 5.3: Web 20 VMs 40 PMs.

it is oblivious to CPU and network usage. In fact, the best *Random* placement yields twice the throughput of the worst, despite using the same number of VMs.

Does a network-aware algorithm accelerate a single Web workload? *Yes. Network-aware algorithms improve performance by up to 42%; network-oblivious optimization (*SA C*), reduces median performance by 9%.* Comparing *Random* to the other optimization algorithms in Figure 5.3a, there is a steady upward progression, where more network knowledge leads to higher total throughput. Surprisingly, simulated annealing with CPU data (*SA C*), which is similar to today's network-oblivious optimization products, performs slightly worse than *Random*. *SA C* evenly spreads CPU load across machines, precluding it from creating network-beneficial placements containing multiple VMs on a single PM, which *Random* can create. *SA CN* adds NIC traffic counters and performs similarly to *Random*; its improvement relative to *SA C* suggests that for this workload, where plenty of cores are available, spreading the network load is more important than spreading the CPU load. Adding the traffic matrix in *SA CNTM* yields another small improvement, as highly-communicating VMs can then be co-scheduled to reduce network load. *SA All* adds topology knowledge to further improve performance by helping the optimizer differentiate between placements that look great for each edge NIC but stress the core network links. *Greedy Net*, which has no CPU info, performs well, indicating the network dependence of this workload. *Greedy Fill*, which has CPU data, performs the best, even outperforming *Model*.

Why does *Greedy Fill* outperform *Model* for this workload? *Model runs for a maximum of 3 hours, bringing only 6% of placements within 10% of the believed optimal solution (gap).* Figure 5.3b plots a CDF of the optimality of the placements created by *Model*, where optimality is a numerical value reported by CPLEX representing its believed distance from the optimal solution. Few solutions are actually solved to within the configured 10% gap; however, the majority are solved to less than 25%. This tells us two things: *Model* would take much longer than the 3 hour cut off to solve all mappings to within the 10% gap; more importantly, a large fraction of placements (those with a gap significantly higher than 10%) are not likely to yield peak performance.

Does network-aware optimization ever harm performance? *Of the 200 placements generated by fully network-aware algorithms, only 3 reduced performance.* Figure 5.3c examines whether an algorithm always improves performance, or whether it occasionally hurts performance. On this and subsequent *relative* graphs, each point on each CDF corresponds to the relative performance difference between an algorithm's chosen placement and the *Random* placement from which it received measurement data: for Web, requests/s after/before; for Hadoop, runtime before/after. In this Web workload, algorithms with full topology knowledge do not harm performance, with the exception of 3 placements for *Greedy Fill*. For these algorithms, performance is good, with median improvements from 1.2× to 1.5×, and a maximum improvement of up to 2.5×. However, algorithms with no

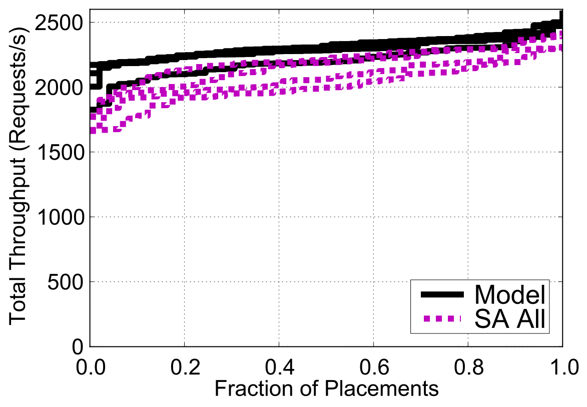
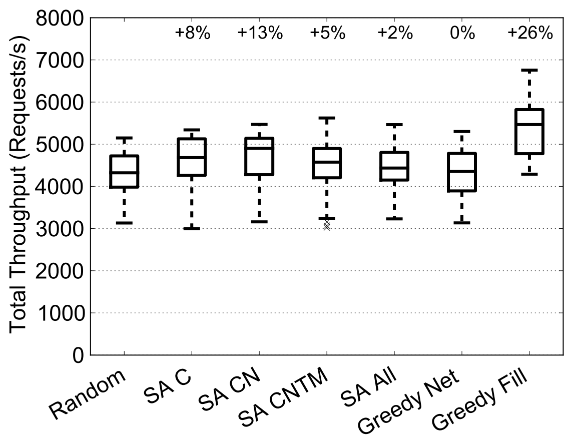


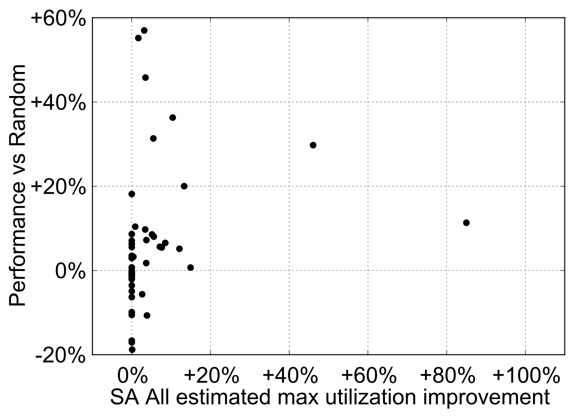
Figure 5.4: Iterative algorithm improvement.

topology knowledge cannot prevent oversubscribing the core. Within this group, *SA CNTM* has the most information and does the best, improving performance in around 70% of the cases and harming performance for the remaining.

