Datacenter Resource Scheduling for Networked Cloud Applications

by

Nodir Kodirov

Master of Science, Konkuk University, 2010

Bachelor's Degree, Tashkent University of Information Technologies, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

October 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Datacenter Resource Scheduling for Networked Cloud Applications

submitted by **Nodir Kodirov** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Department of Computer Science, UBC
Co-supervisor

Alan J. Hu, Department of Computer Science, UBC
Co-supervisor

Margo Seltzer, Department of Computer Science, UBC
Supervisory Committee Member

Mike Feeley, Department of Computer Science, UBC
University Examiner

Sathish Gopalakrishnan, Department of Electrical and Computer Engineering, UBC
University Examiner

Additional Supervisory Committee Members:

Norm Hutchinson, Department of Computer Science, UBC
Supervisory Committee Member

Abstract

Cloud computing is an integral part of modern life, which became increasingly apparent during the COVID-19 pandemic. Applications that run on the cloud facilitate many of our daily activities, including education, retail, and high quality video calls that keep us connected. These applications run on one or more Virtual Machines (VM), where networked cloud applications can benefit from inter-VM network bandwidth guarantees. For example, an entire class of network-intensive big-data processing applications run more quickly with sufficient network bandwidth guarantees.

However, offering inter-VM bandwidth guarantees creates challenges both for resource allocation latency and datacenter utilization, because the resource scheduler must satisfy per-VM resource demands and inter-VM bandwidth requirements.

This dissertation demonstrates that it is feasible to offer inter-VM bandwidth guarantees as a first class cloud service. We develop several algorithms that allow efficient sharing of datacenter network bandwidth across tenants. These algorithms maintain high datacenter utilization while offering low allocation latency. Specifically, we propose constraint-solver-based algorithms that scale well to datacenters with hundreds of servers and heuristic-based algorithms that scale well to large-scale datacenters with thousands of servers. We demonstrate the practicality of these algorithms by integrating them into the OpenStack cloud management framework. We also construct a realistic cloud workload with bandwidth requirements, which we use to evaluate the efficiency of our resource scheduling algorithms.

We demonstrate that selling inter-VM network bandwidth guarantees as a service increases cloud provider revenue. Furthermore, it is possible to do so without changing cloud affordability for the tenants due to shortened job completion times for the tenant applications. Savings from the shortened VM lifetimes can be used to cover the network bandwidth guarantees service cost, which allows tenants to complete their job faster without paying extra. For example, we show that cloud providers can generate up to 63% extra revenue compared to the case when they do not offer network bandwidth guarantees.

Lay Summary

Cloud computing is an integral part of modern life, which became apparent during the COVID-19 pandemic. Applications that run on the cloud facilitate many of our daily activities, including education, retail, and high quality video calls that keep us connected. A subset of these applications rely on having sufficient network bandwidth to function properly. This dissertation explores offering network bandwidth guarantees as a first class cloud service. Specifically, it tries to answer these three questions: *Can we schedule network bandwidth efficiently? Can we integrate this service into an existing cloud management framework? Can we justify the price of this service?* This dissertation answers "Yes" to all three questions, and presents i) efficient datacenter resource scheduling algorithms, ii) a prototype of the solution in OpenStack, and iii) a justified service price that maintains revenue neutrality for the cloud provider without changing cloud affordability for the customers.

Preface

This dissertation includes one published work, which initially appeared at a conference with an extended version published at a journal. Specifically, the material in Chapter 2 was originally published as “Scalable Constraint-based Virtual Data Center Allocation” at the International Joint Conference on Artificial Intelligence (IJCAI), 2017 [30], and the extended version was published in the Artificial Intelligence Journal (AIJ), 2020 [31], with the same title. This was a joint work with Sam Bayless, Syed M. Iqbal, Ivan Beschastnikh, Holger H. Hoos, and Alan J. Hu. I was the lead investigator along with Sam Bayless. In this work, I had the primary role in formulating the problem and dataset and was heavily involved in designing the experimental methodology and editing the manuscript. Sam Bayless implemented the algorithms, conducted the experiments and wrote the manuscript. Syed M. Iqbal developed Integer Linear Programming solver based algorithms.

Chapter 3 and Chapter 4 are based on unpublished work in collaboration with co-authors Syed M. Iqbal, Marlon Ou, Shane Bergsma, Margo Seltzer, Alan J. Hu, and Ivan Beschastnikh. In this work, I was the lead investigator. Syed M. Iqbal was the lead investigator on the Integer Linear Programming parts and Marlon Ou contributed heavily in developing the OpenStack prototype. Shane Bergsma provided datasets and guidance to make our implementation practical. Source codes for these chapters are available in the dissertation artifact repository [81].

Table of Contents

List of Tables

List of Figures

Acknowledgments

I will summarize contributions to this dissertation chronologically, starting from the beginning. Events and emotions are abridged.

1986	I was born to Mayram Mukhidinova and Khomidjon Kodirov.[1]
2001	I get puzzled by Sherzod Ashurov's surprise with my ability to do fractions at age 15. This happens at Gulbahor Pólatova's math tutoring class.[2]
2002	I decide to become a scientist, for everything else seems boring. [3]
2003	Gulbahor introduces Zafar Boltaev and Sherali Ochilov: my new heroes.[4]
2005	Anvar Mirzaev shows that math and computing are complementary.[5]
2006	I meet a fellow undergraduate student, Dilorom, who enters my life as a friend, gets promoted to my wife, and promotes me to become a father.[6]

[1] I am eternally grateful to my parents for their unconditional love and trust, and the freedom to choose my adventures. I am also blessed to have caring siblings, Jakhongir, Nargiza, Erkin, Akmal, and Gulrukh. Being the fifth child in the family helped me to learn from my mistakes at an early age. This skill served me well throughout my life, particularly during my PhD.

[2] Sherzod and Gulbahor started as my math idols and continue to be my close friends. I am also lucky to have studied alongside Bahrom Barnoev and Jamil Kodirov: I dearly cherish the long nights we spent solving math problems.

[3] Science inspirations are partly due to illuminative discussions between my father and a chemist uncle, Orifjon Kodirov, during our regular family gatherings. The discussion topics included the structure of atoms, the definition of infinity, the role of technology in life, and everything in between.

[4] Zafar helped me see the beauty of math, and Sherali made me feel the thrill of physics. I admire Sherali's dedication to educate many generations of scientists. He is one of my lifetime heroes.

[5] I was a sophomore undergraduate deciding on my major. Anvar, my undergraduate math instructor, showed how computer skills, such as programming in Fortran and Pascal, are useful in math. I started there and still did not return to "pure" math. I thank Anvar for introducing me to computing.

[6] Dilorom, this dissertation, and many other achievements in my life, would not be possible without you. You are the love of my life. Katherine and Feyn, my children and dear angels: you have been a source of constant joy during my PhD. You are and will always be the center of my universe.

2008	I complete my bachelor's with Rustam Khamdamov as my advisor.[7]
2010	I complete my master's with Doohyun Kim as my advisor.[8]
2012	I made my first trip to North America (California, USA) and decided to apply for graduate school on this continent.[9]
2013	I started my PhD at UBC.[10]
2014	I took Alan Hu's Introduction to Formal Verification and Analysis class.[11]
2015	I get first-hand experience on the significance of datacenter networking for cloud application performance during my internship at ZeroStack. [12]
2016	Justine Sherry submits her dissertation.[13]
2017	Our constraint-based VDC allocation paper with Sam Bayless et al. gets accepted at the International Joint Conference on Artificial Intelligence.[14]

[7] I thank Rustam for his continued support beyond my undergraduate degree. I also thank Jeisang Jeon (Ulug̀bek), who started as my Java instructor and continues to be my lifetime advisor and friend.

[8] I thank Doohyun Kim for his trust and support for me to become a researcher and for introducing me to the systems research, particularly kernel development. I also thank members of the Embedded Computing Lab, Dongwoon Jeon in particular, for making my studies at Konkuk University and life in Korea pleasant. I am grateful to Christian Rinderknecht, who started as my instructor in functional programming languages and continues to support me beyond my graduate degree. Christian's love of teaching and joy at research still inspires me. I also acknowledge the scholarship support from the Institute of Information Technology Advancement (IITA), Korea.

[9] I thank Jaegi Lee for traveling with me and Taesang Choi for supporting this trip. Many other team members, including Woojik Chun, Doyeon Kim, and Tae-Ho Lee, made my stay in ETRI (Electronics and Telecommunications Research Institute) enjoyable. Tae-Ho, thank you for being a mentor and friend while in ETRI and beyond. I am lucky to have you as a friend.

I was also fortunate to be able to join another team in ETRI, and thank all members of this team, including Soomyeon Pahk, Seonghyuk Byun, Changgyu Lim, and Sunme Kim. 김선미 팀장님, 나를 받아 주셔서 감사합니다!

[10] I thank Andrew Warfield for starting as my advisor and handing me over to Ivan Beschastnikh. Ivan, I am deeply grateful for your support during my PhD. Thank you for being there every time I needed. I am looking forward to having you as a lifetime advisor and friend.

[11] Alan started as my class instructor to eventually become my co-supervisor. Alan, I will always remember how you paused the class to ask how did my Research Proficiency Evaluation (RPE) go, when I walked in late after my RPE. That made me feel valued. I also thank Alan for tireless Zoom meetings to iterate on my earlier drafts: I am a better thinker, writer, and researcher thanks to you.

[12] I thank the entire ZeroStack team for a hands-on and fast-paced internship, particularly my mentors Ajay Gulati, Kiran Bondalapati, Gautam Kulkarni, Ravikant Malpani, and Shane Gibson.

[13] I liked how Justine Sherry formatted the acknowledgments in her PhD dissertation [115], and I decided to adopt it. Thank you, Justine, for letting me use your format!

[14] This was my first paper during my PhD and a confidence booster: I might actually be able to get a PhD. I am grateful to all of my co-authors, including Holger Hoos, Syed M. Iqbal, Marlon Ou, Shane Bergsma, and Fabian Ruffy. Sam, thank you for your research mentorship and heartfelt friendship. My stay at UBC and in Canada was much more pleasant thanks to you.

2018	Margo Seltzer joins UBC.[15]
2020	The Systopia Lab gets founded.[16]
2021	This dissertation is filed.

The work in this dissertation was developed in a collaboration between Systopia and the Integrated Systems Design Laboratory (ISD). Projects described in this dissertation and my PhD have been generously supported by several grants, both national and commercial. These include UBC four-year doctoral fellowship, Huawei, NSERC Discovery, NSERC Create, and Microsoft Azure Cloud Sponsorship. Ivan Beschastnikh acknowledges the support of the NSERC Discovery Grants (RGPIN-2020-05203, RGPIN-2014-04870), and NSERC Engage Grant (EGP 514614-17).

[15] I am eternally grateful to Margo for taking me on as her student, joining my PhD committee, and championing my work throughout the years. Margo, thank you for beating the hand-wavy scientist out of me and accommodating my meeting requests even in non-ideal times. I am also grateful to two other committee members: Norm Hutchinson and the late Bill Aiello. Norm, thank you for always being a bearer of good news. I will always remember our surprising and uplifting morning walk to the department on the day of my PhD proposal defense. Your presence always made me feel confident. I am immensely grateful to Bill for encouraging me to have an independent research taste and opinion of my own, regardless of the hot topics in the top conferences. This dissertation would not be possible without that encouragement, and I dedicate this work to Bill.

[16] Systopia subsumes the Networks, Systems, and Security (NSS) Lab that I was part of and connects other systems researchers across UBC and beyond. I thank many NSS and Systopia members for creating a lovely environment, which included regular soccer games, ice-cream socials, research presentation karaokes, and many other recharging activities. I made many friends from these. I thank Mihir Nanavati, Shriram Rajagopalan, Jean-Sébastien Légaré, Patrick Colp, Yanyan Zhuang, Marjan Alavi, David Williams-King, Kent Williams-King, Robert Jackson Sumi, Peter Chen, Lise Savard, Fabian Ruffy, Clement Fung, Amanda Levin, Anand Jayarajan, Haley Li, Surbhi Palande, Puneet Mehrotra, Tony Mason, Craig Mustard, Swati Goswami and other labmates for camaraderie. Mihir, thank you for the countless late-night lab discussions about research, science, and life.

Dedication

To The Loving Memory of Bill Aiello
A Scientist With a Big Heart

Chapter 1

Introduction

The data center is now the computer.
— Luiz Barroso (2007) [105]

Cloud computing is an integral part of modern life, particularly amplified during the COVID-19 pandemic. It powers remote education [85], health care [56], retail [6] and many other industries [5] and keeps people connected over high quality video calls [4]. The broad use of cloud computing translates into a wide range of applications running in cloud datacenters.

Cloud applications consume datacenter resources by allocating Virtual Machines (VM). Figure 1.1 shows a sample datacenter topology, composed of many servers connected over a datacenter network. A VM includes a *bundle* of hardware resources, such as CPU and RAM. This bundle has been expanding since the initial days of cloud computing. For example, in 2006, Amazon Elastic Compute Cloud (EC2) beta offered one-size-fits-all VMs with one virtual CPU (vCPU), 1.75 GB of RAM, 160 GB of local disk, and 250 Mbps of network bandwidth [26]. However, today, there are many hundred VM types, also called *VM flavors*, and bundles include diverse resources, such as SSD disks, GPUs, and accelerators [11].

The diversity in cloud resources is the product of the wide range of cloud applications deployed today: some run on a single VM, some require multiple VMs [16], while others require multiple VMs connected over a virtual network [15]. Cloud providers need to satisfy increasingly diverse application requirements and grow their capacity to keep up with the volume of the resource demand.

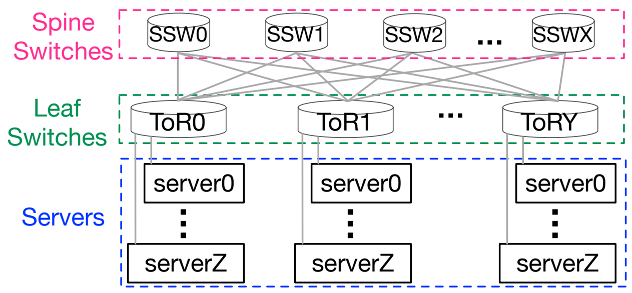


Figure 1.1: Example Datacenter Topology. The datacenter network has three tiers that connect servers with CPU, RAM, and other local resources.

Multi-tenancy is the key factor that enables cloud providers to continuously grow their datacenters in an economically sustainable way. Multi-tenancy allows sharing datacenter resources across a large number of customers, or tenants. The sheer scale of multi-tenancy allows providers to leverage economies of scale and operate datacenters at high utilization. High utilization, in turn, not only enables sustainable growth but also lowers tenant cost – further fueling cloud adoption. Thus, providers strive to efficiently share datacenter resources across the tenants.

Cloud providers need to *isolate* tenant workloads from each other to have practical multi-tenancy. Without isolation, workloads interfere with each other and render the performance of cloud applications unpredictable [128]. In particular, existing studies on cloud performance show that it is impossible to offer predictable application performance without network bandwidth guarantees [21, 128]. For example, Ballani et al. model MapReduce-inspired cloud workloads, and show that job completion time can be shortened by up to 9.2× had the cloud network performance been sufficient [21]. Uta et al. generalize this for big data workloads and demonstrate that these workloads suffer 25% to 50% performance loss due to unpredictable cloud networking [128]. Moreover, recent works show that distributed machine learning training applications converge more quickly, i.e., the training job completes in a shorter time, when they are deployed with network bandwidth guarantees [63, 72, 106]. Providing cloud network bandwidth guarantees is one of the main purposes of this dissertation.

We distinguish two aspects of isolation: *scheduling* and *enforcement*. Scheduling is the focus of this dissertation and is about developing *efficient* algorithms to

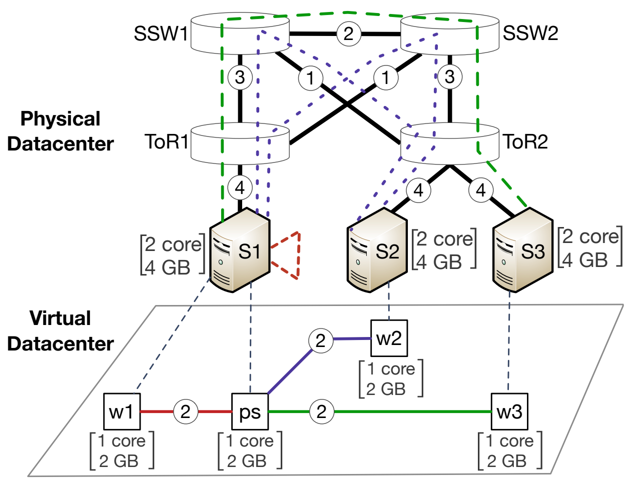


Figure 1.2: Example Virtual Datacenter (VDC) Allocation. The VDC runs machine learning application. The **top** part shows a datacenter with three servers, two top-of-rack (ToR) switches, and two spine switches (SSW). Circled numbers on links denote available bandwidth in Gbps. The **bottom** part shows a sample VDC with a parameter server VM (`ps`) and three worker VMs (`w1`, `w2`, `w3`). Worker VMs connect to VM `ps` with 2 Gbps network bandwidth (shown in circles). VMs also require a certain number of CPU cores and RAM. The VDC gets allocated to the datacenter, as illustrated with the dashed lines.

allocate datacenter resources across multiple tenants over time. Here, efficiency means the scheduler is able to achieve high datacenter **utilization** while maintaining low resource allocation **latency**. High datacenter utilization makes the cloud more affordable. For example, the operators of the Azure cloud report that increasing datacenter utilization by 1%, brings $100 million/year savings [62]. At the same time, low latency improves a tenant's cloud experience by shortening wait times. Short wait times enable agile cloud application development.

In contrast to *scheduling*, *enforcement* is the mechanism to ensure that tenants do not exceed their allocated resources. Isolation enforcement is addressed by virtualization techniques, such as Xen [24], KVM [82] and Andromeda [42] and is not the focus of this work.

A virtual datacenter (VDC) is a construct to provide isolation. It allows tenants to run cloud applications with inter-VM network bandwidth guarantees. The VDC includes a collection of VMs, along with the local resource requirements of each VM (e.g., CPU and RAM), and inter-VM connectivity requirements [59]. As a simple VDC application, consider distributed Machine Learning (ML) training, which performs data-parallel model training [125]. In Figure 1.2, the VDC has three worker VMs (`w1`, `w2`, and `w3`) that communicate model parameter updates to the parameter server VM (`ps`). VDC bandwidth guarantees ensure that the model parameter updates happen in a timely fashion, without getting blocked or delayed due to network bandwidth scarcity between the worker and parameter server VMs.

Figure 1.2 shows VDC allocation on a 3-server datacenter. Here, a 4-node VDC (bottom) is mapped onto a physical datacenter with three servers, two top-of-rack (ToR) switches, and two spine switches (top). The VM placement is indicated with dashed lines. For example, VM `ps` and VM `w1` are placed on the same server (colocated). At the same time, the virtual link `ps-w2` is allocated in a multi-path route where each path provides 1 Gbps network bandwidth. Although no major cloud provider offers VDCs with inter-VM network bandwidth guarantees yet, one can extend the existing cloud resource provisioning formats to support VDCs, e.g., by using AWS CloudFormation templates [13].

VDC allocation has two parts: *local* and *cross-device* resource allocation. Local resource allocation is simpler from the scheduler's perspective, because it reasons only about a single device, such as CPU cores in a server. In this case, the scheduler just needs to find a server with sufficient CPU cores to accommodate the VM. However, cross-device resource allocation, i.e., VDC network bandwidth guarantees, is harder, because it involves reasoning about multiple devices. For example, a VM-to-VM virtual link in a VDC, where peer VMs are allocated in two different servers, such as VM `ps` and VM `w3` in Figure 1.2, involves reasoning about resources in server `S1` and server `S3` as well as every intermediate network node between those two servers, such as `ToR1`, `SSW1`, `SSW2`, and `ToR2`. Thus, VDC scheduling imposes an allocation latency challenge because multiple *connected* VMs in the VDC might require cross-device resource allocation. For example, as a datapoint from our baseline VDC scheduler in Section 4.1.2, a VM allocation latency with only CPU and RAM was around 5 ms, while allocating a VM with one virtual link took 32 ms. The VM allocation latency increases proportionally to the number of virtual links in the VM. Thus, **the main resource scheduling challenge**

is to design a VDC scheduler that offers high datacenter utilization and low resource allocation latency.

Thesis: *Datacenter resources can be efficiently shared by using a VDC scheduler. Here, efficiency means achieving high datacenter utilization while maintaining practical resource allocation latency.*

We make five contributions in support of the thesis:

1. **Workload:** We propose a new technique, Gridiron, to generate a realistic VDC workload to evaluate VDC schedulers. The Gridiron technique augments an existing VM workload with network bandwidth requirements to generate a VDC workload. The existing workload is from Azure cloud's production traces released with the Resource Central paper [40]. Our VDC workload is the first publicly available production-based cloud workload with inter-VM bandwidth requirements. Our VDC workloads and source codes are available in the dissertation artifact repository [81].

2. **Metrics:** We propose *revenue gain* as the metric for evaluating VDC schedulers. We demonstrate how the commonly used static packing metric cannot accurately capture scheduler quality, and we propose an alternative, revenue gain metric, that is better for scheduler evaluation. The static metric measures how many VDCs a scheduler is able to pack into an empty datacenter until the scheduler is no longer able allocate a VDC. The revenue gain metric measures the scheduler's packing quality *over time* — not only until the datacenter is filled but also for newly arriving requests to fill up the resources freed by deallocations. The revenue gain captures what fraction of an ongoing workload the scheduler accommodates in a given datacenter. For example, cloud operators have the maximal revenue gain when a scheduler fully accommodates the workload.

3. **Algorithms:** We develop several heuristic-based and constraint-solver-based VDC scheduling algorithms. For our heuristic algorithms, we extend and enhance the state-of-the-art VM scheduling algorithm of the popular cloud management framework, OpenStack, to support virtual network bandwidth

guarantees. For our constraint-solver-based algorithms, we propose NETSOLVER. NETSOLVER is attractive because it offers *completeness*: a guarantee that the VDC gets allocated, if at all possible. Completeness can yield higher datacenter utilization. Unfortunately, NETSOLVER does not scale to realistic VDC workloads that require datacenters over 1,000 servers. Its VM allocation latencies are prohibitively high, e.g., over 200 minutes for allocating a VDC with 10 VMs (20 minutes per VM) on a datacenter with 6,144 servers. Therefore, based on insights from NETSOLVER, we develop STARNETLA: a heuristic algorithm that offers three order of magnitude lower VM allocation latency while achieving comparable revenue gain to NETSOLVER.

4. **Optimality:** We define *online* and *offline* optimality for VDC scheduling algorithms. In an online setting, the VDC scheduler processes requests in the order they arrive, unaware of characteristics of future requests. In the offline setting, the VDC scheduler is clairvoyant: it has full workload visibility and can optimize VDC scheduling towards an objective, such as minimizing VM allocation failures. We develop an Integer Linear Programming (ILP)-based offline optimal VDC scheduling algorithm: ORACLE. Although it is impossible to deploy ORACLE in practice, because clairvoyance is not realistic, we use ORACLE to study how close our practical VDC schedulers are to the theoretical optimal.

5. **Prototype:** We integrate our heuristic-based VDC scheduling algorithm into OpenStack by extending OpenStack's Nova scheduler. We demonstrate that our enhancements are practical: Nova's existing filtering-based scheduler architecture can easily accommodate our enhancements. Moreover, our prototype shows that the extra latency introduced by end-to-end network bandwidth scheduling is negligible. For example, in a one-node OpenStack deployment, the added per-VM latency of 2.37 ms accounts for only 0.036% of the total time required to allocate a VM.

This dissertation proceeds as follows. In Chapter 2, we propose constraint-solver-based approaches for VDC scheduling. We show how our constraint solvers' completeness property enables more static VDC allocations. In Chapter 3, we describe the Gridiron technique to generate realistic VDC workloads from a VM workload. The realistic workloads overcome the limitations of the synthetic workloads that we used for evaluating constraint-solver-based approaches. In Chapter 4, we study VDC allocation in practice by using realistic VDC workloads. We make a case for offering network bandwidth guarantees as a first-class cloud service and introduce the revenue gain metric for VDC scheduler evaluation. We also extend state-of-the-art heuristic algorithms to perform end-to-end bandwidth allocation, compare heuristic algorithms with their constraint-solver-based alternatives, and study the algorithms' optimality bounds. Finally, we show how we integrate our schedulers into OpenStack. We discuss future work and conclude in Chapter 5.

Chapter 2

Constraint-solver-based VDC Scheduling

The previous chapter introduced virtual datacenters (VDCs) and highlighted the importance of achieving high datacenter utilization and low latency VDC allocation. In this chapter, we consider datacenter utilization as the primary objective and explore using constraint solvers in scheduling VDCs, for they offer *completeness*: the guarantee that they will allocate the VDC if it is at all possible.

Constraint-based techniques, such as Integer Linear Programming (ILP) and SAT Modulo Theories (SMT), play a key role in state-of-the-art approaches for challenging problems across a wide range of applications (e.g., [34, 35, 108, 111]). We demonstrate how VDC scheduling can be tackled using constraint solvers such as Gurobi [61] and MONOSAT [29]. We formalize VDC allocation in terms of multi-commodity flows, allowing us to exploit the efficient handling of network-commodity flow problems in Gurobi and MONOSAT. We implemented our ideas in our constraint-based VDC scheduling algorithm, NETSOLVER.

NETSOLVER is reasonably scalable, sound, and complete, with support for end-to-end, multi-path bandwidth allocation across all the layers of the networking infrastructure, from servers to ToR switches to spine switches. NETSOLVER efficiently allocates VDCs with up to 15 VMs to datacenters with up to 1,000 servers, typically in seconds per VDC allocation. Across a wide variety of datacenter topologies, NETSOLVER can statically pack $150\% - 300\%$ as many total

Table 2.1: VDC Scheduling Algorithms. We compare the features of contemporary sound VDC allocation algorithms and four recent VNE algorithms: GAR-SP/PS (and variant RW-MM), D-ViNE, and ASID, based on linear programming, mixed integer programming, and subgraph isomorphism detection, respectively.

Algorithm	Sound	Complete	Multi-path	Multi-VM	VDC Topology	Datacenter Topology
SecondNet [59]	✓				All	All
Importance Sampling [122]	✓			✓	All	Tree
Oktopus [21]	✓			✓	Star	All
VDCPlanner [137]	✓			✓	All	All
HVC-ACE [112]	✓		✓	✓	Hose	All
GAR-SP/PS [134]	✓		✓	✓	All	<200 nodes
RW-MM-SP/PS [37]	✓		✓		All	<200 nodes
D-ViNE [38]	✓		✓		All	<200 nodes
ASID [87]	✓				All	<200 nodes
VirtualRack [67]	✓	✓			Hose	All
Z3-AR [135]	✓	✓	✓		All	Tree
NETSOLVER (this work)	✓	✓	✓	✓	All	All

VDCs in the same datacenter as SecondNet's VDCAlloc algorithm [59], a state-of-the-art heuristic method. (However, as we demonstrate in Chapter 4, NETSOLVER does not scale to realistic datacenter sizes and VDC workloads. In Chapter 4, we explore more scalable alternatives.)

2.1 Related Work

Table 2.1 summarizes prior work with respect to features of VDC allocation that are relevant to modern datacenters. As we can see, all prior approaches have important limitations relative to NETSOLVER.

Soundness: Sound VDC allocation tools respect end-to-end bandwidth guarantees, while unsound tools attempt to minimize datacenter network traffic only without guaranteeing that VMs will have sufficient dedicated bandwidth. Examples of unsound approaches to VDC allocation include Kakadia et al. [75] and Meng et al. [93], which dynamically identify VM communication patterns through network traffic analysis.

This prior work is in contrast to the approaches proposed in this dissertation, all of which are sound and assume that VDCs and their communication requirements are explicitly known to the scheduler.

Completeness: Most VDC allocation tools that respect bandwidth guarantees are *incomplete*: they can fail to find feasible VDC allocations in cases where such allocations exist (even when given unlimited runtime). Oktopus [21], VDCPlanner [137], and SecondNet [59] are examples of incomplete allocation algorithms. For example, SecondNet's algorithm is greedy in that it maps VMs to servers before checking for available paths, and allocates bandwidth one path at a time; if either of these steps fails, VDC allocation fails. (SecondNet will try this process several times on different subsets of the datacenter before giving up, which does not change its incompleteness.)

Similarly, Hadrian [23], Cicada [83], and CloudMirror [84] use incomplete greedy heuristic algorithms that attempt to colocate VMs of the VDC in the smallest physical datacenter sub-tree.[1] Pulsar [10] uses Hadrian's VDC allocation algorithm and extends it to accommodate VM-appliances (such as SSDs and encryption devices).

HVC-ACE [112] is complete when VDC "bandwidth requirements are negligible", which assumes that datacenter network bandwidth is overprovisioned so that no VDC can fail due to bandwidth scarcity. This assumption does not hold in our setting: VDC allocations fail due to bandwidth scarcity. Thus, HVC-ACE [112] is incomplete for our setting. In general, Rost et al. [112] propose several other VDC allocation algorithms, including VC-ACE, that are complete but make two simplifying assumptions: 1) VDC VMs have identical bandwidth requirements and 2) VM-to-server assignments are *already* fixed. These assumptions do not hold in our VDC allocation setting and we exclude these algorithms from our study.

D-ViNE and its variant R-ViNE (not shown) proposed by Chowdhury et al.

[1] NETSOLVER uses max-flow (and additional constraints) for achieving completeness and multi-path allocations to place VMs with end-to-end network bandwidth guarantees. Hadrian also uses max-flow. However, they use it for a completely different sub-problem. They use max-flow for computing bounds on link bandwidth to decide bandwidth constraints on their hose model [22, 23].

[38] are also incomplete because only a subset of physical nodes (servers) are considered to be "feasible" for placing a virtual node (VM). Feasibility is decided based on the "location" requirement of the VM relative to other VMs in the VDC. The location requirement prevents all datacenter servers from getting included in the feasible region, which reduces the search space but also makes the algorithms incomplete because they miss potential valid VM placements outside the feasible region.

In contrast with this prior work, the constraint-based approaches described in Yuan et al. [135] and NETSOLVER are both complete: they are guaranteed to (eventually) find a feasible allocation if one exists. We show in our experiments that completeness does not merely represent a theoretical benefit but can translate into more VDC allocations. NETSOLVER is the first sound and complete VDC scheduler that can be applied to any VDC and datacenter topology without simplifying abstractions.

Multi-path Allocations: Many datacenters use multi-path allocations to maximize bandwidth and to provide fault-tolerance and load-balancing [3, 109]. Lack of multi-path support in traditional L2/L3-based networks was a primary motive for datacenter operators to develop networking stacks with multi-path support [9, 42, 117].

Despite the increasing importance of multi-path routing, to the best of our knowledge, there is only one previous VDC scheduler that supports multi-path communication between VMs: HVC-ACE [112], a sound but incomplete scheduler that uses a *hose model* for VDCs (we describe hose models below). There are also several incomplete algorithms for virtual network embedding that have support for multi-path allocation for smaller physical networks with 50-150 servers [37, 38, 134]. NETSOLVER is the first sound and complete multi-path VDC scheduler.

Multi-VM Allocations: Some tools simplify VDC placement by assuming that the VMs in a VDC must all be placed on separate servers. For example, SecondNet [59] uses bipartite graph matching to assign VMs to servers; as a result, it can place only a single VM per server when allocating a given VDC.

Similarly, VirtualRack's [67] virtual tree abstraction places each VM into a separate leaf node server. D-ViNE [38] uses mixed-integer programming to perform virtual network embedding, but their encoding does not support allocating multiple virtual nodes per server.

In many cases, it can be advantageous to place multiple VMs on one server, since communication between the colocated VMs is cheap. Multi-VM placement is useful to take advantage of data locality between VMs and can be explicitly requested by a tenant. Conversely, a tenant may want single-VM placement for higher fault tolerance. For example, VMs hosting different database replicas can be assigned to different servers to decrease fate-sharing. By default, NETSOLVER performs multi-VM placement. However, NETSOLVER also supports anti-affinity constraints as well as other advanced placement options, which can be used to force some or all of the VDC VMs to be placed on disjoint servers [31].

Unrestricted Topologies: Many VDC schedulers simplify the problem, either by abstracting VDC topologies into simpler ones that are easier to allocate, or by restricting the physical datacenter to simpler topologies. For example, the abstraction-refinement encodings from Yuan et al. [135] apply only to tree-topology datacenters. Oktopus [21] abstracts VDCs into virtual clusters, which are VMs connected to a central virtual switch in a star topology. VirtualRack [67], HVC-ACE [112], and Hadrian [23] use a less-restricted *hose model* [48] abstraction for VDCs: A hose model allows one to specify only aggregate, rather than pairwise, bandwidth requirements for virtual machines — that is, each VM is guaranteed a certain amount of ingress and egress bandwidth into the VDC network as a whole, but is not guaranteed to have any specific amount of bandwidth to any specific VM. Hose models generalize the star-topology used in Oktopus, but cannot, for example, model virtual networks that include cycles or (non-trivial) trees. NETSOLVER is the first sound and complete VDC scheduler that supports arbitrary VDC and datacenter topologies.

As observed by Ballani et al. [23], VDC allocation is closely related to *virtual network embedding* (VNE) [32, 52]. The VNE literature, however, has fo-

Table 2.2: Notations in the Constraint Solving Equations.

Notation	Description
S	The set of servers in the datacenter.
N	The set of network switches in the datacenter.
L	The set of network edges in the datacenter.
R	Virtual network bandwidth requirements in a VDC, e.g., $R(v,w) = 1$ means that VM v and VM w are connected with 1 unit of bandwidth.
A	VM to server assignment, e.g., $A(v) = s$ means that VM v is placed on the server s.
B	Bandwidth assignment to link, e.g., $B_{v,w}(l) = 1$ means that there is 1 unit of bandwidth from VM v to VM w on link l where $l \in L$.
VM	VMs in a VDC.
V	VMs placed on a server, e.g., $V(s)$ represents all VMs placed on the server s. Formally, $V(s) = \{v \in VM \mid A(v) = s\}$.

cused on allocating virtual networks onto substrate networks that are representative of medium-sized ISPs, with 50–150 servers and few or no intermediate switches. For example, recent VNE tools: GAR-SP/PS [134], RW-MM-SP/PS [37], D-ViNE [38], ASID [87], and ALEVIN [51] all fall into this range. In contrast, work on VDC allocation has typically focused on allocating to larger physical networks with topologies representative of typical datacenters, often with thousands (or even hundreds of thousands) of servers, along with intermediate switches [25, 59]. Therefore, even though the problem definitions in the VNE and VDC literature often overlap, VDC tools have made different trade-offs to focus on scalability. We compare NETSOLVER to several representative VNE approaches in Section 2.4.5 and confirm that these tools perform poorly on typical VDC instances.

2.2 The Multi-path VDC Allocation Problem

We formalize the multi-path VDC allocation problem. Table 2.2 has our notation.

The multi-path VDC allocation problem is defined as follows. We are given the description of a physical datacenter (DC) and a virtual datacenter (VDC). The DC is specified through a set of servers S, switches N, and a directed graph with vertices $(S \cup N)$ and edges L. The graph edges represent network links in a datacenter with capacities $c(i, j)$ for each link in L. The VDC consists of a set of virtual machines, VM, and a function $R : VM \times VM \mapsto \mathbb{Z}^+$ that specifies bandwidth requirement between those machines. For each server $s \in S$, we have CPU core and RAM capacity specifications, $cpu(s)$ and $ram(s)$, and for each virtual machine $v \in VM$, we are given CPU core and RAM requirements, $cpu(v)$ and $ram(v)$.

The objective in the multi-path VDC allocation problem is to find an assignment $A : VM \mapsto S$ of virtual machines to servers S along with an assignment of non-negative bandwidth $B_{v,w}(l)$ to links $l \in L$ for each bandwidth requirement $R(v,w)$, where $v,w \in VM$, satisfying the following constraints:

- **Local VM allocation constraints** ($\mathbf{L}$) ensure two properties: First, that each virtual machine is assigned to exactly one server:

  $$\forall v \in VM : \left(\sum_{s \in S} \big(A(v) = s\big) \right) = 1$$

  Secondly, that each server provides sufficient CPU and RAM resources to accommodate the requirements of all VMs allocated to it:

  $$\forall s \in S : \qquad \left(\left(\sum_{v \in V(s)} cpu(v) \right) \leq cpu(s) \right) \wedge \left(\left(\sum_{v \in V(s)} ram(v) \right) \leq ram(s) \right),$$

  where $V(s) = \{v \in VM \mid A(v) = s\}$.

  Resource requirements are modeled using integer values, and VMs do not share resources.

- **Global bandwidth allocation constraints** ($\mathbf{G}$) ensure that sufficient bandwidth is available in the physical datacenter network to satisfy all bandwidth requirements between pairs of VMs. We formalize this by requiring that for all $(v,w) \in VM$, the bandwidth assignments $B_{v,w}(l)$ must form a valid network flow with its source at $A(v)$ and its sink at $A(w)$. Further, we require that the value of that network flow be greater than or equal to the required bandwidth $R(v,w)$, and that none of the link capacities l in the physical datacenter network is exceeded: $\forall l \in L : \sum_{(v,w) \in VM} B_{v,w}(l) \leq c(l)$. Bandwidths are represented by integer values; bandwidth between VMs allocated on the same server (colocated) is unlimited.

It has been previously observed [38, 60, 121, 134] that when allowing multi-path (also called path-splitting), the global bandwidth allocation constraints give rise to a multi-commodity flow problem, which is NP-complete even for undirected integral flows [50]. Conversely, any multi-commodity flow problem maps directly into bandwidth constraints above, establishing the NP-hardness of the multi-path VDC allocation problem [38]. In principle, this multi-commodity flow problem can be solved efficiently via linear programming for real-valued flows, but this approach does not support the local VM allocation constraints. Approaches that express the global constraints as a linear program therefore either require an additional mechanism to perform local server allocation [134] or use mixed integer programming [38].

2.3 NetSolver

Previous work on constraint-based VM placement used techniques from two methods: Integer Linear Programing (ILP) and SAT modulo Theories (SMT). We now describe how VDC allocation can be implemented and efficiently solved using either Gurobi, a state-of-the-art ILP solver [61], or MONOSAT, an SMT solver with support for network flows [29].

2.3.1 Encoding Multi-path VDC Allocation in ILP

ILP solvers are commonly used for solving maximum flow and multi-commodity flow problems and are widely cited in the literature for that use-case, across a broad range of applications, such as reducing travel distance in dynamic route guidance [76] and minimizing travel distance to reconnect partitioned mobile sensors [118]. CPLEX and Gurobi, for example, are able to automatically recognize properly encoded multi-commodity flow problems and handle them using special-cased techniques [61, 70]. The details of how these solvers handle flow problems are proprietary, but examples of such approaches are discussed in the literature [90].

The local and global constraints **L** and **G** defined in Section 2.2 are directly expressible as ILP constraints. We describe these fully below:

For each $v \in VM$ and each $s \in S$, we introduce a binary variable $A_{v,s}$ to represent whether v is placed on s. In other words, $A(v,s) = 1$ if VM v is assigned to server

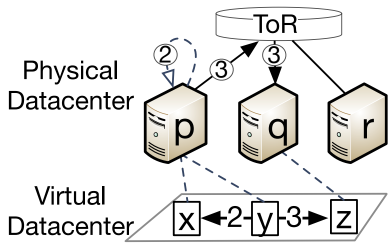


Figure 2.1: Sample VDC Allocation. Dashed lines indicate VM placement where the source VM *y* and the first destination VM *x* are placed on server *p*, and the second destination VM *z* is placed on server *q*. We call VM *x* and VM *y colocated VMs*.

s, $A(v,s) = 0$ otherwise. For example, in Figure 2.1, $A(x,p) = 1$, $A(x,q) = 0$, and $A(x,r) = 0$. We then add a cardinality constraint to enforce that each VM is placed on exactly one server:

$$\forall v \in VM : \left(\sum_{s \in S} A_{v,s} \right) = 1$$

Then, we enforce the constraints for CPU and RAM resources:

$$\forall s \in S : \left(\sum_{v \in VM} A_{v,s} \cdot cpu(v) \right) \leq cpu(s)$$

$$\forall s \in S : \left(\sum_{v \in VM} A_{v,s} \cdot ram(v) \right) \leq ram(s)$$

Together, the above constraints enforce the local constraints **L**.

Similarly, we directly encode the global constraints **G** as multi-commodity flow constraints: For each $v, w \in VM$, and each link $l = (i, j) \in L$, we introduce a non-negative integer variable $B_{v,w}(l)$, representing the bandwidth on link l consumed by the virtual link (v, w). We then assert that for each physical link, the sum of bandwidth consumed by all virtual links (vlinks) does not exceed the capacity of that physical link:

$$\forall (i, j) \in L : \left(\sum_{(v,w) \in VM} (B_{v,w}(i, j)) \right) \leq c(i, j)$$

Next, we enforce network flow constraints on the switches. Since VMs cannot be placed on switches N, this simply enforces that the ingress flow (i,n) equals the egress flow (n, j) for each switch $n \in N$ and each vlink (v,w). Formally:

$$\forall (v,w) \in VM, \forall n \in N : \sum_{(i,n) \in L} B_{v,w}(i,n) = \sum_{(n,j) \in L} B_{v,w}(n,j) \tag{2.1}$$

where i and j can be a server ($i, j \in S$) or a switch ($i, j \in N$) because switches can connect to servers as well as other switches. For example, a top-of-rack switch in Figure 1.2 connects to servers and spine switches (page 3). In other words, this constraint enforces flow conservation on the switches, which is more apparent if we transform Equation 2.1 as follows:

$$\forall (v,w) \in VM, \forall n \in N : \sum_{(n,j) \in L} B_{v,w}(n,j) - \sum_{(i,n) \in L} B_{v,w}(i,n) = 0 \tag{2.2}$$

Informally, the equation above says that the **difference** between switch egress traffic and switch ingress traffic is zero. This must hold because switches neither produce nor consume any traffic. Hence, flows are conserved.

Similarly, network flow constraints on servers enforce flow conservation on each server. Intuitively, the flow conservation rule in Equation 2.2 should hold on each server, unless the server produces or consumes traffic. Given that servers do produce (or consume) traffic when VMs are placed on them, the server flow conservation constraints should account for the VMs placed on the servers. Specifically, the net egress (or ingress) traffic from each server should be equal to the **difference** between traffic produced (or consumed) by the source VMs on that server and traffic consumed (or produced) by the destination VMs on that server. We add two extra terms to capture the server egress traffic ($R(v,w) \cdot A_{v,s}$) and the server ingress traffic ($R(v,w) \cdot A_{w,s}$). Here, v is the source VM (traffic producer) and w is the destination VM (traffic consumer):

$$\forall (v,w) \in VM, \forall s \in S : \sum_{(s,j) \in L} B_{v,w}(s,j) - \sum_{(i,s) \in L} B_{v,w}(i,s) = R(v,w) \cdot A_{v,s} - R(v,w) \cdot A_{w,s} \tag{2.3}$$

Note that this equation also embodies VM colocation, as shown with VM x and

VM y in Figure 2.1. In this case, the server that acts as the traffic producer is also the traffic consumer. We demonstrate an example of this equation by applying it to the VDC allocation shown in Figure 2.1.