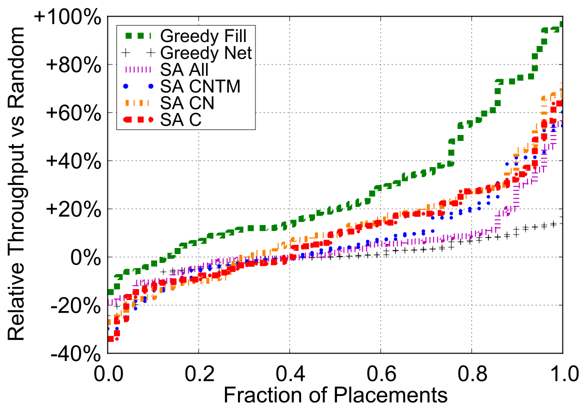
Do multiple optimization cycles improve performance? *Yes, but with diminishing returns.* The optimization algorithms operate without specific workload knowledge, so they must make placement decisions by measuring the infrastructure. However, resource bottlenecks in the infrastructure hide true workload resource demands. If an algorithm knew exactly how a workload would perform absent resource bottlenecks, it could make better placement decisions. To understand how the algorithms could perform with better demand knowledge, I ran four iterations of both *Model* and *SA All*, starting with measurements from a *Random* placement. Each successive iteration should remove bottlenecks, bringing traffic matrix measurements closer to the workload's actual demand and bringing performance closer to its peak. Figure 5.4 shows a CDF comparing the performance of each iteration. *SA All* starts at the bottom line and performs better at each iteration. Between the first and fourth (bottom and top) iterations, there is a 10% median performance improvement. *Model* shows similar behavior between its first and second iterations, finding a 5% median improvement, but the subsequent two show little difference, indicating that *Model* is unable to further improve performance due to its limited view of the world.



(a) Box plots of measured throughputs for each algorithm.



(b) *SA All* optimization metric improvement vs measured throughput improvement.



(c) Relative CDF.

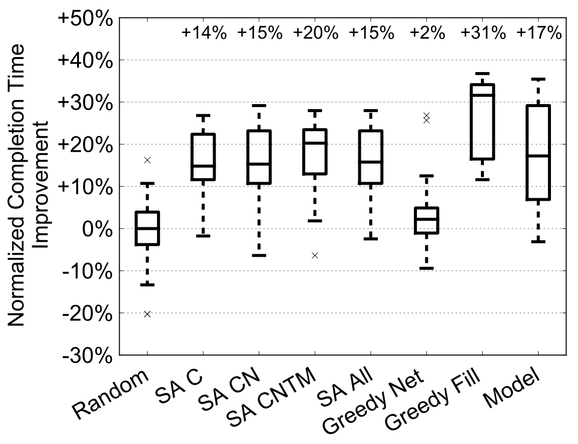
Figure 5.5: Web 120 VMs 80 PMs.

5.2.2 Web Workload, 120 VMs

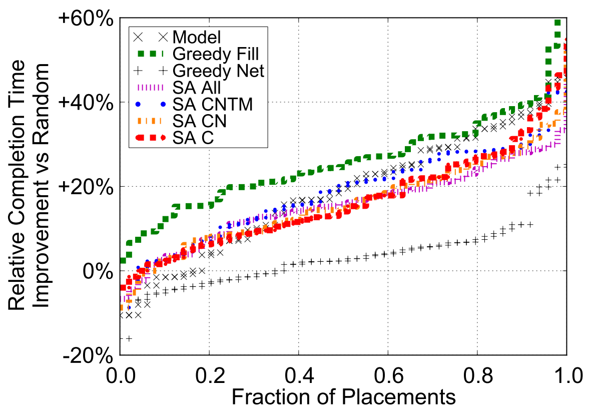
Next the Web workload is scaled up in both VM count (120 VMs: 36 clients, 36 Web, and 48 Memcached) and server count (80). Because there are now 50% more VMs than PMs, it should be more difficult for the Minimax optimization metric to find improvements since the CPU and network resources will be highly utilized. As a reminder, results from *Model* are absent due to its runtime at this scale. Figure 5.5a shows a very different algorithm performance trend compared to the smaller 20 VM experiment.

Does network-aware optimization improve performance? *The results are mixed.* Greedy Fill *improves median performance by 26%, however* SA All *only improves performance by 2%. Greedy Fill* performs the best for this workload. It begins by assigning each of the 48 Memcached VMs to a PM, because these VMs have the least network traffic. This allocation consumes 1 slot on each of the first 48 PMs; then it fills the remaining 32 slots with clients, which have less network traffic than the Web tier, since they do not communicate with the Memcached VMs. Then the allocation spills over and assigns 4 clients onto the first 4 PMs. Next, *Greedy Fill* assigns the Web VMs onto the the right half of the cluster, sharing PMs with the client VMs because amongst each other they have the highest network traffic leading to the lowest maximum link utilization. This layout produces the best placement, because the bulk of the all-to-all communication between the client and Web tiers only traverses one aggregation layer and not the core; the smaller communication from Web to Memcached still traverses the core, but there is less of this traffic. Of note is the large performance range even on *Greedy Fill*; this is an effect of the optimizer receiving imperfect demand information hidden by initial placements.

Why is *Greedy Fill* the only algorithm to make a significant performance improvement? SA *and* Greedy Net *are unable to reach states that improve performance.* With the exception of *Greedy Fill*, the algorithms able to make a measurable improvement are actually those that lack topology information, namely *SA C* and *SA CN*. To understand this unexpected result, I examine the algorithm that was expected to perform the best with the most information – *SA All* – by plotting its view of how much it improved each placement versus the actual measured throughput performance improvement in Figure 5.5b. The majority of placements are at x=0%, meaning that *SA All* was unable to decrease the maximum CPU/link utilization. Because many links must run hot in this more highly-loaded workload, *SA All* is limited by the Minimax optimization metric and thus is unable to consider states that may lead to higher global performance. *SA C* and *SA CN* load-balance the CPUs, which does lead to an improvement for just over 60% of the placements, as shown in Figure 5.5c. *Greedy Fill*'s ability to improve upon *Random* by a median value of 26% vs *SA All*'s 2%, and harm fewer placements (12% vs 42%), indicates there is room for future improvement in both our optimization metric and heuristics.



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



(b) Relative CDF.

Figure 5.6: Hadoop 20 VMs 40 PMs.

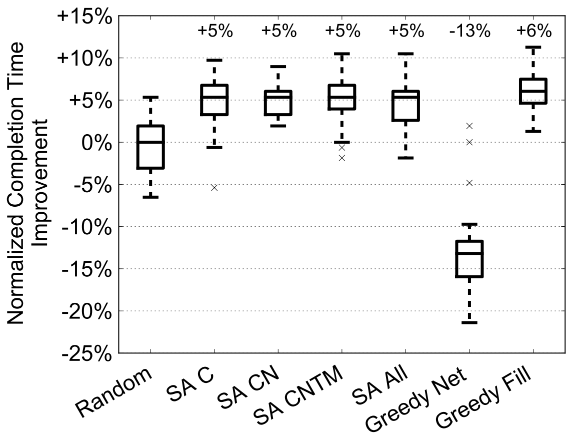
5.3 MapReduce Workload

MapReduce is one of the most frequently run applications in data centers, typified by Amazon's Elastic MapReduce service based on its own version of Hadoop. It is also very different from a Web workload, with MapReduce performance dominated by the CPU, and network usage in two short bursts of all-to-all communication. This experiment uses Hadoop version 0.20 and its Terasort benchmark to represent MapReduce workloads. Terasort uses a set of VMs to sort a data set of configurable size. In each experiment, a data set size was selected so that the benchmark would complete in approximately three minutes – the time of the measurement phase in the optimization cycle — including setup and teardown time. Because MapReduce is CPU-limited, the expectation is to see only a small runtime improvement when moving to network-aware algorithms.

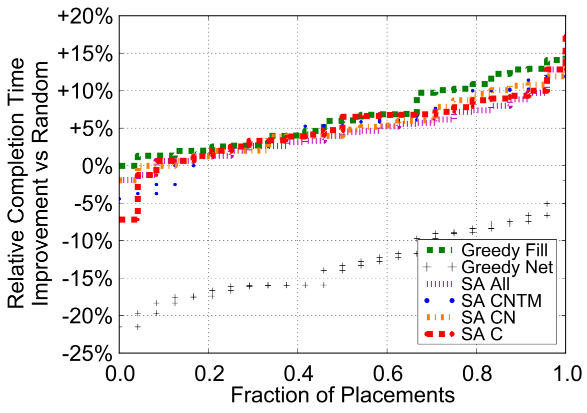
5.3.1 Hadoop, 20 VMs

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

Does a network-aware algorithm help Hadoop finish faster? *Yes, but at most 17% faster than if the algorithm did not use network knowledge.* Figure 5.6a shows a box plot



(a) Completion time normalized to Random's median of 158 seconds.



(b) Relative CDF.

Figure 5.7: Hadoop 120 VMs 80 PMs.

of the time it took the Terasort to complete, for each optimization algorithm, compared to *Random*'s median completion time (139.5 seconds). Because Hadoop is CPU-limited, as expected there is little improvement when moving from *SA C* to network-aware algorithms. *Greedy Net* performs no better than *Random* because it measures low network utilization (and does not have access to CPU usage), so it naively places several VMs on the same PM. *Greedy Fill* performs the best because the optimal placement is one VM per PM across the first two racks, which comes as a consequence of *Greedy Fill*'s strategy to break ties left-to-right when choosing where to place a VM among otherwise identical PMs. *Model* has trouble finding near-optimal solutions in the time allowed, and so its performance varies widely. Figure 5.6b shows the good news: all heuristics other than *Greedy Net* improve more than 90% of the placements.

5.3.2 Hadoop, 120 VMs

Next the Hadoop workload is scaled to 120 VMs using 80 PMs, and the sorted data set size is increased to 6GB.

Does a network-aware algorithm accelerate the Hadoop workload at this scale? *Not any more than algorithms that are only CPU-aware.* Figure 5.7a and 5.7b show the results. A trend similar to the smaller workload size can be seen, where all algorithms other than *Greedy Net* improve relative to *Random* for the majority of placements. At this scale, 1 or 2 VMs must be assigned to each PM for CPU load balancing, so there is little room for optimization by improving network locality. Unsurprisingly, by ignoring the CPU, *Greedy Net* clusters VMs to as few PMs as possible and has the worst workload performance.

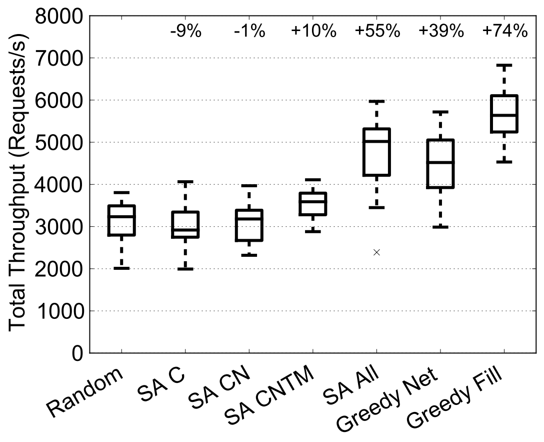
5.4 Combined Web + MapReduce, 120 VMs

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

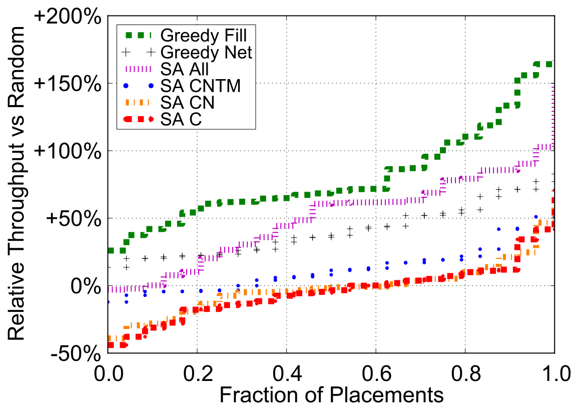
Does a network-aware algorithm accelerate total Web tenant throughput? Without *network knowledge,* SA C *reduced performance by 9% compared to* Random*, whereas optimizing with full network knowledge improved performance by 74%, demonstrating the importance of network-aware algorithms.* Figure 5.8a plots the combined throughput of the Web workloads for different algorithms. The fourth box from the left shows that having NIC and traffic matrix knowledge (*SA CNTM*) makes the workload 10% faster. If the algorithms have full network knowledge, the workload runs 39% faster with *Greedy Net*, 55% faster with *SA All*, and 74% faster with *Greedy Fill*. These three algorithms consistently improve average performance; Figure 5.8b shows that they improve upon *Random* in all but 3 cases (96% of the time).

How do individual tenants fare? *In total, 85% of tenant-placement pairs improve with* SA All. Figure 5.8c plots the per-tenant results using *SA All*. Big workloads are almost all better off with *SA All* than *Random* (with 10% worse off). Small workloads are affected much more (up and down) with 81% doing better, and 19% worse.