Equation 2.3 enforces server flow conservation by instantiating the constraints for each vlink and each server. The example VDC allocation shown in Figure 2.1 has three servers and two vlinks: (x,y) and (y,z). Thus, six instances of this equation are given to the ILP solver for enforcing server flow conservation. We show the expansion of Equation 2.3 for only one instance, when vlink $(v,w) = (y,z)$ and server $s = p$, and omit other instances for brevity:

$$\sum_{(p,j)\in L} B_{y,z}(p,j) - \sum_{(i,p)\in L} B_{y,z}(i,p) = R(y,z)\cdot A_{y,p} - R(y,z)\cdot A_{z,p} \tag{2.4}$$

We transform the left side of the above equation by applying
$L = \{(p,ToR),(ToR,p),(q,ToR),(ToR,q),(r,ToR),(ToR,r)\}$:

$$\sum_{(p,j)\in L} B_{y,z}(p,j) - \sum_{(i,p)\in L} B_{y,z}(i,p) = B_{y,z}(p,ToR) - B_{y,z}(ToR,p) = 3 - 0 = 3 \tag{2.5}$$

Note that we made this transformation by omitting the irrelevant physical links. For example the (p,j) edge cannot get assigned value (q,ToR) because the (p,j) expression requires the first node of the physical link to be p. There is only one such edge (p,ToR) in the datacenter shown in Figure 2.1. Now we transform the right side of the Equation 2.4 by using $R(y,z) = 3$, $A_{y,p} = 1$, $R(y,z) = 3$, and $A_{z,p} = 0$:

$$R(y,z)\cdot A_{y,p} - R(y,z)\cdot A_{z,p} = 3\cdot 1 - 3\cdot 0 = 3 \tag{2.6}$$

This shows that Equation 2.4 holds, because both the left side (Equation 2.5) and the right side (Equation 2.5) have the same value (3). Thus, we conclude that the server flow conservation constraints are satisfied per Equation 2.3 for vlink (y,z) and server p. In other words, this example shows that the net egress traffic (3) from server p is equal to the **difference** between traffic produced (3) by the source VM y on that server and traffic consumed (0) by the destination VM z on that server. Similar constraints are built for the remaining five instances of Equation 2.3: for vlink (y,z) on servers (q,r) and for vlink (x,y) on servers (p,q,r). Combined, these six instances fully enforce flow conservation on the servers.

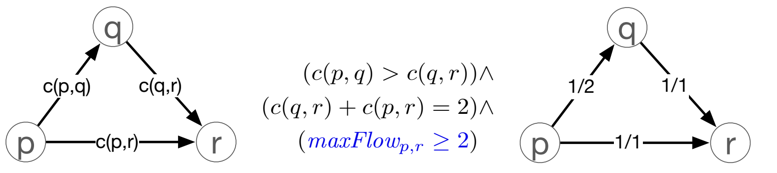
Gurobi has the ability to incrementally re-solve a system of equations after changing the coefficients. In the above equations, the constants that define the resource requirements and bandwidth requirements of the VDC and that define the capacities of each server and physical link in the datacenter network appear as constants. So long as there is a bound on the number of VMs per VDC, the same set of constraints can be re-used for subsequent allocations, after updating each of those constant values appropriately, e.g., to subtract used bandwidth from the capacities of the links of the datacenter network or to alter the bandwidth requirements between two VMs. Our implementation makes use of this incremental solving capability when encoding successive VDC allocations; doing so results in a substantial performance improvement. As we will describe in the next section, the MONOSAT back-end for NETSOLVER takes similar steps to avoid having to re-encode the full constraints after each allocation.

In summary, we demonstrated how our ILP model encodes VDC allocation as a set of constraints. We combined CPU and RAM resource constraints and cardinality constraint to encode the local constraints, **L**. We encoded global constraints, **G**, using multi-commodity flow constraints that combine network flow constraints on switches and servers. Given that a datacenter consists of switches and servers, these constraints together enforce flow conservation on the entire datacenter.

2.3.2 Encoding Multi-path VDC Allocation in SMT

In contrast to ILP solvers, SMT solvers have not traditionally been applied to large multi-commodity flow problems. As a result, techniques for handling network flow problems efficiently in SMT are less mature and require some additional discussion. We now describe how MONOSAT [29] can be used to solve practical multi-commodity network flow problems.

MONOSAT is an SMT solver that extends quantifier-free first-order Boolean logic with highly efficient, built-in support for a wide set of *finite monotonic predicates* [29]. In particular, MONOSAT has built-in predicates for (single-commodity) *s-t* maximum flow. While this does not directly provide support for multi-commodity flows, we show that by expressing multi-commodity flows as a combination of



$$(c(p,q) > c(q,r)) \wedge$$
$$(c(q,r) + c(p,r) = 2) \wedge$$
$$(maxFlow_{p,r} \geq 2)$$

(a) Directed graph G **(b)** Formula to satisfy **(c)** Satisfying flow/capacity

Figure 2.2: Sample Max-flow Encoding in MONOSAT: **(a)** Example symbolic graph, with variable capacities $c(i, j)$ on each edge, **(b)** a formula constraining the graph, **(c)** a solution for the edge capacities, as well as a flow assignment to each edge that satisfies the max-flow constraint. The solution is presented as flow/capacity, e.g., 1/2 means that the edge capacity is 2 and the flow consumes 1 unit of bandwidth on that edge.

single-commodity maximum flow predicates, we can use MONOSAT to solve large multi-commodity flow problems: a first for SMT solvers. By combining this encoding for the global constraints **G** with a pseudo-Boolean encoding of the local constraints **L**, we are able to do multi-path VDC allocation with MONOSAT.

Intuitively, a finite monotonic predicate is a predicate for which increasing the value of its arguments can never change the value of the predicate from true to false, e.g., adding links to a network can only increase the connectedness of the network. MONOSAT supports many common graph constraints, such as reachability, shortest paths, minimum spanning trees, and (single commodity) maximum flows. MONOSAT also supports a subset of the theory of fixed-width bitvectors.

MONOSAT accepts formulas with one or more directed *symbolic graphs*, each of which is composed of a fixed set of nodes and symbolic edges (i, j). Each edge has an integer capacity, $c(i, j)$, which may be either a constant or a variable (a fixed-width bitvector). Finally, MONOSAT supports a number of common graph predicates, of which only one is relevant here: $maxFlow_{s,t,G} \geq f$, where G is a directed graph, s and t are nodes in G, and f is a constant integer or a bitvector term. This predicate is true if and only if the maximum s-t flow in G, *under assignment to the edge capacities associated with* G, is greater or equal to f.

As an example, consider the directed graph G shown in Figure 2.2(a), with variable integer capacities $c(i, j)$, and the formula in Figure 2.2(b). In this example, MONOSAT finds edge capacities that satisfy the constraints and also produces a flow satisfying the maximum flow predicate in Figure 2.2(c).

In the remainder of this section, we first describe how we model integer-value multi-commodity flow in terms of the built-in maximum flow predicates supported by MONOSAT; then we show how to use these multi-commodity flow constraints to express VDC allocation. More extensive discussion about MONOSAT can be found in Bayless et al. [29] and Sam Bayless's PhD dissertation [28].

Multi-commodity Flow in MONOSAT

There are many obvious ways to encode multi-commodity flows in SMT solvers. However, to the best of our knowledge, the one we present here is the only SMT encoding to scale to multi-commodity flow problems with thousands of nodes, i.e., datacenters with over 1000 servers and switches.

Given an arbitrary directed graph $G = (V, E)$, an integer capacity $c(i, j)$ for each edge $(i, j) \in E$, and a set of commodity demands K, where a commodity demand $k \in K$ is a tuple (s_k, t_k, d_k), representing an integer flow demand of d_k from source $s_k \in V$ to target $t_k \in V$, the *integral multi-commodity flow problem* is to find a feasible assignment of flows $f_k(i, j)$ such that each demand d_k is satisfied, while for each edge (i, j) the total flow of all demands (summed) is at most $c(i, j)$:

$$f_k(i,j) \geq 0, \quad \forall (i,j) \in E, k \in K$$

$$\sum_{k \in K} f_k(i,j) \leq c(i,j), \quad \forall (i,j) \in E$$

$$\sum_{j \in V} f_k(i,j) - \sum_{j \in V} f_k(j,i) = \begin{cases} 0, \text{if } i \notin \{s_k, t_k\} \\ d_k, \text{if } i = s_k \\ -d_k, \text{if } i = t_k \end{cases}, \forall i \in E, \forall (s_k, t_k, d_k) \in K$$

We instantiate symbolic graphs $G_{1..|K|}$ with the same topology as G. We set the capacities of each edge $(i, j)_k \in G_k$ to a new integer variable, $c(i, j)_k$, with constraint $0 \leq c(i, j)_k \leq c(i, j)$. Next, we assert that the capacities in each graph sum to no more than the original edge capacity: $\sum_{k=1}^{|K|} c(i, j)_k \leq c(i, j)$. Together, these constraints partition the original capacity graph into $|K|$ separate graphs, one for each demand. To complete the encoding, for each commodity demand (s_k, t_k, d_k), we use MONOSAT's built-in maximum flow constraints to assert that the maximum s_k–t_k

flow in G_k is at least d_k.

In our formulation, we explicitly enforce only that the maximum s_k–t_k flow in G_k is $\geq d_k$, as opposed to enforcing that the maximum flow is exactly d_k. Notice that a flow that is greater than d_k will necessarily contain a flow that is equal to d_k, and that an exact d_k flow can easily be recovered if necessary (e.g., with one extra application of any standard maximum flow algorithm). Alternatively, an extra, external "source" node can be added to the graph, with exactly one edge of capacity d_k leading to the original source node from this new, extra "source" node. This will ensure that the maximum possible s_k–t_k flow is at most d_k. We implement our constraints in this way to improve the performance of the underlying constraint solver. In MONOSAT, it is typically more efficient to enforce one-sided $(>, \geq, \leq, <)$ constraints, rather than two-sided $(=, \neq)$ constraints, because all theory predicates in MONOSAT must be monotonic, so equality needs to be implemented as two (individually monotonic) one-sided comparisons.

Multi-path VDC Allocation in MONOSAT

We now show how the global and local constraints described in Section 2.2 can be encoded into MONOSAT and used to perform VDC allocation. As a running example, we consider a small VDC and datacenter illustrated in Figure 2.3. Note that for readability, our examples are much smaller than the ones we consider in our evaluation (Section 2.4).

The global constraints **G** can be encoded as a multi-commodity flow as described earlier, with up to $|VM|^2$ commodity demands (one for each bandwidth tuple $(u, v, bandwidth) \in R$). However, we can greatly improve on this by merging bandwidth constraints that share a common source into a single commodity demand: Given a set of bandwidth constraints $(u, v_k, bandwidth_k) \in R$ with the same source u, we can convert these into a single commodity demand, by adding an extra node $w \notin VM$, along with edges (v_k, w) with capacity $bandwidth_k$. The commodity demands $(u, v_k, bandwidth_k)$ can then be replaced by a single commodity demand $(u, w, \sum_k bandwidth_k)$. As there are at most $|VM|$ distinct sources in R, this reduces the number of demands from $|VM|^2$ in the worst case to $|VM|$ demands. Converting a single-source, multi-destination flow problem into a single-

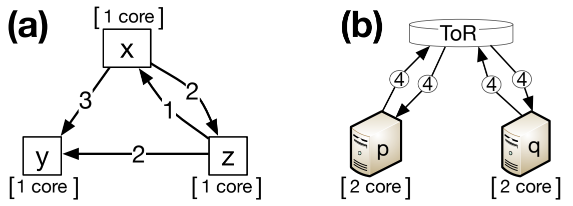
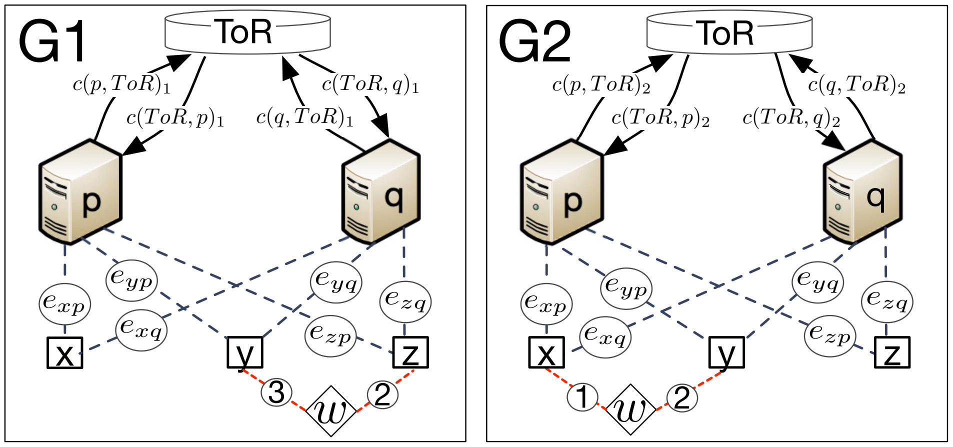


Figure 2.3: Sample Multi-path Encoding in MONOSAT: (**a**) A VDC with three VMs, and four directed bandwidth constraints. In this example, each VM requires 1 core and has no RAM requirements. VM *x* requires 3 Gbps of outgoing bandwidth to VM *y* and 2 Gbps to VM *z*. VM *z* also has bandwidth requirements to VM *x* and *y*, while VM *y* requires no outgoing bandwidth. (**b**) A datacenter with two servers and one Top-of-Rack (ToR) switch. Each server has 2 cores, and has 4 Gbps of bandwidth available to and from the switch.

source, single-destination maximum flow problem is a well-known transformation and safely preserves the maximum possible flow to each destination.

In our example from Figure 2.3, the VDC has four directed bandwidth requirements, but only two distinct bandwidth sources (VM *x* and VM *z*). We can safely merge these four bandwidth requirements into two multi-commodity flow constraints. We construct two graphs (shown in Figure 2.4), $G1$ and $G2$. For each $v \in VM$ and each server $s \in S$, we add a directed symbolic edge e_{vs} from v to s with unlimited capacity to G; this edge controls the server to which each VM is allocated. Next, we assert (using a cardinality constraint) that for each VM v, exactly one edge e_{vs} is enabled, so that the VM is allocated to exactly one server: $\forall v \in VM : \sum_s e_{vs} = 1$. Using the multi-commodity flow encoding described above, we assert that the multi-commodity flow in G satisfies $(u, v, bandwidth)$ for each commodity requirement. The above constraints together enforce global constraints **G**; to enforce local constraints **L**, we use pseudo-Boolean constraints (using the efficient SAT encodings described in Eén and Sorensson [49]) to assert: $\left(\left(\sum_v cpu(v) \leq cpu(s)\right) \wedge \left(\sum_v ram(v) \leq ram(s)\right)\right)$, i.e., that the set of VMs allocated to each server (which may be more than one VM per server) will have sufficient CPU core and RAM resources available on that server. Together these assertions enforce constraint set **L** from our problem definition. A satisfying solution to our running example, implementing all of the above constraints, is shown in Figure 2.5.



$(e_{xp} \vee e_{xq}) \wedge (e_{yp} \vee e_{yq}) \wedge (e_{zp} \vee e_{zq})$ Each VM is assigned to a server (1)

$(\neg e_{xp} \vee \neg e_{xq}) \wedge (\neg e_{yp} \vee \neg e_{yq}) \wedge (\neg e_{zp} \vee \neg e_{zq})$ No VM is assigned to more than 1 server (2)

$(\neg e_{xp} \vee \neg e_{yp} \vee \neg e_{zp}) \wedge (\neg e_{xq} \vee \neg e_{yq} \vee \neg e_{zq})$ Each server has sufficient CPUs for its VMs (3)

$$c(p,ToR)_1 + c(p,ToR)_2 \leq 4$$
$$c(ToR,p)_1 + c(ToR,p)_2 \leq 4$$
$$c(q,ToR)_1 + c(q,ToR)_2 \leq 4$$
$$c(ToR,q)_1 + c(ToR,q)_2 \leq 4$$
Capacities on each physical edge sum to ≤ c (4)

$G1.maxflow(x,w) \geq 5$ Maximum flow from x to w in G1 is ≥ 5 (5)

$G2.maxflow(z,w) \geq 3$ Maximum flow from z to w in G2 is ≥ 3 (6)

Figure 2.4: Sample Multi-commodity Flow Constraints in MONOSAT. We show two symbolic graphs $G1, G2$, and the corresponding constraints enforcing allocation for the VDC and datacenter in Figure 2.3. Edges e_{vs} control which VMs are placed on which servers, and have the same assignments in the two graphs. Edges marked with integers have constant capacities; edges e have unlimited capacity, and edges c have variable, non-negative integer capacities. In this example, constraints 2 and 3 are simple enough to be expressed with a few clauses, but for more complex examples we would use pseudo-Boolean constraints. Constraint 4 is enforced using bitvector arithmetic, while 5 and 6 use the built-in maximum flow predicates of MONOSAT.

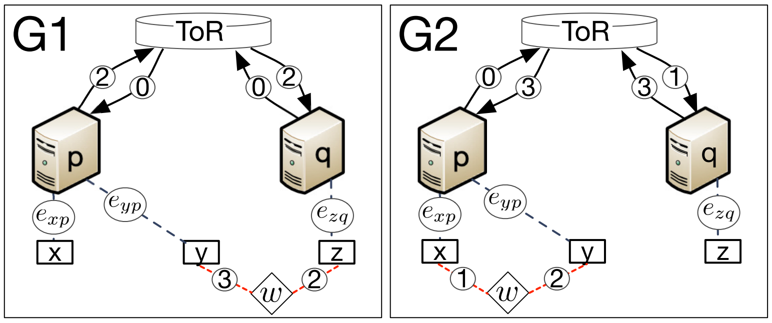


Figure 2.5: SAT Allocation in MONOSAT. A satisfying assignment to the constraints in Figure 2.4; only the edges that are assigned to "true" are shown. Notice that the e_{vs} assignments must be the same in the two graphs. The capacity assignments c are each at least large enough to allow for the required flow between the assigned VMs (but may be larger than required, as is the case for $c(ToR, q)$), and the individual capacities assigned to each edge across the two graphs sum to at most the bandwidth available on each edge of the datacenter (4, in this case).

These encodings are novel contributions and critical to NETSOLVER's performance with an SMT back-end; however, they are empirically efficient only because MONOSAT (unlike other SMT solvers) has built-in support for network flow constraints. As we show in our evaluation in Section 2.4, carefully crafted encodings alone, such as the one developed in Z3-AR [135], are not competitive. Instead, fundamental improvements in the constraint solver, such as the ones we use in MONOSAT, are necessary.

Reusing Constraints

As with the ILP-based approach, it is important to use incremental solving in the SMT-based approach as well. The preceding discussion assumes that the VDC topology is constant and known in advance. However, in practice, it is typically the case that one will want to allocate VDCs of differing topologies. We briefly summarize here how we extend the above encoding to support this use case and demonstrate its benefit in our evaluation in Section 2.4.

Many SAT solvers, including MONOSAT, support an "assumption" mechanism [119] allowing for a formula to be repeatedly solved under multiple, differing restricted portions of the search space (that is, under an *assumed* set of assignments). To support allocating VDCs of differing topologies, without needing to re-encode the entire set of constraints in a new solver at each allocation, which would be prohibitively expensive, we initially encode a VDC topology that is the superset of all the VDCs to be allocated. Then, for each individual VDC to allocate, we use the assumption mechanism to temporarily disable portions of that superset VDC topology in the formula, such that only the edges corresponding to the current VDC to be allocated remain enabled in the solvers search space. In this way, we can efficiently reuse the same solver to perform each allocation, while supporting VDCs of multiple sizes as well as supporting the deallocation of previously allocated VDCs.

2.4 Evaluation

We compare the performance of the ILP and SMT versions of NETSOLVER to that of SecondNet's VDCAlloc [59], a sound VDC allocation algorithm with end-to-end bandwidth allocation, and the Z3-based abstraction-refinement procedure from Yuan et al. [135], which resembles our approach in that it makes use of a constraint solver (SMT). We call NETSOLVER with the ILP back-end NETSOLVER-ILP and with SMT back-end NETSOLVER-SMT.

SecondNet's VDCAlloc algorithm ('SecondNet', except where ambiguous) is an incomplete, heuristic-based algorithm that can allocate VDCs on datacenters with hundreds of thousands of servers. As SecondNet is based on bipartite matching, it fundamentally cannot allocate more than one VM in each VDC to any given server. Furthermore, it can fail to find a feasible allocation, especially in heavily utilized networks, because it performs allocation in an incomplete, greedy fashion. As we will demonstrate, under many realistic circumstances, this happens quite frequently, leading to substantially lower datacenter utilization than can be achieved with a complete method, such as NETSOLVER.

The constraint-solving-based work from Yuan et al. [135] introduced two approaches for performing single-path VDC allocation with end-to-end bandwidth, using the general-purpose SMT solver Z3 [43]. Like almost all SMT solvers, Z3 has no built-in support for network flow predicates. Therefore, to use Z3 for VDC allocation, the global bandwidth and connectivity constraints have to be expressed using a lower-level logical formulation. The first such encoding, which we call Z3-generic, can handle any datacenter topology but scales extremely poorly [135]. The second approach, which we call Z3-AR, makes use of an optimized abstraction-refinement technique; while substantially more scalable than the generic encoding, it is restricted to datacenters with tree topologies. In preliminary experiments, not reported here, we confirmed that Z3-generic performed poorly, often failing to find any allocations within a 1-hour timeout on the benchmarks used in our experiments. We compare NETSOLVER to Z3-AR, the faster and more scalable VDC allocator by Yuan et al. [135].

2.4.1 Methodology

We use the *static packing metric* to evaluate each VDC scheduler's quality. The static packing metric measures how many VDCs (of the same size) a scheduler is able to pack into an empty datacenter. This metric was also used in prior work [59, 135], and it captures datacenter utilization. We give two input files to the scheduler: *workload* and *datacenter*. The workload file contains VDCs that have to be allocated. The datacenter file describes the datacenter topology that consists of servers, which have CPU and RAM resources, and switches, which connect servers (and other switches) with network links with predefined bandwidth.

In each experiment, the algorithms repeatedly allocate VDCs to the datacenter until they are unable to make further allocations or until a 1 CPU hour timeout[2] is reached. The timeout is for an entire experiment, not for an individual VDC allocation. For example, if allocating each VDC takes one second, the experiment ends after allocating 3,600 VDCs (or earlier, if the algorithm cannot place a VDC).

Except where noted, experiments were run on a server with a 2.40 GHz (10 MB L3 cache) Intel Xeon E5-2407 v2 processor with 8 cores across 2 NUMA nodes and hyperthreading disabled. The server uses Ubuntu 16.04.6 LTS with 96 GB RAM that is uniformly distributed (48 GB each) across both NUMA nodes. All experiments are limited to 80 GB RAM; none actually consumed 80 GB.

[2]Timeout is enforced with "time-limit" option in the runlim package http://manpages.org/runlim.

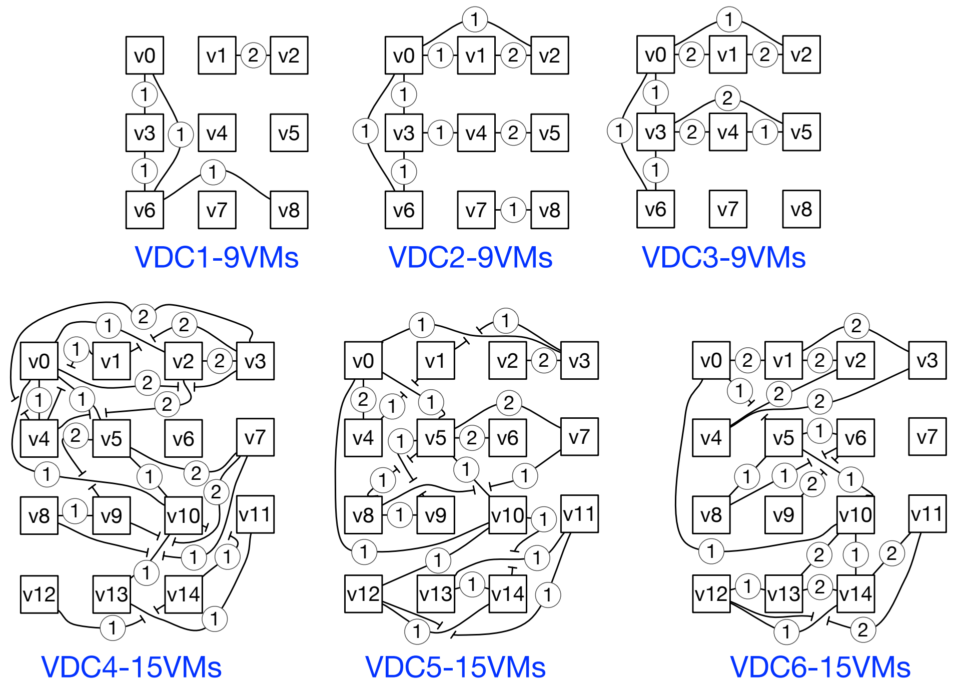


Figure 2.6: VDC Topologies. The circled label on the virtual links signifies the link's bandwidth in Gbps.

2.4.2 Comparison on Datacenters with Tree Topologies

We reproduce and extend experiments from Yuan et al. [135], in which a series of identical VDCs is allocated one-by-one to tree-structured datacenters, until the solver is unable to make further allocations (or a timeout of 1 CPU hour is reached). There are 6 VDC instances considered, each in a separate experiment, three consisting of 9 VMs each and three consisting of 15 VMs each. Each VDC has a unique, randomly generated topology, as shown in Figure 2.6.[3]

We obtained the original implementations of Z3-AR from Yuan et al. [135] and SecondNet from the SecondNet authors [59]. In all experiments in this subsection, VDCs in an experiment have the same topology; this is a restriction introduced

[3]Note that here and in the remainder of this chapter, we allocate individual VDCs one at a time, without looking ahead at the remaining VDCs that have yet to be allocated. This online allocation process can potentially result in a sub-optimal total number of allocations, even though our approach is complete for individual VDC allocations. Our approach is similar in this respect to the previous works of Yuan et al. [135] and SecondNet [59], to which we compare.

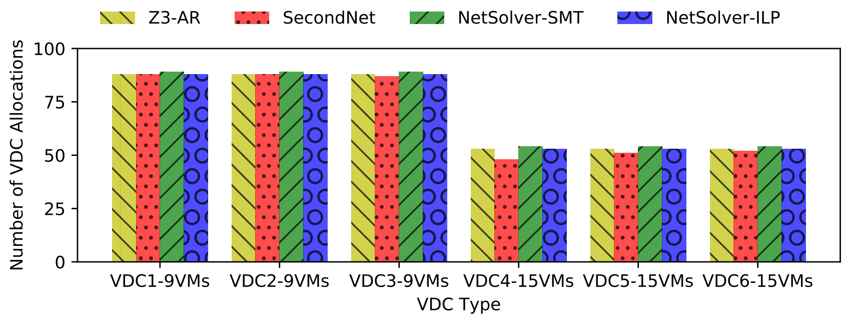


Figure 2.7: VDC Allocation Comparison on Small Tree Datacenter. We show the total number of consecutive VDCs allocated by different algorithms on a datacenter with tree topology from Yuan et al. [135].

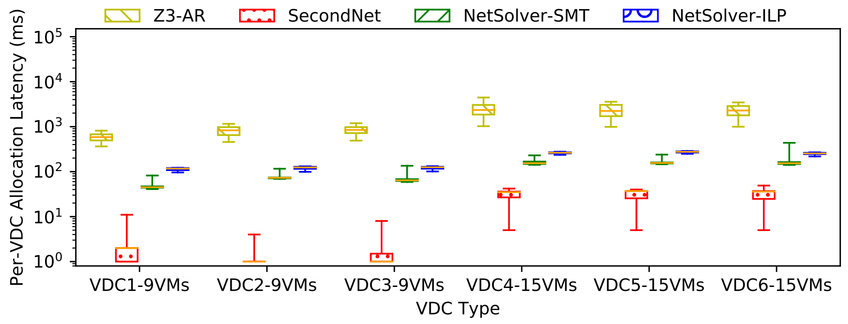


Figure 2.8: Latency Comparison on Small Tree Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.7. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

here for compatibility with the solvers from Yuan et al. [135]. Although this restriction makes the experiment less representative of real-world use cases, it allows all three of the constraint based approaches (Z3-AR, NETSOLVER-SMT, and NETSOLVER-ILP) to avoid substantial costs that would otherwise be incurred to support changing VDC topologies. In our subsequent experiments, Section 2.4.3 onward, we consider cases where VDC topologies vary.

Figure 2.7 compares the number of VDCs allocated by the four algorithms in a small datacenter with 200 servers, where each server has 4 cores. This datacenter

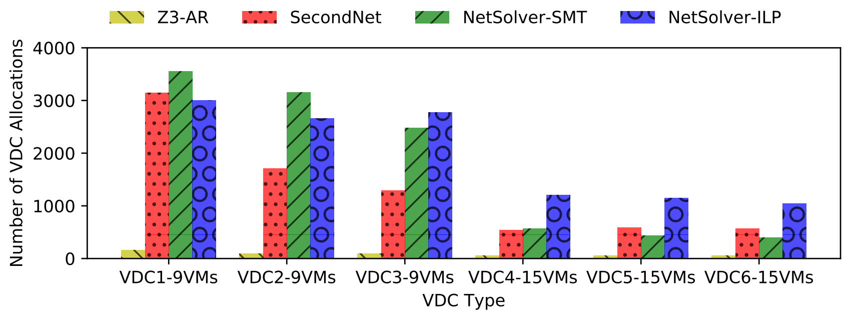


Figure 2.9: VDC Allocation Comparison on Big Tree Datacenter. We show the total number of consecutive VDCs allocated by different algorithms on a datacenter with tree topology from Yuan et al. [135].

has a tree topology and is from Yuan et al. [135]. Figure 2.7 shows allocation results for six different VDC topologies: the first three with 9 VMs each (VDC1, VDC2, VDC3) and the other three with 15 VMs each (VDC4, VDC5, VDC6), as shown in Figure 2.6. In Figure 2.7, all four algorithms allocate similar number of VDCs, because the datacenter is not big enough to highlight the difference between the algorithms. Note that the number of allocated VDCs has decreased on the larger VDCs because the VDC size grew from 9 VMs to 15 VMs.

Figure 2.8 compares the per-VDC allocation latency for the VDC allocations shown in Figure 2.7. Unlike Figure 2.7, Figure 2.8 shows that the time each algorithms takes to allocate VDCs differs significantly. SecondNet, being heuristic and incomplete, has two orders of magnitude lower median VDC allocation latency than NETSOLVER, and three orders of magnitude lower median VDC allocation latency than Z3-AR. Note that the latency difference between the four algorithms becomes less significant as VDC size grows. Figure 2.8 shows that the median latency with SecondNet is at least two orders of magnitude faster than the other algorithms in VDCs with 9 VMs and decreases to one order of magnitude for VDCs with 15 VMs. At the same time, NETSOLVER-SMT is consistently faster than NETSOLVER-ILP for all VDCs, while both of these are an order of magnitude faster than Z3-AR.

Figure 2.9 compares the four algorithms in a bigger datacenter. This datacenter also has a tree topology and is from Yuan et al. [135]. There are 2000 servers in

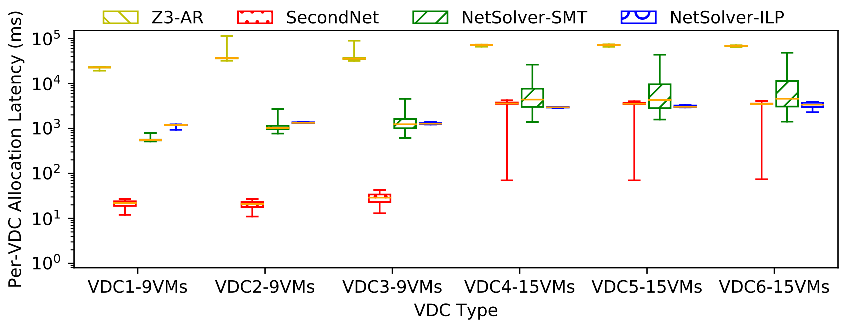


Figure 2.10: Latency Comparison on Big Tree Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.9. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

this datacenter, each server has 16 cores. Unlike our experiments with the small datacenter (200 servers) in Figure 2.7, there is a significant difference between the number of VDCs that the algorithms allocate. NETSOLVER greatly outperforms SecondNet and Z3-AR, often allocating two or even three times as many VDCs on the same datacenter. Figure 2.10 compares per-VDC allocation latency for the VDC allocations shown in Figure 2.9.

Figure 2.9 shows that in most cases NETSOLVER-ILP performs better than NETSOLVER-SMT, but both versions of NETSOLVER scale to thousands of servers, with median per-VDC allocation latency of a few seconds or less. On instances with smaller VDCs, NETSOLVER-SMT tends to have both lower VDC allocation latency and more VDC allocations than NETSOLVER-ILP, while on instances with larger VDCs, NETSOLVER-ILP performs substantially better than NETSOLVER-SMT, sometimes achieving more than twice the allocations of NETSOLVER-SMT. Figure 2.10 also shows that SecondNet's median per-VDC allocation latency becomes comparable to NETSOLVER's latency when both VDCs (15 VMs) and datacenter are large (2000 servers).

Figure 2.11 compares the performance of the four algorithms over time. We plot the number of VDC allocations that the algorithms make until they are unable to make further allocations or until a 1 CPU hour timeout is reached. Figure 2.11(a) shows results for a small VDC with 9 VMs (VDC1 in Figure 2.9) and

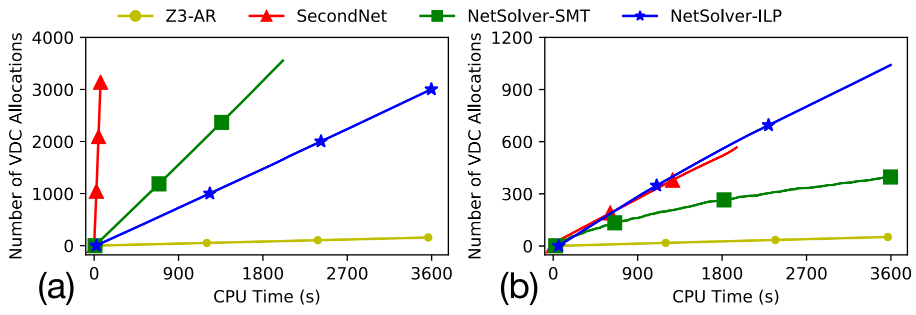


Figure 2.11: VDC Allocations Over Time on Big Tree Datacenter. We show the number of VDC allocations by four algorithms on a datacenter with tree topology from Yuan et al. [135]. The datacenter is the same as the one we used in Figure 2.9. There are 2000 servers, each with 16 cores. (a) reports results for VDC1 with 9 VMs in Figure 2.9. (b) reports results for VDC6 with 15 VMs in Figure 2.9.

Figure 2.11(b) shows results for a big VDC with 15 VMs (VDC6 in Figure 2.9). As we can see in Figure 2.11(a), SecondNet makes all of its VDC allocations quickly and quits. On the other hand, Z3-AR has high VDC allocation latency and therefore is unable to make many VDC allocations within the 3600 second budget. NETSOLVER-SMT and NETSOLVER-ILP steadily allocate VDCs within the time budget and eventually exceed the number of VDC allocations made by the fast, but incomplete, SecondNet. Figure 2.11(a) and Figure 2.11(b) show that SecondNet's and NETSOLVER-SMT's latency profiles are significantly affected by the VDC size. Although Z3-AR's allocations over time remains mostly unchanged across two figures, SecondNet's VDC allocation frequency becomes comparable to that of NETSOLVER-ILP's for VDCs with 15 VMs.

2.4.3 Comparison on FatTree and BCube Datacenters

The second experiment we conducted is based on experiments in the original SecondNet paper [59]. Note Z3-AR is restricted to tree topologies so it could not be included in these experiments.

The SecondNet benchmark instances are extremely large, e.g., in one case exceeding 100,000 servers, but also extremely easy to allocate: the available bandwidth per link is typically $\geq 50\times$ the requested virtual link (vlink) bandwidths in the VDC, so with only 16 cores per server, the bandwidth constraints are mostly irrelevant. For such easy allocations, the fast, incomplete approach that SecondNet uses is the better solution. Although this might be acceptable when a cloud provider is willing to leave a significant portion of their datacenter network bandwidth underutilized, it seems an unlikely scenario in practice, because cloud providers report the datacenter network to be a computation bottleneck with ToR switch uplinks frequently operating above 80% utilization [58]. Therefore, we scaled the SecondNet datacenters down to 432–512 servers: a scale where datacenter network utilization levels approximate a realistic scenario. This datacenter size is realistic for many small scale datacenters.

Unlike the earlier experiments in Section 2.4.2, each experiment in this subsection uses non-identical VDC topologies. We generated sets of 10 VDCs each of several sizes (6, 9, 12 and 15 VMs), following the methodology described in Yuan et al. [135], which generates VDCs that resemble connectivity in the distributed storage system. Here, a VM connects to all other VMs in the VDC with a vlink whose bandwidth ranges from 0 to 2 at random. Thus, we generate 10 VDCs of each VDC size with randomized topology. These VDCs have proportionally greater bandwidth requirements than those originally considered by SecondNet, requiring 5–10% of the smallest physical link capacities in the datacenter. The resulting VDC instances exhibit non-trivial bandwidth constraints. For each of these sets of VDCs, we then repeatedly allocated VDC instances of the same size, e.g., VDCs with 6 VMs, in random order until the datacenter is saturated, or the 1 CPU hour timeout is reached.

Figure 2.12 and Figure 2.13 compare the total number of VDC allocations and the per-VDC allocation latencies by SecondNet and NETSOLVER on a datacenter with FatTree topology with 432 servers. Each server has 16 cores. Figure 2.12 shows that SecondNet consistently allocates fewer VDCs. NETSOLVER-ILP and NETSOLVER-SMT allocate comparable numbers of VDCs when the VDC size is small, but start diverging as it grows. For example, when the VDCs have 15 VMs, NETSOLVER-ILP allocates over $2\times$ more VDCs than NETSOLVER-SMT. Figure 2.13 reinforces our observations from Section 2.4.2: SecondNet's median latency is orders of magnitude lower than both versions of NETSOLVER, and NETSOLVER-ILP has lower median VDC allocation latencies than NETSOLVER-SMT

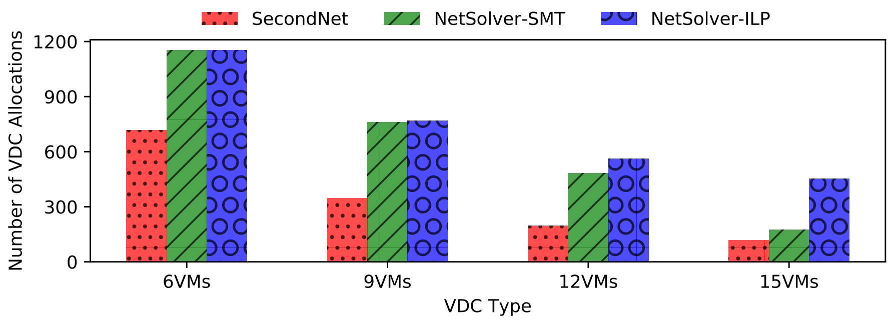


Figure 2.12: VDC Allocation Comparison on FatTree Datacenter. We show the total number of consecutive VDCs allocated by different algorithms on a datacenter with FatTree topology from SecondNet [59].

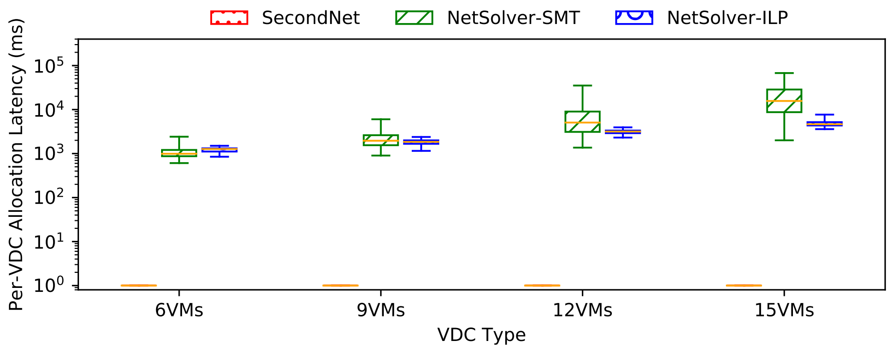


Figure 2.13: Latency Comparison on FatTree Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.12. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

on larger VDC instances.

Figure 2.14 and Figure 2.15 show results for the BCube datacenter from SecondNet [59]. The datacenter has 512 servers, each with 16 cores. The VDC instances used in the BCube experiments are identical to the one we used in the FatTree datacenter (Figure 2.12). These figures confirm our findings from the FatTree experiments. NETSOLVER-ILP has the highest number of VDC allocations, as shown in Figure 2.14, while achieving lower median per-VDC allocation latencies than NETSOLVER-SMT, as shown in Figure 2.15.

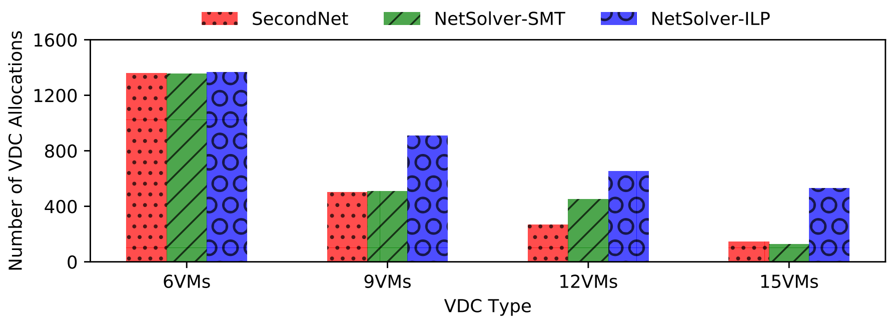


Figure 2.14: VDC Allocation Comparison on BCube Datacenter. We show the total number of consecutive VDCs allocated by different algorithms on a datacenter with BCube topology from SecondNet [59].

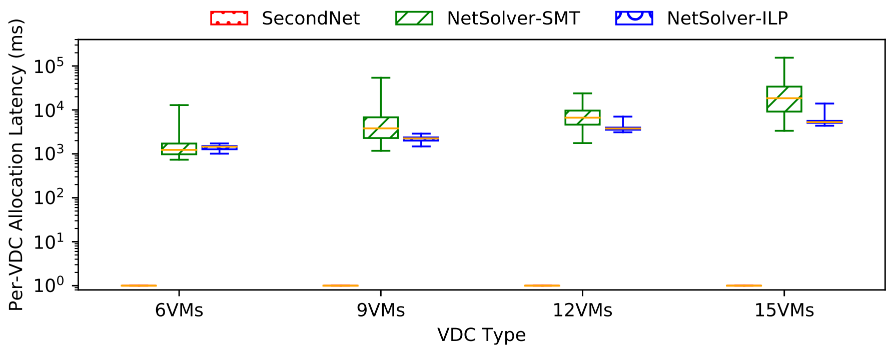


Figure 2.15: Latency Comparison on BCube Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.14. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

Figure 2.16 shows VDC allocations over time in the FatTree and BCube datacenters. This figure also reinforces our observations from earlier experiments with tree datacenters in Section 2.4.2. SecondNet allocates all of its VDCs quickly, but the number of VDCs it allocates are 2–3× fewer than what NETSOLVER steadily allocates during 3600 seconds. We also see that NETSOLVER-ILP can make more allocations more quickly than NETSOLVER-SMT.