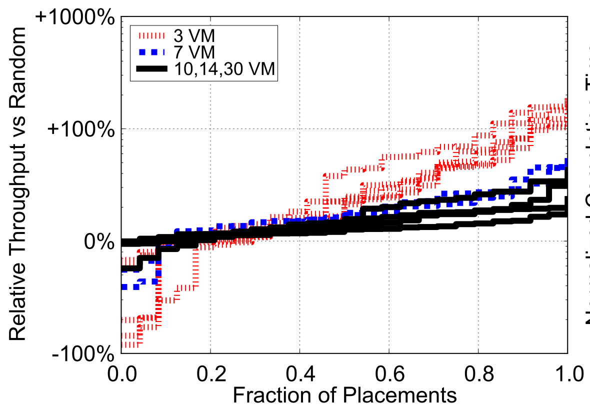
Does a network-aware algorithm help the Hadoop tenant complete faster? *Yes, but marginally, because our Hadoop workload is CPU limited.* Figures 5.8d and 5.8e show the runtime of the 30 Hadoop VMs. *SA C* and *Greedy Fill* make Hadoop finish 9% sooner, because it is sensitive to being co-scheduled with other Hadoop processes. *SA C* minimizes the maximum CPU, so a single Hadoop VM is assigned to each PM, and *Greedy Fill* places



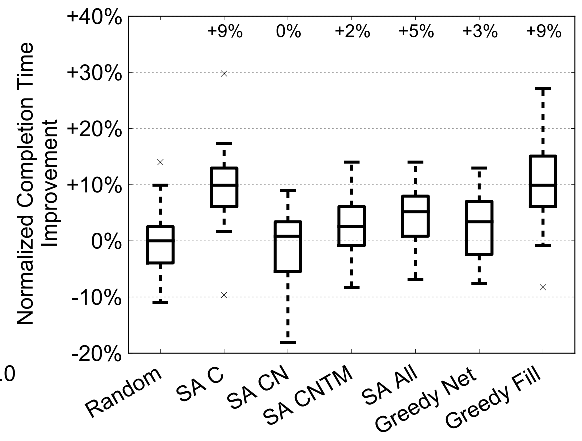
(a) Web Tenant Throughput Sum.



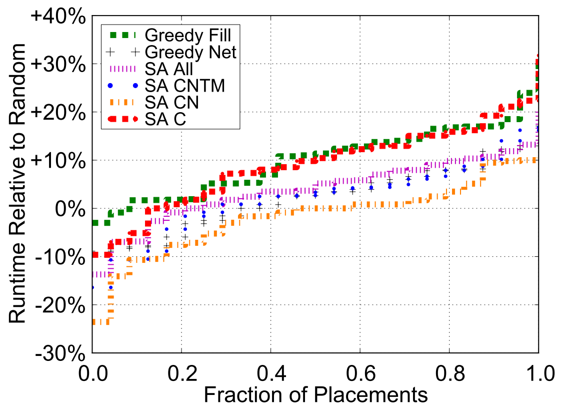
(b) Web Tenant Throughput Sum Relative CDF.



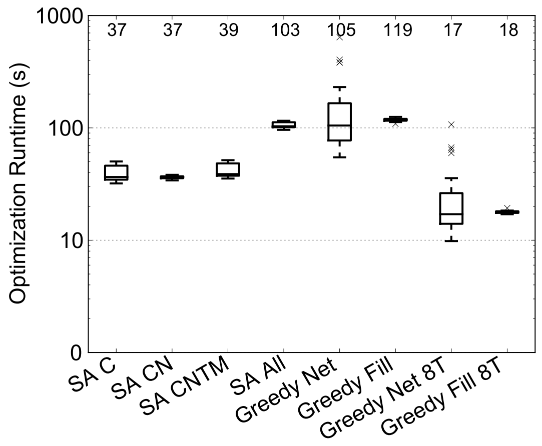
(c) *SA All* relative client throughput.



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



(e) Hadoop Tenant Runtime Relative CDF.



(f) Algorithm Runtime.

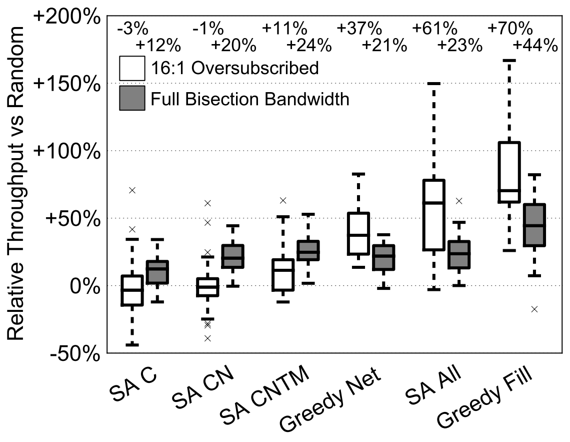
Figure 5.8: Multi-Tenant Workload 120 VMs 80 PMs.

a Hadoop VM on each of the first 30 PMs because their average network use is below any of the Web workload VMs.

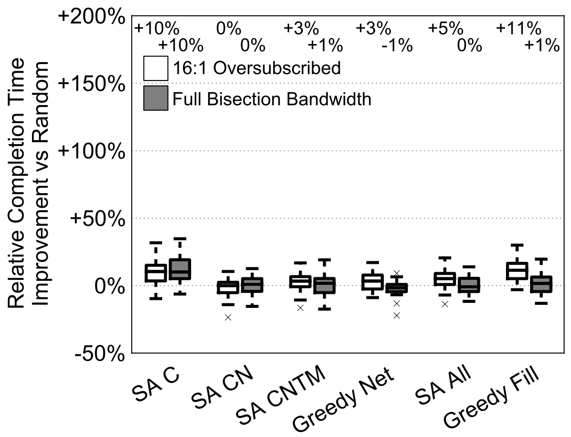
How long do the optimization algorithms take to run? *One thread takes 37-119s, but eight threads only needs 17-18s.* Figure 5.8f compares the runtimes of the (lightly optimized) algorithms. *SA C*, *SA CN*, and *SA CNTM* take about the same time to run (37-39s). *SA All*, *Greedy Net*, and *Greedy Fill* take longer when finding the maximally utilized link, increasing median runtimes to 103-119s. *Greedy Net*'s runtime varies, because it iteratively moves VMs and their traffic, and may evaluate and move the same VM multiple times. Finally, *Greedy Fill*'s runtime is nearly constant because for each VM it examines all possible PM placements exactly once.

Greedy Net and *Greedy Fill* are easy to multi-thread by examining all possible PM locations for a VM in parallel. Both algorithms were run with 8 threads, labeled in Figure 5.8f as *Greedy Net 8T* and *Greedy Fill 8T*. The median runtime of *Greedy Net* decreased from 105s to 17s (617% faster) and *Greedy Fill* from 119s to 18s (661% faster). Due to *SA*'s design, there is no straightforward way to multithread it. Instead of multithreading the *SA* algorithm itself, multiple instances of *SA* could be run in parallel, each with a unique initial random seed. This approach can use additional processing cores to potentially improve the quality of the optimization, but cannot reduce the absolute runtime of the algorithm.

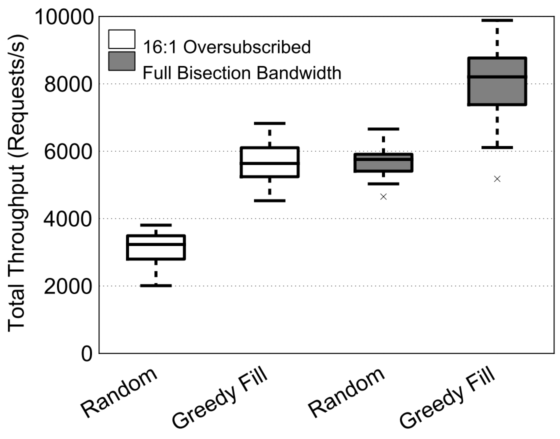
What if the core network is FBB? *The optimization algorithms still improve performance of Web workloads by 20-44%, while Hadoop is unaffected.* Figure 5.9 shows how much faster the Web and Hadoop workloads run when when the network is no longer oversubscribed. Each plot shows two sets of results: 16:1 oversubscription and no oversubscription. Figure 5.9a shows the surprising result that even when the network fabric is not a bottleneck, Web workloads run faster when optimizing with network knowledge; median performance improves by 20% to 44%. The *SA* algorithms all improve by about the same amount because when the network is no longer a bottleneck, they all focus on minimizing the maximum utilization of PM NICs. *Greedy Fill* does much better; its technique of unassigning all VMs then assigning them one by one enables it to reach final states that *SA* does not. As expected, Figure 5.9b shows that CPU-limited Hadoop is unaffected by network oversubscription.



(a) Web.



(b) Hadoop.



(c) Web.

Figure 5.9: How much faster the Web and Hadoop workloads run when the network is not oversubscribed.

As network links get faster, will the optimization algorithms still improve performance? *Yes, performance improves between 33-129% for each tested link speed/network oversubscription combination.* All the results presented so far were for a network with 100Mb/s links at the edge. If the links are faster, keeping everything else constant, one should expect *Random* performance to get better, because the network is less of a bottleneck. At the same time, a reduced network bottleneck leaves less opportunity for the algorithms to improve performance. Indeed, Table 5.2 shows this to be the case. Consider the first column for which the network link increases from 100Mb/s to 750Mb/s while keeping

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

Table 5.2: Median relative performance improvement of *Greedy Fill* vs *Random* by edge link rate and network core oversubscription ratio.

the core oversubscription constant at 16:1. The best algorithm in this evaluation, *Greedy Fill*, still improves upon *Random* but the improvement drops from 129% to 37%[2]. In every combination of link-speed and oversubscription *Greedy Fill* improves performance by at least 33%, suggesting that a network-aware algorithm can improve performance in a wide variety of VDC settings.

If a VDC operator needs to improve networked workload performance, should they improve their network, improve their VM placement, or both? *The results presented here suggest that depending on the workload, all three can lead to significant performance gains.* Figure 5.9c highlights the performance implications of these choices for the multi-tenant web workload, using the 16:1 oversubscribed and FBB network configurations, optimized using *Greedy Fill*. Even on a 16:1 oversubscribed network, *optimization improves performance such that it nearly matches the unoptimized workload placement running on a FBB network.* If, however, a VDC operator chose to upgrade to FBB, the same techniques can further improve the workload's performance.

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

[3]The testbed's software did not yet support this combination of edge link speed and core network bandwidth.

Chapter 6

Conclusion

The introduction posed two key questions that this thesis sought to answer. These questions were:

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

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

Answering these questions convincingly required running experiments on real hardware. No publicly available testbed provided the needed low-level control over the networking equipment, therefore my approach was to build a custom data center cluster for running experiments. This meant setting up, running, and operating a multi-rack testbed with custom control software, combined with thousands of hours of machine time to run workloads on the testbed and the optimization algorithms in the cloud.

Each performance number reported came from direct measurement; none of them were from simulation or estimation. The payoff is more-realistic answers to the original motivating questions. Using *Virtue* I found that an algorithm approximating today's network-oblivious commercial solutions actually *harmed* the median performance of multi-tenant network-heavy workloads by 9%. In contrast, the best network-aware placement algorithm was able to *increase* performance of the same workload by 74%, and 129% on the latest hardware. Other placement heuristics were also evaluated that had differing levels of network-awareness, and correspondingly achieved workload performance improvements and algorithm runtimes that varied between these ranges.

Perhaps the biggest takeaway from our results is that VDC operators could save upgrade costs by using network-aware placement algorithms rather than upgrading to an expensive FBB network. Even if the operator did upgrade to FBB, the same techniques are able to further improve the workload's performance.

Where does this research go from here? I believe that the use of network-aware VM placement is inevitable using these or similar techniques. Public clouds are still relatively new and as time goes on more cloud providers will enter the market increasing competition. A key differentiator amongst the different providers is performance. Cloud providers all use the same core hardware providers, thus the base hardware performance with random-style VM placement will be largely similar. Improving upon this level of base performance will require VM placement optimization using as many resources as possible, of which the network is absolutely critical, as demonstrated in this thesis.

My hope is that the techniques and results presented here will spur additional research in this area, and hasten commercial deployments. As the cloud continues to grow in both size and scope, it will affect more and more of our lives, so improving it can have an enormous impact.