In all experiments of this chapter, all solvers are restricted to a single CPU core. However, as Gurobi supports parallel execution, we also tried running this

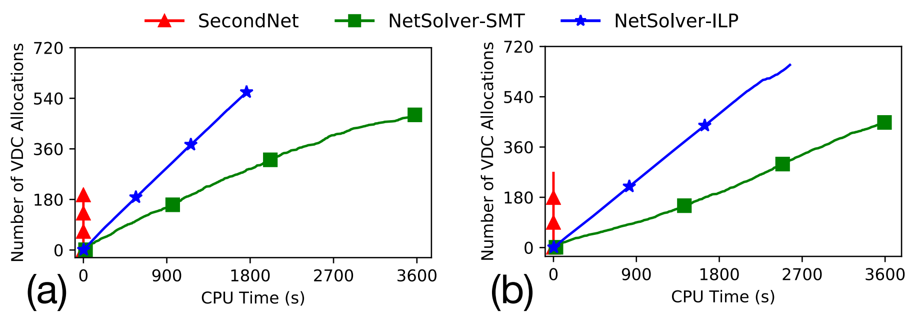


Figure 2.16: VDC Allocations Over Time on FatTree and BCube Datacenters. We show the number of VDC allocations by three algorithms on datacenter topologies from SecondNet [59]. (a) reports results for the FatTree datacenter with 432 servers. (b) reports results for the BCube datacenter with 512 servers. A server in both datacenters has 16 cores. Both figures show results for allocating VDC instances with 12 VMs.

experiment with Gurobi's multi-threaded support enabled, using up to 8 CPU cores. We found that the results were similar to those for single-threaded execution and in particular, neither consistently better nor worse, so we report only single-threaded execution results.

2.4.4 Comparison on Commercial Datacenters

The comparisons so far showed how NetSolver compares to existing VDC allocation tools on several datacenter network topologies from the VDC literature. To examine NetSolver's allocation behavior on real workloads, we also considered a deployment of a commonly used Hadoop VDC, on a set of commercial datacenter topologies. We collaborated with a private cloud provider, ZeroStack Inc. [136], to devise a practical Hadoop VDC to run the Terasort workload [39]. Each Hadoop VDC consists of a single leader VM connected to 3–11 worker VMs. We considered five different VM sizes, ranging from 1 CPU and 1 GB RAM, to 8 CPUs and 16 GB of RAM. The worker VMs were selected at random from this set, with the leader VM also randomized but always at least as large as the largest worker VM. The Hadoop leader VM connects to other worker VMs in a tree topology, analogous to the sample VDC in Figure 1.2, where each virtual link has either 1 Gbps or 2 Gbps bandwidth.

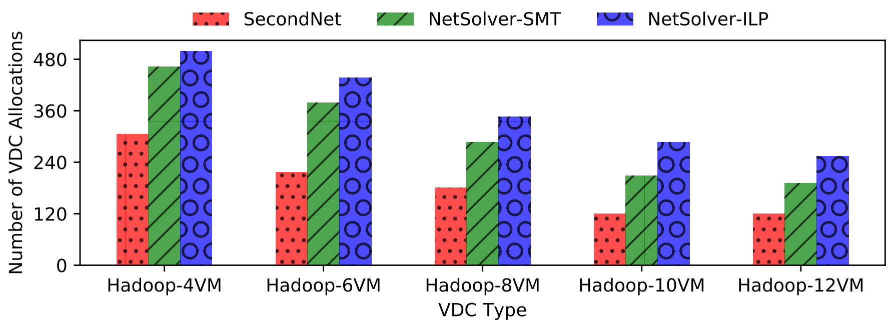


Figure 2.17: VDC Allocation Comparison on US-West1 Datacenter with 1200 Servers. We show the total number of consecutive VDCs allocated by different algorithms.

The datacenter topologies were provided by another company, who requested to remain anonymous. This company uses a private cloud deployed across four datacenters in two geographic availability zones (AZs): *US-West* and *US-Middle*. Each datacenter contains between 280 and 1200 servers, spread across one to four clusters with 14 and 40 racks. Each server has 16 cores, 32 GB RAM, 20 Gbps network bandwidth (via two 10 Gbps links). The network in each datacenter has a leaf-spine topology, where all ToR switches connect to two distinct spine switches over 40 Gbps links each (a total of two links with 80 Gbps; one on each spine switch) and spine switches are interconnected with four 40 Gbps links each. For each cluster, there is a gateway switch with a 240 Gbps link connected to each spine switch. Appendix A shows topologies of all four datacenters.

We evaluated SecondNet and NETSOLVER in this setting, consecutively allocating Hadoop VDCs of several sizes, ranging from 4 to 12 VMs, until no further allocations could be made. Note that in addition to using a realistic datacenter topology, the CPU/memory, bandwidth values, and the VDCs being allocated are all real-world VDCs derived from Hadoop jobs. By contrast, the previous experiments used synthetic VDCs from Yuan et al. [135] and SecondNet [59].

Figure 2.17 and Figure 2.18 show the results for the largest of these datacenters (US-West1), results for the smaller datacenters were similar, e.g., Figure 2.19 and Figure 2.20 (US-Middle1). As observed in our previous experiments, although SecondNet had much lower VDC allocation latency than either version of NET-

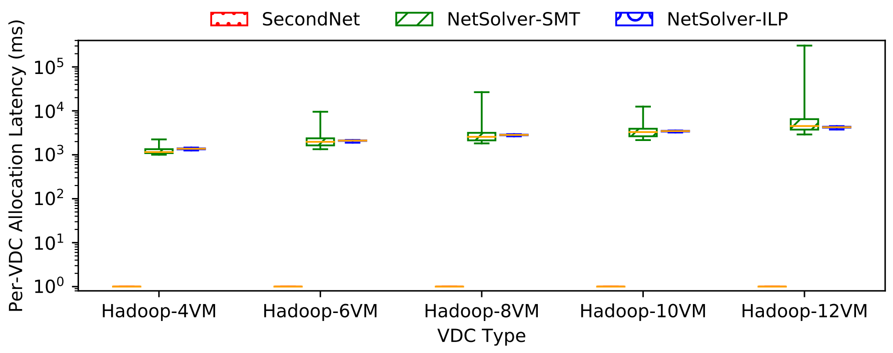


Figure 2.18: Latency Comparison on US-West1 Datacenter with 1200 Servers. We show per-VDC allocation latency distribution for allocations shown in Figure 2.17. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

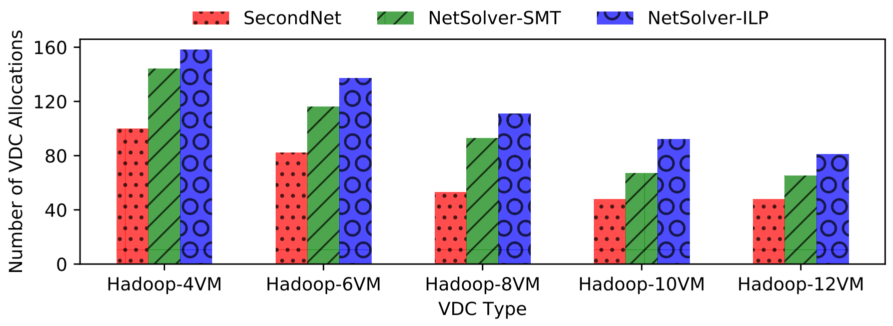


Figure 2.19: VDC Allocation Comparison on US-Middle1 Datacenter with 800 Servers. We show the total number of consecutive VDCs allocated by different algorithms.

SOLVER, NETSOLVER's per-VDC allocation latency was typically just a few seconds, which might be reasonable for long-running applications, such as the Hadoop jobs considered here. Again, NETSOLVER was able to allocate many more VDCs than SecondNet (here, 1.5–2× as many), across a range of datacenters and VDC sizes, including a commercial datacenter with over 1000 servers. Moreover, with increasing VDC size, NETSOLVER was able to allocate many more VMs, while respecting end-to-end bandwidth constraints. Often NETSOLVER allocated several

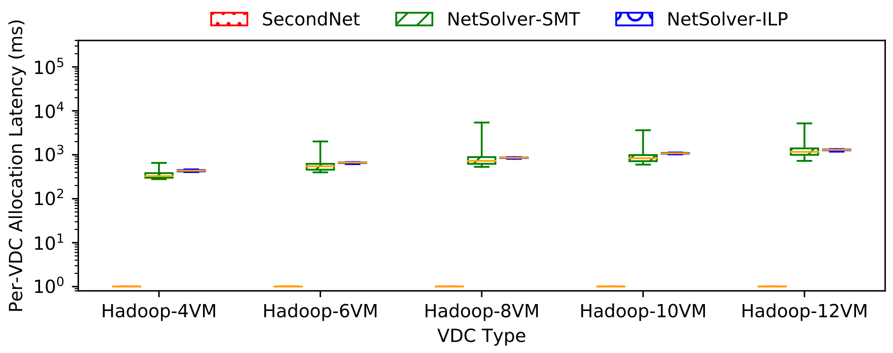


Figure 2.20: Latency Comparison on US-Middle1 Datacenter with 800 Servers. We show per-VDC allocation latency distribution for allocations shown in Figure 2.19. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

times as many VDCs as SecondNet, and in extreme cases, it found hundreds of allocations, while SecondNet was unable to make any allocations (not shown for brevity). Similarly, keeping the VDC the same size, but doubling the bandwidth requirements of each VM greatly decreased the number of allocations made by SecondNet, while NETSOLVER showed considerably more robust performance in these high network utilization settings.

Figure 2.21 shows VDC allocations over time in the commercial datacenters. This figure reinforces our observations from the earlier experiments. SecondNet allocates all of its VDCs quickly, but the number of VDCs it allocates are 2–3$\times$ less than what NETSOLVER can allocate over a longer time. Note that in these experiments the datacenter filled up long before the 3600s time budget was reached.

The experiments on FatTree, BCube, and the commercial datacenter networks reinforce our observations from the earlier experiments with synthetic datacenter tree topologies: both versions of NETSOLVER improve greatly on state-of-the-art VDC allocation, i.e., SecondNet and Z3. Further, the ILP version of NETSOLVER generally out-performs the SMT version, typically allocating 10–30% more VDCs.

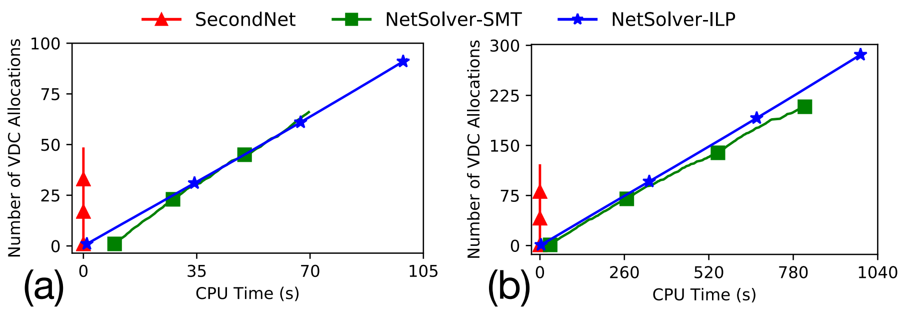


Figure 2.21: VDC Allocations Over Time on Commercial Datacenters. We show the number of VDC allocations by three algorithms. (a) reports results for the US-Middle1 datacenter with 800 servers. (b) reports results for the US-West1 datacenter with 1200 servers. Both figures show results for allocating Hadoop VDCs with 10 VMs. Note scale difference in axes across (a) and (b) figures.

2.4.5 Comparison to Virtual Network Embedding Approaches

In addition to the VDC allocation tools we considered above, we also compare to several state-of-the-art virtual network embedding (VNE) tools, as implemented in the VNE testing framework ALEVIN [51]. We provide these comparisons mainly for reference, as the VNE tools we consider here were neither designed nor optimized for allocating to these large and sparsely connected networks. As VNE algorithms are technically capable of performing VDC allocation, it is relevant to ask how they perform in this setting.

The VNE experimental framework we tested, ALEVIN [51], uses a GUI, and so we employed a (significantly faster) 3.4 GHz Intel Core-i7-2600K processor with 32 GB of RAM for these VNE experiments. Moreover, ALEVIN reports only the median per-VDC allocation latency. So, we show only median latencies for all algorithms (in Figure 2.23 and Figure 2.25).

In Figure 2.22 and Figure 2.24, we show two variants of the "Greedy Allocation Resources" algorithm from Yu et al. [134]. The PS ("path-splitting") variant supports multi-path allocation, while the SP ("shortest-paths") variant does not. Both of these are greedy, incomplete, linear programming based algorithms, and are appropriate to consider as they are two of the fastest and simplest VNE algorithms from the literature. Unlike SecondNet, both of these algorithms do support allocat-

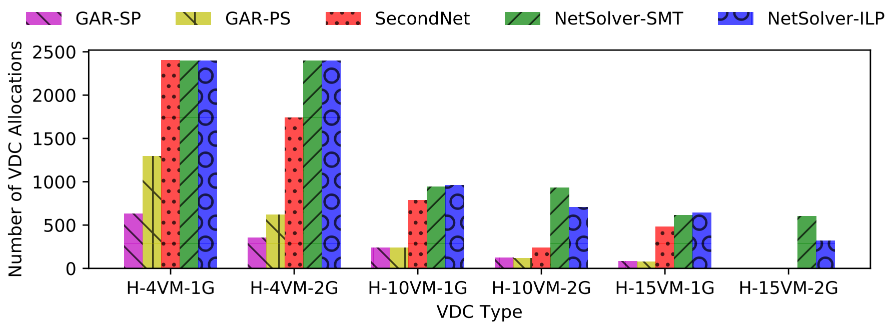


Figure 2.22: VDC Allocation Comparison with VNE Approaches on US-West1 Datacenter with 1200 Servers. We show the total number of consecutive VDCs allocated by different algorithms. The VNE solvers perform poorly in this setting, achieving a small fraction of the allocations that NETSOLVER-SMT or NETSOLVER-ILP achieves.

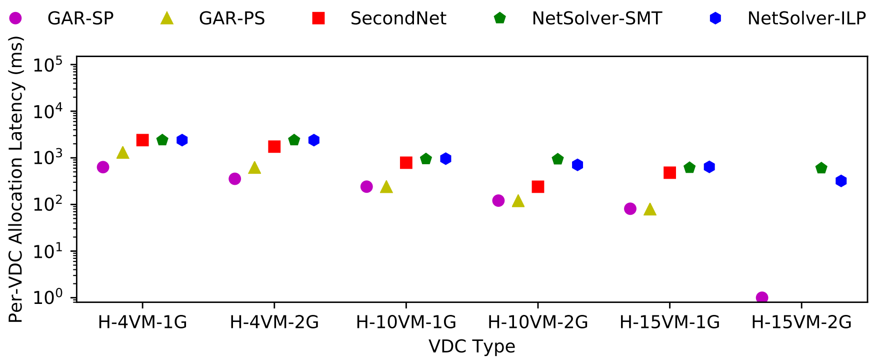


Figure 2.23: Latency Comparison with VNE Approaches on US-West1 Datacenter with 1200 Servers. We show the median per-VDC allocation latency for allocations shown in Figure 2.22. Note that no latency is reported for SecondNet and GAR-PS in "H-15VMs-2Gbps" because they are unable to make any VDC allocations there.

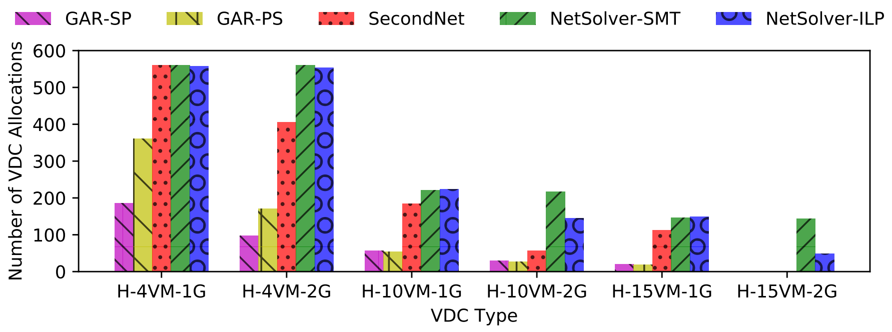


Figure 2.24: VDC Allocation Comparison with VNE Approaches on US-West2 Datacenter with 280 Servers. We show the total number of consecutive VDCs allocated by different algorithms. The VNE solvers perform poorly in this setting, achieving a small fraction of the allocations that NETSOLVER-SMT or NETSOLVER-ILP achieves.

ing multiple VMs per server. We applied these algorithm to two of the largest (US-West1 datacenter with 1200 servers in Figure 2.22) and smallest (US-West2 datacenter with 280 servers in Figure 2.24) commercial datacenters from Section 2.4.4, on the same VDC instances. Note that due to limitations in the ALEVIN platform, for these experiments, we consider just a single VDC instance of each size, rather than a set of such instances. Figure 2.22 and Figure 2.24 show that these two VNE algorithms, while faster, perform worse than both SecondNet and NETSOLVER, in many cases finding less than a quarter of the allocations of either.

We also tested several variants of three other families of state-of-the-art VNE algorithms from the ALEVIN framework: RW-MM-SP/PS [37], DViNE [38], and ASID [87]. Unfortunately, none of these were able find *any* allocations within several hundred seconds. This strongly suggests that at least the VNE algorithms we evaluated are not sufficiently scalable for VDC allocation. Our findings here are consistent with those reported in the SecondNet paper [59].

2.4.6 Allocation Robustness

In the above experiments, we showed that across many conditions, NETSOLVER was able to make many (often hundreds) more allocations than SecondNet or Z3-AR. One may wonder whether these additional allocations are the result of NET-

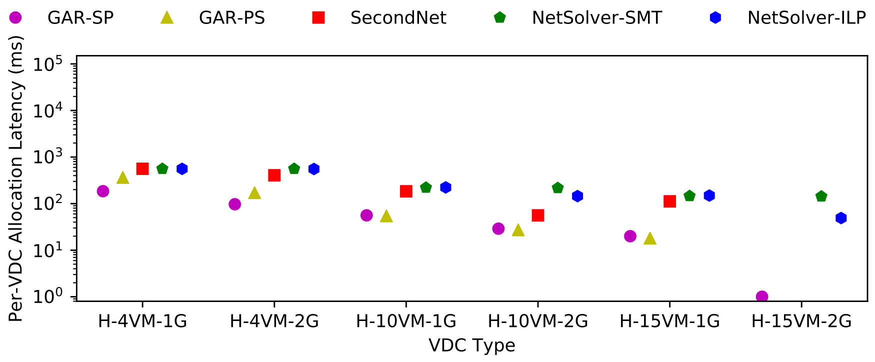


Figure 2.25: Latency Comparison with VNE Approaches on US-West2 Datacenter with 280 Servers. We show the median per-VDC allocation latency for allocations shown in Figure 2.24. Note that no latency is reported for SecondNet and GAR-PS in “H-15VMs 2Gbps” because these algorithms make no VDC allocations there.

SOLVER having a better ability to solve challenging allocations quickly (completeness and efficiency), or if NETSOLVER is somehow making “smarter” allocations early on that leave more space for later VDC allocations.

In the experiments where Z3-AR makes fewer VDC allocations (e.g., Figure 2.7), Z3-AR’s problem is excessively high allocation latency, allocating only a handful of VDCs in datacenters with room for hundreds or thousands. In those cases, both NETSOLVER and SecondNet can make hundreds of further allocations starting from where Z3-AR was cut off after the 1 CPU hour limit.

The robustness question is more apropos versus SecondNet. We repeated the experiments with the FatTree (Figure 2.12) and BCube datacenters (Figure 2.14) by first using SecondNet to allocate as many VDCs as it can into an empty datacenter. Then, starting from that already partially utilized datacenter, we used NETSOLVER to allocate further VDCs. We found conclusive evidence that good early allocations cannot be entirely responsible for NETSOLVER’s performance, by observing that NETSOLVER can continue to allocate VDCs in cases where SecondNet can no longer make any further allocations.

The results of this experiment are shown in Figure 2.26 and Figure 2.28. Similarly to the earlier experiment, NETSOLVER can still allocate hundreds of addi-

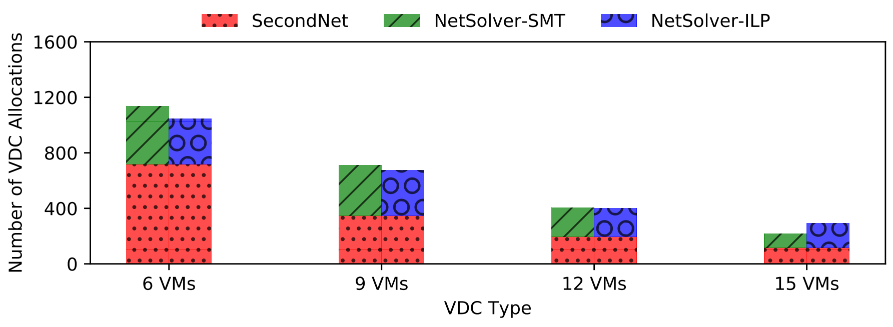


Figure 2.26: Additional Allocations by NETSOLVER in FatTree Datacenter. We show the number of additional VDCs allocated by NETSOLVER-SMT and NETSOLVER-ILP, after SecondNet has allocated its maximum number of VDCs. These experiments used the same VDCs and FatTree datacenter as in Figure 2.12.

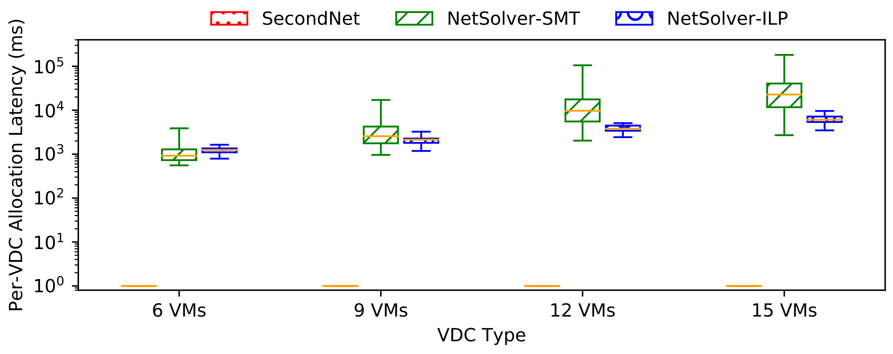


Figure 2.27: Latencies of Additional Allocations on FatTree Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.26. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

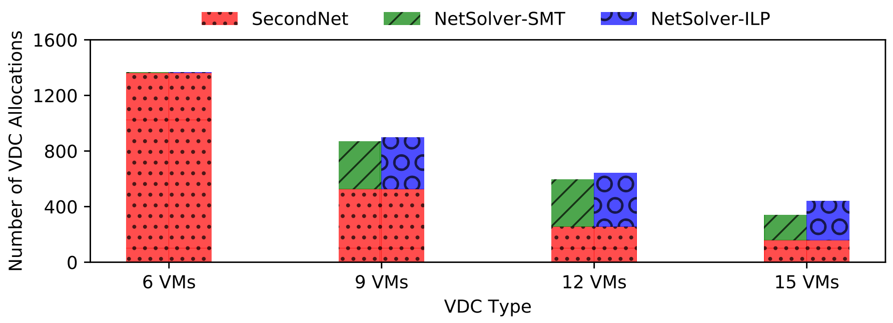


Figure 2.28: Additional Allocations by NETSOLVER in BCube Datacenter. We show the number of additional VDCs allocated by NETSOLVER-SMT and NETSOLVER-ILP, after SecondNet has allocated its maximum number of VDCs. These experiments used the same VDCs and BCube datacenter as in Figure 2.14.

tional VDCs starting from SecondNet's final allocation. For example, for VDCs with 6 VMs in Figure 2.26, SecondNet makes only 718 VDC allocations. NETSOLVER-ILP, however, allocates 326 more VDCs (+45%) on the partially utilized datacenter after SecondNet's termination. Thus, the aggregate number of VDCs allocated by NETSOLVER-ILP and SecondNet together are 1044 VDCs. On the other hand, for VDCs with 6 VMs in Figure 2.28, SecondNet terminates after allocating 1363 VDCs. Running NETSOLVER-ILP in this partially utilized datacenter yields only 3 more VDC allocations (+0.22%) and NETSOLVER-SMT yields only 2 more VDC allocations (+0.14%). Thus, bars for NETSOLVER-SMT and NETSOLVER-ILP are invisible. Both versions of NETSOLVER allocate substantially more additional VDCs when the VDCs are larger. In these additional VDC allocations, NETSOLVER-ILP's median per-VDC allocation is around 10 seconds or less, as shown in Figure 2.27 and Figure 2.29.

An interesting comparison is between NETSOLVER-ILP and NETSOLVER-SMT. In this case, both solvers are quite fast, and both solvers are complete in the sense that they will find an allocation if one exists. Therefore, in the cases where both solvers could find no more allocations, the additional allocations for NETSOLVER-ILP *must* be due to NETSOLVER-ILP somehow finding "smarter" allocations. In close examinations of the output of some of our experiments, we

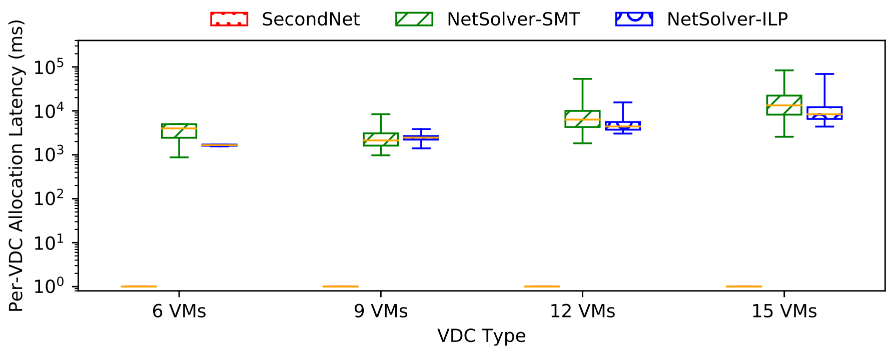


Figure 2.29: Latencies of Additional Allocations on BCube Datacenter. We show per-VDC allocation latency distribution for allocations shown in Figure 2.28. The latency boxes show the first and third quartiles, and whiskers show the min and max. The horizontal line inside the box is the median.

indeed found this to be the case, with NETSOLVER-ILP packing early allocations more tightly, thereby consuming less overall bandwidth with the early allocations, whereas NETSOLVER-SMT makes more spread-out allocations that consume more overall bandwidth.

For example, consider one of the largest examples from experiments on commercial datacenters, shown in Figure 2.17. Here, we allocate VDCs with 12 VMs in the US-West1 datacenter with 1200 servers. We analyze and compare the first 100 VDC allocations done by each algorithm. In the first 100 VDCs allocated by NETSOLVER-ILP, just 130 connections are to a top-of-rack (ToR) switch, and just 71 connections pass between racks, e.g., passing through gateway or aggregation switches. Note that as each placement is for 12 VMs with multiple connections between them, there are many more total connections between VMs than there are VDCs allocated. As all VMs are placed on servers, and all servers are contained in racks, anytime that connected VMs in a VDC are placed on different servers, connections to the ToR switch will be required. Similarly, anytime that VMs from a VDC are placed on multiple racks, connections between ToR switches will be required. In contrast to NETSOLVER-ILP, NETSOLVER-SMT's first 100 allocations require 603 connections to the ToR, and 449 connections between rack switches.

We hypothesize that this is due to the different approaches to incremental solving in ILP and SMT: an ILP solver will typically attempt to re-use a previous solution, whereas an SMT solver's main re-use strategy is to retain learned clauses. Therefore, during the early, highly unconstrained phase of the experiments, NETSOLVER-ILP will tend to allocate VDCs repeatedly onto the same machines, packing them in more tightly, whereas NETSOLVER-SMT spreads the allocations more arbitrarily around the datacenter. Although it is possible to extend NETSOLVER-SMT to be *locality-aware*, the way NETSOLVER-ILP is by construction, the empirical evidence suggests that NETSOLVER-ILP generally outperforms NETSOLVER-SMT without further enhancements to NETSOLVER-SMT. (We will revisit the value of being locality-aware in Chapter 4.)

2.5 Conclusions

We explored using constraint-solvers for VDC scheduling. In particular, we exploited the completeness property offered by constraint solvers for achieving high datacenter utilization. We introduced a new, constraint-based VDC scheduler, NETSOLVER, for multi-path VDC allocation with end-to-end bandwidth. Our approach differs from previous constraint-based approaches by making use of efficient network flow encodings in the underlying constraint solvers.

NETSOLVER overcomes major limitations of current state-of-the-art approaches for VDC scheduling. Unlike SecondNet, our approach is complete and, as a result, is able to continue making allocations in bandwidth-constrained networks. Unlike the abstraction-refinement techniques from Yuan et al. [135], NETSOLVER supports arbitrary datacenter topologies and has lower VDC allocation latency. Our constraint-based approach represents the first complete VDC scheduler supporting multi-path bandwidth allocation for arbitrary network topologies — an important capability in modern datacenters.

NETSOLVER scales well to datacenters with up to around 1000 servers, while substantially improving datacenter utilization as compared to previous methods. Notably, we have demonstrated that in several settings, NETSOLVER allocates up to 3 times as many VDCs as previous approaches, with a median per-VDC allocation latency around one second. We also saw scalability limitations of NETSOLVER, where it consumed over 10 seconds to allocate VDCs with 15 VMs on a datacenter with 800 servers. As we will see in Chapter 4, this limitation is exacerbated when the datacenter size increases and the density of the VDC topology grows.

In the rest of this dissertation our goal is to overcome these limitations. In addition to scalability issues, in practice, there are other limitations of the experiments so far that we need to consider. Therefore, we first ask, *what do real VDC workloads look like*? Unlike the synthetic VDC workloads we used in this chapter, Chapter 3 presents VDC workloads constructed from production traces from the Azure public cloud. We highlight the major differences between the production-based VDC workload and the synthetic one across several dimensions, including the dynamic nature of the production workload that includes VDC allocations, mutations, and deallocations. We then ask *what is the right metric for VDC scheduler evaluation* and *how does* NETSOLVER *compare to heuristic algorithms used in practice*? In Chapter 4, we propose the revenue gain metric for evaluating VDC schedulers. We also show that NETSOLVER has prohibitively high resource allocation latency for datacenters with over a thousand servers and VDCs with dense topologies. Thus, although one might be able to use NETSOLVER in datacenters of the limited scale and for a subset of VDC topologies, Chapter 4 demonstrates that the heuristics-based VDC schedulers are better suited for large-scale datacenters, which include public clouds.

Chapter 3

VDC Workload

Workloads are important for measuring system performance, and a cloud system for scheduling VDCs is no exception. Ideally, we would use a VDC workload from a production cloud. However, no such workloads exist, because no cloud provider offers VDCs with network bandwidth guarantees. Thus, we resort to approximating a *realistic* VDC workload. We use production traces from the Azure cloud as the workload source [40] and augment its VMs with bandwidth requirements. The outcome is a VM workload with bandwidth requirements, which we call a *VDC workload*. In this chapter, we describe our VDC workload generation methodology.

In approximating a realistic VDC workload, we face the following question: **How much inter-VM network bandwidth should the VDC workload demand?** To answer this question, we take inspiration from cloud computing history. In the early days of cloud computing, cloud providers faced the same question, but for compute. For example, in 2006, public cloud pioneer Amazon had to decide how much compute capacity to offer in a VM flavor in the EC2 product. EC2 beta started with only one VM flavor that included one virtual CPU, which offered an equivalent of Intel Xeon 1.7 GHz processor [26]. They also released several customer use-cases that might benefit from this VM flavor, such as a web-based back-office inventory application, and three-tier web application (presentation tier, application tier, data tier), which were popular applications at that time [26]. These use-cases are evidence that the decision for a VM's compute capacity was driven by what EC2 cloud operators *expected* tenants to run and pay for in the cloud.

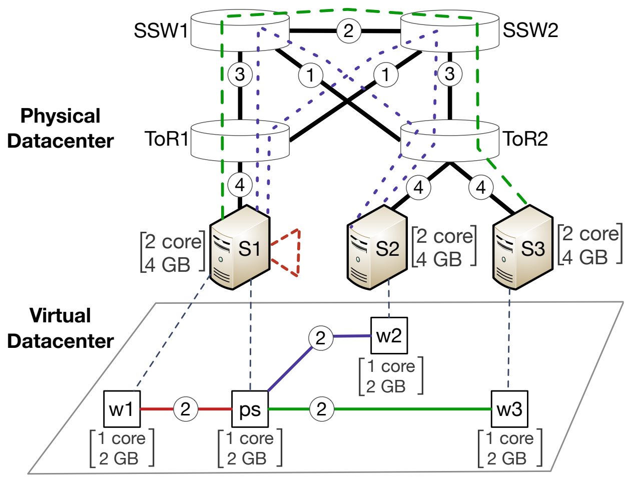


Figure 3.1: Example VDC Application. The VDC runs distributed ML training on a parameter server VM (`ps`) and three worker VMs (`w1`, `w2`, `w3`). The figure is reproduced from Figure 1.2 (on page 3).

We expect that VDC network bandwidth guarantees will evolve in the same way: driven by what *networked* applications providers expect tenants to run and pay for in the cloud. Note that this is different from today's networked cloud applications for which free, best-effort networking is the only option. The networked cloud applications that will run in VDCs are those that will see significant performance benefit, such as job completion time reduced by half.

A distributed Machine Learning (ML) training is a cloud workload that was demonstrated to significantly benefit from network bandwidth guarantees [63, 72, 106]. ML training is typically deployed across multiple *communicating* VMs, or as a VDC, where each VM trains an ML model on a subset of the data. The P3 authors show that distributed ML training jobs complete 2–3× faster with consistent inter-VM network bandwidth guarantees [72]. These findings agree with other work in the literature, such as TicTac [63] and ByteScheduler [106].

We will use distributed ML training as the sample VDC application in the rest of this dissertation because it is one of the major cloud workloads [73]. Several other big data workloads, such as HiBench and TPC-DS benchmarks [128], also benefit from network bandwidth guarantees in the cloud. Figure 3.1 shows an example of a distributed ML training application deployed as a VDC.

Distributed training is an instance of parallel computing with both *sequential* and *parallel* phases [131]. The sequential part, parameter aggregation, is run by the parameter server, while the parallel part, model training, is split and run across multiple worker VMs. The worker VMs synchronize their local state over the network. Thus, we can apply Amdahl's second law (Amdahl's I/O law) [7] to estimate the network bandwidth requirements of distributed training.

Amdahl's second law estimates the required network bandwidth to have a *balanced* system. A system is balanced when it can perform a compute task without memory or I/O bottlenecks [7, 86]. The law states that for every one instruction per second of processing speed (Hz), a balanced computing system needs to provide one bit per second of I/O rate (bps) and one byte of main memory capacity [86]. For example, the initial VM flavor offered by EC2 beta with a 1.7 GHz processor was a balanced system in terms of memory (1.75 GB of RAM), but not in terms of I/O (250 Mbps) [26]. In other words, the VM had

$$Amdahl\ memory\ number = \frac{memory\ size\ (GB)}{CPU\ speed\ (GHz)} = 1.75/1.7 \approx 1$$

and

$$Amdahl\ I/O\ number = \frac{bandwidth\ (Gbps)}{CPU\ speed\ (GHz)} = 0.25/1.7 \approx 0.15$$

Note that we use the network bandwidth, not the disk bandwidth, to derive the Amdahl I/O number because our distributed training application communicates parameter updates (from the worker VMs) to the parameter server VM over the network. Moreover, even though parallel and sequential phases are temporally disjoint in theory, they are optimized to overlap in practice. These optimizations avoid bursty traffic and ensure continuous, high throughput network utilization [63, 72, 106].

We can use Amdahl's second law to answer our question: **a VM's network bandwidth should be proportional to its compute capacity, for example one bit per second for every one CPU instruction in a balanced VDC application**. We call this the *compute-proportional-bandwidth* approach for VDC workload generation. This approach is similar to how the Google Compute Engine (GCE) scales its VM's *egress* network bandwidth as a function of vCPUs [92]. Note that unlike VDCs in our model, GCE's egress bandwidth is best-effort and is not between a pair of VMs. Even though egress bandwidth is from an individual VM perspective, the GCE example demonstrates the practicality of a compute-proportional-bandwidth approach in a cloud environment.

As an example, we analyze the sample VDC used in P3 [72] using Amdahl's second law. The VDC was deployed on an EC2 cluster of g3.4xlarge VMs where each VM had 16 vCPUs, 122 GB memory, and up to 10 Gbps network bandwidth [12]. Each g3.4xlarge vCPU is a 2.7 GHz Intel Xeon processor. Thus, each VM has $16 * 2.7$ GHz ≈ 43 GHz compute capacity, which yields:

$$Amdahl\ memory\ number = 122/43 \approx 2.84$$

$$Amdahl\ I/O\ number = 10/43 \approx 0.23$$

This shows the P3 VDC was "over-balanced" for memory and "under-balanced" for network I/O. If this VDC was running an application that requires $Amdahl\ I/O\ number = 1$, i.e., VDC was balanced for network I/O, each VM should have had 43 Gbps network bandwidth. Although VMs with $Amdahl\ I/O\ number = 1$ are not common in public clouds today, cloud network operators are already planning to increase their datacenter network bandwidth to offer such VMs [129].

An optimal Amdahl I/O number is deployment dependent, i.e., it depends on the characteristics of the cloud application and the datacenter the application is running on. For parallel applications, Amdahl's second law recommends a value of 1, which is one datapoint in a spectrum. Not all applications need a balanced system. For example, Liang et al. demonstrate that an optimal Amdahl I/O number for MPI-like (Message Passing Interface) applications ranges between 0.02–0.21, while it ranges between 0.02–0.85 for Hadoop-like applications [86]. Therefore, our generated VDC workload should be *adaptable*, i.e., we should be able to adjust the network bandwidth demand of the workload for the datacenter under test. In Section 3.2.3, we describe a parameterized VDC workload generation methodology that allows adapting the VDC workload's network bandwidth demand.

In summary, we now better understand VDCs' inter-VM network bandwidth requirements. We analyzed the characteristics of an emerging VDC application, distributed ML training, established its analogy with parallel computing applications, and applied Amdahl's second law, to derive the network bandwidth requirements of the sample VDC application.

In the rest of this chapter, we apply these findings to generate a realistic VDC workload from a VM workload, which has no explicit network bandwidth requirements. In Section 3.1, we describe the VM workload. Section 3.2 describes the Gridiron technique[1] to generate a VDC workload, and Section 3.3 applies this technique to construct a VDC workload for distributed ML training. In Section 3.4, we discuss other production traces released by public cloud providers and describe our rational for choosing the Resource Central dataset [40] as the basis for our VDC workload. We conclude in Section 3.5.

3.1 The Base Workload

We generate a VDC workload by augmenting the Azure cloud's production trace with network bandwidth requirements. The Azure trace [20] was released with the Resource Central paper [40]. We first describe the characteristics of the Azure trace, such as the number of VMs in the workload, and their CPU and RAM footprints. We then describe how we use the deployment IDs, which are included in the trace, to group VMs into VDCs.

The Azure trace contains a list of 2,013,767 unique VMs. For each VM, the dataset includes 11 fields shown in Table 3.1. We use only five of these fields to construct the base workload: deployment ID, VM creation time, VM deletion time, VM cores, and VM memory. The timestamps are reported in seconds with five min granularity (300 seconds); we call each such five minute interval a *tick*. The trace contains 8,640 ticks.

We preprocess the Azure trace to have valid timestamps. There are two sources of invalidity: 1) capture and 2) life span. A VM has an invalid capture timestamp if the timestamp is not the multiple of 300 seconds (capture interval). In other words, all timestamps should satisfy ($t \ mod \ 300 = 0$) equality, which 27 VMs do not. We round invalid timestamps to the closest valid ones, as suggested by the Azure trace authors [80]. For example, an invalid $t = 1000$ is rounded to a valid $t = 900$.

[1]The term "gridiron" is borrowed from the urban planning literature. Just like gridiron planning converts implicitly connected village houses into explicitly connected network of urban residences, the Gridiron technique converts implicitly connected tenant VMs into VDCs: a network of explicitly connected VMs.

Table 3.1: Information Released in the Azure Trace [20]. These are the fields in `vmtable.csv` file. The ones we use in our base workload are in bold.

#	Name	Description
1	VM ID	Unique ID for each VM.
2	Subscription ID	One needs to subscribe to Azure to create a VM. It is similar to customer IDs, although one customer can create multiple subscriptions. There are 5,958 subscriptions in the dataset.
3	**Deployment ID**	VMs are grouped and managed in a deployment. A subscription can have multiple deployments and a deployment belongs to only one subscription. There are 35,941 deployments in the dataset.
4	**Timestamp VM Created**	VM creation time in seconds. It is reported in five minute intervals. Thus, values are in increment of 300.
5	**Timestamp VM Deleted**	VM deletion time in seconds. Reported in the same interval as the VM create timestamps.
6	Max CPU	Maximum CPU utilization reading during the VM's lifetime. CPU utilizations are read every five minutes and are reported in percentages.
7	Average CPU	Average CPU utilization during the VM's lifetime.
8	P95 of Max CPU utilization	The 95th percentile (p95) of max CPU utilization readings during the VM's lifetime.
9	VM Category	VMs are put into one of three categorized based on their performance characteristics: delay-sensitive, delay-insensitive, or unknown.
10	**VM Cores**	The number of virtual cores in the VM.
11	**VM Memory**	The amount of RAM in the VM.

A VM has an invalid life span timestamp if its create and delete ticks are identical. We eliminate these VMs, which we call *instant-VMs*. There are 53,467 instant-VMs (less than 2% of total), which leaves 1,960,300 VMs in the preprocessed workload. Resource Central refers to instant-VMs as "control plane latency test VMs" [40]; these test VMs are used to continuously evaluate VM creation latency.

In summary, there are 2,013,767 VMs in 35,941 deployments before preprocessing the trace and 1,960,300 VMs in 35,870 deployments afterwards. We call the Azure trace after preprocessing the *base workload*. The base workload's compute and memory demand varies by around 10% across all 8640 ticks: between 321,043 cores and 346,755 cores and between 730,314 GB and 781,767 GB.

Our base workload uses only a subset of VM information in the Azure trace [20]: five out of 11 shown in Table 3.1. The Azure dataset also includes CPU readings (max, average, p95) of each VM every five minutes (in `vm_cpu_readings.csv` file [20]). Although we do not use CPU utilization readings to generate a VDC workload, one could use them to generate a VDC workload with different network bandwidth requirements. We leave this to future work. We now describe our VDC workload generation technique based on VMs' *requested* resources.

3.2 From VMs to VDCs: Gridiron Technique

We use deployments to represent VDCs. There are four additional steps in the Gridiron technique. First, VDCs have inter-VM communication topologies, which the base workload lacks. In Section 3.2.1, we describe various topologies VDCs can have and explain our rational for adopting an all-to-all topology. Second, VDC sizes should be appropriate for the application(s) that these VDCs encapsulate. In Section 3.2.2, we explain the significance of the peak VDC sizes and show peak sizes in the base workload. Third, virtual links (vlinks) in a VDC should have a bandwidth value. In Section 3.2.3, we use the compute-proportional-bandwidth approach for assigning vlink bandwidths and constructing a parameterized VDC workload. The parameterization allows us to tailor the network bandwidth demand of the VDC workload. Fourth, the VDC workload should be feasible by construction, e.g., an empty datacenter should be able to accommodate the VDC with the highest bandwidth demand. Section 3.2.4 explains several kinds of scenarios to be aware of when generating the VDC workload, and Section 3.2.5 describes a model to avoid these scenarios.

3.2.1 VDC Topologies

The design space for VDC topologies ranges from low to high connectivity. Figure 3.2 shows three different VDC topologies with sparse and dense connectivity. An example VDC with sparse connectivity, as in Figure 3.2(a), runs a distributed ML training application in a data-parallel method where all worker VMs connect to a centralized parameter server in a star topology [125]. The VDC in Figure 3.1 for distributed ML training, which we used as the sample VDC application earlier, is an

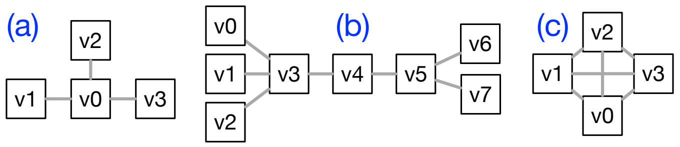


Figure 3.2: VDC Topologies with Varying Connectivity: (a) star topology with sparse connectivity, (b) pipeline topology with sparse connectivity, and (c) all-to-all topology with dense connectivity.

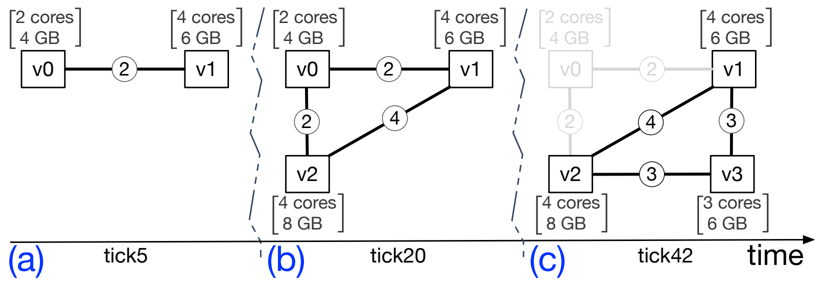


Figure 3.3: VDC Mutation Over Time: (a) shows VDC creation with two VMs, (b) shows VM v2 allocation, and (c) shows VM v0 deallocation and VM v3 allocation. The VDC will continue with three VMs (v1, v2, v3) after tick 42.

instance of VDC topology with sparse connectivity. Moreover, VDCs we used from the existing literature in Chapter 2, such as Yuan et al. [135] and SecondNet [59], also have sparse connectivity. A different example also with sparse connectivity, as in Figure 3.2(b), is a big data application, such as the ETL (Extract-Transform-Load) Spark pipeline [41]. Here, VMs load data from various sources, process it with map-reduce-style operations, and present the data or store the results for later processing. Finally, an example VDC with dense connectivity, as in Figure 3.2(c), runs a distributed ML training application in a model-parallel method where every VM acts as both a worker and a parameter server. Here, each VM stores a full copy of the ML model and trains its local model on its distinct data slice while periodically communicating the local parameter updates to all other VMs. This method of distributed training is also called training with a mirrored strategy [124].

Figure 3.3 shows network bandwidth assignment for a sample VDC with dense connectivity, as in Figure 3.2(c). It also shows VDC size mutation as VMs are

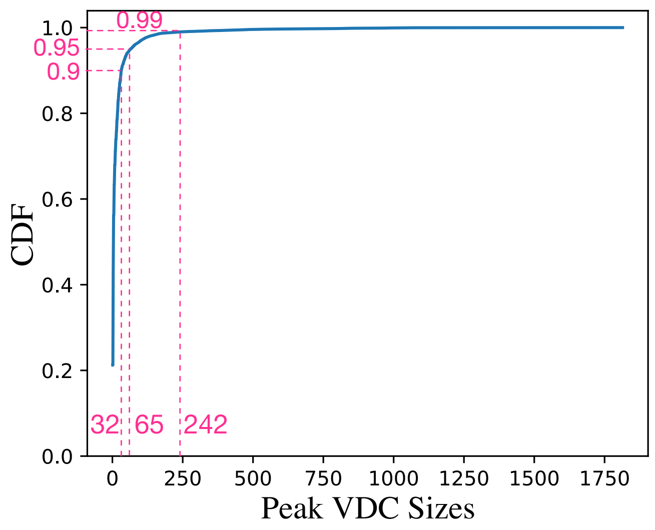


Figure 3.4: Peak VDC Sizes in the Base Workload.

added to and deleted from the VDC. Figure 3.3(a) shows an initial VDC created in time tick 5 with two VMs: v0 and v1. The v0-v1 vlink has two "units of network bandwidth".[2] We use the term *unit of network bandwidth* throughout our examples for generality. We convert it to a specific unit, such as 6 Mbps, when we tailor the VDC workload to a specific datacenter. The VDC expands to include VM v2 in tick 20, as shown in Figure 3.3(b). This requires allocating v0-v1 and v1-v2 vlinks. A VDC can also shrink with VM deallocation(s), as shown in tick 42 (Figure 3.3(c)). In tick 42, VM v0 is deallocated and VM v3 is allocated. The VM v0 deallocation requires deallocating the v0-v1 and v0-v2 vlinks. In the Gridiron technique, *VM deletions always precede VM allocations within a tick* so that datacenter resources are first released before being consumed. The VM v3 connects to all other alive VDC VMs, v1 and v2, which requires allocating v1-v3 and v2-v3 vlinks. Note that VM deletions in a VDC do not have to adhere to FIFO (first in first out), LIFO (last in first out), or any other order. The ordering is inherited from the base workload.

3.2.2 Peak VDC Sizes

A VDC reaches its peak size when it has the maximum number of VMs alive in the same tick. For example, the VDC in Figure 3.3 reaches peak size `P=3` in tick 20 and maintains that size until tick 42 and beyond. Figure 3.4 shows the distribution of peak VDC sizes in the base workload. It shows that the 90th percentile (p90) of the VDCs has 32 VMs and VDC sizes can reach as many as 1,814 VMs.

A VDC with 1,814 VMs is not common. For example, running distributed ML training at this scale is challenging [57]. Therefore, we should be able to cap the peak VDC size when generating a VDC workload for a particular cloud application. In the Gridiron technique, we cap the peak VDC sizes by splitting a deployment into multiple VDCs where each VDC's peak size is below the cap.

We split a too-large deployments into multiple VDCs via a rolling-overflow mechanism. VDCs are processed one-by-one, keeping track of the peak VDC size so far. If that size exceeds the cap, then subsequent allocations to the same deployment are rolled to a new VDC. In other words, a VDC will have no more VM create events after it reaches the cap size. At the same time, if the base workload VDC has a peak size below the cap, no capping is applied, and it will be added to the VDC workload as-is, as a single VDC. Given that the rolling-overflow mechanism splits a single deployment into multiple VDCs, the number of VDCs in the capped workload will potentially exceed the number of deployments in the base workload. Appendix B shows the pseudocode for constructing the VDC workload from the base workload while applying the capping mechanism. The actual source code is available in the dissertation artifact repository [81].

3.2.3 Parameterizing VDC Workload's Network Load

We assign a bandwidth value to each vlink using a compute-proportional-bandwidth approach. However, a vlink connects two VMs: Which VM's compute capacity should be used as the reference point? In the Gridiron technique, we choose the VM with the smaller capacity, because doing otherwise introduces a *flooding* effect. Flooding happens when a VM with more compute capacity sends a higher traffic volume to the VM with less compute capacity, or the *weaker* VM, to the

[2]We will describe how we derive vlink bandwidths in Section 3.2.3.

point where the weaker VM is no longer able to process the traffic. In practice, the weaker VM drops the subset of packets it cannot process. Thus, the vlink bandwidth is more realistic if the excess packets were not sent to begin with. For example, in Figure 3.3, the v0-v1 vlink has two units of bandwidth as the VM with the weaker processing capacity has two vCPU cores (VM v0). Similarly, the v1-v2 vlink has four units of bandwidth because both VMs that it connects have four vCPU cores. The rationale for avoiding a flood is similar to what TCP flow control is designed for, except VDC vlinks in this work are (transport) protocol-agnostic.

We generate VDC workloads with varying bandwidth demand by parameterizing each vlink's bandwidth. The key insight in parameterization is that we can use different "units" to represent the unit of bandwidth. For example, in Figure 3.3, the v0-v1 vlink gets assigned 2 Mbps bandwidth when we use 1 Mbps as the unit, and 10 Mbps bandwidth when we use 5 Mbps as the unit. Given that the unit of bandwidth of the vlink is determined by the compute capacity of the weaker VM (that it connects), we can directly make the unit as the function of the vCPU cores in the weaker VM. We call this unit the *bandwidth per core* (`bpc`). In our earlier example, the v0-v1 vlink gets assigned 2 Mbps bandwidth when `bpc=1Mbps`, and 10 Mbps bandwidth when `bpc=5Mbps` (because the weaker VM has 2 vCPUs). Thus, we can generate VDC workloads with varying bandwidth demand by assigning different values to `bpc`. For example, the workload with `bpc=2Mbps` has twice higher bandwidth demand than the workload with `bpc=1Mbps`.

We use the `bpc` parameter to avoid the *over-provisioned* network pitfall that SecondNet's VDC workload suffered from. The VDC workload used in the SecondNet [59] consumed 1/50th of the datacenter bandwidth provided (Section 2.4.3). In other words, the datacenter's network capacity was $50\times$ over-provisioned. Thus, VDCs' bandwidth requirements were mostly irrelevant during resource scheduling despite the fact that networking was the central focus of the paper. Although this might be acceptable when a cloud provider is willing to leave a significant portion of their datacenter network bandwidth underutilized, it seems an unlikely scenario in practice, because cloud providers report that datacenter network is a computation bottleneck with ToR switch uplinks frequently operating above 80% utilization [58]. Thus, VDC schedulers should be evaluated with workloads that consume significant portion of the datacenter network bandwidth.

We can tune the `bpc` parameter to create a VDC workload with the right amount of bandwidth demand for the datacenter under test. As we will show in Section 4.2.6, varying the demand allows us to evaluate the bandwidth allocation quality of different VDC schedulers. Note that it is possible to diversify vlink bandwidths across different VDCs by adopting a finer tuning mechanism for the `bpc` parameter. For example, we could generate a VDC workload with more diverse vlink bandwidths by using `bpc=2Mbps` in some subset of VDCs and `bpc=5Mbps` in other subsets. Even though we do not see a reason for such diversity to influence the scheduler evaluation, i.e., we hypothesize that the best VDC scheduler remains the best regardless of the vlink diversity in VDC workload, empirical demonstration of this hypothesis is left as future work.

3.2.4 Network-bound VM Allocation Failures

Cloud services depend on hardware capabilities of the datacenter(s) they run on. For example, cloud providers will not offer a VM flavor with 70 vCPUs if their datacenter does not have a server with that many cores. We call these phenomena *datacenter-level constraints*. Similar constraints apply to all cloud services, including network bandwidth guarantees. We take datacenter-level constraints into account when generating a VDC workload, because otherwise we risk generating VDCs that are *infeasible by construction*. In this section, we discuss VM allocation failures that surface if datacenter-level-constraints are not taken into account. A VM allocation failure happens when a tenant VM allocation request is rejected because of insufficient residual capacity in the datacenter.

We call a VM allocation failure a network-bound VM allocation failure, or *network-bound failure*, when the VM allocation request is rejected because of insufficient network bandwidth in the datacenter. When bandwidth is offered as a first class cloud service, just like the compute service, cloud providers would want to avoid, or minimize, network-bound failures. Avoiding network-bound failures is more complex than avoiding compute-bound failures, because unlike vCPUs, inter-VM bandwidth is not a server-local resource: it is a cross-device resource. In particular, we identified three specific scenarios that can produce network-bound failures by construction. We first elaborate on each scenario and its significance.

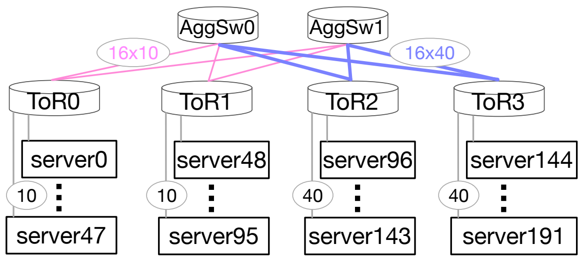


Figure 3.5: Example Datacenter with Four Racks. Here, racks are connected to two aggregation switches.

Later in Section 3.2.5, we present three constraints to enforce on VDCs such that the maximum bandwidth each VDC might ever impose on a single server is feasible to accommodate.

The first scenario is allowing excessive bandwidth in a virtual link (*vlink-caused*). Vlink-caused failures happen when a vlink's bandwidth capacity exceeds the *aggregate uplink capacity at the most network-bandwidth-intensive server*, which we call the "fattest server-uplink". If servers have two or more uplinks (multi-homed), the fattest server-uplink's capacity will be equal to a server's aggregate uplink bandwidth. Vlink-caused failure is analogous to the number of vCPUs exceeding the number of CPUs at the most compute-intensive server. For example, for the datacenter shown in Figure 3.5, a vlink with over 40 Gbps is guaranteed to cause a network-bound failure. (The only exception is if both ends of the vlink are colocated on the same server.)

We define the "fattest link" in terms of "server uplink" because of the network multi-path feature, which is commonly used in modern datacenters [58]. For example, when two VDC VMs are placed across different racks, such as on server0 and on server144 in Figure 3.5, a vlink between these VMs can span multiple paths, such as ToR1-AggSw1-ToR2 and ToR1-AggSw2-ToR2, to pool the bandwidth amount required for this vlink. In general, the vlink bandwidth should not exceed the bandwidth across any cut in the network between the servers that host two ends of the vlink. However, this is non-trivial to compute and depends on the placement of the VMs, but the fattest server-uplink is always a cut, and therefore an upper bound on the vlink bandwidth.

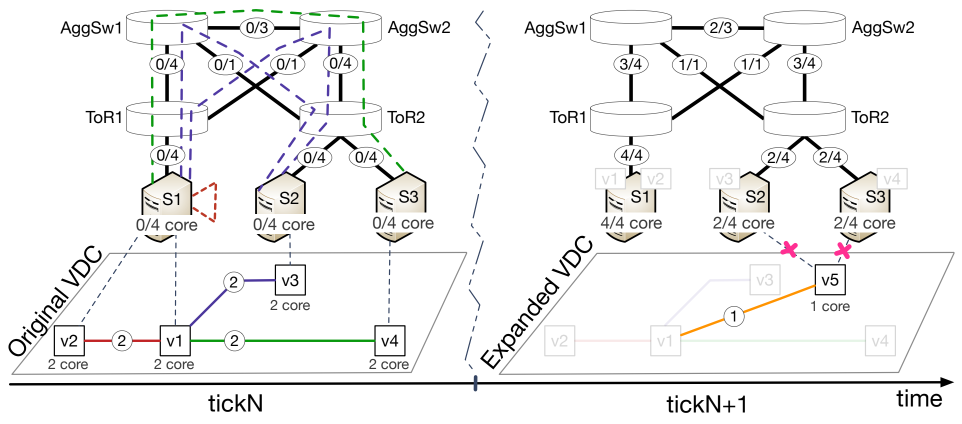


Figure 3.6: This is an example of overpeering-caused VM allocation failures where the initial VDC allocation succeeds in tick N but the expansion of the VDC with VM v5 in tick (N+1) fails because of overpeering of VM v1. Datacenter resource capacities are shows as used / original. For example, 1/4 in link bandwidth means that "1" unit of bandwidth is used out of total "4" units. We omit VM and server memory capacities for brevity. The datacenter is empty in tick N.

The second scenario is allowing excessive VM overpeering[3] (*overpeering-caused*). Overpeering-caused failures happen when a VDCs' already allocated VM(s) get too many peering requests such that the server hosting an already-allocated VM becomes bandwidth bottlenecked. Figure 3.6 shows an overpeering-caused failure. In tick N, a tenant requests a VDC with four VMs (v1, v2, v3, v4). These four VMs get placed on three servers (S1, S2, S3) as shown with the dashed lines in the left figure. We show VM-to-server assignment with grayed boxes placed on the servers in tick (N+1). The tenant requests to expand the original VDC with VM v5 in tick (N+1). The v5 needs to connect to v1 with one unit of bandwidth. However, as we can see in tick (N+1), the only two servers with sufficient cores and memory to accommodate v5 (S2, S3) do not have sufficient bandwidth to connect to S1, which hosts the already allocated peer VM (v1). Thus, the VDC scheduler fails v5 allocation due to insufficient bandwidth.

[3]We could say "oversubscription" instead of "overpeering" because peer VM(s) actually *subscribe* to the already-allocated peers. However, the term "oversubscription" is already used in the cloud's compute service. For example, saying "a server is CPU oversubscribed" means that "the number of vCPUs VMs are consuming exceed what the host server offers".

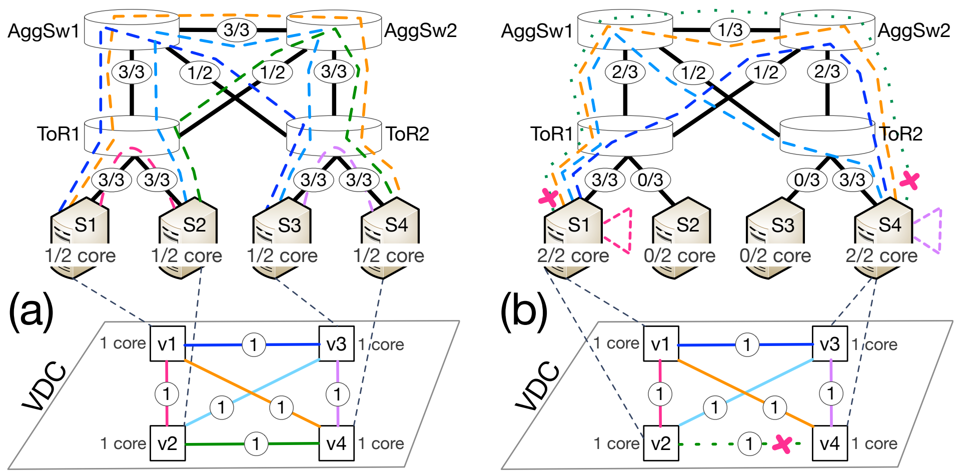


Figure 3.7: Effect of Colocation on Datacenter Network Bandwidth. We show how colocation can over-stress datacenter network: (a) shows successful VDC allocation, and (b) shows colocation-caused VM allocation failure (v4). Datacenter resource capacities are shown as used / original. For example, 1/4 in link bandwidth means that "1" unit of bandwidth is used out of total "4" units. The datacenter hosts only this VDC.

The overpeering-caused failures happen for the same reason as the vlink-caused failures: insufficient bandwidth at the server uplink level (server-to-ToR switch links). Similarly to the vlink-caused failures, we do not include failures that happen due to bandwidth scarcity in other datacenter network levels, such as the spine links (because of multi-pathing). For example, in Figure 3.6, VM v5 fails because of the S1-ToR1 server uplink. Otherwise, S2(or S3)-ToR2-AggSw2-AggSw1-ToR1 does have one unit of bandwidth available to accommodate the v1-v5 vlink.

The third scenario is allowing excessive VM colocation, which concentrates aggregate vlink bandwidth on a single server (*colocation-caused*). We say that two VDC VMs are *colocated* when they are placed on the same server. Colocation-caused failures happen when the aggregate bandwidth to colocated VDC VMs exceeds the bandwidth offered by the host server. An example is shown in Figure 3.7 where a VDC with four VMs needs to be placed on a datacenter with four servers. VDC VMs arrive and are placed one-by-one. Figure 3.7(a) shows successful VDC allocation when no VMs are colocated. Figure 3.7(b) shows VM v4 allocation failure because of colocation. This VDC consumes the maximal bandwidth on a single physical link when VDC VMs are equally split across two servers, e.g., v1 and v2 VMs are placed on server S1, and two other VMs are placed on server S4. (We proof this theorem in the next paragraph.) Colocated VMs do not consume any datacenter network bandwidth to communicate with each other since they communicate locally (through the hypervisor), but they do consume datacenter network bandwidth to communicate with *every* VM placed on the other server. Thus, there are 2×2=4 vlinks, quadratic in the number of VMs placed on each server, traversing the path between server S1 and server S4. However, the S1-ToR1 link can accommodate only three vlinks. Hence, v2-v4 vlink allocation fails, causing VM v4 allocation failure. Notice that VM v4 in Figure 3.7(b) fails allocation even though colocation reduces the overall demand on datacenter network bandwidth (four units) compared to fix units of overall bandwidth without colocation (Figure 3.7(a)). This demonstrates that the colocation-caused failures happen due to demand concentration, not necessarily demand increase.

Theorem: *A VDC imposes the greatest demand on a single server link when its VMs are equally split across two servers.*

Proof. Imagine a datacenter with servers $(s \in S)$ and VDC with n VMs: x VMs placed on server s_x, y VMs placed on server s_y, and all remaining $(VDC \setminus \{x,y\})$ VMs placed on other datacenter servers $(S \setminus \{s_x, s_y\})$. The aggregate network bandwidth demand of this VDC (VDC_{bw}) is the sum of all VDC vlink bandwidths:

$$VDC_{bw} = \sum_{vlink \in VDC} bw_{vlink}$$

VDC allocation spreads VDC_{bw} among all servers that host VDC VMs, i.e.,

$$bw_{(s_x,s_y)} + \sum_{i \in \{S \setminus s_y\}} bw_{(s_x,i)} + \sum_{j \in \{S \setminus s_x\}} bw_{(s_y,j)} = VDC_{bw} \tag{3.1}$$

Given that we want to find an allocation that generates the greatest demand on (s_x, s_y) link, i.e., maximize $bw_{(s_x,s_y)}$ addend, we need to assign zero to two other addends in Equation 3.1. Informally, so far we established that the demand on (s_x, s_y) link is the greatest when all VDC VMs are allocated on s_x and s_y.

Now we prove that splitting VDC VMs equally between s_x and s_y generates the greatest demand, i.e., maximizes $bw_{(s_x,s_y)}$. When k VMs are on s_x, there are $(n-k)$ VMs on s_y. With all-to-all VDC topology, there are $(k*(n-k))$ vlinks on (s_x,s_y) link. Assuming that vlinks require identical bandwidths, we need to simply maximize the number of vlinks traversing (s_x,s_y) link. We can reduce this maximization problem to maximizing $(k*n-k^2)$ quadratic equation, which has the solution at $k=n/2$. Thus, we proved that a VDC imposes the greatest demand on a single server link when $(k=n/2)$ VMs are on server s_x and $(n-k=n/2)$ VMs are on server s_y, i.e., VDC VMs are equally split across two servers. $\square$

In summary, we described three scenarios that can cause network-bound failures due to datacenter-level constraints. These scenarios should be taken into account when generating any VDC workload. Otherwise, the VDC workload can have VDCs that are infeasible by construction, which can produce vlink-caused, overpeering-caused, and colocation-caused failures. Moreover, these scenarios are not exhaustive. There could be other scenarios that cause network-bound failures. However, in our experience these three scenarios are the most relevant ones to take into account during VDC workload generation.

Note that in this dissertation, we assume that VMs cannot be migrated, i.e., VMs cannot be relocated to another server after the initial placement. Migrating VDC VMs is particularly challenging, because a to-be-migrated VM might already be communicating with other peer VMs. Cloud providers avoid VM migration, because it affects application performance and might even cause VM downtime [40]. In fact, the Resource Central [40] authors propose using prediction-based VM allocation techniques to avoid "problematic live VM migration in practice ... and place VMs where they can stay".

3.2.5 Avoiding Network-bound VM Allocation Failures

Now that we have described three scenarios, we have to ensure that the VDC workload we generate is feasible by construction. We call such a workload *datacenter-aware*. In this section, we present a series of three constraints for generating datacenter-aware VDC workloads, which avoid vlink-caused, overpeering-caused, and colocation-caused failures.

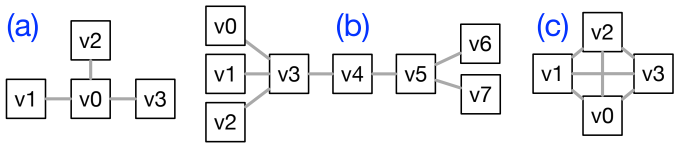


Figure 3.8: VDC Topologies: (a) and (b) have sparse connectivity, and (c) has dense connectivity. This is reproduced from Figure 3.2 for readability.

There are three knobs to control: (1) maximum bandwidth per vlink, (2) VDC topology, and (3) peak VDC size. The first knob is similar to the number of vCPUs in a VM flavor. We can cap vlink capacities to ensure that the highest bandwidth a vlink offers does not exceed the capacity of the fattest server-uplink. For example, for the datacenter shown in Figure 3.5, no vlink should offer over 40 Gbps. Two other knobs, VDC topology and peak VDC size, are related to each other and can be constrained to avoid overpeering-caused and colocation-caused failures.

Figure 3.6 shows that in constructing a VDC workload, we need to budget not only for a VM's current bandwidth requirements but also for its *growth potential*. A VM's growth potential is the difference between the bandwidth it consumes at its allocation and how much more bandwidth it can consume in the future. For example, if a VM is created with only one vlink that has 100 Mbps bandwidth but it can create 10 more such vlinks during its lifetime, this VM's growth potential is 1000 Mbps (11*100-100). A VMs' growth potential is a function of two things: the topology of its VDC, which determines the number of vlinks the VM can have, and the peak VDC size, which defines the maximum number of VMs allowed in a VDC at the same time. Figure 3.8 shows VDC topologies with sparse and dense connectivity. VMs in a dense connectivity, as in Figure 3.8(c), have the highest growth potential. This potential can induce overpeering- and colocation-caused failures, because VMs have the highest aggregate bandwidth when they peer with every other VM in the VDC, i.e., in an all-to-all topology.

We can impose restrictions on the VDC topology to avoid VDCs with dense connectivity in the workload. For example, we could allow only two vlinks per VM to ensure that all VDCs have a sparse, e.g., a chain-like, topology. However, this would be too restrictive. As an example, the generated VDC workload could not contain distributed-training-like applications that have all-to-all connectivity. In-

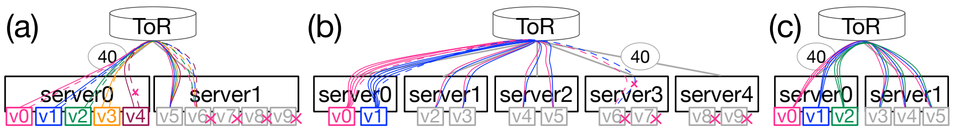


Figure 3.9: Effect of Colocation on Datacenter Network Bandwidth. We show how colocation can create bottleneck(s) in datacenter network. VM-to-server placement is shown as VMs placed on the servers: (a) and (b) show four colocation-caused VM failures (v6, v7, v8, v9), and (c) shows failure-free VDC placement. VDCs in all figures have all-to-all connectivity. Figure (b) shows connectivity for only VM v0 and VM v1 for brevity.

stead, we can just cap the peak VDC size while granting tenants complete freedom in the topology choice. Put differently, we can guard against the overpeering- and colocation-caused failures by controlling only the peak VDC size knob, and leave the topology knob free by assuming (the worst case) all-to-all VDC topology.

For example, consider the sample four-rack datacenter topology in Figure 3.5 (we generalize this example later):

1. We can avoid vlink-caused failures by capping vlink capacities to $C = 40$ Gbps, i.e., the fattest server-uplink capacity.

2. Avoiding overpeering-caused failures is about capping VDC size such that the aggregate bandwidth of the VM vlinks does not exceed C. For the sample datacenter in Figure 3.5, we need to limit the per-VM aggregate bandwidth to 40 Gbps. Assume that the peak VDC size $P = 10$. A VDC VM can therefore can have up to $P-1 = 9$ vlinks. There, we need to ensure that the aggregate bandwidth of nine vlinks does not exceed 40 Gbps. Thus, we conservatively cap each vlink to have at most $B = 40/9 \approx 4.4$ Gbps bandwidth to avoid overpeering-caused VM allocation failures.

3. We can prevent colocation-caused failures by avoiding VM colocation or by capping the VDC VM *colocation degree*. A VDC has colocation degree of D when the largest number of VMs of the VDC colocated on a server is D.

   For example, imagine the VDC with 10 VMs shown in Figure 3.9(a). The VDC has an all-to-all topology where each vlink has 4.4 Gbps bandwidth

(B=4.4 Gbps). VMs arrive and are placed one-by-one. The first five VMs (v0, v1, v2, v3, v4) are colocated on server0, after which server0 becomes compute bound. Although server1 has sufficient compute capacity to accommodate the remaining five VMs (v5, v6, v7, v8, v9), only VM v5 gets successfully allocated, because server0's uplink is exhausted after nine vlinks (five vlinks to connect v5 to the first five VMs in server0, and four vlinks to connect v6 to the first four VMs; $4.4\times9\approx40$) and has no bandwidth left for v4-v6 vlink. Thus, v6, v7, v8, v9 VMs fail allocation (when $D=5$).

Imagine placing this VDC on a five-server datacenter, shown in Figure 3.9(b). Here also, VDC VMs are placed one-by-one and each vlink has 4.4 Gbps bandwidth. Assume that servers' compute capacity suffices to accommodate only two VMs. The first six VMs get successfully allocated, after which server0's uplink is exhausted with nine vlinks ($4.4\times9\approx40$) and has no bandwidth left for v1-v6 vlink. Thus, the last four VMs fail allocation even when $D=2$. Therefore, for the datacenter shown in Figure 3.5, where $C=40$ Gbps, $P=10$, and $B=4.4$ Gbps, disabling colocation altogether ($D=1$), is the only way to avoid colocation-caused failures.

At the same time, Figure 3.9(c) shows that it is possible to leave the colocation degree unconstrained when $P=6$ (while $C=40$ Gbps and $B=4.4$ Gbps) because 40 Gbps server-to-ToR links can accommodate the maximal aggregate bandwidth of the smaller, 6-VM VDC ($D=3$). Thus, Figure 3.9(c) demonstrates that we can avoid colocation-caused failures by controlling peak VDC size (`P`).

Now we generalize our findings from the sample datacenter and propose a method for generating datacenter-aware VDC workload. The datacenter-aware VDC workload should satisfy the following three constraints to avoid all three causes of network-bound failures:

1. vlink-caused failures:

$$B \leq C \tag{3.2}$$

where `B` is the maximal per vlink bandwidth and `C` is the capacity of the fattest server-uplink.

2. overpeering-caused failures:

$$B \leq C/(P-1) \tag{3.3}$$

   where P is the peak VDC size.

3. colocation-caused failures:

$$B \leq C/(P/2)^2 \tag{3.4}$$

Note that Equation 3.3 supersedes Equation 3.2, and Equation 3.4 supersedes Equation 3.3, when $P \geq 2$. That is, limiting vlink bandwidth (B) to satisfy the constraint in Equation 3.4 automatically satisfies the two other constraints. At the same time, $P \geq 2$ is always true by VDC construction, because a VDC of size 1 is a degenerate VDC, requires no bandwidth, and therefore, is not considered. Thus, we can exclusively focus on satisfying Equation 3.4 (avoiding colocation-caused failures). This is what we do when generating a datacenter-aware VDC workload for our VDC scheduler evaluation in Section 4.2.6.

The method's purpose is not to prevent all possible network-bound failures, but to bound vlink bandwidths such that the generated VDC workload is datacenter-aware. The method is useful in generating VDC workloads that are feasible by construction and sufficiently network intensive to evaluate the VDC schedulers.

3.3 Case Study: Applying Gridiron Technique to ML Training

Deployment sizes in the base workload reach up to 1,814 VMs. This is not realistic for distributed ML training applications, because distributed ML training at this scale is challenging [57]. Thus, we cap the peak VDC sizes to a realistic size.

A common way to scale distributed ML training is by parallelism. For example, Goyal et al. scale DNN training by increasing the training batch size and executing data-parallel training across multiple machines/devices [57]. In a VDC, multiple VMs would execute the data-parallel training: the training data is split across VDC VMs, and each VM runs computation on a slice of the local data (mini-batch). The

Table 3.2: Common Distributed DNN Training Applications. Models are sorted by their size.

Task	Model Name	Model Size (MB)	Batch Size	Mini-Batch Size	Number of VMs
Recommendation	DeepLight [45]	2319	2^{11}–2^{13}	2048	1–4
Translation	LSTM [66]	1627	8–64	8–32	1–8
Translation	BERT [46]	1274	4–256	4–32	8–64
Image classification	VGG19 [116]	548	64–256	32–256	1–8
Translation	UGATIT [79]	511	1–2	1	1–2
Recommendation	NCF [65]	121	128–2^{17}	128–2^{14}	1–8
Object detection	SSD [89]	98	1–8	1–8	1–8
Image classification	ResNet-50 [64]	87	64–2^{21}	32–8192	8–256

result of the computation on a mini-batch (gradients) is broadcast to all other VDC VMs. A VM applies gradients from all VDC VMs to its local model. For example, in a VDC with 4 VMs using batch size 64, each VM will have mini-batch size 16. In general, we use the following formula to derive the number of VMs in a VDC:

$$Number\ of\ VMs = \frac{Batch\ Size}{Mini\ Batch\ Size}$$

However, higher batch sizes hurt the learning rate, impeding linear scalability beyond a certain batch size [57, 72, 78]. An optimal batch size differs by DNN. Table 3.2 lists eight common DNN applications identified by Sapio et al. [113]. These applications cover five tasks, out of six total, that were selected as the representative applications by the MLPerf training benchmarking organization [91], which has the broadest recognition across academia and industry [95]. We surveyed the recent literature to study the batch sizes and mini-batch sizes used to train these DNNs, which we then used to derive the VDC size range [36, 45, 57, 64, 65, 79, 89, 97, 113, 116].[4]

We chose ResNet-50 training as the most common workload. ResNet-50 is the state-of-the-art model in image classification, and is the most widely studied model in the literature [91]. ResNet-50 achieves linear scalability for batch sizes up to 1024 [57, 72]. Given that the most commonly used mini-batch size in the

[4]We report the studies that use only CPUs and GPUs. We exclude batch sizes when the model is trained with vendor-specific accelerators, such as TPUs.

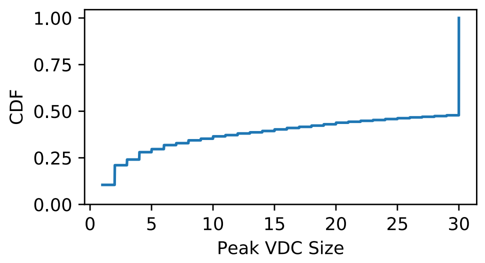


Figure 3.10: Peak VDC Sizes in the ML Training Workload: 48% of the VDCs have a peak size <30, the rest have peak size of exactly 30.

literature is 32, a VDC to train ResNet-50 can have up to 32 VMs (32x32=1024), which we round to 30. Our observations from the existing ML training literature is that a VDC size of 30 is realistic for modern cloud environments. This size also generalizes to models other than ResNet-50 because the workload analysis study by Jeon et al. shows that distributed DNN training in clusters with up to 16 VMs were already common in 2017 [73], and the cluster size has been increasing due to increased DNN model size [113].[5]

Figure 3.10 shows the peak VDC size distribution after we cap the peak VDC size at 30. The VDC workload we construct has $\approx 2\times$ more VDCs (73,872) than the deployments (35,870) in the Azure trace. Although a $2\times$ increase in VDC numbers has no influence for evaluating VDC schedulers, the cap threshold (30) does have an influence on the VDC workload's potential to introduce network-bound VM allocation failures (Section 3.2.4).

Next, to make the generated workload datacenter-aware, we need to target a specific datacenter. We demonstrate the vlink bandwidth assignment on the sample datacenter in Figure 3.5. Given the peak VDC size `P=30` and the fattest server-uplink capacity in the sample datacenter `C=40,000` Mbps, from Equation 3.4:

$$B \leq C/(P/2)^2 = 40{,}000/(30/2)^2 \approx 177.78\ Mbps$$

which means that the VDC workloads will not have VDCs that are infeasible by

[5]Note that the Jeon et al. study [73] is different from the Resource Central paper [40]. Jeon et al. analyzed DNN training workloads deployed on a multi-tenant GPU cluster in Microsoft during a 75-day period (from 2017.10 to 2017.12). Their analysis contained 96,260 jobs.

Table 3.3: Steps in the Gridiron Technique.

#	Name	Our Approach in the ML Training Workload
1	VDC Topology Selection	Use all-to-all topology.
2	Capping the Peak VDC Sizes	Cap the peak VDC sizes to 30.
3	Per-vlink Bandwidth Assignment	Adopt compute-proportional-bandwidth approach.
4	Generating Workloads with Bandwidth Demand	Parameterize using `bpc` values.
5	Making Workload Datacenter-aware	Constrain the peak VDC sizes and vlink bandwidths to avoid network-bound failures.

construction as long as vlinks' bandwidth do not exceed 177 Mbps.

From the cap on vlink, we can derive bandwidth-per-core (`bpc`). In the Azure workload, the largest VM has 16 cores. Therefore, we limit `bpc≤11Mbps` (177 Mbps / 16 cores) to ensure that vlinks never exceed 177 Mbps. Furthermore, `bpc` is just a parameter. We can use different values to generate ML workloads with different network demand. The reference VDC workload (`bpc=1Mbps`) consumes between 5,828 Gbps and 6,581 Gbps. This 10% variance is similar to the variance in CPU and RAM resources in the base workload (Section 3.1). The similarity is by design: `bpc` is just a multiplier on the workload's CPU footprint.

Table 3.3 summarizes five steps in the Gridiron technique and our approach in generating a VDC workload that approximates ML training. One can adopt different approaches in one or more steps to generate different VDC workloads. For example, it is possible to change the first step to use sparse connectivity instead of all-to-all connectivity to generate a less network-intensive VDC workload. Similarly, in the second step, one can use different value (not 30) to cap the peak VDC sizes, although this change might require adjustments on the fifth step to avoid network-bound failures. We leave exploring these variations to future work.

In summary, we applied the Gridiron technique to generate a realistic VDC workload with characteristics of the distributed training application. In Chapter 4, we will use this VDC workload (with a realistic datacenter) to evaluate different VDC scheduling algorithms. Future work can apply the Gridiron technique to generate other VDC workloads, and use them for their scheduler evaluations.

3.4 Related Work

We divide the related work into two areas. The first area covers existing techniques for VDC workload generation. We explain how the Gridiron technique differs from the existing techniques. The second area covers publicly available traces from production clouds. We explain our rationale for choosing the Resource Central dataset [20] in the Gridiron technique.

Existing Techniques for Generating VDC Workloads

To the best of our knowledge, no existing work uses production cloud workload traces to construct a VDC workload. Instead, all existing work uses a randomization-based approach for generating VDCs and their lifetimes [8, 21, 59, 120, 133, 135]. For example, Amokrane et al. [8] generate VDCs with 5–200 VMs, where a pair of VMs connect with a probability 0.5 with a bandwidth demand uniformly distributed between 10 and 50 Mbps. The VDC allocation requests arrive according to a Poisson process, and VDC lifetimes follow an exponential distribution. Although synthetic VDC workloads suffice for scheduling algorithm feasibility studies, such as the NETSOLVER evaluation in the previous chapter, these workloads lack important properties, such as VDC mutation, needed to evaluate VDC scheduler performance in practice.

We are the first to generate a VDC workload from production cloud traces where VDC sizes and VDC allocation requests are directly derived from the production traces. The next chapter will show the importance of the realistic VDC workloads for evaluating VDC schedulers in practice.

Rational for Choosing the Resource Central Dataset

We generated VDC workloads based on production traces from the Azure cloud, which are described in and released with the Resource Central paper [40]. Table 3.4 compares features of this dataset with other datasets released by public cloud providers. The Resource Central 2017 dataset is well-suited for VDC workload generation, because a VDC is a collection of *VMs*, and the Resource Central 2017 dataset contains VMs. Moreover, the dataset describes VMs with their absolute number of CPU cores, which is essential for our compute-proportional-

Table 3.4: Publicly Available Production Cloud Workloads.

Workload	Virtualization	CPU cores	Grouping	Volume (million)	Duration	Year	Provider
Resource Central [40]	VM	Absolute	✓	2	30 days	2017	Azure
Resource Central V2 [17]	VM	Buckets	✓	2.6	30 days	2019	Azure
Protean [62]	VM	Normalized		5.5	14 days	2020	Azure
Serverless [114]	Functions		✓	0.6	14 days	2019	Azure
Borg 2011 [110]	Container	Normalized	✓	25.4	30 days	2011	Google
Borg 2019 [127]	Container	Normalized	✓	$\sim$732	31 days	2019	Google
Alibaba 2017 [88]	Container	Absolute	✓	0.09	0.5 day	2017	Alibaba
Alibaba 2018 [126]	Container	Absolute	✓	14.3	8 days	2018	Alibaba

bandwidth generation approach. Lastly, VMs are grouped into “deployments” that we can use as a proxy for establishing VDC VM membership.

The same Azure team released version 2 (V2) of the Resource Central dataset in 2019 [17]. Although this V2 dataset has $\approx$33% more VMs and is more recent, it is ill-suited as the VDC workload source because it does not describe VMs’ CPU cores in absolute terms. In the V2 dataset, the authors anonymized the number of VM CPU cores by grouping the number of cores into six buckets.[6] Although it is possible to generate VDCs’ inter-VM network bandwidth requirements in the same granularity as bucket compute capacities, using the more precise CPU core numbers, via the Resource Central 2017 dataset [40], allows us to construct a more realistic VDC workload.

The Azure cloud team also released a larger dataset, containing 5.5 million VM allocations and deallocations, collected over a period of 14 days. This dataset is described in and released with the Protean paper [62]. There are two disadvantages to using this dataset as a basis for a VDC workload. First, VM CPU cores are not given in absolute numbers. They are normalized to the server that the VM is placed on. For example, if the VM requested 4 cores and the server it is placed on has 40 cores, the VM will be recorded as having 0.1 cores. Reverse engineering this dataset to extract absolute cores cannot be precise because the dataset includes multiple servers configurations and these configurations are not released. Moreover, VMs in the Protean dataset do not have grouping, i.e., all VMs are solo-VMs.

[6]The first bucket contains all VMs with $<$ 2 cores, or bucket1$<$ 2 cores for short, 2 cores $\leq$ bucket2 $<$ 4 cores, 4 cores $\leq$ bucket3 $<$ 8 cores, 8 cores $\leq$ bucket4 $<$ 12 cores, 12 cores $\leq$ bucket5 $<$ 24 cores, and bucket6 $\geq$ 24 cores.

We cannot construct a VDC from solo-VMs (Section 3.1). Therefore, the Protean dataset is also ill-suited for using as the basis for VDC workload generation.