Appendix A

Controller Evaluation Settings

The following list shows the OpenFlow controllers and their individual settings as recommended by the author(s) used in this evaluation.

- **Beacon**. Version 1.0.2, Oracle JVM 1.6 Update 37, using the configurationSwitch launch config, and the following JVM flags: -XX:+AggressiveOpts -Xmx16G -XX:InlineSmallCode=16384 -XX:MaxInlineSize=16384 -XX:FreqInlineSize=16384 -XX:-UseParallelOldGC -XX:-UseParallelGC -XX:+UseNUMA.

- **Floodlight**. Version 0.90, Oracle JVM 1.6 Update 37, using the following floodlight.modules:

  floodlight.modules =
  net.floodlightcontroller.learningswitch.LearningSwitch,
  net.floodlightcontroller.counter.NullCounterStore,
  net.floodlightcontroller.perfmon.NullPktInProcessingTime.

  Launched using a modified floodlight.sh which uses the floodlight.properties file containing the list of modules, and modifies the maximum heap size to 16G (-Xmx16g). All other settings remain unmodified.

- **Maestro**. Version 0.2.1, Oracle JVM 1.6 Update 37. Command line:

  java -cp build/ sys.Main conf/openflow.conf conf/learningswitch.dag daemon

- **NOX**. Verity branch, commit f75d2f31d934185fe0ce7c2482d0f8ae950b34f9 (9/6/12). Configured with –enable-ndebug, launched using only the switch module.
- **POX**. Betta branch, commit 52712bcaf0c230ba6d3915f56ab32b281a0d8de3 (12/2/12). Launched using PyPy 1.9.0 and Python 2.7.2, running forwarding.l2_learning.
- **Ryu**. Version 1.5, Python 2.7.2. Launched running a modified ryu/app/simple_switch.py that does not emit log messages per packet in.

Cbench was used as the test harness for controller evaluations, the Git commit used was f848965336c1275c7847149c0089c16645ad8d32. Cbench's throughput mode tests were launched either with one or two instances of Cbench, based on the number of threads the controller under test was running. The following command launched a single Cbench throughput test:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 64
```

The following two commands launched two Cbench instances for throughput tests:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 32
taskset -c 8 cbench -c localhost -p 6633 -m 10000 -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 32 -o 33
```

And latency mode tests were launched with the following command:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 -l 13 -w 3 -M 100000 -i 50 -I 5
```

Appendix B

EDF

An example Experiment Description File.