The bottom five datasets in Table 3.4 do not use VMs as the virtualization unit. They use functions, e.g., Azure Functions [19], or containers, e.g., the Azure Container Service [18]. Unfortunately, VM lifetimes are radically different from containers and function lifetimes: VM lifetimes are on the order of *hours*, container lifetimes are on the order of *minutes*, and function lifetimes are on the order of *seconds*.[7] Thus, a container trace or function trace is not an ideal starting point for a VM-based VDC workload.

However, cloud applications are evolving and VDCs might need to be extended to include, or be redefined in terms of, non-VM constructs, such as containers and functions. We do not rule out non-VM VDCs dominating the future of VDCs [74]. Our work focuses on several techniques that are useful for VM-based VDCs. These techniques may serve as the foundation for non-VM VDC research in the future.

3.5 Conclusions

We described the Gridiron technique to construct a realistic VDC workload. We used a VM trace from the Azure production cloud as the base workload and augmented its VMs with network bandwidth requirements. We used the notion of "deployment" that is present in the Azure trace to derive a VM's VDC membership. We considered various VDC topologies and selected all-to-all connectivity. We also capped the peak VDC size to account for realistic application properties.

We proposed the compute-proportional-bandwidth approach to develop a parameterized VDC workload generation mechanism. This mechanism allows us to adapt VDC workloads to different datacenters. For example, we can use this mechanism to scale up a workload's bandwidth requirements to make sure that the datacenter network is not over-provisioned to the point that hinders evaluation of VDC scheduling algorithms' efficacy.

We studied the datacenter-level constraints for VDC workloads. We described three scenarios that may produce network-bound VM allocation failures and pro-

[7]More precisely, 50% of functions run for less than 3 seconds, or p50=3 seconds for short, and p90=60 seconds [114]. Lifetime for containers are: p50=50 seconds and p90=1000 seconds [126]. VMs lifetimes are the longest: p50=900 seconds (15 mins) and p90=86,400 seconds (24 hours) [40].

posed a model to avoid these scenarios. The model can also be used by cloud operators to decide the volume of network bandwidth guarantees to offer to their tenants based on the datacenter capacities.

Finally, we applied the Gridiron technique to generate a VDC workload that captures the characteristics of distributed ML training applications. We capped the peak VDC size at 30 VMs to capture the scalability limitations of the distributed training. The capping allows us to generate realistic VDC workloads, which we use for VDC scheduler evaluation in practice, as we describe in the next chapter.

Chapter 4

VDC Scheduling in Practice

We collaborated with a public cloud provider, Huawei Cloud [68], to evaluate constraint-based VDC schedulers, such as NETSOLVER, in practice. Huawei Cloud operates dozens of datacenters across four different continents [69]. We identified four practical concerns that needed to be addressed to have confidence in attempting to deploy NETSOLVER in practice.

The first practical concern was that NETSOLVER was evaluated using synthetic VDC workloads on datacenters of modest size. As we described in Section 2.4, we used three different VDC workloads to evaluate NETSOLVER: two from the previous literature, Yuan et al. [135] and SecondNet [59], and one from our industrial collaborator: ZeroStack. Even though we used three different sources to capture the essential properties of a realistic VDC workload, the changes in the input sizes, such as a significantly larger Azure-cloud-based VDC workload and the datacenter size needed to accommodate it, warrant a reevaluation of the scheduler.

The Azure-cloud-based VDC workload that we generated in Chapter 3, which we refer to as the *realistic VDC workload* hereafter, is long and large. The workload contains every VM in an Azure datacenter cluster over 30 days. The Azure dataset has two orders of magnitude more VDCs than did the synthetic workloads, i.e., a few hundred VDCs in our synthetic workloads vs. tens of thousands in the Azure workload. The datacenter size needed to accommodate the Azure workload is also an order of magnitude larger than what we used for the NETSOLVER evaluation in Section 2.4, i.e., a few hundred servers vs. a few thousand servers.

These changes in workload and datacenter sizes likely pose scalability challenges for NETSOLVER.

VDCs in the Azure workload are also dynamic: they grow and shrink as VMs join and leave — events that are triggered by tenants allocating and deallocating their VMs. This motivates the second practical concern: the metric used in the VDC scheduler evaluation. The *static* metric, which was used in Yuan et al. [135], SecondNet [59], and Section 2.4, evaluates the number of VDCs allocated on an empty datacenter until the scheduler is no longer able to allocate any VDC. However, real workloads have VDC allocations and *deallocations*. High datacenter utilization is an ongoing objective that matters not only during the initial stage of resource allocation, which the static metric captures, but also during *steady state* operation when tenants request resource allocation as well as deallocation.

The third practical concern is that NETSOLVER is architecturally incompatible with state-of-the-art resource scheduling algorithms commonly deployed in practice, such as OpenStack's Nova filtering-based algorithm [100]. We call OpenStack Nova's filtering-based algorithm NOVAFILTER. NOVAFILTER handles each resource requirement, such as CPU and RAM, separately while NETSOLVER handles all resources together. Adopting a new, NETSOLVER-like, architecturally intrusive approach means diverging from the established industrial approach, which is already deployed and operational.

In theory, it is possible to combine all filtering operations into a set of constraints. Solving these constraints together will be functionally equivalent to running filters. However, constraint-based approaches face several operational challenges, such as scalability and extensibility. For example, extending filtering-based algorithms with an additional filter, such as server network bandwidth filter, adds a predictable VM allocation latency overhead because, in the worst case, the new filter exhaustively searches over all servers. This search completes in a constant time. However, predictability does not hold for constraint solvers. For example, as we show in Section 4.3.5, doubling the number of constraints might increase the constraint solving time by over 1000×. In general, our experience with constraint-solvers is that it is hard to predict the effect of additional constraints on latency. On the other hand, filtering-based algorithms have been demonstrated to be scalable and extensible in Huawei Cloud [77] and Microsoft Azure [62]. That said, cloud operators might consider adopting a constraint-solver-based approach only when it brings a major advantage that is not attainable by incremental changes to the already deployed, filtering-based scheduler.

And that leads to the fourth practical concern: absence of performance comparisons between NETSOLVER and state-of-the-art resource scheduling algorithms, such as NOVAFILTER. Given a realistic VDC workload and scheduler evaluation metric, we can make this comparison. Unfortunately, as we show in Section 4.3.2, NETSOLVER does not scale to the realistic VDC workloads. NETSOLVER's VM allocation latency becomes prohibitively high when a datacenter has over a thousand servers and a VDC has all-to-all connectivity. NETSOLVER's median per-VM allocation latency in such a large datacenter is 19 minutes, which is impractical. Therefore, in this chapter, we design algorithms that scale to the realistic VDC workloads and are compatible with existing cloud resource schedulers.

We address all 4 practical concerns and make the following 4 contributions:

1. **Metrics:** We propose the revenue gain metric for evaluating VDC schedulers. We also make the case for attributing a dollar value for bandwidth, e.g., \$0.58 per Gbps/hour, and use it for revenue gain computation.

2. **Algorithms:** We extend and enhance NOVAFILTER to support end-to-end network bandwidth allocation (STARNET). We demonstrate that enhancing STARNET with locality-awareness (STARNETLA) yields comparable revenue gain as NETSOLVER, which is complete, while offering three orders of magnitude faster resource allocation latency than NETSOLVER.

3. **Prototype:** We integrate STARNETLA into OpenStack by extending OpenStack's Nova scheduler. We demonstrate that our locality-awareness enhancements are compatible with Nova's existing filtering-based architecture.

4. **Optimality:** We develop an ILP-based offline optimal VDC scheduling algorithm: ORACLE. Although it is not possible to deploy ORACLE in practice, we use it to study how closely our practical algorithms approximate the theoretically optimal scheduler. Our experiments show that ORACLE can produce 50% higher revenue gain than STARNETLA.

Table 4.1: Resource Scheduling Algorithms. The “net” suffix signifies an algorithm’s support for end-to-end network bandwidth allocation.

	CPU& RAM	Server Network	End-to-end Network	Complete
NOVASIM (NOVAFILTER)	✓	✓	×	×
STARNET	✓	✓	✓	×
NETSOLVER	✓	✓	✓	✓
STARNETLA	✓	✓	✓	×
STARNETILP	✓	✓	✓	✓
STARNETLAILP	✓	✓	✓	✓

We proceed as follows: First, we describe the evolution of our improved VDC scheduling algorithms. Then, we describe our evaluation methodology, including the revenue gain metric. Next, we present results for all algorithms, including our OpenStack prototype and ORACLE. We place this work in the context of existing work and conclude in Section 4.4 and Section 4.5, respectively.

4.1 Algorithms

We explored a wide spectrum of VDC scheduling algorithms, trying to achieve the combination of architectural compatibility, high datacenter utilization, and acceptably low resource allocation latency. Table 4.1 summarizes the algorithms we consider. We start with NOVAFILTER: the resource scheduling algorithm used in OpenStack. NOVASIM is our implementation of NOVAFILTER in our simulation environment. NOVAFILTER, and therefore NOVASIM, do not support end-to-end network bandwidth allocation. Thus, we extend NOVASIM with end-to-end network bandwidth allocation to produce our baseline network bandwidth guaranteeing VDC scheduler: STARNET. We add locality-awareness and retries to STARNET to produce STARNETLA. Our hybrid algorithms, STARNETILP and STARNETLAILP, combine fast heuristic algorithm, STARNET and STARNETLA respectively, with NETSOLVER, which is complete, to achieve high datacenter utilization and low resource allocation latency. We now elaborate on each algorithm.

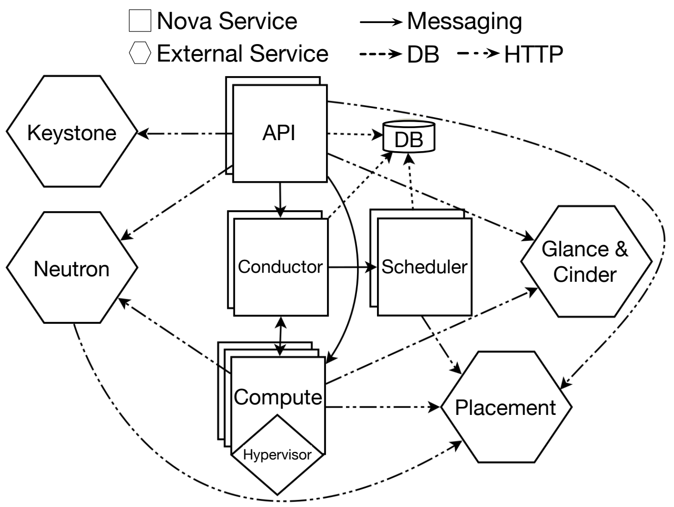


Figure 4.1: OpenStack Architecture. OpenStack modules interact to manage cloud resources. For example, several services of the Nova module, such as API, Conductor, and Scheduler manage CPU and RAM resources. Neutron manages networking. Keystone manages authentication.

4.1.1 NOVAFILTER and NOVASIM

OpenStack is a popular open-source cloud management framework [99]. Figure 4.1 shows OpenStack's architecture.[1] OpenStack's uses filtering-based resource scheduling algorithms. An algorithm takes as input a set of servers and narrows down the set by a sequence of filters. A filter applies a resource constraint, e.g., servers with at least X free CPUs or servers with at least X GB free memory. For example, if VM-New requires four cores, the CPU filter will remove all servers that have fewer than four free cores. The input to the filter pipeline is a list of all datacenter servers or, optionally, a subset of those servers.

Today's OpenStack implementation consists of three filters that are relevant to our VDC scheduling: a filter on CPU, a filter on memory and a filter on a server's network bandwidth. Note that the server network bandwidth filter is different from network bandwidth guarantees required by VDCs because the existing filter is not end-to-end. This filter only reasons about the bandwidth available at the servers.

[1]The figure is based off the OpenStack manual at https://docs.openstack.org/nova/latest/user/architecture.html

Algorithm 1 : VM allocation with NOVAFILTER

```
S: all servers in datacenter, v: to-be-allocated VM
procedure AllocateVM(v)
1:  cpu_passed = {∀ s ∈ S | s.avail_cores ≥ v.cores}
2:  ram_passed = {∀ s ∈ cpu_passed | s.avail_ram ≥ v.ram}
3:  net_passed = {∀ s ∈ ram_passed | s.avail_band ≥ v.band}
4:  candidate_server = Weigher(net_passed)
5:  if candidate_server then
6:      ▷ allocate v & record its resource usage
7:  else
8:      ▷ fail v
end procedure
procedure Weigher(S')
9:  return random.choice(S')
end procedure
```

The end-to-end network bandwidth VDCs require includes not only two communicating servers, but also every network node between them.

OpenStack's CPU and memory filters are part of Nova module and server network bandwidth filter is part of Neutron module (Figure 4.1). We will use the name NOVAFILTER to describe all three filters that exist in OpenStack. Algorithm 1 shows NOVAFILTER's pseudocode. For each VM allocation request, `AllocateVM` starts the search over the entire datacenter server pool and gradually prunes servers with insufficient CPU cores and RAM (Nova), or server network bandwidth (Neutron) (lines 1-3). Once all filters have been applied, the `Weigher` selects the final candidate server based on higher-level placement policies, such as avoiding racks reaching their operational end of life. These policies are deployment-dependant; the default OpenStack weigher chooses a random server from the final feasible set. We also use the random policy (although one could imagine much better policies). If there is at least one server that passes all three filters, the algorithm places the VM on that server (line 6). Otherwise, the VM is not allocated (line 8).

Ideally, we would use NOVAFILTER in a real OpenStack deployment to study its performance in practice, including VM allocation latencies. However, we do not have a datacenter with thousands of servers to replicate the production environment. Thus, we resort to DevStack — an emulation environment OpenStack developers use for functional testing [101].

Unfortunately, DevStack is unfit for testing schedulers at scale. As we show in our experiments in Section 4.3.6, VM allocation latency in DevStack is on the order of 5 seconds for a small datacenter with 192 servers. Thus, processing the entire VDC workload with $\approx$2 million VMs would take around 115 days. Our VM allocation latency measurements in DevStack are an underestimate, because they include only CPU and RAM filters, not the server bandwidth filter shown in Algorithm 1 (line 3). The server bandwidth filter requires enabling OpenStack Neutron, which further increases the VM allocation latency [98]. Our estimate above excludes the latency Neutron would introduce, if enabled.

We integrate our VDC scheduling algorithms to DevStack in Section 4.3.6 and to evaluate algorithms in a small scale. However, for evaluating algorithms in large scale — using VDC workload with $\approx$2M VMs and datacenters with over 6000 servers — we develop a lightweight simulation environment, VDCSIM. VDCSIM architecture is described in Section 4.2.2. We implement NOVAFILTER's filtering functions to run in VDCSIM. We call NOVAFILTER's VDCSIM-tailored implementation NOVASIM. NOVASIM's pseudocode is identical to that of NOVAFILTER, which we described in Algorithm 1. In other words, NOVASIM is identical to NOVAFILTER, except NOVASIM runs in VDCSIM while NOVAFILTER runs in real OpenStack and DevStack deployment. We need to distinguish these two algorithms, because we evaluate scheduler performance in VDCSIM (Section 4.3) and study a scheduler's practical latency in DevStack (Section 4.3.6). VDCSIM's lightweight architecture allows NOVASIM to achieve three orders of magnitude lower VM allocation latencies than NOVAFILTER running in DevStack. Therefore, NOVASIM can process the entire base workload in around 3 hours.

Recall that NOVAFILTER, hence NOVASIM, does not support end-to-end network bandwidth allocation.[2] Thus, we cannot use NOVASIM for VDC scheduling. We extend NOVASIM with end-to-end bandwidth allocation and call it STARNET.

Algorithm 2 : VM allocation with STARNET

```
net_passed: servers that passed server bandwidth filter, v: to-be-allocated VM
procedure AllocateEnd2EndNetwork(v, net_passed)
 1: server = random.choice(net_passed)
 2: vlinks = list()
 3: foreach peer, band in v.peers
 4:   if server == ServerOf(peer) then continue          ▷ peer VMs are colocated
 5:   vlink = AllocateVlink(server, ServerOf(peer), band)
 6:   if vlink == None then DeallocateVlinks(vlinks) return None
 7:   else vlinks.append(vlink)       ▷ vlink alloc. succeeded; handle the next peer
 8: ▷ record vlinks to belong to v; this is outside the for loop
 9: return server                               ▷ successfully connected v to all peers
end procedure
procedure AllocateVlink(src_server, peer_server, band)
10: allocated_band = 0; paths = list()
11: while allocated_band < band
12:   remaining_band = band - allocated_band;
13:   path = Dijkstra(src_server, peer_server)
14:   if path == None then DeallocatePaths(paths) return None
15:   end2end_band = GetMinBand(path)
16:   if end2end_band ≥ remaining_band then band2consume = remaining_band
17:   else band2consume = end2end_band
18:   foreach hop1, hop2 in path
19:     ▷ deduct band2consume from hops; append hops&band2consume to paths
20:   allocated_band += band2consume
21: return paths
end procedure
```

4.1.2 STARNET

STARNET adds end-to-end bandwidth allocation to NOVASIM. We implement it by adding another filter, `AllocateEnd2EndNetwork`, which is called after the server bandwidth filter (Algorithm 1 line 3) but before the `Weigher` (Algorithm 1 line 4). `AllocateEnd2EndNetwork` filter takes the to-be-allocated VM (VM-New) and `net_passed` servers as input and returns one server that can accommodate the end-to-end bandwidth requirements of the VM-New. That server is

[2]As of March 4, 2019, the date when we forked off OpenStack upstream for further development. This is still true in January 10, 2021.

passed to the `Weigher` (Algorithm 1 line 4). Here, `Weigher` is redundant, because `AllocateEnd2EndNetwork` filter returns at most one server, which the `Weigher` is guaranteed to select. We keep `Weigher` for compatibility with the Nova architecture. Cloud operators can change `AllocateEnd2EndNetwork` to return multiple servers and choose the final server using their own weigher. We leave this to future work. Algorithm 2 shows pseudocode for STARNET's end-to-end bandwidth allocation.

`AllocateEnd2EndNetwork` allocates physical path(s) between the *candidate* server and *peer* server(s) to accommodate the virtual link(s) between VM-New and already-allocated VM peer(s). The peer server(s) host VM-New's already-allocated VM peer(s), or VM-Existing VM(s) for short. The candidate server is the one that is selected among `net_passed` servers at random (Algorithm 2 line 1) as the potential host for VM-New. We fail the VM allocation if we make an unlucky random selection. However, as we show in Section 4.3.1, random candidate selection among `net_passed` (without checking end-to-end bandwidth between the servers) works well. Section 4.3.3 improves this further using locality-awareness.

In `AllocateEnd2EndNetwork`, we iterate through VM-Existing to allocate a vlink between each VM in VM-Existing and VM-New (line 3). We first check if VM-Existing is also allocated on the candidate server. If yes, we skip vlink allocation, because we assume that network bandwidth within the same server, i.e., localhost interface capacity, is always available (line 4). If the servers differ, we attempt vlink allocation by calling `AllocateVlink`, which takes three inputs: two server endpoints and the bandwidth amount to connect them with (line 5).

`AllocateEnd2EndNetwork` expects `AllocateVlink` to return one of two values. `AllocateVlink` can return `None` to indicate that vlink allocation is impossible. This could happen, for example, when the peer server (`ServerOf(peer)` in line 5) has less than the required bandwidth (`band`) available. In this case, we deallocate all successful vlink allocations for VM-New, if any, and indicate VM allocation failure by returning `None` (line 6). Alternatively, `AllocateVlink` can return a list of vlinks to indicate successful bandwidth allocation between server endpoints (line 7). Once vlink allocation is successful for all peer VMs, we record these vlinks and return the candidate server to indicate successful placement of the VM-New on it (lines 8–9).

`AllocateVlink` allocates vlinks that are end-to-end between server endpoints, which include top-of-rack and spine level switches. We allocate vlinks on each hop between these two servers, using multi-path, if needed. We allocate a Dijkstra path on an unweighted graph as a shortest path between two servers [96]. `Dijkstra` returns either `None`, to indicate absence of a path between server endpoints (line 14) or a single path with non-zero bandwidth (lines 15–20). We use `GetMinBand` to find the bottleneck hop along the path. The bandwidth of this bottleneck hop is the end-to-end bandwidth available along this path. We repeat `Dijkstra` calls to map the vlink onto multiple paths, if necessary (lines 11–20).

STARNET is the first algorithm in Table 4.1 with end-to-end network bandwidth allocation. We use STARNET as the baseline VDC scheduler in our evaluations (Section 4.3), because it allows us to evaluate the advantages of NETSOLVER over state-of-the-art VDC schedulers in practice. Although the end-to-end bandwidth allocation in STARNET is a contribution of our own, e.g., does not exist in OpenStack, we expect this Dijkstra-based virtual link allocation to be the default way to implement end-to-end path allocation.

Table 4.1 also shows that STARNET is incomplete. STARNET is complete with respect to a VM's CPU and memory requirements, but is not complete with respect to inter-VM bandwidth requirements. Thus, it might fail to find an allocation even when one exists. There are two reasons for its incompleteness. First, it is greedy — it allocates VMs to servers one-by-one, which might cause a bad placement of earlier VMs in a VDC, making it impossible to find servers for placing later VMs in the VDC. Second, VM allocation might fail even though there may have been a *different candidate server* that could have accommodated the VM. This failure happens because end-to-end bandwidth allocation is done *after* the candidate server selection (Algorithm 2 line 1). Although STARNET's server network bandwidth filter guarantees availability of the requested bandwidth at each server (otherwise these servers would not pass that filter), intermediate network hops might not have the required bandwidth available. We show results for different retries in Section 4.3.3.

Figure 4.2(b) illustrates the limitation of the server-only bandwidth filtering. Here, although server S2 and server S3 have sufficient server network bandwidth to accommodate VM v5, no bandwidth is available at the top-of-rack switch level (ToR2) to connect these servers to the peer server S1, which hosts the already-

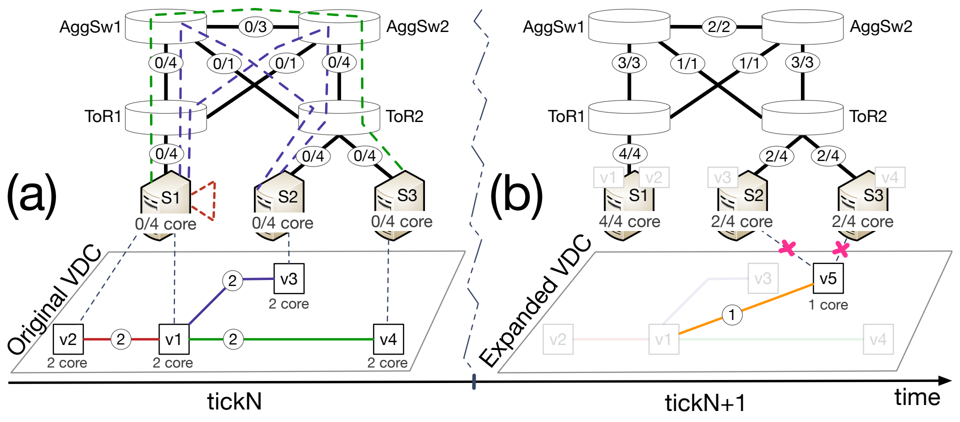


Figure 4.2: Oversubscribed Datacenter Spine: (a) successful allocation of the original VDC, and (b) unsuccessful allocation of the expanded VDC. Datacenter resource capacities are shows as used / original. For example, 1/4 in link bandwidth means that "1" unit of bandwidth is used out of total "4" units. The datacenter is empty in tick N. (This is a slightly modified version of Figure 3.6 on page 62 to illustrate oversubscription.)

placed peer VM v1. Bandwidth scarcity in datacenter upstream networks is common due to oversubscription — modern datacenter topologies are designed to have more network bandwidth capacity on the edge (servers-to-ToR switch) with less capacity on the spine nodes for cost-effectiveness [1, 2, 27].

4.1.3 NetSolver

NetSolver eliminates both sources of incompleteness in StarNet by formulating VDC allocation as a constraint satisfaction problem. The scheduler tries to find a solution that simultaneously maps all VMs in a VDC to servers in a way that satisfies all CPU, memory, and end-to-end bandwidth constraints. As we described in Section 2.3, NetSolver can use one of two back-end constraint solvers: an SMT (Satisfiability Modulo Theories) solver and an ILP (Integer Linear Programming) solver. We use NetSolver with the ILP solver, NetSolver-ILP, as it was shown to be more scalable and faster than NetSolver-SMT (Section 2.4.6).

The weakness of a constraint-solving approach is poor scalability, which surfaces as high VM allocation latency. The ILP solver has a worst-case runtime that is exponential in the model size, and since the model considers placing each VM in the VDC on any server, the model size grows as $\Omega(nm)$, where n and m are the VDC and datacenter sizes, respectively. The synthetic VDCs in Section 2.4 had up to 15 VMs. The realistic VDC workload we described in Section 3.2.2 has up to 30 VMs. Moreover, the realistic VDC workload has all-to-all connectivity, instead of the sparse connectivity in the synthetic VDCs. The density of a VDC topology also influences the ILP solver's model size. As we will see in Section 4.3.2, these added complexities significantly increase VM allocation latency: over 1,140 seconds ($\approx$20m) median per-VM allocation latency for a VDC of size 10 on a datacenter topology with 6,144 servers. NETSOLVER terminates with an out-of-memory error (50 GB) for larger VDC sizes (Figure 4.12).

Observing significant latencies in NETSOLVER, we ask *what is a practical per-VM allocation latency*? Azure cloud operators state 20ms and 100ms to be the typical and maximum per-VM allocation latency budgets [33, 62], respectively. Note that Azure's latency budgets do not account for bandwidth allocation latency because Azure does not offer end-to-end bandwidth guarantees. Although the latency budget is cloud- and deployment-dependent, in an environment where a sub-second latency is the norm [33, 62], one minute is generous. For example, a tenant is unlikely to wait for more than 10 minutes to allocate a VDC with 10 VMs. We use this one minute budget as our practicality threshold and require schedulers' 99th percentile per-VM allocation latency to be under 60 seconds.[3]

We explored a range of options to trade off some completeness for better scheduler latency. In particular, we explored breaking up large VDC requests into smaller batches of VMs, e.g., break a VDC with 30 VMs into six batches of five VMs. The five VMs in each batch are scheduled by the ILP solver all-at-once, so we maintain completeness within a batch — if there is any possible allocation for the batch, the scheduler will find it. We call this the batch-level completeness. However, a bad decision when scheduling an earlier batch could affect the feasibility of scheduling later batches, so there is no completeness guarantee *across* batches. A larger batch

[3]The 99th percentile, or p99 for short, is an arbitrary choice. For example, one could require p99.9 or p100 to be the threshold. However, p99 covers the common cases and one can, in addition, use tail latency reduction techniques, such as running multiple schedulers [62], to further reduce the bottom one percentile latency.

size makes the scheduler more complete, but slower; a smaller batch size makes the scheduler less complete, but faster.

We started with the batch size of 30 VMs and tried smaller and smaller sizes in an effort to make the allocation latency low enough. However, we found that even with a batch size of two, NETSOLVER's 99th percentile VM allocation latency is 110s in a datacenter with 6,144 servers. The 99th percentile latency with batch size of one is 48s, which is below the practical threshold, though still high. We decreased the datacenter size to 192 servers (in four racks; Figure 4.13) to evaluate the latency at a smaller scale. Here, NETSOLVER's 99th percentile latency with batch size of one is 544ms, which we consider to be a practical latency. We show full experimental results in Section 4.3.2. Note that even with a batch size of one, NETSOLVER is still more complete than STARNET because NETSOLVER eliminates the second source of incompleteness: STARNET's inability to consider all servers. Given NETSOLVER's limited scalability for datacenters of up to 192 servers, we develop STARNETLA, which we describe next.

4.1.4 STARNETLA

STARNETLA extends STARNET with two ideas: **l**ocality-**a**wareness and retries. We reflect these optimizations in the algorithm's name by using an "LA" suffix. Locality-awareness tries colocate the communicating VMs (on the same server) to save datacenter network bandwidth. When a VM request arrives, we first check if there is an already-placed peer VM in the network. If so, we try to colocate the new VM with its peer(s). If not, we retry placement on other servers.

We also added a retry heuristic. Failing to allocate a VM after one attempt is pessimistic. Thus, we retry several times. We show experimental results for different retry values in Section 4.3.3. In general, the larger the number of retries, *N*, the closer STARNETLA comes to achieving NETSOLVER's completeness. For example, the brute-force search across all datacenter servers produces completeness. However, increasing *N* also increases VM allocation latency. Increasing *N* too much can make the latency impractical.

`AllocateEnd2EndNetwork` implements both STARNETLA extensions, as we show in Algorithm 3. STARNETLA's locality and retry enhancements are

Algorithm 3 : AllocateEnd2EndNetwork function in StarNetLA

```
v: to-be-allocated VM, N: number of retries
procedure AllocateEnd2EndNetwork(v, net_passed)
 1: sorted_servers = Sort(v, net_passed)
 2: retries = 0
 3: while retries < N
 4:   candidate_server = pop(sorted_servers)
 5:   success = TryCandidateServer(v, candidate_server)
 6:   if success then return candidate_server
 7:   if IsEmpty(sorted_servers) then return None    ▷ no more candidates; fail v
 8:   retries += 1
 9: return None
end procedure
procedure Sort(v, S')
10: savings = map(s ← 0 | ∀ s ∈ S')    ▷ assign 0 savings to all candidate servers
11: foreach peer_vm in v.peers
12:   savings[ServerOf(peer_vm)] += BandwidthOf(v, peer_vm)
13: return sorted(savings, key=lambda x: x[1], reverse=True)
end procedure
procedure TryCandidateServer(v, server)
14: vlinks = list()
15: foreach peer, band in v.peers
16:   if server == ServerOf(peer) then continue          ▷ peer VMs are colocated
17:   vlink = AllocateVlink(server, ServerOf(peer), band)
18:   if vlink == None then DeallocateVlinks(vlinks) return False
19:   else vlinks.append(vlink)      ▷ vlink alloc. succeeded; handle the next peer
20: ▷ record vlinks to belong to v; this is outside the for loop
21: return True                             ▷ successfully connected v to all peers
end procedure
```

applied to the servers that already pass CPU, RAM, and server bandwidth filters. When multiple servers pass the previous filters, we sort them by their *bandwidth savings*, as shown in Algorithm 3 line 1 with the call to `Sort`. In `Sort`, we iterate through every VM-Existing (already-allocated peer VM) to compute the network bandwidth between that and VM-New (to-be-allocated VM) (lines 11-12). These per-candidate-server bandwidth values capture the amount of network bandwidth we save should the VM-New be placed on that candidate server, which means colocation of VM-New and VM-Existing on that candidate server.[4] `Sort` returns the candidate servers in the *savings descending order* such that the server with the highest bandwidth savings is used first, i.e., considered as the colocation server.

STARNETLA's retry optimization considers candidate servers in the order dictated by `sorted_servers`, as shown in Algorithm 3 line 4. Here, we keep popping candidate servers from the top of the list until we find a candidate server that can accommodate VM-New's network bandwidth or until we exhaust the number of retries (*N*) (lines 3-9). Note that `TryCandidateServer` in STARNETLA is almost identical to `AllocateEnd2EndNetwork` in STARNET. Both functions consider placing VM-New on candidate server.

4.1.5 Hybrid Algorithms

Locality awareness and retry optimizations in STARNETLA only *approximate* completeness. We can extend STARNETLA to a complete solution by using NETSOLVER as a fallback scheduler. We call this a *hybrid* approach.

Our hybrid algorithms strive to make the scheduler complete without incurring the high latency of the ILP solver. We begin with a heuristic filter-based algorithm, STARNET or STARNETLA and fall back to the ILP solver approach only when the heuristic fails. We can implement this by making the `AllocateVM` in Algorithm 1 call the ILP solver before failing the VM (in line 8). We call these hybrid methods STARNETILP and STARNETLAILP. In both hybrid algorithms, a VM allocation fails only if the ILP solver is unable to find an allocation.

However, our heuristic algorithms still do VM-at-a-time allocation. Accordingly, when they fall back to NETSOLVER-ILP for allocating an individual VM, NETSOLVER-ILP should also do VM-at-a-time allocation. Recall that NETSOLVER-ILP provides all-or-nothing VDC allocation semantics (Chapter 2). In all-or-nothing VDC allocation, all VDC VMs within the tick fail allocation if any VM fails. The VDC VMs that were already allocated in the earlier ticks remain allocated. (See the full description of all-or-nothing semantics in Section 2.3.) We relaxed NETSOLVER-ILP's all-or-nothing semantics to *best-effort* semantics. With best-effort

[4]Note that the candidate server might be hosting more than VM-Existing. The per-server bandwidth summation in Algorithm 3 line 12 accommodates this multi-peer-hosting by creating a per server entry in the `savings`, not per VM.

semantics, NETSOLVER-ILP tries to allocate as many VMs of the VDC while failing the rest. The best-effort semantics can directly support VM-at-a-time allocation. Here, NETSOLVER-ILP considers all datacenter servers to allocate the VM, guaranteeing the VM allocation, if possible. We call this VM-level completeness. Note that VM-level completeness is theoretically superior to our heuristic algorithms, which consider only one server (STARNET) or a subset of servers (STARNETLA with *N* retries). Our hybrid algorithms provide VM-level completeness. Note that we could begin the ILP solving in parallel with the heuristic algorithm, but we have left that optimization for future work.

The intuition for hybrid algorithms, especially for STARNETLAILP, is as follows: Hybrid algorithms have the potential to reduce VM allocation failures without increasing VM allocation latency in the common case. For example, locality in STARNETLA is an approximation of the locality in NETSOLVER, because the former is restricted to server scope. However, locality in NETSOLVER covers server-, rack-, cluster-, and pod scopes. STARNETLA performs only the initial step. Thus, STARNETLA has less potential to reduce datacenter bandwidth consumption, which might translate into more VM allocation failures, hence, the lower revenue gain. Similarly, retries in STARNETLA are an approximation of the exhaustive search in NETSOLVER, since retries in STARNETLA are bounded (e.g., up to 100 in our experiments in Section 4.3.3) while an ILP solver explores the entire (server) search space.

As we show in Section 4.3.4, this intuition is both elusive and unsuccessful. It is elusive, because our experiments in Section 4.3.3 show that although NETSOLVER is able to allocate the VM when STARNETLA is unable to (this happens $\approx$18% of the time), NETSOLVER's VM allocation latency is prohibitively high (99th percentile is 60.45s). Therefore, in practice, cloud providers would not benefit from falling back to the ILP solver, because they cannot afford to wait for an ILP allocation. Thus, STARNETLAILP being fast in the common case is elusive, since the ILP component is still prohibitively slow. Moreover, the intuition that STARNETLAILP generates higher revenue gain is unsuccessful in the long run, because STARNETLA leaves only small room for improvement. That is, although the number of successful fall backs in which NETSOLVER succeeds are reasonably high ($\approx$18%), the number of VMs allocation failures STARNETLA produces is small

(0.17%). In our experiments (Section 4.3.4), we therefore find that the revenue gain between StarNetLA and StarNetLAILP is statistically insignificant.

In summary, we design hybrid algorithms because they offer completeness, which StarNetLA lacks. However, our experiments show that this completeness is prohibitively slow when exploited and does not actually offer a higher revenue gain. We present our evaluation results in Section 4.3.4.

We presented seven algorithms, five of which are VDC scheduling algorithms. The first two algorithms, NovaFilter and NovaSim, cannot be used for VDC scheduling, because they lack support for end-to-end bandwidth allocation. In the next section, we describe our methodology for evaluating VDC schedulers.

4.2 Evaluation Methodology Overview

Two of our practical concerns are about VDC scheduler evaluation with realistic VDC workloads using a realistic metric. We address the first concern by constructing realistic VDC workloads using the Gridiron technique that we introduced in Chapter 3. We address the second concern by proposing the revenue gain metric. We also use realistic datacenter topologies for our VDC scheduler evaluations. We start by describing these datacenters.

4.2.1 Datacenter Topologies

We use publicly available datacenter topologies. Jupiter is a widely studied topology used in Google's datacenters [117]. The full Jupiter topology can accommodate 64 pods, each with 1536 servers, for a total of 98,304$\approx$100K servers. However, the Jupiter paper [117] does not describe compute and memory specifications of these servers. We use a modern enterprise-grade server with 60 cores and 256 GB RAM, such as Dell PowerEdge R940 [44], in our Jupiter topology.

We use the ML training workload for our scheduler evaluation. Recall that the Gridiron technique allows us to adapt the bandwidth-per-core (`bpc`) parameter to match the VDC workload's network demand to the target datacenter (Section 3.2.3). Also recall that the reference VDC workload used `bpc=1Mbps`.

The full Jupiter topology is too big for our reference VDC workload. The reference VDC workload consumes up to 346,755 cores, 781,767 GB RAM, and 6,581

Gbps bandwidth (Section 3.2.2). With 60 cores, 256 GB RAM, and 40 Gbps network bandwidth per server (specified as a server-to-ToR switch link bandwidth in the Jupiter paper [117]), four pods offer sufficient compute, memory, and bandwidth to accommodate the reference VDC workload. More precisely, four pods with 6,144 servers offer 1.06× extra compute (6,144*60/346,755≈1.06), 2× extra memory (6,144*256/781,767≈2), and 37× extra server bandwidth capacity (6,144*40/6,581≈37) of the reference VDC workload.

The disparity between compute (1.06×) and memory (2×) capacities in a 4-pod Jupiter datacenter reflect reality, because modern datacenters are compute-bound [40]. Network bandwidth disparity is also realistic by construction, as the server-to-ToR switch and other node bandwidths are taken directly from the full Jupiter topology. Moreover, the VDC workload's network bandwidth requirement is configurable. We scale it up to evaluate the efficacy of the VDC schedulers in handling network bandwidth requirements (Section 4.2.6). Thus, the experimental results from evaluating VDC schedulers on the 4-pod Jupiter datacenter will qualitatively hold for other datacenters. We give more elaborate description of 4-pod and full Jupiter datacenter topologies in Appendix C.

4.2.2 VDC Scheduler Simulator: VDCSIM

Simulation-based cloud scheduler evaluation is common (e.g., [40, 55, 62]). In fact, the Resource Central [40] and Protean [62] papers state that scheduler evaluation in a simulator is a production deployment prerequisite. In an absence of the existing simulator for VDC scheduling, we built our own lightweight simulator, VDCSIM.

Figure 4.3 shows the VDCSIM architecture, which is designed to evaluate algorithms in a plug-in fashion. We pass a runtime flag to indicate the algorithm we want to evaluate. VDCSIM replays the workload from a JSON file and outputs allocation results to another JSON file. The Replayer can operate in two modes: VDC-at-a-time or VM-at-a-time. In VDC-at-a-time mode, the Replayer feeds an entire VDC to the scheduler for processing. Analogously, in VM-at-a-time mode, the Replayer feeds individual VM events to the scheduler.

We run VDCSIM on a Dell PowerEdge R940 server with 2.30 GHz (30 MB L3 cache) Intel Xeon E7-4870 v2 processor with 60 cores across four NUMA nodes.

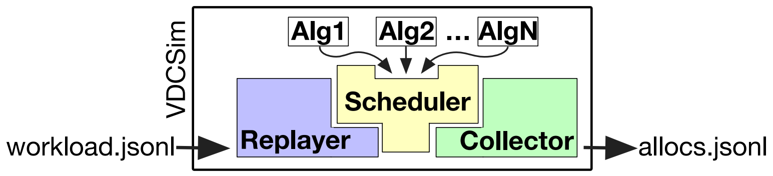


Figure 4.3: VDCSim Architecture. The Replayer consumes a tick from the workload and makes resource (de)allocation requests to the Scheduler. The Scheduler performs (de)allocation decisions and passes the outcome to the Collector. The Collector keeps outcomes in memory until the entire tick is processed. The Collector writes all processed events to the output file once all events in the tick are processed.

The server has 512 GB RAM that is uniformly distributed across four NUMA nodes (128 GB each). All the algorithms are single threaded, so we disable hyperthreading. Thus, all algorithms use only one CPU core. The host OS is Ubuntu 20.04.1 LTS with Linux 5.4.0-58-generic kernel. Most of our experiments use under 10 GB of RAM, so, we generously impose a 20 GB limit on RAM.[5] Some experiments with an ILP solver require over 20 GB memory. We relax the 20 GB restriction for these experiments so that no experiment fails due to memory restrictions, unless stated otherwise.

4.2.3 Revenue Gain Metric

Cloud providers should gain extra revenue when they monetize datacenter network bandwidth in addition to the compute, memory, and other resources they already monetize today. However, revenue gain is not always guaranteed, because a subset of VMs can fail allocation when datacenter network bandwidth is insufficient to accommodate the VDC workload's bandwidth requirements.

Revenue gain quantifies how much a cloud provider will benefit from selling network bandwidth guarantees. It is computed in percentages relative to the baseline revenue. Formally,

$$gain = (computeRevenue + networkRevenue)/baseRevenue \tag{4.1}$$

[5] We use the runlim tool to enforce memory restrictions [123].

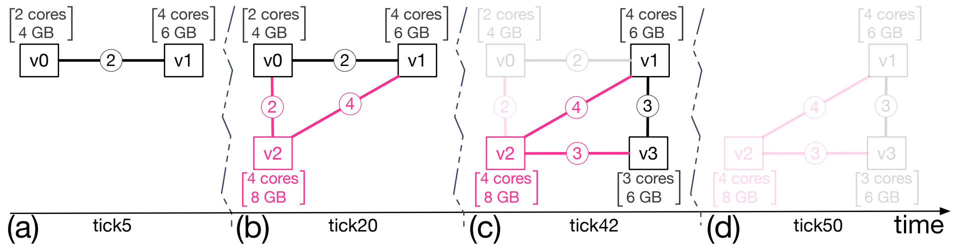


Figure 4.4: Example VDC Workload: (a) shows VDC allocation with two VMs, (b) shows VM v2 allocation, (c) shows VM v0 deallocation and VM v3 allocation, and (d) shows deallocation of all VMs in the VDC. We show VM v2 and its vlinks in red. (This figure is a modified version of Figure 3.3 on page 56.)

Here, the base revenue (`baseRevenue`) is a revenue generated from selling VMs with only compute resources (CPU and RAM). Our VDC schedulers fail to allocate VMs only because of network bandwidth requirements. Since the base workload has no network bandwidth requirements, `baseRevenue` includes revenue from all VMs. The compute revenue (`computeRevenue`) similar to `baseRevenue` in that it covers the revenue from compute resources, but only for VMs that are successfully allocated. Thus, `computeRevenue<baseRevenue` holds when a scheduler fails VM(s), otherwise `computeRevenue=baseRevenue`. Analogously, `networkRevenue` is a revenue generated from selling network bandwidth for successfully allocates VMs.

We illustrate the revenue gain computation for the example workload in Figure 4.4. Assuming no VM allocation failures and given that tick duration is 5 mins, we compute `baseRevenue` as the sum of compute revenue for all VMs:

- VM v0: lives for 37 ticks, from tick 5 to tick 42. The VM's lifetime is $37 * (5/60)hour = 3.08hour$. If VM's price is \$0.2/hour, the revenue from this VM is \$0.62 (from $3.08 * 0.2$).
- VM v1: lives for 45 ticks, from tick 5 to tick 50. The VM's lifetime is $45 * (5/60)hour = 3.75hour$. If VM's price is \$0.4/hour, the revenue from this VM is \$1.5 (from $3.75 * 0.4$).
- VM v2: lives for 30 ticks, from tick 20 to tick 50. The VM's lifetime is

$30*(5/60)hour = 2.5hour$. If VM's price is \$0.5/hour, the revenue from this VM is \$1.25 (from $2.5*0.5$).

- VM v3: lives for 8 ticks, from tick 42 to tick 50. The VM's lifetime is $8*(5/60)hour = 0.67hour$. If VM's price is \$0.3/hour, the revenue from this VM is \$0.2 (from $0.67*0.3$).

Thus, $baseRevenue = \sum_{i\in\{v0,v1,v2,v3\}} vmRevenue_i = (0.62+1.5+1.25+0.2) = 3.57$. Hence, the workload's `baseRevenue` is \$3.57.

Now we compute the nominator in Equation 4.1 for the example workload in Figure 4.4. When no VMs fail, the workload's `computeRevenue` is equal to `baseRevenue`. Hence, `computeRevenue` is also \$3.57. `networkRevenue` quantifies the revenue from network bandwidth guarantees, which consist of revenue from successfully allocated vlinks. When no VMs fail, no vlinks fail. Conversely, in this work, a vlink failure is always accompanied by the VM failure because bandwidth allocation is a prerequisite for VM allocation. (See our discussion on VM allocation failures in practice in Appendix D.) Thus, `networkRevenue` is the sum of vlink revenues, which are computed as follows:

- vlink v0-v1: lives for 37 ticks, from tick 5 to tick 42. The vlink's lifetime is $37*(5/60)hour = 3.08hour$.
- vlink v0-v2: lives for 22 ticks, from tick 20 to tick 42. The vlink's lifetime is $22*(5/60)hour = 1.83hour$.
- vlink v1-v2: lives for 30 ticks, from tick 20 to tick 50. The vlink's lifetime is $30*(5/60)hour = 2.5hour$.
- vlink v1-v3: lives for 8 ticks, from tick 42 to tick 50. The vlink's lifetime is $8*(5/60)hour = 0.67hour$.
- vlink v2-v3: lives for 8 ticks, from tick 42 to tick 50. The vlink's lifetime is $8*(5/60)hour = 0.67hour$.

Note that vlink is a pairwise construct. Therefore, a vlink's lifetime depends on the lifetimes of the two VMs it connects, say, VM `src` and VM `dst`. Formally,

$$lifetime_{vlink} = overlap(lifetime_{vlink_{src}}, lifetime_{vlink_{dst}})$$

Assuming that the unit of bandwidth in this workload is `bpc=1Mbps` and network bandwidth price (`bwPrice`) is \$10 per 1 Gbps/hour (\$0.01 per 1 Mbps/hour), the bandwidth-hours consumed is:

- vlink v0-v1: $2Mbps * 3.08hour * \$0.01Mbps/hour = \$0.0616$.
- vlink v0-v2: $2Mbps * 1.83hour * \$0.01 * Mbps/hour = \$0.0366$.
- vlink v1-v2: $4Mbps * 2.5hour * \$0.01Mbps/hour = \$0.1$.
- vlink v1-v3: $3Mbps * 0.67hour * \$0.01Mbps/hour = \$0.02$.
- vlink v2-v3: $3Mbps * 0.67hour * \$0.1 * Mbps/hour = \$0.02$.

Thus, $networkRevenue = \sum_{j \in \{v0-v1, v0-v2, v1-v2, v1-v3, v2-v3\}} vlinkRevenue_j = (0.0616 + 0.0366 + 0.1 + 0.02 + 0.02) = \$0.2382$. Assigning `computeRevenue`, `networkRevenue`, and `baseRevenue` values to Equation 4.1 we get $gain = (3.57 + 0.2382)/3.57 = 1.0667 = 6.67\%$.

Now we demonstrate `gain` computation in presence of VM allocation failures. Assume that in Figure 4.4, VM v2 fails allocation because the datacenter network does not have enough bandwidth to accommodate it. This failure causes VM v2's `computeRevenue` and `networkRevenue` loss. Note that `baseRevenue` does not get affected because it captures the VM revenues without network bandwidth guarantees; should the VM v2 not require network bandwidth guarantees, it would not have failed. We omit revenue for VM v2 and its vlinks (v0-v2, v1-v2, v2-v3) in the earlier equations to compute `gain` as follows:

$$gain = (computeRevenue + networkRevenue)/baseRevenue =$$
$$((0.62 + 1.5 + 0.2) + (0.0616 + 0.02))/3.57 = 0.67 = -33\%.$$

In this case, the revenue generated from selling network bandwidth guarantees ($0.0616 + 0.02 = \$0.0816$) was not enough to cover the revenue loss from VM v2 allocation failure (\$1.25). Thus, we have -33% revenue gain.

The general formula to compute `baseRevenue`, `computeRevenue`, and `networkRevenue` are as follows:

$$baseRevenue = \sum_{i \in allVMs} (vmPrice_i * lifetime_i)$$

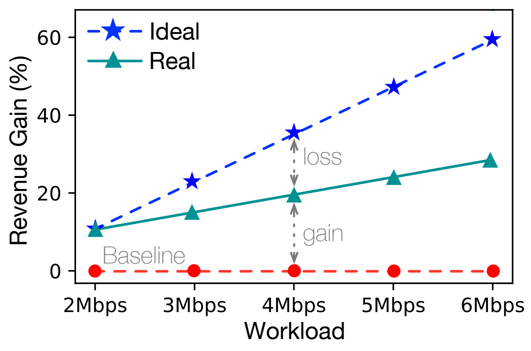


Figure 4.5: Revenue Gain Example. These results are not from an experiment. They are manual drawings for illustration purposes.

$$computeRevenue = \sum_{j \in allocatedVMs} (vmPrice_j * lifetime_j)$$

$$networkRevenue = \sum_{k \in allocatedVlinks} (bwAmount_k * lifetime_k * bwPrice)$$

We use a simple example to demonstrate how different schedulers achieve different revenue gains. Figure 4.5 shows the percent revenue gain, relative to the baseline (`Baseline`), as a function of the value of `bpc`. The red line with circles is `Baseline` that does not provide bandwidth guarantees: thus it is a horizontal line at zero revenue gain. Now, assume that the total revenue for allocating all VMs without bandwidth guarantees (`baseRevenue`) is \$100 and that we charge \$0.01 per 1 Mbps/hour (`bwPrice`). Consider an ideal scheduler that never fails a VM The blue line with stars (`Ideal`) shows that as `bpc` increase, so does revenue gain: linearly with the ratio of bandwidth price to VM price. Next, consider a realistic scheduler (`Real`) that fails VMs, it's revenue will fall somewhere in the shaded region between `Ideal` and `Baseline`. For example, the turquoise line with triangles shows what happens when we have:

- `bpc=2Mbps`: `Real` fails no VMs. We generate \$12 from selling network bandwidth guarantees (`networkRevenue`) and `computeRevenue` is equal to `baseRevenue` (\$100). Thus, per Equation 4.1, $gain = (100 + 12)/100 = 1.12 = 12\%$.

- `bpc=3Mbps`: `Real` fails 3%. We have `networkRevenue=`\$19 and `computeRevenue`=\$97. Thus, per Equation 4.1, $gain = (97+19)/100 = 1.16 = 16\%$.

- `bpc=4Mbps`: `Real` fails 4%. We have `networkRevenue`=\$24 and `computeRevenue`=\$96. Thus, per Equation 4.1, $gain = (96+24)/100 = 1.2 = 20\%$.
- `bpc=5Mbps`: `Real` fails 5%. We have `networkRevenue`=\$29 and `computeRevenue`=\$95. Thus, per Equation 4.1, $gain = (95+29)/100 = 1.24 = 24\%$.
- `bpc=6Mbps`: `Real` fails 6%. We have `networkRevenue`=\$35 and `computeRevenue`=\$94. Thus, per Equation 4.1, $gain = (94+35)/100 = 1.29 = 29\%$.

Note that in this simple example, we assume identical VDC configurations to illustrate the basic revenue gain metric. Also note that in Figure 4.5, `gain` with network-intensive VDC workloads `bpc`={`4Mbps, 5Mbps, 6Mbps`} still generate a positive `gain` despite the non-zero revenue loss, because the extra revenue collected from selling network bandwidth outweighs the loss. This outweighing is contingent on the price of network bandwidth guarantees. A low price could be insufficient to cover the loss and result in a net negative revenue gain. We now discuss prices for VMs and network bandwidth guarantees.

4.2.4 VM Pricing

We use VM pricing rates from the Azure cloud since we generated our VDC workload from the Azure cloud traces. The Azure trace contains 16 VM flavors (Table 4.2): see “vCPU cores” and “RAM” columns (in italics) that are part of the trace. The other two columns, “Flavor Name” and “Price (\$/hour)”, are not part of the trace and we describe them below. The first column, “Flavor #”, is added for ease of reference to each flavor.

Recall that the Azure traces were collected over 30 days, starting from November 16, 2016. We retrieved Azure’s hourly VM rates for that period from Azure’s pricing page snapshot saved in the Internet Archive’s Wayback Machine [94]. The

Table 4.2: Virtual Machine Pricing in Azure Cloud. The "vCPU cores" and "RAM" columns are part of the Azure workload. We use VM prices in the "Price" column in this work.

Flavor #	Flavor Name	*vCPU cores*	*RAM (GB)*	Price ($/hour)
1	A0 Basic	1	0.75	0.018
2	A1 Basic	1	1.75	0.044
3	A1 v2 Standard	1	2	0.043
4	A2 Basic	2	3.5	0.088
5	A2 v2 Standard	2	4	0.091
6	D11	2	14	0.175
7	A2m v2 Standard	2	16	0.149
8	A3 Basic	4	7	0.176
9	A4 v2 Standard	4	8	0.191
10	D12	4	28	0.35
11	A4m v2 Standard	4	32	0.297
12	A4 Basic	8	14	0.352
13	A8 v2 Standard	8	16	0.4
14	D13	8	56	0.7
15	A8m v2 Standard	8	64	0.594
16	D14	16	112	1.387

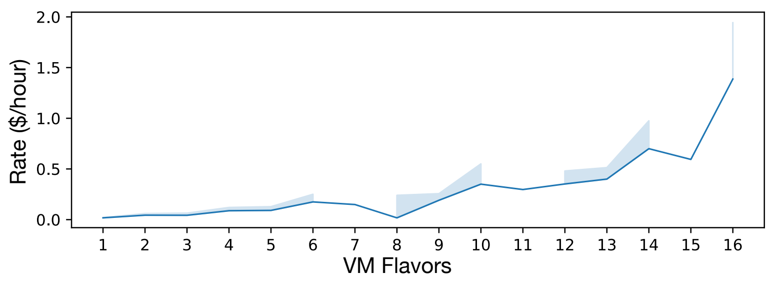


Figure 4.6: Virtual Machine (VM) Pricing in Azure Cloud. We show hourly rates on Dec. 24, 2016. The solid line connects minimum rates for all VM flavors. Spikes on the line show the price range for that VM flavor.

Wayback Machine has multiple snapshots in that period. We use the Dec. 24, 2016 snapshot as it is the first one after the trace collection start day.

Figure 4.6 shows price ranges for all 16 VM flavors in the Azure trace. The flavor indices are consistent across Table 4.2 and Figure 4.6. For example, flavor #8 in Table 4.2 shows the properties of that flavor, including the hourly price we use for it, while flavor #8 in Figure 4.6 shows the possible price range for that flavor. Figure 4.6 shows price range because the Azure traces contain incomplete information about the VMs. As we explained in the base workload description (Section 3.1) and showed in Table 4.2, the Azure traces contain only the amount of CPU and RAM. The trace omits other resources VMs might have had such as storage space and networking capabilities. For example, as we can see in Figure 4.6, the lowest hourly rate for flavor D14 with 16 cores and 112 GB RAM is \$1.387, while the highest rate is \$2.14, for flavor H16r (not shown). The H16r flavor is more expensive, because it has 2,000 GB storage space and is equipped with low-latency network interface (RDMA) while D14 has only 800 GB disk space and no low-latency networking. We cannot tell which flavor the Azure trace contains, because the disk space and other information, are not recorded in the trace. The collected trace could actually contain both D14 and H16r flavors but collapse them into the single flavor in the released dataset.

However, the released information is sufficient to compare revenue gain across different VDC schedulers. The revenue gain of VDC scheduler A and VDC scheduler B is identical if both schedulers fail identical VMs (e.g., as in Figure 4.4) and identical VM pricing is used for both schedulers. In theory, when schedulers fail different VMs, it is possible for scheduler A's revenue gain to be higher than scheduler B's revenue gain with pricing X, and vice-versa with pricing Y.[6] However, we do not expect minor variations in pricing to change our qualitative results in Section 4.3, because the number of failed VMs are 60× higher with scheduler A versus scheduler B.[7] Thus, for all of our evaluations (Section 4.3), we use the minimum rates for all VMs, as shown with the continuous line in Figure 4.6.

[6]As a contrived example, consider pricing X where VM1 is \$1/hour and VM2 is \$2/hour. Scheduler A successfully allocated all VMs, except VM1. Thus, scheduler A's gain is (`max_gain-$1`). Similarly, scheduler B successfully allocated all VMs, except VM2. Thus, scheduler B's gain is (`max_gain-$2`). Therefore, with pricing X, scheduler A is better than scheduler B (`A>B`). If we use pricing Y where VM prices are swapped, our scheduler preference will also be swapped. That is, with pricing Y, VM1 is \$2/hour and VM2 is \$1/hour. Just like before, scheduler A fails VM1 and scheduler B fails VM2. Thus, scheduler A's gain is (`max_gain-$2`) and scheduler B's gain is (`max_gain-$1`). Hence, scheduler B is better than scheduler A (`A<B`).

[7]For example, in Figure 4.17(b), when `bpc=6Mbps`, STARNET fails 190,640 VMs out of 1,960,300 VMs (9.73%) and STARNETLA fails 3,427 VMs out of 1,960,300 VMs (0.17%) .

4.2.5 Virtual Network Bandwidth Guarantee Pricing

Charging for Network Bandwidth Guarantees

Oktopus was the first work to propose virtual network bandwidth guarantees as a *standalone* cloud service for which tenants can be charged independently from the compute service [21]. The Oktopus authors' observation is that the cost of network bandwidth is *already* part of the bill that tenants pay for the compute service, because even though it is not directly visible to a tenant (the invoice does not list the cost of the network bandwidth) the total running time for application on a VDC depends on the network. For example, some data intensive job might complete twice as quickly if the VDC were given sufficient network bandwidth to avoid stalling on network I/O. Every second the VM stalls for network I/O, the compute service generates no value but incurs additional compute cost. Tenants might actually reduce their bill by explicitly paying for network bandwidth guarantees so network I/O stalls do not happen. Building on this idea, we develop a model to show how much the cloud operators can charge for network bandwidth guarantees and how much revenue they can generate from offering these guarantees.

A Case Study with ML Training Application

We present a case study demonstrating that billing tenants for the network bandwidth guarantees does not change cloud *affordability*, at least for a subset of cloud applications that share performance characteristics with the application we study. A network bandwidth price is called affordable if tenants get an equivalent or better utility by paying for the network bandwidth guarantee (compared to best-effort networking offered today). Analogously, cloud providers get revenue *neutrality* if the price tenants pay for that utility, e.g., completing a data processing job, does not change with and without network bandwidth guarantees.

We use the ML training workload in P3 [72] for our case study. The P3 authors run ML training workloads on a cluster of g3.4xlarge Linux VMs, where each VM has 16 vCPUs, 122 GB memory, and up to 20 Gbps (best-effort) network bandwidth using EC2's "enhanced networking" feature [14]. The g3.4xlarge VMs cost \$1.14/hour[8]. However, Uta et al. observe that the actual bandwidth available is significantly lower when VMs generate continuous traffic [128]. For example, EC2's c5.large flavor is listed as offering up to 10 Gbps but achieves only 1 Gbps when Uta et al. make c5.large VMs generate continuous traffic. Although Uta et al. do not analyze the g3.4xlarge flavor, which the P3 authors use, it is likely to behave similarly, because both c5.large and g3.4xlarge flavors use EC2's enhanced networking feature. For illustration purposes, we assume that g3.4xlarge VMs offer 2 Gbps continuous network bandwidth.

The P3 authors show that network-intensive ML training workloads complete 2–3$\times$ faster with consistent 5 Gbps inter-VM bandwidth instead of consistent 2 Gbps bandwidth. Thus, we can conclude that *network-intensive ML workloads' job completion time decreases by at least half with 5 Gbps bandwidth compared to 2 Gbps bandwidth*. Therefore, for this workload, selling network bandwidth guarantees at the same price rate as the compute service does not change cloud affordability for tenants, because bandwidth guarantees allow tenants to shorten their VM runtimes. In other words, a 2$\times$ increase in tenant billing rate is canceled out by 2$\times$ shorter VM rental time. Given that the g3.4xlarge VM's hourly price is \$1.14, the tenants can pay \$1.14/hour for 5 Gbps bandwidth guarantees and get the same utility from the cloud provider. Thus, the bandwidth cost is \$1.14 for 5 Gbps/hour, which we can use for deriving the price of 1 Gbps/hour, as follows:

`bwPrice` = (\$1.14 / 5) Gbps/hour = \$0.228 per 1 Gbps/hour

Note that this is a pessimistic view on cloud affordability, because tenants' willingness to pay the same amount for completing the same job in a shorter time is an underestimate. In reality, tenants might be willing to pay more when their jobs complete more quickly, in addition to the bandwidth guarantee price they have already paid. Thus, our pricing for bandwidth is conservative, because we do not attribute any dollar cost for the speedup introduced by the bandwidth guarantees.

Our case study demonstrates that the cost of 1 Gbps bandwidth can be a function of the VM's compute cost, depending on how much performance improvement the guaranteed 1 Gbps bandwidth adds to the application running on the VM. If a VM's performance, e.g., job completion time, increases twice with 1 Gbps band-

[8]As of March 3, 2021 in US East (Northern Virginia) AWS Region [14]. Also, note that g3.4xlarge is the VM used only in our case study. It is not the VM flavor in Azure traces.

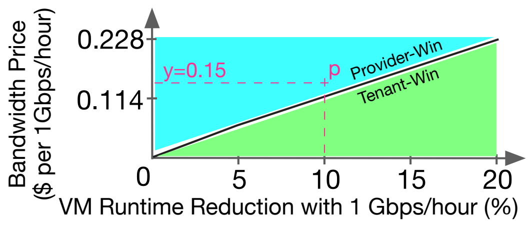


Figure 4.7: Network Bandwidth Price in ML Training Application. The line shows the equilibrium pricing that offers a revenue neutrality for the cloud provider without changing cloud affordability for the tenants. The region below the line (Tenant-Win) makes the bandwidth price more affordable for the tenants. The region above the line (Provider-Win) offers a higher provider revenue by charging tenants more for the bandwidth.

width, we can maintain tenant's affordability and cloud provider's revenue neutrality by pricing 1 Gbps bandwidth at the identical rate of the VM. Similarly, if VM's performance increases only by 10%, 1 Gbps should cost 10% of the VM's rate.

Figure 4.7 visualizes the relationship between the bandwidth guarantee price and a VM's compute price for g3.4xlarge VMs in the ML training application. We call the prices on the equilibrium line *justified* because they offer revenue neutrality (for cloud providers) without changing cloud affordability (for the tenants). For example, when bandwidth is priced at \$0.228 per 1 Gbps/hour (\$0.228 $\times$ 5 = \$1.14 for 5 Gbps/hour) and the VM speeds up by 20% with a 1 Gbps bandwidth (2$\times$ with 5 Gbps), a tenant's \$0.228 per 1 Gbps/hour expense is canceled out by 2$\times$ shorter VM rental time. Similarly, assuming a linear VM speedup with a unit of bandwidth, a tenant's \$0.114 per 1 Gbps/hour payment is canceled out by 10% reduction of VM runtime. The prices that are outside the line are not justified. For example, if a tenant pays \$0.15 per 1 Gbps/hour when the application they run on g3.4xlarge VM speeds up by only 10%, as shown with the point *p* in Figure 4.7, a cloud provider's revenue would increase at the tenant's expense. Thus, prices above the equilibrium line are bad for the tenants (decreased cloud affordability) and prices below the line are bad for the cloud provider (negative revenue gain).[9]

[9]We use a game-theoretic terminology for readability. The revenue from network bandwidth guarantees does not always have zero-sum nature for tenants and providers. For example, as we show in Section 4.3.1, cloud providers can operate their datacenter network in a moderate utilization level so that their revenue increases without extra expense on the tenants.

We apply the justified bandwidth price to all VDC workloads to study the worst case scenario from a VDC scheduling perspective. The worst case scenario happens when datacenter network bandwidth is insufficient to accommodate all tenant requests such that a subset of requests fail. Assigning a uniform bandwidth (i.e., compute-proportional-bandwidth) to all VDCs and attributing the same 1 Gbps/hour dollar price to the entire cloud workload lets us evaluate the worst case (minimum) revenue gain produced by the VDC scheduler. If part of the cloud workload does not require network bandwidth guarantees, i.e., the workload consumes only CPU and RAM (as data analytics workloads in Ousterhout et al. [104]), the bandwidth offered by the existing best effort networking setup would be sufficient and would not cause any failures. This is analogous to running a network-light workload in the cloud where no VMs fail due to network bandwidth scarcity. As we show in Section 4.3.1, these network-light cloud workloads, e.g., `bpc=2Mbps` VDC workload, have only positive revenue gain. Therefore, augmenting the entire cloud workload with network bandwidth requirements and studying the revenue gain using that workload is about evaluating VDC schedulers in the pathological setting. The reality, where not all VDCs require network bandwidth, can only be better. That is, the revenue gain in reality is always higher than the one in the pathological case, which means that the VDC scheduler we propose will do better (more revenue) than what we demonstrate in our evaluations (Section 4.3).

The case study is constructive. It shows an example for pricing network bandwidth guarantees by deriving 1 Gbps/hour price without changing the cloud affordability for the tenants. Cloud providers can use this example to encourage tenants to start using bandwidth guarantees in their (network-heavy) applications. This example is analogous to EC2-beta advertising the three-tier web application as a sample use case of the cloud VMs [26] or like adding GPU (or SSD, persistent memory, or any other new hardware) service in the cloud: only applications that benefit from GPUs use the newly added GPU service; others do not. The case study paves the way for an incremental adoption of a network bandwidth guarantee service.

Finally, our case study with the ML training workload is descriptive, not prescriptive. We do not require characteristics of training application to hold in other workloads. For example, Ousterhout et al. observe that several cloud data analytics workloads have little dependency on the network so that even infinite bandwidth would improve application performance by only 2% [104]. Although this might seem like it contradicts the findings by Uta et al. [128], P3 [72], TicTac [63], and ByteScheduler [106], the contrasting findings only point to the fact that cloud workloads are more diverse than any individual study.

In summary, our case study with ML training application shows that *both* tenants and cloud providers will benefit from a network bandwidth guarantees service. The case study outlines a methodology for deriving a justified price for 1 Gbps/hour service. Later, we use this methodology to derive the 1 Gbps/hour service price in other cloud environments. Moreover, augmenting the entire cloud workload with network bandwidth guarantees, as in our case study application, and applying the bandwidth price to the entire workload allows us to stress test the VDC scheduler under conditions that produce the largest VM allocation failures. Finally, our case study is constructive. It shows the kind of existing cloud applications that are ripe for adopting a network bandwidth guarantee service.

Network Bandwidth Price in the Reference VDC Workload

Now we apply our findings from the case study to scheduling a VDC workload on our 4-pod Jupiter datacenter. Here, we restrict discussion to the parts that are relevant for deriving bandwidth cost in the Azure workload and defer the more elaborate discussion (of algorithm evaluation results) until Section 4.3.

Figure 4.8(a) compares a cloud provider's revenue for VDC workloads with and without network bandwidth guarantees, using the baseline VDC scheduling algorithm, STARNET, on a 4-pod Jupiter datacenter. Each line in Figure 4.8(a) represents a workload with a different bandwidth demand. Higher `bpc` values correspond to higher bandwidth demand, because we scale each vlink's bandwidth proportionally to the number of VM vCPUs. Note that we study the effect of only bandwidth price on the revenue. We keep the VM prices constant (Table 4.2). One could also consider changing VM prices, which would also influence revenue. However, many other factors outside our control influence VM pricing, such as capital and operational expenses for servers. Thus, we keep VM prices constant in our study.

Figure 4.8(a) also shows a cloud provider's revenue change when a unit of

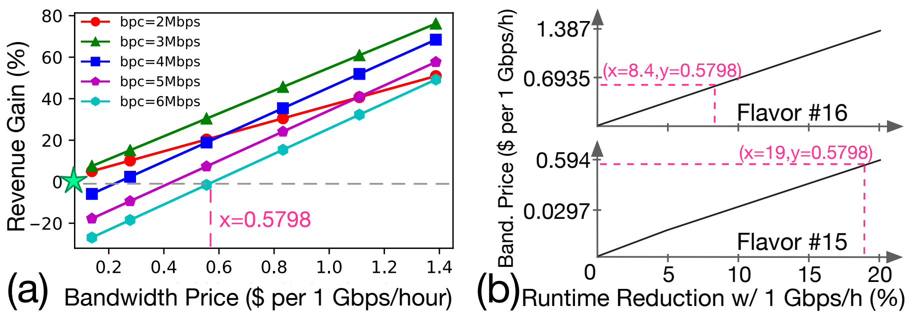


Figure 4.8: Effect of Network Bandwidth Price: (a) shows how cloud provider's revenue changes when 1 Gbps/hour network bandwidth guarantee is priced differently across various VDC workloads (`bpc=[2-6]Mbps`) that are allocated on the 4-pod Jupiter datacenter using StarNet, (b) shows expectations on VM performance improvements to make the \$0.5798 per 1 Gbps/hour price justified. Note that in figure (a), for readability, we label the horizontal axis ticks differently from the bandwidth price we use in our experiments. Thus, the markers on lines are misaligned with the ticks. The actual prices used for this experiment are (0.1387, 0.2774, 0.5548, 0.8322, 1.1096, 1.387).

bandwidth (Gbps) is priced differently. The origin (marked with a star) shows the revenue with best-effort networking offered today where tenants do not explicitly pay for network bandwidth. The `bpc` lines show the revenue change when tenants explicitly pay for network bandwidth guarantees. (Note that offering network bandwidth guarantees also increases VM allocation latencies, which Figure 4.8(a) does not show, but we discuss it in Section 4.3.1.)

Recall that in Section 4.2.5, we derived 1 Gbps/hour price by using the price of the VM in the case study application from P3 [72]. Specifically, we used AWS EC2 g3.4xlarge VM's price, \$1.14/hour, to derive \$0.228 per Gbps/hour price because a 5 Gbps network bandwidth guarantee increased VM's performance (or shortened the VM allocation time) by 2×. Our VDC workload is based on the traces from the Azure cloud, not AWS, so we cannot use the EC2 g3.4xlarge VM's price as the reference point. However, we can imagine an identical ML training application running in the Azure cloud. We derive the price of 1 Gbps/hour bandwidth guarantees with this assumption. As we can see in Table 4.2, the D14 VM flavor in the Azure cloud has identical CPU and RAM specs as the EC2 g3.4xlarge VM.

Table 4.2 also shows that the Azure D14 VM costs \$1.387/hour. Assuming that the Azure D14 VM has a similar performance profile as the EC2 g3.4xlarge VM, the price of the 5 Gbps/hour bandwidth guarantee in the Azure cloud should also be equivalent to D14 VM's hourly rate (\$1.387/hour). This gives the (\$1.387 per 5 Gbps/hour) = (\$0.2774 per 1 Gbps/hour) bandwidth guarantees in the Azure cloud.

The horizontal axis in Figure 4.8 covers a range of prices for 1 Gbps/hour, including the \$0.2774 per 1 Gbps/hour price point. The lowest price in this range (\$0.1387 per 1 Gbps/hour) captures the case when a 5 Gbps network bandwidth guarantee improves application performance by 10% (1.387/0.1387); hence, 1 Gbps improves the performance by 2%. On the other end, the highest price in this range (\$1.387 per 1 Gbps/hour) captures the case when a 5 Gbps network bandwidth guarantee improves application performance by 100% (1.387/1.387); hence, 1 Gbps improves the performance by 20%.

We study the revenue change with different VDC workloads to find the 1 Gbps/hour price that provides cloud revenue neutrality across all studied workloads. For example, Figure 4.8(a) shows that in the `bpc=2Mbps` workload, a cloud provider's revenue increases 5–50% compared to the best-effort case offered today. The extra revenue is generated from the monetization of bandwidth. Figure 4.8(a) also shows that a cloud provider loses revenue when a VDC workload's network demand is too high and the bandwidth price is too low. The loss happens when some VM allocations fail due to insufficient datacenter network capacity. Unallocated VMs account for the lost revenue. For example, the highest revenue loss of 27% happens in the most network-intensive VDC workload (`bpc=6Mbps`) and the lowest bandwidth price of \$0.1387 per 1 Gbps/hour, because the scheduler fails to allocate the largest number of VMs (9.73% of VMs; not shown) in this case.[10]

However, cloud providers can recover the lost revenue by increasing the bandwidth price. For example, \$0.5798 per 1 Gbps/hour and higher rates increase cloud provider's revenue for all workloads shown in Figure 4.8(a). Another way to interpret this rate is that if cloud providers want to utilize their datacenter network as highly as in the `bpc=6Mbps` workload (where $\approx$10% of VMs might fail allocation due to bandwidth scarcity), they should price 1 Gbps/hour service as 42%

[10]There are 190,640 failed VMs out of 1,960,300 VMs (9.73%) in the workload (Section 3.1).

(0.5798/1.387) of D14 flavor's price to remain revenue neutral. From a tenant's perspective, it means that tenants should not buy network bandwidth guarantees unless their workload performance improves by at least 42% with 1 Gbps bandwidth.

Figure 4.8(b) illustrates the effect of $0.5798 per 1 Gbps/hour rate from a tenant's perspective. The top plot shows the equilibrium line for the D14 flavor (flavor #16 in Table 4.2) where 1 Gbps/hour bandwidth guarantee improves D14 VM's performance (reduces its runtime) by 8.4% (42% for 5 Gbps). Hence, the $0.5798 per 1 Gbps/hour price is justified. If the tenant decides to run the application on a different VM flavor, that flavor's performance speedup should increase to make the $0.5798 per 1 Gbps/hour price justified. The bottom plot in Figure 4.8(b) demonstrates the bandwidth price versus performance speedup for flavor #15 (Table 4.2). A flavor #16 VM's compute rate ($1.387/hour) is more expensive than flavor #15's ($0.594/hour), so X% performance speedup in an expensive VM saves more compute money; hence, more money is available to pay for bandwidth guarantees. However, a VM with a lower compute rate saves less money with X% performance speedup; hence, less money is available to pay for bandwidth guarantees. Therefore, an application running on flavor #15 VMs should achieve higher performance speedup than the application running on flavor #16 VMs to make the $0.5798 per 1 Gbps/hour price justified. The bottom plot in Figure 4.8(b) shows that an application running on the flavor #15 VMs should achieve 19% performance speedup to justify the $0.5798 per Gbps/hour price. The lower speedup point for the $0.5798 per 1 Gbps/hour price goes above the equilibrium line, which make the cloud less affordable for the tenant (Figure 4.7).

Note that the bandwidth price versus performance speedup points above apply to the datacenter that accommodates `bpc=6Mbps` VDC workload, i.e., the datacenter network utilization is high. A justified price for 1 Gbps network bandwidth guarantee would be lower, or higher, should the datacenter network utilization be lower, or higher, respectively. For example, if a cloud provider runs `bpc=2Mbps` like workload that generates low network datacenter utilization (where no VMs fail; Figure 4.8(a)) a justified price for 1 Gbps/hour service can be 10% (0.1387/1.387) of D14 flavor's compute price ($0.1387 per Gbps/hour), or even 1%, to increase the cloud provider revenue. This low pricing allows tenants to purchase network bandwidth guarantees even if their workload (or VM) gets only

1% performance speedup from 1 Gbps network bandwidth guarantees.

The exact network bandwidth price depends on multiple factors, including the utilization level cloud providers operate their datacenter network at, the significance of bandwidth guarantees for tenant applications, the price of VM flavors (without bandwidth guarantees), and even the tenants' behavior when a cloud provider introduces VM flavors with network bandwidth guarantees. In our earlier example, the VDC workloads with different network demand (`bpc=[2-6]Mbps`), the case study with an ML training application, Azure VM pricing (Table 4.2), and the range of 1 Gbps/hour price rates in Figure 4.8 illustrate one plausible scenario. Although, cloud providers are likely to cap their datacenter network utilization to a similar level as the `bpc=4Mbps` VDC workload to avoid substantial VM allocation failures. This is similar to how compute services are operated today, i.e., VM scheduling failures in the Azure cloud normally do not exceed 0.1% [40].[11]

In summary, the price of a 1 Gbps/hour network bandwidth guarantee is deployment dependent. Cloud providers need to choose a price by following the factors we have outlined above, including the significance of network bandwidth guarantees for VDC applications and the datacenter network utilization levels providers target. In the rest of this chapter, we use \$0.5798 per 1 Gbps/hour price for all VDC workloads. As we showed in Figure 4.8, \$0.5798 per 1 Gbps/hour price point is the lowest price that achieves cloud provider revenue neutrality for the VDCs workload shown in that figure. Although all of our experiments use \$0.5798 per 1 Gbps/hour price, we round this number to \$0.58 and use a reader friendly \$0.58 value in the rest of this dissertation for readability. We use the same 4-pod Jupiter datacenter and VDC workloads in `bpc=[2-6]Mbps` range for evaluating VDC schedulers, as we explain next.

4.2.6 VDC Workloads for Scheduler Evaluation

In our VDC scheduler evaluation, we allocate VDC workloads on a 4-pod Jupiter datacenter. We use the bandwidth-per-core (`bpc`) parameter to adapt the reference VDC workload to the 4-pod Jupiter topology and to vary the workloads' network demand. The `bpc` parameter plays a key role in deciding compliance of the VDC

[11]The difference between VM allocation failures in this work versus the "failures" (under-performance operation) in the Resource Central paper [40] is discussed in Appendix D.

workload with the network-bound failure avoidance model in the Gridiron technique (Section 3.2.5). In that model, we need to input values for the peak VDC size (`P`) and capacity of the fattest server-uplink (`C`) to derive the maximal vlink bandwidth (`B`). Recall that `P=30` in our reference VDC workload (Section 3.3). Given that `C=40,000` Mbps in the Jupiter topology (because all server uplinks are 40 Gbps as described in Section 4.2.1), we get the following from Equation 3.4:

$$B \leq C/(P/2)^2 = 40,000/(30/2)^2 \approx 177.78 \text{ Mbps}$$

which means that it is possible to allocate VDCs without causing network-bound failures as long as vlinks do not exceed the 177 Mbps bandwidth cap.

The compute-proportional-bandwidth approach that we used for deriving vlink bandwidths makes each vlink's maximal bandwidth a function of the number of VM cores. Given that the most compute-intensive VM flavor in the Azure traces has 16 cores (Table 4.2), we need to satisfy the `bpc≤11Mbps` (177 Mbps / 16 cores) constraint. Note that VM allocation failures can still happen, because a datacenter network is oversubscribed, i.e., even though server uplinks have network bandwidth to accommodate more VMs, the network links above ToR switches might become a bottleneck, causing VM allocation failures.

We use `bpc=[2-6]Mbps` range based on empirical evidence. Recall that the reference VDC workload has `bpc=1Mbps`. Our experiments with the baseline algorithm, StarNet, showed that StarNet is able to fully accommodate the reference VDC workload on a 4-pod Jupiter datacenter, i.e., StarNet produced no VM allocation failures. Thus, we generated VDC workloads with higher `bpc` values, i.e., more network-intensive workloads, so that StarNet fails a subset of VMs, and we can evaluate if other algorithms can do better. We used `bpc=[2-10]Mbps`, and saw that values above `bpc=6Mbps` produce over 10% VM allocation failures (e.g., 17% failures for `bpc=7Mbps`), which we thought to be unrealistically high. Thus, we decided to use a `bpc=[2-6]Mbps` range in our evaluations.

In summary, this section outlined our methodology for VDC scheduler evaluation. First, we described full and 4-pod Jupiter datacenter topologies on which we allocate our VDC workloads. Second, we described the lightweight simulation environment, VDCSim, that allows rapid evaluation of VDC scheduling algorithms. Third, we described the revenue gain metric for VDC scheduler evaluation. Fourth, we showed how we derive pricing for VM flavors and 1 Gbps/hour network bandwidth guarantees. We used an example ML training application to justify the bandwidth price and described how cloud providers can use this example application to pave the way for adoption of the network bandwidth guarantees service. Finally, we generated VDC workloads by using the network load parameterization mechanism in the Gridiron technique (Section 3.2.3). Next, we use these VDC workloads for evaluating several VDC scheduling algorithms.

4.3 Evaluation Results

We answer six research questions designed to address all four concerns regarding the VDC scheduler deployment in practice:

1. **What is the latency overhead of end-to-end network bandwidth allocation in the baseline algorithm, STARNET?** Today, public cloud operators, such as Microsoft Azure, budget under 100ms to allocate a VM [33, 62]. In Section 4.3.1, we evaluate VM allocation latencies with and without network bandwidth requirements. Our experiments show that the VM allocation latency budget needs to be increased by an order of magnitude to accommodate end-to-end bandwidth allocation. However, as we show in Section 4.3.6, the scheduler latency conservatively accounts for less than 0.1% of the total VM allocation time in our OpenStack prototype; increasing our budget to 1s still (conservatively) consumes only 15% of the total VM allocation time.

2. **Does NETSOLVER scale to our environment?** In Section 4.3.2, we evaluate NETSOLVER's VM allocation latency and find that NETSOLVER does not scale to datacenters with over 6,000 servers, which is the size of datacenter needed to accommodate our realistic VDC workloads. NETSOLVER's VM allocation latency is practical only in a small datacenter, e.g., 4-rack datacenter with around 200 servers.

3. **How does STARNETLA compare with STARNET in terms of revenue gain and VM allocation latency?** STARNETLA is an enhanced version of STARNET with locality-awareness and retries. As we will see in Section 4.3.3, STARNETLA generates up to 63% higher revenue than STARNET by reducing VM allocation failures by up to 9%. STARNETLA also reduces

the tail (99th percentile) VM allocation latency by up to 45% by colocating 38% more virtual links than STARNET.

4. **How do hybrid algorithms, STARNETILP and STARNETLAILP, compare to STARNETLA in terms of revenue gain and latency?** In Section 4.3.4, we evaluate hybrid algorithms, combining STARNET variants with NETSOLVER to demonstrate the revenue and the latency difference between hybrid algorithms and STARNETLA is statistically insignificant.

5. **How far is STARNETLA from optimal?** In Section 4.3.5, we design an ILP solver based VM allocation engine that we call ORACLE. ORACLE is a clairvoyant offline algorithm, e.g., it uses the full workload to minimize VM allocation failures. We show that ORACLE can produce 50% higher revenue gain than STARNETLA. Although ORACLE is not practical, because clairvoyance is not practical, it shows that STARNETLA works quite well, although there is still room for improvement.

6. **How much should existing cloud management frameworks change to support end-to-end bandwidth allocation?** In Section 4.3.6, we extend OpenStack to support end-to-end bandwidth allocation. Our prototype shows that OpenStack Nova's existing filtering-based scheduler readily accommodates the end-to-end bandwidth allocation filter. We allocate VDCs in our prototype and show that the extra latency introduced by bandwidth allocation is insignificant in the context of the full VM allocation pipeline latency.

Note that most of this section focuses on the VM instead of the VDC because our VDC scheduler evaluation metric, revenue gain, relies on VMs. As we demonstrated in Section 4.2.3, the revenue gain metric is descriptive. It captures the full compute and network bandwidth capacity the scheduler provided. Using VDCs, on the other hand, is a too coarse grained and ill-suited unit for evaluating a scheduler's efficacy. The VDC metric neither captures the compute capacity of the VDC VMs nor their inter-VM network bandwidth requirements. The only place we use a VDC as the unit of resource allocation is in Section 4.3.2 where we evaluate the scalability of NETSOLVER's VDC-at-a-time allocation feature. However, even

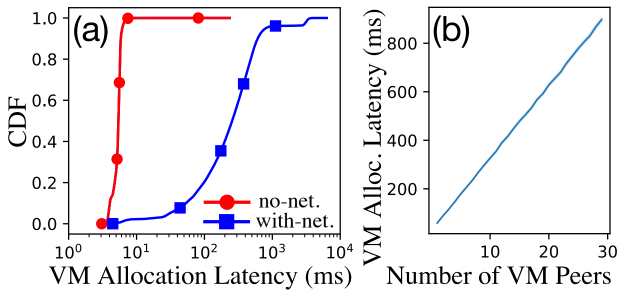


Figure 4.9: End-to-end Network Bandwidth Allocation Overhead: (a) shows the CDF of latencies for workloads with and without network bandwidth requirements, and (b) shows latencies as a function of the number of peers of a VM. The line in (b) shows latencies for all VMs with confidence internals; medians are in bold. Confidence intervals are invisible in the plot because of their tight bounds.

there, we evaluate the scheduler's efficacy in terms of revenue gain and per-VM allocation latency.

4.3.1 STARNET

We evaluate STARNET's VM allocation latencies and revenue gain. Recall that STARNET extends NOVASIM with the end-to-end bandwidth allocation feature. This Dijkstra-based multi-path allocation is also inherited by STARNETLA and our hybrid algorithms. Thus, it is important to understand the latency overhead of this feature. We first study this overhead using the reference VDC workload (`bpc=1Mbps`). Afterwards, we demonstrate STARNET's VM allocation latencies on VDC workloads with higher network demands (`bpc=[2-6]Mbps`). We also compare STARNET's revenue to the case where the cloud provider does not monetize network bandwidth.

The Latency Overhead of End-to-end Network Bandwidth Allocation

We evaluate the latency overhead by comparing scheduler latencies for workloads with and without network bandwidth requirements. We use the reference VDC workload as the networked workload and remove its network bandwidth requirements to generate a workload without networking (*no-network* workload).

Figure 4.9 shows StarNet's VM allocation latencies when allocating the reference VDC workload (`bpc=1Mbps`) on the 4-pod Jupiter datacenter. As we can see in Figure 4.9(a), end-to-end bandwidth allocation increases the median VM allocation latency by 47$\times$ and the tail (99th percentile (p99)) latency by $\approx$470$\times$. StarNet has a 5.41ms median VM allocation latency on the no-network workload and a 255ms (47$\times$) median VM allocation latency on the networked workload. In the p99 range, StarNet's 6.86ms latency (470$\times$) on no-network workload soars to 3,288ms on the networked workload. Here, and in other experiments we analyze p99 latency, not p100. One could apply tail latency reduction techniques, such as running multiple schedulers [62], to further reduce the tail one percentile latency. Moreover, all of our schedulers are single-threaded. One could explore improving latencies by using multiple threads. We leave these improvements to future work.