```
{
    "parameters": {
        "CPU_COUNT": "2",
        "CPU_SPEED": "2.8E9",
        "GREEDY_FILL_UTILIZATION": "0.860927",
        "INTRA_PM_LINK_SPEED": "200000000",
        "LINK_SPEED": "1000000000",
        "OPTIMIZER": "greedyFill",
        "OPTIMIZER_DATE": "1358473028844",
        "OPTIMIZER_INPUT": "/data/input/v20on403tier/greedyfill/edf-8392.json",
        "OPTIMIZER_RUNTIME": "4494",
        "PROPERTIES": "",
        "experimentDuration": "180",
        "generator": "org.virtuesource.edf.generators.WebApp3TierGenerator",
        "mappingSeed": "-2935031425998787072",
        "persistent": "true"
    },
    "pms": [
        {
            "id": "host0001",
            "parameters": {},
            "vms": {
                "vm0001": {
                    "id": "vm0001",
                    "parameters": {
                        "APP": "org.virtuesource.shared.app.impl.ClientApplication",
                        "MAC": "00:16:3E:01:00:01",
                        "destinationAddresses": "vm0002",
                        "parallelRequests": "20",
                        "reportType": "client",
                        "writeMax": "800",
                        "writeMin": "700"
                    }
                },
                "vm0002": {
                    "id": "vm0002",
                    "parameters": {
                        "APP": "org.virtuesource.shared.app.impl.
                            TierNodeApplication",
                        "MAC": "00:16:3E:01:00:02",
                        "cpuPerRequestMax": "0",
                        "cpuPerRequestMin": "0",
                        "facebookMemcacheDistRequest": "true",
                        "listenPort": "30000",
                        "nextTierAddresses": "vm0003",
                        "nextTierConnections": "10",
                        "nextTierPort": "30001",
                        "nextTierWriteMax": "100",
                        "nextTierWriteMin": "100",
                        "reportType": "tier1",
                        "responseMax": "34467",
                        "responseMin": "34467"
                    }
                }
            }
        },
        {
            "id": "host0002",
            "parameters": {},
            "vms": {
                "vm0003": {
                    "id": "vm0003",
                    "parameters": {
                        "APP": "org.virtuesource.shared.app.impl.
                            TierNodeApplication",
                        "MAC": "00:16:3E:01:00:03",
                        "cpuPerRequestMax": "0",
                        "cpuPerRequestMin": "0",
                        "facebookMemcacheDistResponse": "true",
                        "listenPort": "30001",
                        "nextTierAddresses": "",
                        "nextTierConnections": "0",
                        "nextTierPort": "0",
                        "nextTierWriteMax": "0",
                        "nextTierWriteMin": "0",
                        "reportType": "tier2",
                        "responseMax": "16384",
                        "responseMin": "16384"
                    }
                }
            }
        }
    ],
    "switches": [
        {
            "id": "00:00:00:00:00:00:00:01",
            "parameters": {},
            "software": false
        },
        {
            "id": "01:00:00:00:00:00:00:01",
            "parameters": {},
            "software": true
        },
        {
            "id": "01:00:00:00:00:00:00:02",
            "parameters": {},
            "software": true
        }
    ],
    "links": [
        {
            "destinationId": "01:00:00:00:00:00:00:01",
            "destinationPort": 1,
            "id": "00:00:00:00:00:00:00:01:1->01:00:00:00:00:00:00:01:1"
                ,
            "parameters": {
                "LINK_SPEED": "100000000"
            },
            "sourceId": "00:00:00:00:00:00:00:01",
            "sourcePort": 1
        },
        {
            "destinationId": "00:00:00:00:00:00:00:01",
            "destinationPort": 1,
            "id": "01:00:00:00:00:00:00:01:1->00:00:00:00:00:00:00:01:1"
                ,
            "parameters": {
                "LINK_SPEED": "100000000"
            },
            "sourceId": "01:00:00:00:00:00:00:01",
            "sourcePort": 1
        },
        {
            ”destinationId”: ”01:00:00:00:00:00:00:02”,
            ”destinationPort”: 1,
            ”id”: ”00:00:00:00:00:00:00:01:1->01:00:00:00:00:00:00:02:1”,
            ”parameters”: {
                ”LINK_SPEED”: ”100000000”
            },
            ”sourceId”: ”00:00:00:00:00:00:00:01”,
            ”sourcePort”: 2
        },
        {
            ”destinationId”: ”00:00:00:00:00:00:00:01”,
            ”destinationPort”: 2,
            ”id”: ”01:00:00:00:00:00:00:02:1->00:00:00:00:00:00:00:01:2”,
            ”parameters”: {
                ”LINK_SPEED”: ”100000000”
            },
            ”sourceId”: ”01:00:00:00:00:00:00:02”,
            ”sourcePort”: 1
        },
        {
            ”destinationId”: ”01:00:00:00:00:00:00:01”,
            ”destinationPort”: 0,
            ”id”: ”vm0001:0->01:00:00:00:00:00:00:01:0”,
            ”parameters”: {},
            ”sourceId”: ”vm0001”,
            ”sourcePort”: 0
        },
        {
            ”destinationId”: ”vm0001”,
            ”destinationPort”: 0,
            ”id”: ”01:00:00:00:00:00:00:01:0->vm0001:0”,
            ”parameters”: {},
            ”sourceId”: ”01:00:00:00:00:00:00:01”,
            ”sourcePort”: 0
        },
        {
            ”destinationId”: ”01:00:00:00:00:00:00:01”,
            ”destinationPort”: 0,
            ”id”: ”vm0002:0->01:00:00:00:00:00:00:01:0”,
            ”parameters”: {},
            ”sourceId”: ”vm0002”,
            ”sourcePort”: 0
        },
        {
                "destinationId": "vm0002",
                "destinationPort": 0,
                "id": "01:00:00:00:00:00:00:01:0->vm0002:0",
                "parameters": {},
                "sourceId": "01:00:00:00:00:00:00:01",
                "sourcePort": 0
            },
            {
                "destinationId": "01:00:00:00:00:00:00:02",
                "destinationPort": 0,
                "id": "vm0003:0->01:00:00:00:00:00:00:02:0",
                "parameters": {},
                "sourceId": "vm0003",
                "sourcePort": 0
            },
            {
                "destinationId": "vm0003",
                "destinationPort": 0,
                "id": "01:00:00:00:00:00:00:02:0->vm0003:0",
                "parameters": {},
                "sourceId": "01:00:00:00:00:00:00:02",
                "sourcePort": 0
            }
        ],
        "routes": [
            {
                "destinationId": "vm0002",
                "linkIds": [
                    "vm0001:0->01:00:00:00:00:00:00:01:0",
                    "01:00:00:00:00:00:00:01:0->vm0002:0"
                ],
                "sourceId": "vm0001"
            },
            {
                "destinationId": "vm0001",
                "linkIds": [
                    "vm0002:0->01:00:00:00:00:00:00:01:0",
                    "01:00:00:00:00:00:00:01:0->vm0001:0"
                ],
                "sourceId": "vm0002"
            },
            {
                "destinationId": "vm0003",
                "linkIds": [
                    "vm0002:0->01:00:00:00:00:00:00:01:0",
                    "01:00:00:00:00:00:00:01:1->00:00:00:00:00:00:00:01:1",
                    "00:00:00:00:00:00:00:01:2->01:00:00:00:00:00:00:02:1",
                    "01:00:00:00:00:00:00:02:0->vm0003:0"
            ],
            "sourceId": "vm0002"
        },
        {
            "destinationId": "vm0002",
            "linkIds": [
                "vm0003:0->01:00:00:00:00:00:02:0",
                "01:00:00:00:00:00:00:02:1->00:00:00:00:00:00:01:2",
                "00:00:00:00:00:00:00:01:1->01:00:00:00:00:00:00:01:1",
                "01:00:00:00:00:00:00:01:0->vm0002:0"
            ],
            "sourceId": "vm0003"
        },
    ]
}
```

Appendix C

Model

Variables.

$X_{a,b} \in \{0,1\}$	VM a is allocated on PM b or not
$f_{a,b,u,v} \in \{0,1\}$	Traffic from VM a to VM b is on link (u,v)

Constants.

VM	Set of all VMs
PM	Set of all PMs
SW	Set of all switches
V	PM $\cup$ SW
E	Set of all links
Cv_i	The CPU usage of VM i
Cp_i	The CPU capacity of PM i
$T_{a,b}$	The traffic between VM a and b
$L_{u,v}$	The link capacity of (u, v)

Optimization. The goal is to minimize y, the maximum CPU or link utilization.

Constraints.

The maximum CPU utilization is $\leq y$

$$\forall j \in \text{PM} : (\sum_{i \in \text{VM}} X_{i,j} Cv_i)/Cp_j \leq y$$

The maximum link utilization is $\leq y$

$\forall(u,v) \in E : (\sum_{a,b \in VM} T_{a,b} f_{a,b,u,v})/L_{u,v} \le y$

Each VM must be allocated to exactly one server.

$\forall i \in \text{VM} : \sum_{j \in \text{PM}} X_{i,j} = 1$

The VMs on a server cannot exceed its CPU capacity.

$\forall j \in \text{PM} : \sum_{i \in \text{VM}} (X_{i,j} C v_i) \le C p_j$

The traffic on a link cannot exceed the link capacity.

$\forall(u,v) \in E : \sum_{a,b \in VM} T_{a,b} f_{a,b,u,v} \le L_{u,v}$

If a flow comes into a switch, it must also go out. $\forall a,b \in \text{VM}, \forall u \in \text{SW} : \sum_{w \in V} f_{a,b,u,w} = \sum_{w \in V} f_{a,b,w,u}$

The flow to and from a VM must exit the source VM on exactly one link and enter the target VM on exactly one link. This equation is linearized using the method in [79].

$\forall a,b \in \text{VM} : \sum_{u,w \in V} f_{a,b,u,w} X_{a,u} = \sum_{u,w \in V} f_{a,b,w,u} X_{b,u} = 1$

Flows to/from a VM only enter/exit from the PM the VM is assigned to.

$\forall a,b \in \text{VM}, \forall u \in \text{PM}, \forall w \in \text{V}\ f_{a,b,u,w} \le X_{a,u}$

$\forall a,b \in \text{VM}, \forall u \in \text{PM}, \forall w \in \text{V}\ f_{a,b,w,u} \le X_{b,u}$

Bibliography

[1] Amazon AWS High Performance Computing. http://aws.amazon.com/hpc-applications/.

[2] Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[4] Apache Cassandra. http://cassandra.apache.org/.

[5] Apache Hadoop. http://hadoop.apache.org/.

[6] Apache Lucene. http://lucene.apache.org/.

[7] Apache Maven. http://maven.apache.org/.

[8] Apache Tomcat. http://tomcat.apache.org/.

[9] Apache ZooKeeper. http://zookeeper.apache.org/.

[10] Big Network Controller. http://www.bigswitch.com/products/SDN-Controller.

[11] Citrix XenServer Workload Balancing, 6.0 User Guide. http://support.citrix.com/article/CTX130429.

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

[13] Eclipse. http://www.eclipse.org/.

[14] Equinox. `http://www.eclipse.org/equinox/`.

[15] Floodlight. `http://floodlight.openflowhub.org/`.

[16] Flowscale. `http://www.openflowhub.org/display/FlowScale/FlowScale+Home`.

[17] IBM Workload Deployer. `http://www-01.ibm.com/software/webservers/workload-deployer/`.

[18] Indigo: OpenFlow for Hardware Switches. `http://www.openflowhub.org/display/Indigo/`.

[19] Jetty WebServer. `http://jetty.codehaus.org/`.

[20] KVM. `http://www.linux-kvm.org/`.

[21] Lanamark Suite. `http://www.lanamark.com/`.

[22] Mono. `http://www.mono-project.com/`.

[23] MUL. `http://sourceforge.net/projects/mul/`.

[24] Novell PlateSpin Recon. `https://www.netiq.com/products/recon/`.

[25] OpenDaylight — A Linux Foundation Collaborative Project. `http://www.opendaylight.org/`.

[26] OpenStack: Open source software for building private and public clouds. `http://www.openstack.org/`.

[27] POX. `http://www.noxrepo.org/pox/about-pox/`.

[28] ProgrammableFlow Controller. `http://www.necam.com/SDN/doc.cfm?t=PFlowController`.

[29] Rackspace Cloud. `http://www.rackspace.com/cloud/`.

[30] The Rise of Soft Switching Part II: Soft Switching is Awesome. `http://networkheresy.wordpress.com/2011/06/25/the-rise-of-soft-switching-part-ii-soft-switching-is-awesome-tm/`.

[31] Ryu. `http://osrg.github.com/ryu/`.

[32] Spring Framework. `http://www.springsource.org/`.

[33] StarCluster. `http://star.mit.edu/cluster/`.

[34] Sun Grid Engine. `http://gridscheduler.sourceforge.net/`.

[35] Trema. `http://trema.github.com/trema/`.

[36] VMware. `http://www.vmware.com/`.

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

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

[39] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[40] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[41] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012.

[42] G. Attiya and Y. Hamam. Optimal allocation of tasks onto networked heterogeneous computers using minimax. In *International Network Optimization Conference*, 2003.

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

[44] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[45] Muli Ben-Yehuda, David Breitgand, Michael Factor, Hillel Kolodner, Valentin Kravtsov, and Dan Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, pages 179–188, New York, NY, 2009.

[46] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[47] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. Sane: A protection architecture for enterprise networks. In *USENIX Security Symposium*, 2006.

[48] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[49] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[50] Camil Demetrescu and Giuseppe F Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, 51(6):968–992, 2004.

[51] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2010.

[52] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[53] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[54] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 2012.

[55] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: Challenges and techniques. In *HotCloud*, 2011.

[56] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, 2010.

[57] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[58] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[59] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[60] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. *Distributed Computing*, pages 350–364, 2008.

[61] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX ATC*, 2008.

[62] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[63] Liting Hu, Karsten Schwan, Ajay Gulati, Junjie Zhang, and Chengwei Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *ICAC*, 2012.

[64] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K Panda. High performance virtual machine migration with RDMA over modern interconnects. In *ICCC*, 2007.

[65] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[66] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. *OSDI*, 2010.

[67] Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese. Net-Share: Virtualizing data center networks across services. *University of California, San Deigo, Tech. Rep. CS2010-0957*, 2010.

[68] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: comparing public cloud providers. In *IMC*, 2010.

[69] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[70] Marvin Mcnett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker. Usher: An extensible framework for managing clusters of virtual machines. In *USENIX LISA*, 2007.

[71] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via VM multiplexing. In *ICAC*, 2010.

[72] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[73] Eugene Ng. Maestro: A system for scalable openflow control.

[74] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. *SIGCOMM*, 2011.

[75] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *ICAC*, 2009.

[76] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *NSDI*, 2011.

[77] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. 2012.

[78] Alexander Stage and Thomas Setzer. Network-aware migration control and scheduling of differentiated virtual machine workloads. In *ICSE Cloud*, 2009.

[79] Kathryn E Stecke. Formulation and solution of nonlinear integer production planning problems for flexible manufacturing systems. *Management Science*, 29(3):273–288, 1983.

[80] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: Accelerating resource allocation in virtualized environments. In *ASPLOS*, 2012.

[81] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. *PADL*, 2011.

[82] Andreas Voellmy and Junchang Wang. Scalable software defined network controllers. In *SIGCOMM*, 2012.

[83] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *SIGCOMM*, 2010.

[84] Guohui Wang and TS Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM*, 2010.

[85] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*, 2007.

[86] V Yazici, MO Sunay, and AO Ercan. Architecture for a distributed openflow controller. In *Signal Processing and Communications Applications Conference (SIU), 2012 20th*, pages 1–4. IEEE, 2012.

[87] Volkan Yazıcı *et al*. Controlling a Software-Defined Network via Distributed Controllers. In *NEM Summit*, 2012.

[88] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *IEEE CLUSTER*, 2010.

[89] Ming Zhao and Renato J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC*, 2007.