Figure 4.9(b) shows VM allocation latencies as a function of the number of peers of a VM. For example, a VM with 10 peers requires allocating 10 virtual links (vlinks). The maximum number of vlinks a VM can have is 29 because we cap the peak VDC size at 30 in our VDC workloads. We omit solo-VMs, which have no peer VMs, and VMs with colocated vlinks, which do not actually allocate a vlink, from Figure 4.9(b) because they obscure the relationship between a VM's allocation latency and its number of vlinks. The reference VDC workload has 2.22% solo-VMs and StarNet colocates 3.53% of the non-solo VMs. Thus, 5.75% of the VMs are omitted from Figure 4.9(b). As we can see in Figure 4.9(b), VM allocation latency grows linearly with the number of its (non-colocated) vlinks. For example, a VM with 10 peers takes around 320ms to be allocated while a VM with 20 vlinks takes around 640ms. Thus, we can conclude that a vlink allocation in the 4-pod Jupiter datacenter takes around 32ms. Figure 4.9(b) also plots the confidence intervals for VM allocation latencies, which are barely visible.

VM Allocation Latencies with Other VDC Workloads

Figure 4.10 shows StarNet's VM allocation latencies when allocating the full VDC workloads (`bpc=[2-6]Mbps`) in the 4-pod Jupiter datacenter. The full VDC workload has $\approx$2M VMs and latencies are shown for a single run of the experiment. Overall, our observations in Figure 4.9 hold for these workloads as

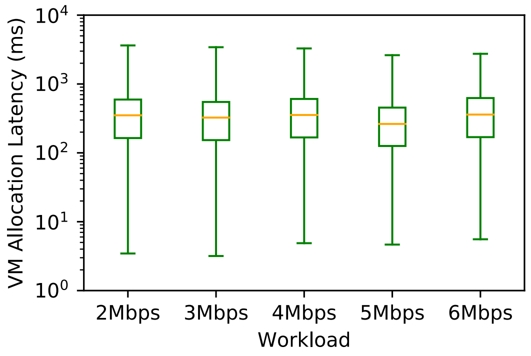


Figure 4.10: VM Allocation Latencies in StarNet. The latency boxes show the first and third quartiles, and whiskers show the min and 99th percentile. The horizontal line inside the box is the median.

well. StarNet's median latency across all workloads is around 300ms while p99 is around 3s. These latencies show that the 100ms VM allocation latency budget adopted by cloud operators today [62] needs to be relaxed for VDC allocation with end-to-end network bandwidth.

Figure 4.10 also shows a noticeable variation in VM allocation latencies. We attribute these variations to the non-deterministic host server selection during VM placement, which can place two communicating VMs on servers with varying network diameter across the datacenter. Here, the time consumed by the Dijkstra-based path allocation is proportional to the network diameter, i.e., allocating a vlink with a shorter diameter is faster.

Revenue Gain

Figure 4.11 shows StarNet performance when allocating full VDC workloads in the 4-pod Jupiter datacenter. Figure 4.11(a) shows revenue gain[12] and Figure 4.11(b) shows VM allocation failure percentages that cause StarNet's revenue gain to fall short of the Ideal. StarNet's revenue is identical to that of Ideal in the `bpc=2Mbps` VDC workload, because StarNet fails no VM allocations. The revenue gains diverge when StarNet start to fail VMs due to insufficient datacenter network bandwidth. The divergence is 0.13% with the `bpc=3Mbps` work-

[12]Our plots do not show absolute revenues. The Baseline revenue is US $11,416,553.

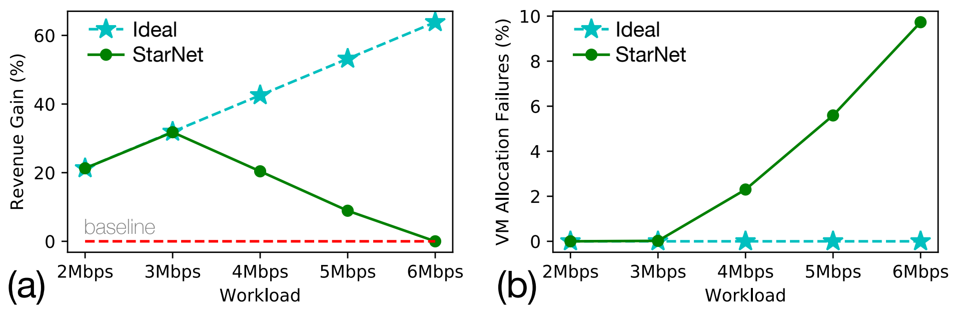


Figure 4.11: Revenue Gain with STARNET: (a) revenue gain and (b) VM allocation failures that cause the revenue loss. Results are for allocating full VDC workloads in the 4-pod Jupiter datacenter.

load but is invisible because the percentage of failed VMs is small (0.02%: 397 out of 1,960,300 VMs). The divergence grows as STARNET fails to allocate more VMs, 9.73% VMs fail with the `bpc=6Mbps`, and drops to almost zero, because the extra revenue generated by network bandwidth guarantees in successful VMs is canceled out by the lost revenue from the failed VMs.

Note that the revenue equality between STARNET and the Ideal in `bpc=6Mbps` workload is not accidental. They match because we intentionally priced the network bandwidth guarantees, $0.58 for 1 Gbps/hour such that the cloud provider remains revenue neutral with and without network bandwidth guarantees. In other words, should the bandwidth guarantees be priced differently, e.g., lower (or higher), or slightly more (or less) VMs fail due to non-determinism in STARNET, the provider would end up with a negative (or a positive) revenue gain.

STARNET Summary

In summary, the 100ms VM allocation latency budget adopted by cloud operators today needs to be relaxed or implementation get improved for VDC allocation with end-to-end bandwidth. For example, allocating a VM with 29 vlinks (or peers) can take around 1s, which is an order of magnitude higher than the current latency budget. In Section 4.3.3, we show how VM allocation latency can be reduced by favoring VDC VM colocation, but the tail (p99) latency is still on the order of a second. In Section 4.3.6, we prototype network bandwidth allocation in OpenStack, which shows that scheduler latency on the order of a second might be acceptable in practice, because this latency is insignificant when we consider the time consumed by other OpenStack modules during VM allocation.

Cloud providers can adjust their revenue by pricing network bandwidth guarantees differently. Setting the price right is particularly important for maintaining (cloud provider) revenue neutrality when datacenter network bandwidth is scarce, which happens when cloud providers operate their datacenter networks at high utilization. On the other hand, when datacenter network bandwidth is not scarce, cloud providers can generate up to 32% revenue gain (`bpc=3Mbps` in Figure 4.11(a)) by offering network bandwidth guarantees.

4.3.2 NETSOLVER

STARNET fails VMs because it is not complete. NETSOLVER is complete but does it scale? Scalability is the major limitation of constraint-solver based resource allocation tools, such as NETSOLVER [31]. Depending on the VDC size, the constraint solver might take over a dozen minutes to allocate a VM, because the size of the encoded ILP constraints grows in proportion to the number of VMs in the VDC. We quantify this growth in our environment by measuring per-VM allocation time as a function of VDC size. We use all-or-nothing VDC allocation semantics where all VDC VMs within the tick fail allocation if any VM fails. The VDC VMs that were already allocated in the earlier ticks remain allocated. For example, AWS CloudFormation, a tool for creating and managing cloud application stacks, allows customers to delete all VMs in their stack if any one or more VMs fail [107]. (See description of all-or-nothing semantics in Section 2.3.)

The Micro-benchmarking Results

We micro-benchmark NETSOLVER to evaluate its scalability in a controlled environment. Recall that the datacenter size required to accommodate the reference VDC workloads is at least 6$\times$ bigger than the datacenters we used to evaluate NETSOLVER in Chapter 2: $\approx$6,000 servers versus $\approx$1,000 servers. Moreover, the reference VDC workload has dense connectivity; the VDC VMs are connected in an all-to-all topology (Figure 3.2).

We evaluate NETSOLVER's scalability by manually constructing two kinds of workloads: VDC workloads and no-network workloads. Each VDC workload has only one VDC containing *N* VMs created in one tick and deleted in the next tick. Each VM has 8 cores, 16 GB RAM, and 1 Mbps network bandwidth to every other VDC VM in an all-to-all topology. We also evaluate the VM allocation latency using a no-network workload, because it allows us to deconstruct the VDC allocation latency into the VM allocation component and the vlink allocation component. We generate the no-network version of each VDC workload by removing the network bandwidth requirements from all VMs. This is similar to how we generated the no-network workload for STARNET evaluation (Section 4.3.1). We allocate workloads on an empty 4-pod Jupiter datacenter where each server has 60 cores and 256 GB RAM (Section 4.2.1). Note that only 7 VMs (with 8 cores each) can be colocated on a server; beyond this the server becomes CPU-bound. Also note that, no allocation fails in this micro-benchmark as the 4-pod Jupiter topology (with 6,144 servers) has ample space to accommodate even the largest VDC (with 30 VMs). We do not batch VMs either: VDC VMs are allocated as all-at-once.[13]

Figure 4.12 shows VM allocation latencies for workloads with and without network bandwidth requirements. As we can see, the median VM allocation latency for the no-network workload ranges between 13ms for VDCs of size 1 and 8,505ms when the VDC size is 15. The experiments for VDC sizes of 20 and higher terminate by hitting the memory limit of 50 GB. We analyze NETSOLVER's vlink allocation latency for the completed runs, instead of rerunning the experiments with a higher memory limit. Notice a significant jump in VM allocation latency for the VDC workloads. The jump is between VDC size 7, with under 1 second median VM allocation latency, and VDC size 8, with $\approx$300s median VM allocation latency. The jump is due to VDC size 8 exceeding a server's maximum colocation capacity. Recall that each server in the 4-pod Jupiter datacenter has 60 CPU cores, which can hold up to 7 VMs, each with 8 cores. When the VDC size is 8, the 8th VM gets allocated on another server that requires NETSOLVER to allocate

[13]Recall that "batching" groups multiple VMs of the VDC for allocation at the same time. For example, a VDC allocation request with 30 VMs can be broken into six batches of five VMs (Section 4.1.3). The all-or-nothing VDC allocation semantics still apply with batching, i.e., a VM failure in any batch (e.g., the 24th VM; in the 5th batch) will fail the entire VDC.

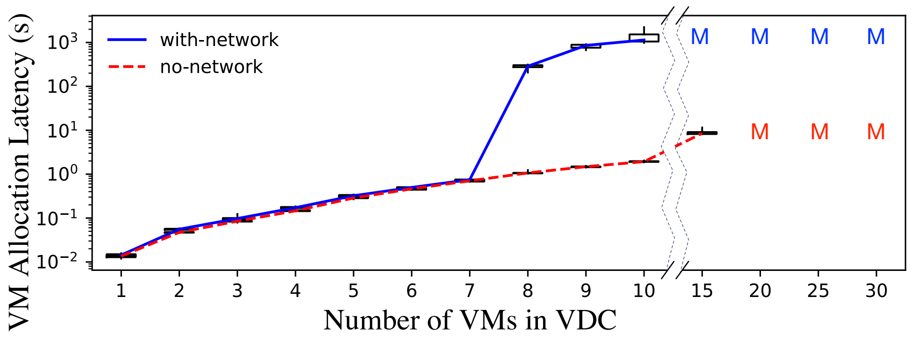


Figure 4.12: VM Allocation Latencies with NETSOLVER. Here, NETSOLVER allocates two kinds of VDC workloads in the 4-pod Jupiter datacenter. We allocate the whole VDC at once and divide that latency by the VDC size to get the average per-VM latency. We plot the results of 10 runs for each data point. The boxes show the first and third quartiles, and whiskers show the min and max values. "M" stands for out-of-memory with 50 GB RAM.

7 vlinks between the two servers. We do not observe this phenomena when VDC size is 7 because NETSOLVER colocates all VDC VMs, completely obviating the vlink allocation. The VM allocation latency grows further for VDC size 9 with almost 1000s median VM allocation latency, and VDC size 10 with well over 1000s median latency, after which NETSOLVER terminates due to memory limit (50 GB).

We derive NETSOLVER's vlink allocation latency by contrasting latencies in the VDC workload and in its no-network counterpart. More precisely, we take the difference between VM allocation latency in a VDC size N workload and in no-network workload with N VMs. The outcome gives us the allocation latency for $(N-1)$ (non-colocated) vlinks that are present in the VDC workload but not in its no-network counterpart. For example, when VDC size is 8, the median VM allocation latency is 288s while it is $\approx$1s in the no-network workload with 8 VMs. Thus, NETSOLVER's vlink allocation latency is $\approx$41s (288/7) when VDC size is 8. The vlink allocation latency grows further for larger VDCs: 106s when VDC size is 9 and 126s when VDC size is 10. Therefore, we conclude that NETSOLVER's vlink allocation latency in the 4-pod Jupiter datacenter is at least 41 seconds, which is three orders of magnitude (1280$\times$) higher than that of STARNET (32ms).

The micro-benchmarking results show that NETSOLVER does not scale to a

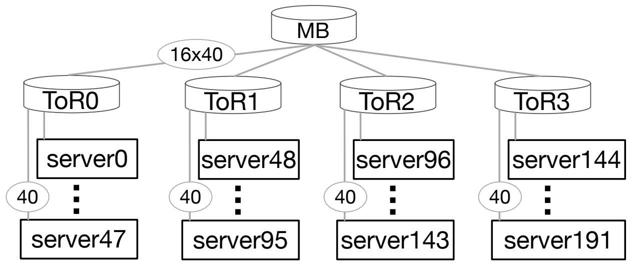


Figure 4.13: Four-rack Jupiter Datacenter Topology.

4-pod datacenter. Its VM allocation latency is prohibitively high when VDC VMs cannot be colocated, e.g., allocating a VM with only two non-colocated vlinks takes at least 82 seconds, which exceeds the per-VM latency budget of 60s we consider to be practical (Section 4.1.3). For example, 82 seconds are 12× higher than VM allocation latency (6.5s) in our OpenStack prototype (Section 4.3.6).

Smaller Datacenter

We study NETSOLVER's scalability at smaller scales. There are two ways to reduce NETSOLVER's ILP model size so that latency will drop: smaller VDCs or smaller datacenters. Reducing the VDC size is too restrictive for the tenants. As we described in VDC workload generation (Section 3.2.2), popular machine learning applications that commonly run in the cloud today, including the one we presented in our case study (Section 4.2.5), require dozens or more VMs in a VDC. Thus, we explore the second option: reducing the datacenter size. This option is practical because, cloud management frameworks support splitting a datacenter into multiple allocation units, e.g., OpenStack Cells [102].

Figure 4.13 shows a scaled down Jupiter datacenter with four racks. The 4-rack Jupiter topology maintains the rack-level properties of the 4-pod Jupiter datacenter that we use in our full experiments. Just like in the 4-pod datacenter, servers in the 4-rack datacenter are connected to one top-of-rack (ToR) switch with 40 Gbps links. This gives 16x40G downlink capacity to each ToR switch. At the same time, each ToR switch connects to a single Middle Block (MB) switch with 16x40G links. A Jupiter MB offers 64x40G downlink capacity, which suffices for four ToR switches: 16x40G uplink for each ToR switch (Section 4.2.1). Thus, ToR switches

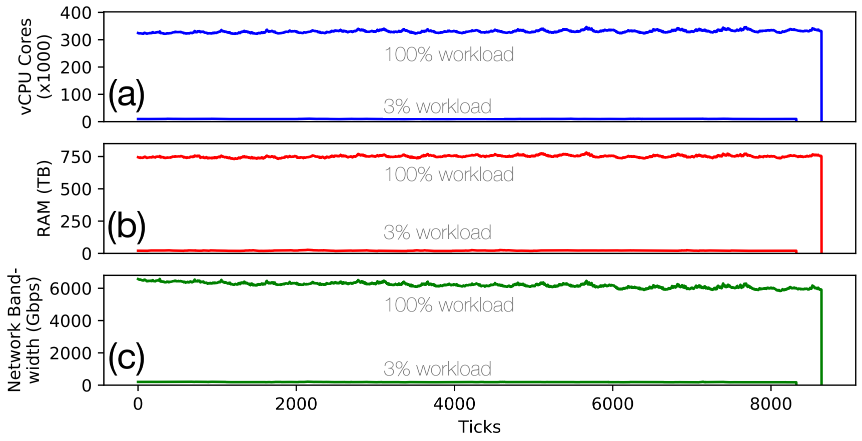


Figure 4.14: Resource Footprint of the 3% VDC Workload. The 100% workload footprint is described in Section 3.1 and Section 3.3. Both 3% and 100% workloads use `bpc=1Mbps` for network bandwidth.

in both 4-rack and 4-pod topologies have 48:16=3:1 downlink:uplink oversubscription ratio. Moreover, servers have 60 cores and 256 GB RAM in both topologies. There are 192 servers in the 4-rack datacenter.

Partial VDC Workload

We also scale down the reference VDC workload *volume* to ensure that the peak CPU and memory consumed by the VDC workload does not exceed the 4-rack datacenter's capacity. Note that reducing workload volume is different from reducing *peak VDC size*, which we discounted because it is too restrictive a mechanism for reducing the ILP model size. In fact, we strive to maintain a consistent VDC size distribution across the full and partial VDC workloads by adopting randomized VDC sampling. Here, we collect a list of all VDC UUIDs in the reference VDC workload and randomly select $k\%$ of these VDCs to be included in the partial workload. We choose $k = 3\%$, because the number of servers in the 4-rack datacenter is reduced by $32\times$ (6144/192).

We summarize the CPU, RAM, and network bandwidth footprints of the 3% VDC workload. This summary highlights the scale of workload NetSolver is able to handle without introducing prohibitively high resource allocation latency.

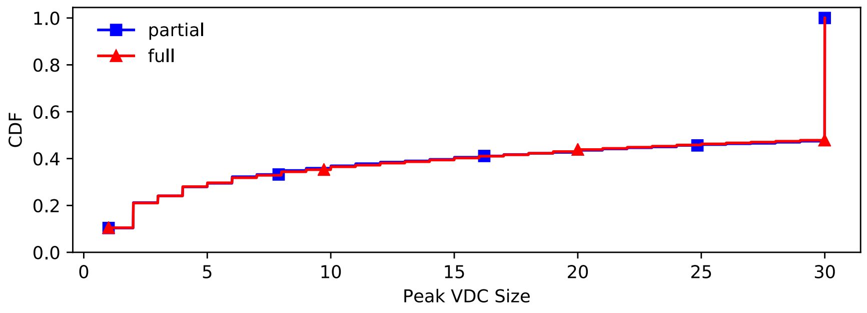


Figure 4.15: Peak VDC Sizes in Partial (3%) and Full VDC Workloads.

Figure 4.14 compares 3% VDC workload's (`bpc=1Mbps`) footprints with the reference VDC workload's footprints (100% workload, `bpc=1Mbps`). As we can see in Figure 4.14, the 3% workload's duration is also (coincidentally) 3.6% shorter (8322 ticks) than the the reference VDC workload's duration (8640 ticks). Figure 4.14(a) shows that the number of consumed cores in the 3% workload ranges between 8,789–10,920 cores, while it ranges between 321,943–341,279 ($\approx$36$\times$) in the reference VDC workload. Similarly, Figure 4.14(b) shows that the memory footprint of the 3% workload ranges between 18,360–27,939 GB, while it ranges between 730,314–781,767 GB ($\approx$40$\times$) in the reference VDC workload. Finally, Figure 4.14(c) shows that the network bandwidth footprint of the 3% workload ranges between 171–214 Gbps, while it is between 5,828–6,580 Gbps in the reference workload ($\approx$34$\times$).

Figure 4.15 compares peak VDC sizes in the 3% and the reference VDC workloads. As we can see, the CDF lines overlap, meaning that VDC size distribution in the 3% VDC workload closely reflects the one in the reference VDC workload. There are 2,216 unique VDCs in the 3% VDC workload, while there are 73,872 VDCs in the reference VDC workload ($\approx$33$\times$) (Section 3.2.2). Moreover, the 3% VDC workload has 57,809 VMs while there are 1,960,300 VMs in the full VDC workload ($\approx$34$\times$).

VM Allocation Latencies

Figure 4.16 compares VM allocation latencies in NETSOLVER and STARNET on the 3% VDC workload in the 4-rack datacenter. We show NETSOLVER's alloca-

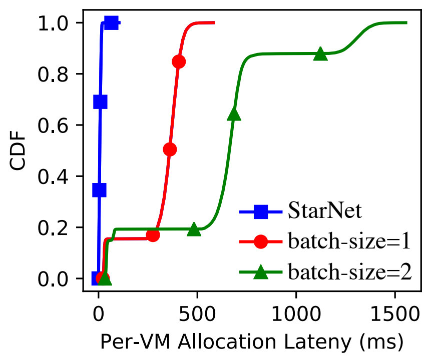


Figure 4.16: NETSOLVER's Allocation Latencies on the 3% VDC Workload with Two Batch Sizes. We also show STARNET's VM allocation latencies for comparison.

tion latencies for batch size of one VM (batch-size=1) and batch size of one VMs (batch-size=2), which we contrast with that of STARNET's. Recall that we can batch VDC VMs in NETSOLVER to trade-off completeness with lower allocation latencies. As we can see in Figure 4.16, the larger batch size increases VM allocation latency: by 84% in 50th percentile (from 360 ms in batch-size=1 to 661 ms in batch-size=2) and by ≈300% in 99th percentile (from 465 ms in batch-size=1 to 1,387 ms in batch-size=2). However, NETSOLVER's VM allocation latencies are at least 24× higher than those of STARNET, which has 6 ms in 50th percentile and 17 ms in 99th percentile.

We conclude that NETSOLVER's VM allocation latencies are too high in a 4-pod datacenter with over 6000 servers. NETSOLVER can handle smaller datacenters, such as the one with 192 servers we showed in Figure 4.13. Even at this small scale, we need to constrain NETSOLVER to tiny batch size (e.g., batch-size=1, or VM-at-a-time) to get something approximating acceptable latency (≈1s).

4.3.3 STARNETLA

Since NETSOLVER is impractical at scale, we enhance STARNET with locality-awareness and retries (STARNETLA) to see if we can obtain some of the benefit that NETSOLVER provides. We first evaluate STARNETLA with locality enhancement and only then enable retries to distinguish the relative contribution of each

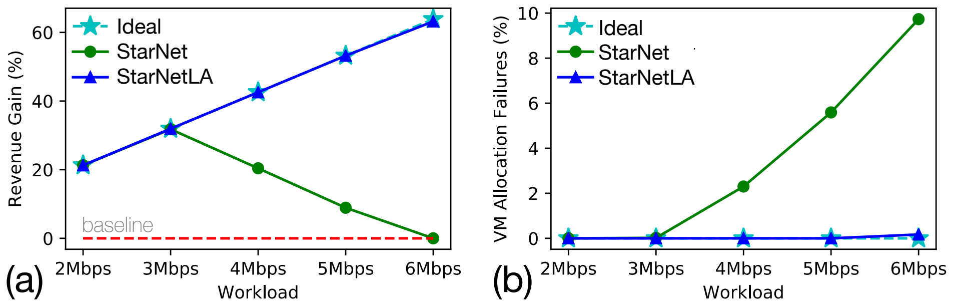


Figure 4.17: Revenue Comparison with StarNet and StarNetLA. Revenue gain and VM allocation failures by StarNet and StarNetLA on 4-pod Jupiter datacenter with full VDC workload with varying network load: (a) shows revenue gain, and (b) shows VM allocation failures. The revenue gain and latencies for StarNet in this figure are identical to the ones in Figure 4.11.

enhancement on revenue gain and VM allocation latency. We use StarNetLA to indicate that retries=0, and use "StarNetLA (N)" to explicitly state the number of retries (N) in StarNetLA.

Figure 4.17 shows the effects of locality-awareness on revenue gain and VM allocation failures. As we can see in Figure 4.17(a), which shows the `bpc=2Mbps` workload, both StarNet and StarNetLA generate an identical 21% revenue gain compared to the baseline where the cloud provider does not offer network bandwidth guarantees. The revenue from StarNet and StarNetLA in the `bpc=2Mbps` workload is equal to the Ideal revenue because neither of these algorithms fail any VM allocations. Zero VM failures is also shown for `bpc=2Mbps` workload in Figure 4.17(b). The number of VM allocation failures, hence the revenue gain, start to diverge between StarNet and StarNetLA from `bpc=3Mbps` workload onward, reaching the peak VM allocation failures of 9.73% as shown in Figure 4.17(b) and no revenue gain at `bpc=6Mbps`, as shown in Figure 4.17(a). On the other hand, StarNetLA fails no VM allocations until `bpc=6Mbps` workload. Thus, the revenue gain with StarNetLA equals the Ideal revenue gain for `bpc=[3,4,5]Mbps` workloads. With the `bpc=6Mbps` workload, StarNetLA fails only 0.1748% of VM allocations (3427 out of 1,960,300 VMs) generating 63.19% revenue gain (0.66% less than the Ideal). These exper-

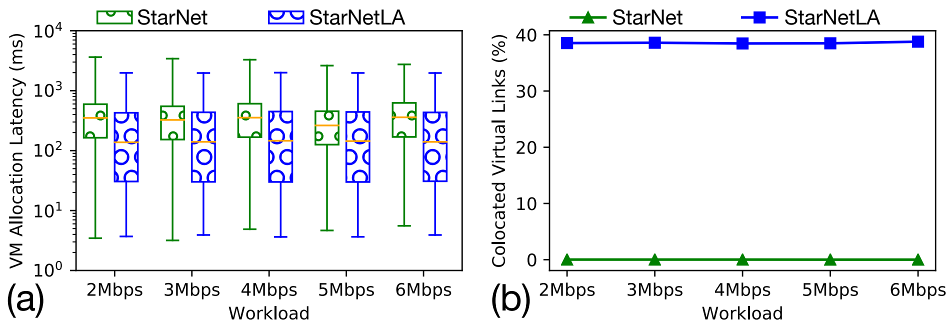


Figure 4.18: VM Allocation Latencies with STARNET and STARNETLA. We show results for allocating full VDC workloads with varying network demand in the 4-pod Jupiter datacenter: (a) shows VM allocation latencies, and (b) shows the percentage of colocated virtual links. The latency boxes show the first and third quartiles, and whiskers show the min and 99th percentile. The horizontal line inside the box is the median. The VM allocation latencies for STARNET in this figure are identical to the ones in Figure 4.10 (on page 117).

iments show that VM colocation significantly decreases VM allocation failures, producing up to 63% revenue gain compared to the STARNET algorithm, which lacks locality-awareness.

The VM colocation optimization also reduces VM allocation latencies, as shown in Figure 4.18(a). Figure 4.18(a) shows that STARNETLA has 45–61% (119–220ms) lower median VM allocation latency than STARNET. The tail latency (99th percentile) difference between STARNETLA and STARNET ranges in 24–45% (645–1651ms); STARNETLA is faster. Figure 4.18(b) shows the reason behind STARNETLA's lower latency: VM colocation. Recall that we use the Dijkstra-based virtual link (vlink) allocation technique when we cannot colocate VMs. Avoiding vlink allocation altogether for all (or some) vlinks in a VM reduces VM allocation latency. As we see in Figure 4.18(b), STARNET consistently colocates only ≈0.02% of the vlinks across all workloads, while STARNETLA consistently colocates ≈38% of the vlinks.

We now describe experimental results with different retry values. As we explained in Section 4.1.4, the intuition why more retries might help is because retries approximate NETSOLVER's completeness optimization: trying more servers makes

StarNetLA more complete by decreasing the probability of missing the server that can accommodate the to-be-allocated VM's end-to-end bandwidth requirements. For this experiment, we chose the full VDC workload with `bpc=6Mbps` because StarNetLA produces zero VM allocation failures for VDC workloads with lower `bpc` values. We want to see if retries reduce the number of VM allocation failures and thereby generate a higher revenue gain. We experiment with 10 different retry values: 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, and 100. Here, retry=N means that StarNetLA attempts VM allocation N times, e.g., N=1 means StarNetLA tried only one server, which is identical to the StarNetLA analyzed above.

Figure 4.19 shows the results from our experiment. Contrary to our intuition, Figure 4.19(a) shows that the revenue gain does not increase with more retries. The revenue gain by StarNetLA is almost equal across all retry values. Figure 4.19(b) shows StarNetLA VM allocation latencies with different numbers of retries. Similar to the revenue gain, the latencies (for successful VM allocations) also have no dependency on the number of retries. As we can see in Figure 4.19(b), p50 VM allocation latencies are in 137–147ms (7% variance) range while p99 latencies range in 1,950–2,032ms (5% variance). These latencies neither consistently increase nor decrease as the number of retries goes up. We repeated the experiment with `bpc=7Mbps` workload and its results were qualitatively similar, leading us to the same conclusions.

More retries do not improve StarNetLA's revenue gain, because the locality-awareness optimization itself failed only 0.1748% of all allocations; it already achieves high datacenter utilization. To confirm this hypothesis, we collected utilization levels of every physical link in the 4-pod Jupiter datacenter at the end of each tick. That is, we record the link utilization percentage after all events in the tick are processed by the scheduler. We then generate the heatmap of the datacenter network utilization over time (i.e., across all ticks), as shown in Figure 4.20.

Figure 4.20 compares the heatmap from StarNet with the heatmap from StarNetLA (with no retries). On the horizontal axis, we show the number of ticks (8640) in the workload. On the vertical axis, we enumerate individual physical links in the datacenter. We divide physical links into three categories and plot them separately (bottom, middle, top) to visualize link utilizations across the datacenter network hierarchy. Recall that in a leaf-spine datacenter topology, the

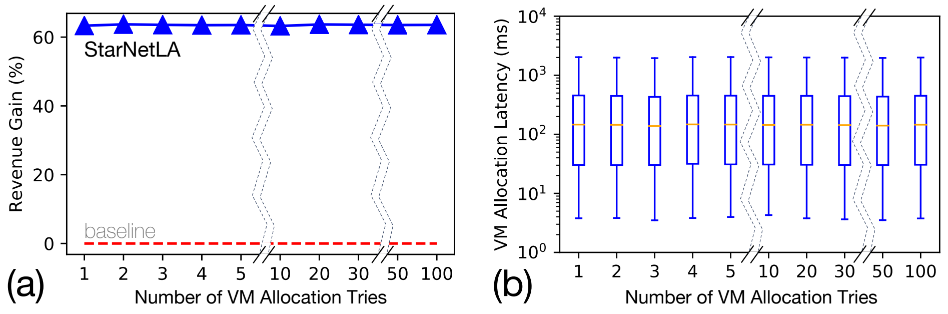


Figure 4.19: StarNetLA Performance with Different Retries. We show results for allocating the full VDC workloads with `bpc=6Mbps` in the 4-pod Jupiter datacenter: (a) shows revenue gains, and (b) shows VM allocation latencies for different retry values. The latency boxes show the first and third quartiles, and whiskers show the min and 99th percentile. The horizontal line inside the box is the median.

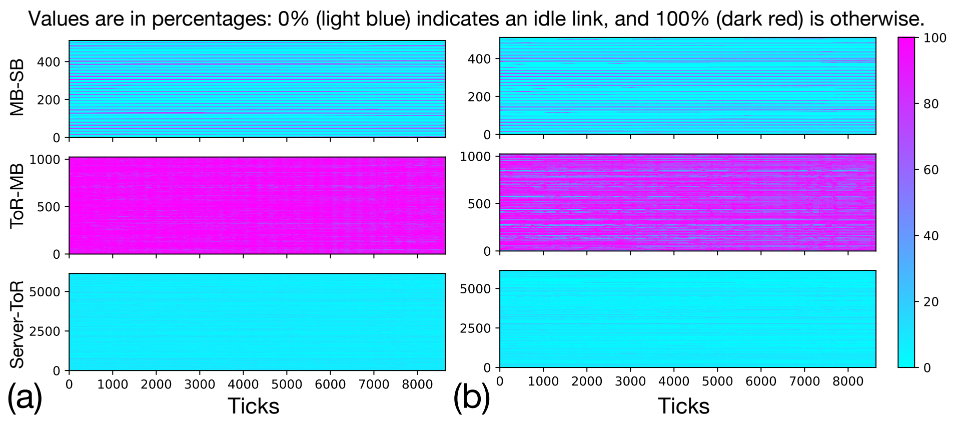


Figure 4.20: Datacenter Network Bandwidth Utilization Heatmap Over Time. We show results for allocating the full VDC workload with `bpc=6Mbps` in the 4-pod Jupiter datacenter: (a) shows utilization with StarNet, and (b) shows utilization with StarNetLA (no retries).

bottom-most level of the network hierarchy is the leaf level that connects servers to ToR switches. The leaf link utilizations are shown on the bottom-most plots in Figure 4.20. There are 6144 leaf links in the 4-pod Jupiter datacenter, which correspond to the length of the vertical axis of the bottom-most plots. The middle plots show link utilizations in the aggregation level that connect ToR switches to Middle Blocks (MB) in the Jupiter datacenter topology. There are 1024 aggregation links. Finally, the top-most plots show link utilizations of the core links that connect MBs to the Spine Blocks (SB) in the Jupiter datacenter topology. There are 512 core links. A colorful horizontal line in each plot shows the utilization level of a physical link across all 8640 ticks. For example, the point corresponding to x=1000 and y=500 on the bottom-most plot of Figure 4.20(a) shows the utilization level of the 500th leaf link after the scheduler processed all events in tick number 1000. The darker the (purple) color of the point the higher the link utilization. The link's color will change to the opposite side of the spectrum (light blue) in tick number 1001 if all VMs that are consuming bandwidth on the 500th leaf link are deallocated in tick number 1001.

As we see in Figure 4.20(b), datacenter network bandwidth utilization levels with STARNETLA are high. The links at the ToR-MB layer are the bottleneck in this datacenter. Thus, increasing the number of STARNETLA retries is fruitless, because link utilization levels are already high even without retries. More retries are not able to overcome this bottleneck.

On the other hand, Figure 4.20(a) shows that it is possible to achieve even higher utilization at ToR-MB layer as the heatmap of the middle plot in Figure 4.20(a) is darker than that of Figure 4.20(b). However, higher utilization levels in Figure 4.20(a) are the result of inefficient bandwidth allocation because STARNET reaches this utilization level after failing to allocate 9.73% of VMs while STARNETLA achieves the utilization level in Figure 4.20(b) failing only 0.1748% of VMs. Thus, we conclude that STARNETLA's bandwidth allocation is efficient, even without retries. (The heatmap plots like in Figure 4.20 also show where the network operators should add more capacity to increase datacenter utilization.)

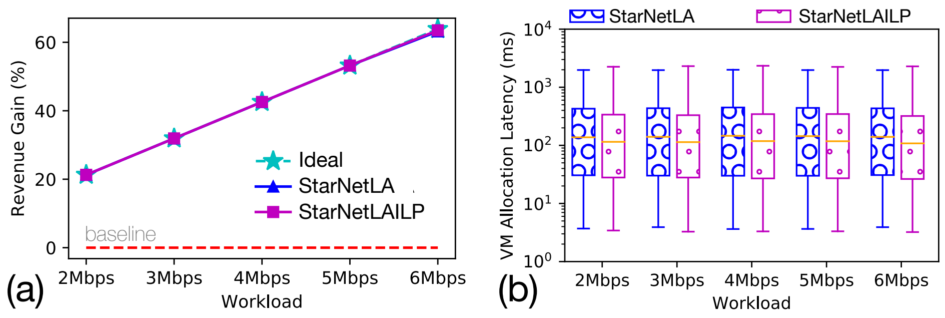


Figure 4.21: STARNETLA vs. STARNETLAILP Revenue and Latency. We show results for allocating full VDC workload with `bpc=[2-6]Mbps` in the 4-pod Jupiter datacenter: (a) compares revenue gains, and (b) compares VM allocation latencies.

4.3.4 Hybrid Algorithms

Since retries did not enable STARNETLA to achieve completeness, we experiment with two hybrid algorithms: STARNETILP and STARNETLAILP. Both of these algorithms use NETSOLVER-ILP as the fall-back engine to the corresponding heuristic algorithm. Intuitively, hybrid algorithms should reduce the number of VM allocation failures, hence generate a higher revenue gain, compared to the incomplete heuristic algorithms. Hybrid algorithms, on the other hand, provide VM-level completeness. We first discuss STARNETLAILP results, followed by discussion of the results with STARNETILP.

We repeated the STARNETLA experiments in Section 4.3.3 with STARNETLAILP to compare these two algorithms. In other words, we allocated the full VDC workloads with `bpc=[2-6]Mbps` on the 4-pod Jupiter datacenter. Figure 4.21 compares the performance of these two algorithms: both algorithms produce identical revenue gain (Figure 4.21(a)) and VM allocation latency (Figure 4.21(b)) for `bpc=[2-5]Mbps` workloads because the primary algorithm (STARNETLA) did not fall back to the secondary algorithm (NETSOLVER-ILP). The fallback does not happen because the primary algorithm does not fail any VMs for these workloads. Although barely visible in Figure 4.21, the algorithms' performance does differ slightly at the `bpc=6Mbps` workload for which STARNETLA fails to allocate 0.1748% of VMs and thereby falls back to NETSOLVER-ILP to reattempt the allocation.

We analyze StarNetLAILP's performance for `bpc=6Mbps` workload by re-running the experiment with debug output, which allows us to track the fallbacks. Collecting debugging information is intrusive and it increases VM allocation latency. Therefore, Figure 4.21 shows results from the debug disabled run. However, the number of VM allocation failures are similar[14] in both runs and the conclusions regarding fallbacks from one run generalize to others.

StarNetLAILP has 4,111 fallbacks of which 746 are successful. Thus, StarNetLAILP fails to allocate 3,365 VMs, which represent 0.1717% of the full VDC workload. The allocation latency of these 746 VMs ranges between 26 and 65 seconds where the 50th percentile (p50) latency is 43.49s and the 99th percentile (p99) latency is 60.45s. Here, p50 latency of the fallback scheduler (NetSolver-ILP) is 310$\times$ higher and p99 latency is 30$\times$ higher than the primary scheduler's (StarNetLA) respective latencies. The results demonstrate that fallback scheduler's latencies are impractical. Therefore, although it is possible to reduce the VM allocation failures by leveraging NetSolver-ILP's VM-level completeness, the latency of these allocations render them useless in practice.

Our experiments with the other hybrid algorithm, StarNetILP, are likely to lead to the same conclusion, except that it will take us longer to arrive at that conclusion. The reason for the longer experimental time is because there are significantly more fallbacks to the secondary scheduler in StarNetILP, which are caused by the primary scheduler (StarNet) failing to allocate a significantly larger number of VMs compared to StarNetLA. For example, in the full VDC workload with `bpc=6Mbps`, StarNet fails to allocate 190,640 VMs and even with 26s per-VM allocation latency (the minimum latency of the fallback scheduler in our earlier experiment with StarNetLAILP), the StarNetILP experiment will require 57 days. In our actual experiments, StarNetILP completed the `bpc=2Mbps` and `bpc=3Mbps` workloads in 9 days and 14 days, respectively. There were no fallbacks with these workloads because StarNet

[14]There are 0.28% VM allocation failures (5,472 VMs out of 1,960,300 VMs) with debug disabled (shown in Figure 4.21) and 0.1748% failures (3,237 VMs out of 1,960,300 VMs) with debug enabled (not shown). The outcome differs across the multiple experimental runs due to non-deterministic server selection in the `Weigher` function in StarNet (see Algorithm 2 line 1 on page 84).

successfully allocated all VMs by itself. The experiments with other workloads (`bpc=[4,5,6]Mbps`) were still running after 25 days, after which we terminated them. Thus, those experiments are unlikely to produce any new finding, i.e., the fallback scheduler will still have high latency, because the fallback scheduler is identical in STARNETILP and STARNETLAILP. Thus, we conclude that hybrid algorithms are not useful in practice.

4.3.5 STARNETLA Optimality Approximation

In Section 4.3.3, we saw that locality-awareness reduces VM allocation failures by up to $\approx$9%, which generated up to 63% revenue gain. Can we do better? Constraint-solvers can be used to answer this question. However, the constraint encodings we proposed for NETSOLVER (Section 2.3) are not the right ones. These encodings are *online optimal*, while we need *globally optimal* to answer the question above. In online optimal, a constraint-solver makes piecemeal decisions about VDC (or a VM batch) placement, i.e., given a VDC (or VM batch) allocation request the solver finds a satisfying solution that minimizes VM allocation failures in each allocation. However, an optimal decision in each allocation step is *greedy* and it does not necessarily lead to an optimal decision across multiple allocations. For example, when allocating 10 VMs in a piecemeal fashion, the online algorithm might fail to allocate VM_5 and VM_8, resulting in two VM allocation failures. On the other hand, if we encode constraints for all 10 VMs together and minimize for failures at once, the constraint-solver could deliberately fail an earlier VM, e.g., VM_3, to avoid failing VM_5 and VM_8. Thus, the globally optimal constraints may produce fewer VM allocation failures than the online optimal constraints.

We developed a globally optimal algorithm that we call ORACLE. ORACLE has two major changes on top of NETSOLVER-ILP. First, ORACLE removes the temporal aspect from NETSOLVER-ILP: instead of building constraints for each VM allocation and solving them separately, we build a collective constraint model for all VMs in the workload. Second, in addition to VM allocations, VM deallocations are also part of the collective model. The deallocation constraints are the duals of their allocate constraints, where, if the constraint-solver satisfies the allocate event it must also satisfy the respective deallocate event. ORACLE solves the collective

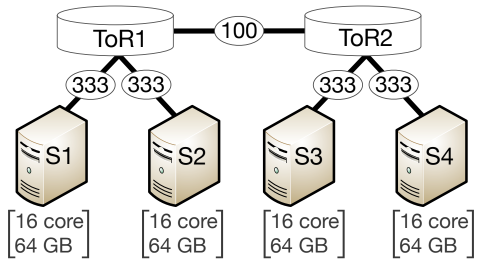


Figure 4.22: Four Server Datacenter used in ORACLE. Network links are in Mbps. We maintain 3:1 downlink:uplink oversubscription ratio in the ToR level as in the Jupiter topology.

model by maximizing the number of VM allocations. Although ORACLE faces scalability challenges for large workloads and datacenter sizes, we compared VM allocation failures in ORACLE and STARNETLA (our algorithm with the least VM allocation failures) on a limited scale.

We gradually increased the workload and datacenter size, and here we report results from the biggest scale that ORACLE completed in ≈5 CPU minutes (295 seconds).[15] We use the small four server datacenter show in Figure 4.22. We select a single VDC from the reference VDC workload that has the peak size of 20 VMs. Overall, this VDC has 30 VMs that are created/deleted over 12 ticks. Evaluation with a single VDC workload is desired as it presents the most challenging resource allocation scenario that imposes network bandwidth scarcity. We make sure no VMs fail with `bpc=1Mbps` and we gradually increase the `bpc` value until a subset of VDC VMs fail (because of insufficient datacenter network bandwidth). We report results for `bpc=6Mbps`, which consumes 1278 Mbps peak bandwidth.

ORACLE failed to allocate one VM while STARNETLA failed to allocate three VMs out of 30 VMs total. These correspond to 51% revenue gain by ORACLE, and 35% revenue gain by STARNETLA. Thus, ORACLE has 50% ((51-35)/35) higher revenue gain than STARNETLA. The ideal revenue gain is 60% for this workload. This result shows that STARNETLA works quite well, although there is still room

[15] ORACLE is slow. We give examples of larger workloads that we tried but ORACLE could not complete (on time). A VDC from the full Azure workload that has the peak size of 30 VMs, with 55 VMs spread over 18 ticks did not complete after 4 CPU days (102 hours). A larger VDC with the peak size of 30 VMs, and 512 VMs spread over 70 ticks did not complete after 28 CPU days (673h).

for improvement. The major advantage ORACLE has compared to other algorithms, including STARNETLA, is *clairvoyance*. As we described in our example above, ORACLE can deliberately fail some (early) VMs to keep room for (a bigger set of) future VMs. This large improvement in scheduler quality shows great potential for *prediction based* algorithms that can approximate ORACLE's perfect knowledge about the workload. For example, the Resource Central paper [40] and the follow-up paper [33] by Microsoft Azure describe machine learning based techniques to predict VM lifetimes and how these predictions can improve the VM scheduler quality. We would like to explore this direction in the future.

4.3.6 OpenStack Prototype

We prototyped STARNETLA in OpenStack with two goals: (1) to confirm STARNETLA's compatibility with OpenStack Nova's filtering-based scheduler architecture, and (2) to evaluate the latency impact of our scheduler on the overall VM allocation pipeline. Achieving the first goal required several changes to OpenStack, such as modeling every physical hop with its capacity to enable end-to-end bandwidth allocation, making the scheduler aware of VM delete events, and adding inter-VM bandwidth requirements to the VM creation CLI. To achieve the second goal, we did lightweight instrumentation of OpenStack's Nova module and quantified the latency contribution of each Nova submodule in the VM creation pipeline. We elaborate on each goal.

Scheduler's OpenStack Integration

We integrated STARNETLA as an additional Nova scheduler filter. We call our filter NovaNet. NovaNet runs as part of the Nova scheduler and has scheduler and DeleteNotifier components, as shown in Figure 4.23.[16] NovaNet translates OpenStack-specific VM create requests into STARNETLA input. This approach allows us to plug our algorithms directly into Nova.

DeleteNotifier keeps the datacenter inventory consistent between NovaNet and DB. Nova's *stateless* scheduler design processes VM delete events without sched-

[16]The figure is based off the OpenStack manual at https://docs.openstack.org/nova/latest/user/architecture.html

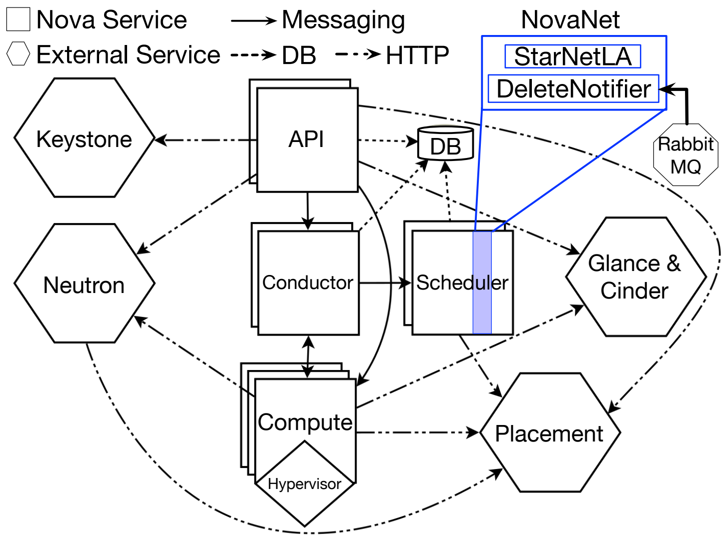


Figure 4.23: OpenStack Prototype Architecture. All modules exists in OpenStack except the NovaNet scheduler on the top-right. This is the modified version of Figure 4.1 to illustrate new modules, in blue.

uler involvement. That is, when a VM is deleted, the Placement module directly updates the DB. For quick prototyping, we designed NovaNet to be *stateful*: instead of fetching datacenter inventory from the DB in every VM create request, NovaNet maintains internal state. The DeleteNotifier makes NovaNet VM-delete-aware by updating STARNETLA's internal state in each VM delete event.

Scheduler's Latency Contribution

We added lightweight instrumentation to Nova submodules to quantify the latency contribution of each submodule. The instrumented submodules include API, Conductor, Scheduler, and NovaNet scheduler, as shown in Figure 4.23. We run our OpenStack deployment on a server separate from the simulation (VDCSIM) server to avoid performance interference. Our single-node OpenStack deployment (DevStack) runs on a server with 2.40 GHz (10 MB L3 cache) Intel Xeon E5-2407 v2 processor with eight cores across two NUMA nodes and hyperthreading disabled. The host OS is Ubuntu 16.04.6 LTS with 32 GB RAM that is uniformly distributed (16 GB each) across both NUMA nodes.

Table 4.3: Latency Contribution of Different OpenStack Submodules. We show the relative latency contribution of each Nova submodule to the VM creation pipeline. The numbers are average of 30 samples (VM creations). The contribution of the StarNetLA algorithm makes up only 0.036% of the overall time.

Submodule Name	Absolute Latency (ms)	Percentage of Total Latency
API	949.42	14%
Conductor	362.79	5.964%
Scheduler	1284	20%
StarNetLA	**2.37**	**0.036%**
Compute	3953	60%
Total	6552	100%

Table 4.3 shows results from our runtime analysis. We used the same single VDC workload we used for Oracle evaluation. The VDC has the peak size of 20 VMs, with 30 VMs spread over 12 ticks. We allocated the VDC on the four-rack Jupiter topology (Figure 4.13 on page 122). This experiment had no VM failures. The average VM allocation latency with StarNetLA is 6.5s while the Scheduler submodule takes only 1.28s (20% of total). The StarNetLA algorithm is executed entirely within the Scheduler submodule, and it completes in 2.37 milliseconds (0.036% of total). Note that the runtime for the slowest submodule, Compute, does not include VM creation time by the hypervisor as our prototype does not actually create the VM. Thus, our reported StarNetLA latency contribution is an overestimate as the actual VM creation takes even longer, making StarNetLA's latency contribution even smaller. Moreover, VM creation in production can be even longer as our analysis does not include other modules involved in VM creation, such as Keystone (authentication service), Cinder (block storage service), Glance (VM image service), and Neutron (networking service) [103].

4.4 Related Work

In Section 2.1, we already discussed existing literature on VDC scheduling. Here, we revisit some of those works to highlight the parts that are relevant to the topics we discussed in this chapter. We start by stating our rationale for omitting performance comparison between SecondNet's VDCAlloc algorithm [59] and the heuristic algorithms introduced in this chapter, such as StarNetLA.

In the SecondNet paper [59], Guo et al. introduce the VDCAlloc algorithm for VDC scheduling. VDCAlloc is a heuristic algorithm that scales well to datacenters with up to 100,000 servers. We do not directly compare StarNetLA to VDCAlloc, because we have already demonstrated the superiority of NetSolver in its VDC packing ability in Chapter 2, where NetSolver allocated up to $3\times$ more VDCs than VDCAlloc, although NetSolver's latency is impractical for realistic VDC workloads and datacenter with thousands of servers. As we demonstrated in Chapter 2, VDCAlloc's limitations come from its *bipartite graph matching* property, which is used to reduce VDC allocation latency.

VDCAlloc uses bipartite graph matching where it models VDC topology as a graph to be matched with the datacenter topology graph. The datacenter servers are grouped into clusters of varying size and sorted in an ascending order by the number of servers in the cluster. VDC VMs are also sorted in a descending order by their bandwidth requirements such that the VM with the highest network bandwidth requirement is allocated first. These two sorting techniques produce a *fail-fast* property that allows VDCAlloc to quickly fail clusters without sufficient capacity to accommodate a VDC's largest bandwidth requirement. VDCAlloc exhaustively retries clusters with more servers until it finds a cluster that can accommodate the entire VDC.

The bipartite matching heuristics fundamentally prevents VDC VM colocation by strictly assigning VDC VMs to separate servers. In fact, VDCAlloc considers VM colocation only when tenants explicitly request it. This assumption does not hold in our VDC workload, because neither the Azure cloud nor any other public cloud provider offer an option for tenants to express their colocation preferences. As we saw in Section 4.3.3, colocation can be inferred by the cloud provider, without tenant input and is useful in practice. Colocation reduces median VM allocation latencies between 45–61% and generates up to 63% extra revenue.

Zhani et al. introduce VDC Planner for VDC scheduling [137]. Similar to SecondNet [59], VDC Planner models VDC scheduling as a graph embedding problem but mainly focuses on VDC VM migration for minimizing datacenter resource defragmentation. VDC Planner associates a cost with migrating VMs to different parts of the datacenter and maximizes the datacenter utilization by minimizing the migration cost. The production workload traces from Azure cloud that we use in this work do not have a VM migration construct nor is there a cost model for migrating VMs [40]. In fact, the Resource Central [40] authors propose to use a prediction-based VM allocation techniques to avoid "problematic live VM migration in practice ... and place VMs where they can stay". Thus, we do not compare our work to VDC Planner.

4.5 Conclusions

We reexamined VDC scheduling in practice. In particular, we addressed four practical concerns for the deployment of VDC schedulers in cloud datacenters. We addressed these practical concerns in five steps. First, we constructed realistic VDC workloads using the Gridiron technique. Second, we proposed the revenue gain metric for VDC scheduler evaluation. Third, we developed STARNET: NOVAFILTER with end-to-end network bandwidth allocations. Fourth, we compared NETSOLVER with STARNET using realistic VDC workloads and the revenue metric. Fifth, we enhanced STARNET with locality-awareness and retries to further improve state-of-the-art algorithms' performance.

The revenue gain metric not only captures a VDC scheduler's ability to efficiently allocate datacenter network bandwidth, but it also allows cloud operators to directly measure the extra generated revenue when using a particular VDC scheduler. We also made a case for attributing a dollar value, \$0.58 per 1 Gbps/hour, for network bandwidth guarantees. We demonstrated that this price offers revenue neutrality for cloud providers without changing cloud affordability for the tenants.

Our evaluations show that NETSOLVER has a prohibitively high VM allocation latency on allocating a realistic VDC workload on the 4-pod Jupiter datacenter with over 6,000 servers. NETSOLVER's average vlink allocation latency is 41s, which is three orders of magnitude higher than that of STARNET (32ms). Moreover, NETSOLVER's average VM allocation time is $\approx$20mins when allocating a VDC with 10 VMs, and it is not able to allocate VDCs with more VMs because it runs out of memory (50 GB) when given larger VDCs. These results show that NETSOLVER does not scale to datacenters with thousands of servers, but can handle scheduling

VDCs in a smaller datacenters, such as the ones with $\approx$200 servers.

We demonstrated the importance of locality-awareness when scheduling VDCs. Our locality-aware heuristic VDC scheduling algorithm, STARNETLA, reduces the median VM allocation latencies between 45–61% and, as a result achieves under 220ms median VM allocation latency. Moreover, STARNETLA generates up to 63% revenue gain by being locality-aware.

We also showed that, unlike NETSOLVER, STARNETLA does not raise a scheduler compatibility concern. We confirmed this by integrating STARNETLA into OpenStack. Our OpenStack prototype demonstrated that end-to-end bandwidth allocation can be integrated to OpenStack without significant changes. In fact, Nova scheduler's existing filtering-based architecture readily accommodates STARNETLA as one of the scheduler filters. Moreover, our VM allocation latency evaluations in OpenStack prototype demonstrate that the extra latency introduced by end-to-end bandwidth allocation is insignificant (0.036%) when we take into account latencies of other OpenStack modules. Thus, we conclude that OpenStack-based cloud deployments, and perhaps cloud deployments with other cloud management frameworks, can readily support end-to-end bandwidth allocation.

In conclusion, our close examination improves our confidence in the practicality of VDC scheduling. Our revenue gain analysis shows that cloud providers can maintain revenue neutrality and cloud affordability by setting the right price for the virtual network bandwidth guarantees. Our scheduler latency analysis shows that it is possible to achieve practical VM allocation latencies when VMs request end-to-end network bandwidth allocation. Our OpenStack prototype also shows that the extra latency introduced by end-to-end bandwidth allocation is insignificant in practice. Therefore, this chapter brings us one step closer to offering virtual network bandwidth guarantees as a new cloud service.

Chapter 5

Conclusions and Future Work

We have constructed efficient VDC schedulers that can achieve high datacenter utilization while offering practical resource allocation latency. Specifically, we made the following five contributions:

1. **Workload:** We propose a new technique, Gridiron, to generate a realistic VDC workload to evaluate VDC schedulers. The Gridiron technique augments an existing production VM workload with network bandwidth requirements to generate a VDC workload. Our VDC workload is the first publicly available production-based cloud workload with inter-VM network bandwidth requirements.

2. **Metrics:** We proposed the revenue gain metric for evaluating VDC schedulers. We also demonstrated how charging for network bandwidth guarantees can increase a cloud provider's revenue by up to 63% without changing cloud affordability for the tenants.

3. **Algorithms:** We developed several constraint-solver-based and heuristic-based VDC scheduling algorithms. Specifically, we proposed a constraint-solver-based VDC scheduling algorithm, NETSOLVER-ILP, that scales to datacenters with over hundred of servers, and a heuristic-based algorithm, STARNETLA, that scales well to datacenters with thousands of servers.

4. **Optimality:** We constructed a constraint-solver-based, offline VDC schedul-

ing algorithm, ORACLE, for minimizing VM allocation failures. We showed that ORACLE can generate 50% higher revenue gain compared to STARNETLA. This shows that STARNETLA works quite well, although there is still room for improvement.

5. **Prototype:** We demonstrated the practicality of STARNETLA by integrating it into OpenStack. We also showed that the additional latency introduced by end-to-end network bandwidth allocation is insignificant in practice. The additional per-VM latency of 2.37ms makes up only 0.036% of the total time required to allocate a VM in our OpenStack deployment.

Future Work

There are three directions in which we could extend this work:

1. **Workload:** We studied the previous literature to identify classes of cloud workloads that would benefit from network bandwidth guarantees. Directly deploying a wide range of modern cloud workloads and quantifying their performance improvements when the cloud offers network bandwidth guarantees remains important work to be done.

2. **Prototype:** We evaluated our OpenStack prototype on a single-node deployment where we simulated a datacenter with 192 servers. Moreover, we disabled actual VM creation and did not enforce inter-VM network bandwidth reservations. Deploying our prototype on a multi-node OpenStack cloud, evaluating VM creation latencies by actually creating VMs, and enforcing inter-VM network bandwidth across server NICs and datacenter switches will bring our prototype closer to a minimum viable product.

3. **Scheduler:** We demonstrated that ORACLE outperforms STARNETLA by exploiting clairvoyance: perfect knowledge of the future. The Resource Central paper [40] and the followup paper [33] by the same group in the Azure cloud demonstrate that an imperfect future knowledge is also useful for the cloud scheduler. Specifically, these works demonstrate how predictions about various VM characteristics, such as VM lifetimes and its peak deployment sizes (peak VDC size) can be used to reduce VM scheduling failures

by 2.5× (from 0.25% to 0.1%). Integrating similar prediction models into StarNetLA will further increase its revenue.

Closing Thoughts

Network bandwidth guarantees will become increasingly important for the future of the cloud. We foresee at least three other use cases for bandwidth guarantees:

1. **Heterogeneity:** Datacenter hardware is becoming increasingly heterogeneous to support a wide range of tenant applications. For example, the global technology analysis firm Gartner forecasts that

   > *Infrastructure as a Service will grow 24% year over year, which is the highest growth rate across all market segments. This growth is attributed to the demands of modern applications and workloads, which require infrastructure that traditional data centers cannot meet.* [54]

   One example of the new infrastructure for modern workloads is domain specific accelerators [11]. Using accelerators for large scale workloads requires connecting them with sufficient network bandwidth. Tenants' ability to request and reserve sufficient network bandwidth for their VMs and accelerators allows them to efficiently use heterogeneous cloud resources.

2. **Disaggregation:** High performance networking is important for new datacenter architectures, such as Disaggregated Datacenters (DDC) [53]. DDCs have a resource-centric architecture, instead of the server-centric architecture in traditional datacenters. In DDCs, compute, memory, storage, accelerators, and other resources are decoupled into separate *blades* where each blade hosts one type of resource. These blades are interconnected via high performance network fabric [71], because applications running on DDCs rely on low latency and network bandwidth guarantees for their performance [53]. Tenants consume a slice of these resources to run their workload. The VDC abstraction directly captures the resource slice tenants want to allocate in the cloud, and cloud providers can directly apply our VDC schedulers to efficiently share DDC resources.

3. **Inter-Cloud:** An emerging application that requires inter-cloud network bandwidth guarantees is Software Defined Wide Area Network (SD-WAN) product, such as VMWare SD-WAN [130]. Service providers use SD-WAN to build a secure, reliable, and scalable WAN over the (public) cloud infrastructure. The SD-WAN can be used by many types of applications, such as for setting up a remote office that requires connectivity between office staff, which are potentially distributed across the globe. The SD-WAN product is typically offered as a scalable alternative to existing, perhaps more expensive, MPLS links offered by telecommunication providers [132].

   Scheduling VDCs in a datacenter is analogous to scheduling SD-WANs over clouds. Here, a VDC is akin to SD-WAN, for they both capture bandwidth requirements between the nodes in the virtual network, and a datacenter is akin to a cloud availability zone that offers network bandwidth for the SD-WAN. Our VDC scheduling algorithms, particularly locality-awareness optimization, are directly applicable to scheduling SD-WANs.

In summary, network bandwidth guarantees will become increasingly important for the future of the cloud. It will be critical for supporting datacenter heterogeneity, new datacenter architectures, emerging inter-cloud services, and offering new kinds of cloud services, such as the inter-VM network bandwidth guarantees service that we focused on this dissertation. As one of the cloud infrastructure veterans, Luiz Barroso at Google, put it "The data center is now the computer" [105]. The datacenter network is for a cloud what a system bus is for a computer. They are both the (network) fabric of the computing infrastructure. Just as mainframes and personal computers relied on the system bus bandwidth for connecting compute, memory, storage, and other peripherals, and enabled a wide range of applications over the past 50+ years, future cloud applications will rely on the datacenter network. This dissertation, a work on VDC scheduling, describes several approaches for efficiently sharing datacenter network bandwidth between such networked cloud applications.

Bibliography

[1] D. Abts and B. Felderman. A Guided Tour through Data-Center Networking: A Good User Experience Depends on Predictable Performance within the Data-Center Network. *Queue*, 10(5):10–23, 2012. URL https://doi.org/10.1145/2208917.2208919. → page 87

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74. ACM, 2008. URL https://doi.org/10.1145/1402958.1402967. → page 87

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014. → page 11

[4] Amazon. AWS and Zoom Extend Strategic Relationship, 2020. URL https://press.aboutamazon.com/news-releases/news-release-details/aws-and-zoom-extend-strategic-relationship. → page 1

[5] Amazon. AWS: Case Studies, 2021. URL https://aws.amazon.com/solutions/case-studies. → page 1

[6] Amazon. AWS: Retail Case Studies, 2021. URL https://aws.amazon.com/retail/case-studies. → page 1

[7] G. M. Amdahl. Computer Architecture and Amdahl's Law. *Computer*, 46 (12):38–46, 2013. URL https://doi.org/10.1109/MC.2013.418. → page 51

[8] A. Amokrane, M. F. Zhani, R. Langar, R. Boutaba, and G. Pujolle. Greenhead: Virtual data center embedding across distributed infrastructures. *IEEE transactions on cloud computing*, 1(1):36–49, 2013. URL https://doi.org/10.1109/TCC.2013.5. → page 73

[9] A. Andreyev, X. Wang, and A. Eckert. Reinventing Facebook's data center network, 2019. URL https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack. → page 11

[10] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 233–248, USA, 2014. USENIX Association. URL https://dl.acm.org/doi/abs/10.5555/2685048.2685067. → page 10

[11] AWS. Amazon EC2 F1 Instances, 2020. URL https://aws.amazon.com/ec2/instance-types/f1. → pages 1, 143

[12] AWS. Amazon EC2 G3 Instances, 2020. URL https://aws.amazon.com/ec2/instance-types/g3. → page 52

[13] AWS. AWS CloudFormation - Infrastructure as Code and AWS Resource Provisioning, 2021. URL https://aws.amazon.com/cloudformation. → page 4

[14] AWS. Amazon EC2 G3 Instances Price; March 3, 2021 snapshot, 2021. URL https://web.archive.org/web/20210303021517/https://aws.amazon.com/ec2/instance-types/g3. → pages 103, 104

[15] AWS. Amazon Virtual Private Cloud Documentation, 2021. URL https://docs.aws.amazon.com/vpc. → page 1

[16] AWS Solutions Builder Team. Real-Time Analytics with Spark Streaming, 2021. URL https://docs.aws.amazon.com/solutions/latest/real-time-analytics-spark-streaming/welcome.html. → page 1

[17] Azure. AzurePublicDatasetV2, 2019. URL https://github.com/Azure/AzurePublicDataset/blob/master/AzurePublicDatasetV2.md. → page 74

[18] Azure. Azure Kubernetes Service, 2021. URL https://azure.microsoft.com/en-ca/services/kubernetes-service. → page 75

[19] Azure. Azure Functions documentation, 2021. URL https://docs.microsoft.com/en-us/azure/azure-functions. → page 75

[20] M. Azure. Microsoft Azure VM Traces (AzurePublicDatasetV1), 2017. URL https://github.com/Azure/AzurePublicDataset/blob/master/AzurePublicDatasetV1.md. → pages 53, 54, 55, 73

[21] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 242–253, New York, NY, USA, 2011. Association for Computing Machinery. URL https://doi.org/10.1145/2018436.2018465. → pages 2, 9, 10, 12, 73, 103

[22] H. Ballani, D. Gunawardena, and T. Karagiannis. Network sharing in multi-tenant datacenters. Technical Report MSR-TR-2012-39, February 2012. URL https://www.microsoft.com/en-us/research/publication/network-sharing-in-multi-tenant-datacenters. → page 10

[23] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, page 171–184, USA, 2013. USENIX Association. URL https://dl.acm.org/doi/abs/10.5555/2482626.2482644. → pages 10, 12

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. SOSP '03, page 164–177. ACM, 2003. URL https://doi.org/10.1145/945445.945462. → page 3

[25] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 15(2):909–928, 2013. → page 13

[26] J. Barr. Amazon EC2 Beta, 2006. URL https://aws.amazon.com/blogs/aws/amazon_ec2_beta. → pages 1, 49, 51, 106

[27] L. A. Barroso, U. Hölzle, and P. Ranganathan. The datacenter as a computer: Designing warehouse-scale machines, third edition. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018. URL https://doi.org/10.2200/S00874ED3V01Y201809CAC046. → page 87

[28] S. Bayless. *SAT Modulo Monotonic Theories*. PhD thesis, University of British Columbia, 2017. URL https://dx.doi.org/10.14288/1.0343418. → page 21

[29] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. Sat modulo monotonic theories. AAAI'15, page 3702–3709. AAAI Press, 2015. ISBN 0262511290. URL https://dl.acm.org/doi/10.5555/2888116.2888230. → pages 8, 15, 19, 21

[30] S. Bayless, N. Kodirov, I. Beschastnikh, H. H. Hoos, and A. J. Hu. Scalable constraint-based virtual data center allocation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 546–554, 2017. URL https://doi.org/10.24963/ijcai.2017/77. → page vi

[31] S. Bayless, N. Kodirov, S. M. Iqbal, I. Beschastnikh, H. H. Hoos, and A. J. Hu. Scalable constraint-based virtual data center allocation. *Artificial Intelligence*, 278, 2020. URL https://doi.org/10.1016/j.artint.2019.103196. → pages vi, 12, 119

[32] A. Belbekkouche, M. M. Hasan, and A. Karmouch. Resource discovery and allocation in network virtualization. *IEEE Communications Surveys & Tutorials*, 14(4):1114–1128, 2012. → page 12

[33] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, A.-M. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich. Toward ML-Centric Cloud Platforms. *Commun. ACM*, 63(2):50–59, 2020. URL https://doi.org/10.1145/3364684. → pages 88, 113, 135, 142

[34] A. Biere and D. Kröning. SAT-based model checking. In *Handbook of Model Checking*, pages 277–303. Springer, 2018. → page 8

[35] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008. URL https://dl.acm.org/doi/10.5555/1855741.1855756. → page 8

[36] K. Chen and Q. Huo. Training Deep Bidirectional LSTM Acoustic Model for LVCSR by a Context-Sensitive-Chunk BPTT Approach. 24(7): 1185–1193, 2016. ISSN 2329-9290. URL https://dl.acm.org/doi/10.5555/2992803.2992806. → page 70

[37] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, and J. Wang. Virtual network embedding through topology-aware node ranking. *ACM SIGCOMM Computer Communication Review*, 41(2):38–47, 2011. → pages 9, 11, 13, 42

[38] N. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual network embedding with coordinated node and link mapping. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 783–791. IEEE, 2009. → pages 9, 11, 12, 13, 15, 42

[39] S. B. committee. Sort benchmark home page, 2021. URL http://sortbenchmark.org. → page 36

[40] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167. USENIX Association, 2017. URL https://doi.org/10.1145/3132747.3132772. → pages 5, 49, 53, 54, 65, 71, 73, 74, 75, 94, 111, 135, 139, 142, 174

[41] Coursera. How Coursera Manages Large-Scale ETL using AWS Data Pipeline and Dataduct, 2015. URL https://aws.amazon.com/blogs/big-data/how-coursera-manages-large-scale-etl-using-aws-data-pipeline-and-dataduct. → page 56

[42] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. De Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 373–387. USENIX Association, 2018. URL https://www.usenix.org/conference/nsdi18/presentation/dalton. → pages 3, 11

[43] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. → page 26

[44] Dell. PowerEdge R940 Rack Server, 2020. URL https://www.dell.com/en-ca/work/shop/productdetailstxn/poweredge-r940. → page 93

[45] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving, 2021. URL https://arxiv.org/abs/2002.06987. → page 70

[46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019. URL https://arxiv.org/abs/1810.04805. → page 70

[47] N. Dogovic. Understanding and Interpreting CPU Steal Time on Virtual Machines, 2020. URL https://blog.leaseweb.com/2020/09/24/understanding-and-interpreting-cpu-steal-time-on-virtual-machines. → page 174

[48] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 95–108. ACM, 1999. → page 12

[49] N. Eén and N. Sorensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2: 1–26, 2006. → page 23

[50] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 184–193. IEEE, 1975. → page 15

[51] A. Fischer, J. F. Botero Vega, M. Duelli, D. Schlosser, X. Hesselbach Serra, and H. De Meer. Alevin - a framework to develop, compare, and analyze virtual network embedding algorithms. In *Electronic Communications of the EASST*, pages 1–12, 2011. → pages 13, 40

[52] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013. → page 12

[53] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao. → page 143

[54] Gartner. Gartner forecasts worldwide public cloud revenue to grow 17% in 2020, 2019. URL https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020. → page 143

[55] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI' 16, page 99–115. USENIX Association, 2016. → page 94

[56] Google Cloud Blog. How Schrödinger is advancing COVID-19 drug discovery efforts with Google Cloud, 2021. URL https://cloud.google.com/blog/topics/healthcare-life-sciences/how-schrodinger-is-advancing-covid-19-drug-discovery-with-google-cloud. → page 1

[57] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018. URL https://arxiv.org/abs/1706.02677. → pages 58, 69, 70

[58] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009. → pages 33, 59, 61

[59] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference on emerging Networking EXperiments and Technologies*, Co-NEXT '10, New York, NY, USA, 2010. Association for Computing Machinery. URL https://doi.org/10.1145/1921168.1921188. → pages 4, 9, 10, 11, 13, 26, 27, 28, 32, 34, 35, 36, 37, 42, 56, 59, 73, 77, 78, 138

[60] A. Gupta, J. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. Provisioning a virtual private network: a network design problem for multicommodity flow. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 389–398. ACM, 2001. → page 15

[61] Gurobi. Gurobi optimizer reference manual, 2021. URL https://www.gurobi.com. → pages 8, 15

[62] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861.

USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/hadary. → pages 3, 74, 78, 88, 94, 113, 116, 117

[63] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling, 2018. URL https://arxiv.org/abs/1803.03288. → pages 2, 50, 51, 107

[64] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385. → page 70

[65] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural collaborative filtering, 2017. URL https://arxiv.org/abs/1708.05031. → page 70

[66] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. → page 70

[67] T. Huang, C. Rong, Y. Tang, C. Hu, J. Li, and P. Zhang. Virtualrack: Bandwidth-aware virtual network allocation for multi-tenant datacenters. In *2014 IEEE International Conference on Communications (ICC)*, pages 3620–3625. IEEE, 2014. → pages 9, 12

[68] Huawei. Huawei Public Cloud, 2021. URL https://www.huaweicloud.com/intl/en-us. → page 77

[69] Huawei. Huawei Cloud Worldwide Infrastructure, 2021. URL https://www.huaweicloud.com/intl/en-us/global. → page 77

[70] IBM. IBM ILOG CPLEX Optimization Studio CPLEX User's Manual; Chapter 16: Solving mixed integer programming problems (MIP); Multi-commodity flow cuts, 2017. URL https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.1/ilog.odms.studio.help/pdf/usrcplex.pdf. → page 15

[71] Intel. Intel, Facebook Collaborate on Future Data Center Rack Technologies, 2013. URL https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies. → page 143

[72] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training, 2019. URL https://arxiv.org/abs/1905.03960. → pages 2, 50, 51, 52, 70, 103, 107, 108

[73] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/jeon. → pages 50, 71

[74] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html. → page 75

[75] D. Kakadia, N. Kopri, and V. Varma. Network-aware virtual machine consolidation for large data centers. In *Proceedings of the Third International Workshop on Network-Aware Data Management*, page 6. ACM, 2013. → page 9

[76] D. E. Kaufman, J. Nonis, and R. L. Smith. A mixed integer linear programming model for dynamic route guidance. *Transportation Research Part B: Methodological*, 32(6):431–440, 1998. → page 15

[77] X. Ke, C. Guo, S. Ji, S. Bergsma, Z. Hu, and L. Guo. Fundy: A scalable and extensible resource manager for cloud resources. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021. → page 78

[78] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017. URL https://arxiv.org/abs/1609.04836. → page 70

[79] J. Kim, M. Kim, H. Kang, and K. Lee. U-gat-it: Unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation, 2020. URL https://arxiv.org/abs/1907.10830. → page 70

[80] N. Kodirov. VM creates with invalid timestamps, 2019. URL https://github.com/Azure/AzurePublicDataset/issues/4. → page 53

[81] N. Kodirov. PhD dissertation artifacts, 2021. URL https://github.com/DCResourceManage/nodir-phd-dissertation. → pages vi, 5, 58

[82] KVM. Kernel Virtual Machine, 2021. URL https://www.linux-kvm.org. → page 3

[83] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. volume 14, pages 14–19, 2014. → page 10

[84] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 467–478. ACM, 2014. → page 10

[85] L. Lee. How cloud software and service providers addressed an education crisis to enable remote learning at scale, 2020. URL https://www.cio.com/article/3571408. → page 1

[86] F. Liang, C. Feng, X. Lu, and Z. Xu. Performance Characterization of Hadoop and Data MPI Based on Amdahl's Second Law. In *The 9th IEEE International Conference on Networking, Architecture, and Storage*, pages 207–215, 2014. URL https://doi.org/10.1109/NAS.2014.39. → pages 51, 52

[87] J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, pages 81–88. ACM, 2009. → pages 9, 13, 42

[88] Q. Liu and Z. Yu. The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 347–360. ACM, 2018. ISBN 9781450360111. URL https://doi.org/10.1145/3267809.3267830. → page 74

[89] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single Shot MultiBox Detector. *Lecture Notes in Computer Science*, page 21–37, 2016. ISSN 1611-3349. URL http://dx.doi.org/10.1007/978-3-319-46448-0_2. → page 70

[90] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, 2002. → page 15

[91] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*, volume 2, pages 336–349, 2020. URL https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf. → page 70

[92] C. McAnlis. 5 steps to better GCP network performance, 2017. URL https://cloud.google.com/blog/products/gcp/5-steps-to-better-gcp-network-performance. → page 51

[93] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010. → page 9

[94] Microsoft Azure. Linux virtual machines pricing; december 24, 2016 snapshot, 2016. URL https://web.archive.org/web/20161224165940/https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux. → page 100

[95] MLCommons. MLCommons, 2021. URL https://mlcommons.org. → page 70

[96] NetworkX. Network Analysis in Python: networkx.algorithms.shortest_paths.weighted.dijkstra_path, 2021. URL https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.dijkstra_path.html. → page 86

[97] NIVIDIA-AI. BERT Meets GPUs, 2019. URL https://medium.com/future-vision/bert-meets-gpus-403d3fbed848. → page 70

[98] OpenStack. Quality of service (qos): Guaranteed minimum bandwidth, 2020. URL

https://docs.openstack.org/neutron/latest/admin/config-qos-min-bw.html. → page 83

[99] OpenStack. Open source cloud computing infrastructure, 2020. URL https://www.openstack.org. → page 81

[100] OpenStack. Nova Filter Scheduler, 2020. URL https://docs.openstack.org/nova/latest/user/filter-scheduler.html. → page 78

[101] OpenStack. OpenStack Docs: DevStack, 2021. URL https://docs.openstack.org/devstack. → page 83

[102] OpenStack. OpenStack Cells, 2021. URL https://docs.openstack.org/newton/config-reference/compute/cells.html. → page 122

[103] OpenStack. OpenStack Services, 2021. URL https://www.openstack.org/software/project-navigator/openstack-components#openstack-services. → page 137

[104] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 293–307, USA, 2015. USENIX Association. ISBN 9781931971218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout. → pages 106, 107

[105] D. A. Patterson. Technical perspective: The data center is the computer. *Commun. ACM*, 51(1):105, 2008. ISSN 0001-0782. URL https://doi.org/10.1145/1327452.1327491. → pages 1, 144

[106] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29. ACM, 2019. URL https://doi.org/10.1145/3341301.3359642. → pages 2, 50, 51, 107

[107] D. Poccia. New for AWS CloudFormation – Quickly Retry Stack Operations from the Point of Failure, 2021. URL https://aws.amazon.com/blogs/aws/new-for-aws-cloudformation-quickly-retry-stack-operations-from-the-point-of-failure. → page 119

[108] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005. ISSN 1433-2779. URL https://doi.org/10.1007/s10009-004-0183-4. → page 8

[109] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011. → page 11

[110] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. Association for Computing Machinery, 2012. ISBN 9781450317610. URL https://doi.org/10.1145/2391229.2391236. → page 74

[111] J. Rintanen. Madagascar: Efficient planning with SAT. *The 2011 International Planning Competition*, page 61, 2011. → page 8

[112] M. Rost, C. Fuerst, and S. Schmid. Beyond the stars: Revisiting virtual cluster embeddings. *ACM SIGCOMM Computer Communication Review*, 45(3):12–18, 2015. → pages 9, 10, 11, 12

[113] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik. Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/sapio. → pages 70, 71

[114] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference*, pages 205–218. USENIX Association, 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad. → pages 74, 75

[115] J. Sherry. *Middleboxes as a Cloud Service*. PhD thesis, EECS Department, University of California, Berkeley, Nov 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-165.html. → page xv

[116] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. URL https://arxiv.org/abs/1409.1556. → page 70

[117] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM '15, page 183–197. ACM, 2015. URL https://doi.org/10.1145/2785956.2787508. → pages 11, 93, 94, 169

[118] M. Y. Sir, I. F. Senturk, E. Sisikoglu, and K. Akkaya. An optimization-based approach for connecting partitioned mobile sensor/actuator networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 525–530. IEEE, 2011. → page 15

[119] N. Sorensson and N. Een. Minisat - a SAT solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005. → page 25

[120] G. Sun, D. Liao, S. Bu, H. Yu, Z. Sun, and V. Chang. The efficient framework and algorithm for provisioning evolving VDC in federated data centers. *Future Generation Computer Systems*, 73:79–89, 2017. URL https://doi.org/10.1016/j.future.2016.12.019. → page 73

[121] W. Szeto, Y. Iraqi, and R. Boutaba. A multi-commodity flow based approach to virtual network resource allocation. In *Global Telecommunications Conference (GLOBECOM)*, volume 6, pages 3004–3008. IEEE, 2003. → page 15

[122] A. N. Tantawi. A scalable algorithm for placement of virtual clusters in large data centers. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 3–10. IEEE, 2012. → page 9

[123] R. team. Runlim: Linux program to control benchmarks, 2021. URL http://fmv.jku.at/runlim. → page 95

[124] Tensorflow. Distributed training with TensorFlow, 2020. URL https://www.tensorflow.org/guide/distributed_training#mirroredstrategy. → page 56

[125] Tensorflow. Parameter server training, 2020. URL https://www.tensorflow.org/tutorials/distribute/parameter_server_training. → pages 4, 55

[126] H. Tian, Y. Zheng, and W. Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 139–151. ACM, 2019. ISBN 9781450369732. URL https://doi.org/10.1145/3357223.3362710. → pages 74, 75

[127] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. ACM, 2020. ISBN 9781450368827. URL https://doi.org/10.1145/3342195.3387517. → page 74

[128] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 513–527. USENIX Association, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/uta. → pages 2, 50, 104, 107

[129] A. Vadhat. SIGCOMM 2020 Keynote: Coming of Age in the Fifth Epoch of Distributed Computing, 2020. URL https://youtu.be/27zuReojDVw. → page 52

[130] VMWare. VMWare SD-WAN (Software-Defined Wide Area Network) and SASE (Secure Access Service Edge), 2021. URL https://sase.vmware.com. → page 144

[131] Wikipedia. Parallel computing, 2020. URL https://en.wikipedia.org/wiki/Parallel_computing. → page 51

[132] Wikipedia. SD-WAN, 2021. URL https://en.wikipedia.org/wiki/SD-WAN. → page 144

[133] Y. Yang, X. Chang, J. Liu, and L. Li. Towards robust green virtual cloud data center provisioning. *IEEE Transactions on Cloud Computing*, 5(2): 168–181, 2015. URL https://doi.org/10.1016/j.future.2016.12.019. → page 73

[134] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008. → pages 9, 11, 13, 15, 40

[135] Y. Yuan, A. Wang, R. Alur, and B. T. Loo. On the feasibility of automation for bandwidth allocation problems in data centers. In *Formal Methods in Computer-Aided Design, 2013*, FMCAD' 13, pages 42–45, 2013. URL https://doi.org/10.1109/FMCAD.2013.6679389. → pages 9, 11, 12, 25, 26, 27, 28, 29, 30, 32, 33, 37, 47, 56, 73, 77, 78

[136] ZeroStack. Zerostack private cloud, 2016. URL http://zerostack.com. → page 36

[137] M. F. Zhani, Q. Zhang, G. Simona, and R. Boutaba. VDC Planner: Dynamic migration-aware Virtual Data Center embedding for clouds. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 18–25, 2013. URL https://ieeexplore.ieee.org/document/6572965. → pages 9, 10, 138

Appendix A

Datacenters in a Private Cloud

We show topologies of four commercial datacenters used in Section 2.4.4. Servers have 16 cores and 32 GB RAM. Link bandwidths are in Gbps. Datacenters are sorted in an ascending order by the number of servers in them.

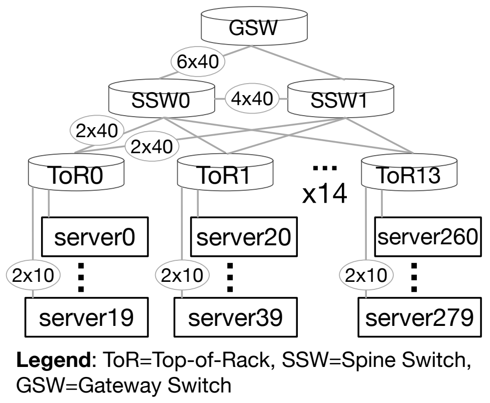


Figure A.1: US-West2 Datacenter Topology with 280 Servers.

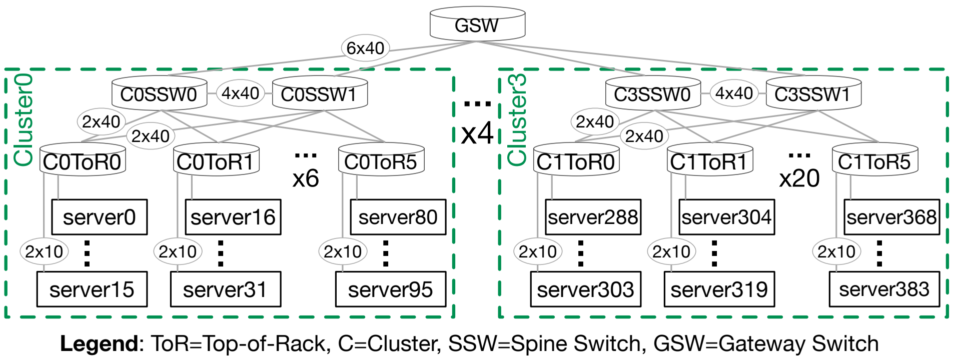


Figure A.2: US-Mid1 Datacenter Topology with 384 Servers.

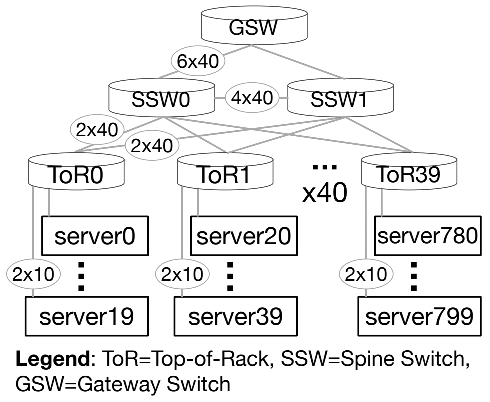
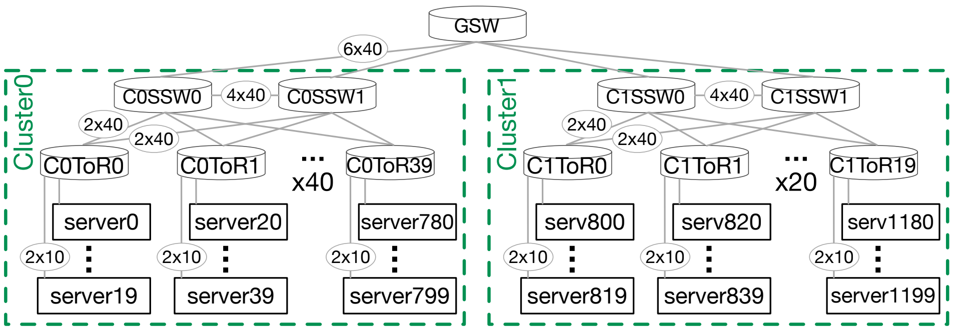


Figure A.3: US-Mid2 Datacenter Topology with 800 Servers.



Legend: ToR=Top-of-Rack, C=Cluster, SSW=Spine Switch, GSW=Gateway Switch

Figure A.4: US-West1 Datacenter Topology with 1200 Servers.

Appendix B

VDC Workload Generation Pseudocode

We show the pseudocode for generating VDC workload, which we call `workload.json` from Azure traces, which we call `azure.csv`. The `azure.csv` is the preprocessed CSV file where we already removed *instant-VMs*, VMs with `create_tick` equal to `delete_tick`, and already fixed the VM create/delete timestamps by rounding their values to the nearest valid tick (see Section 3.1).

All deployments in `azure.csv` will be present in the generated `workload.json`. If deployment's `peak_deploy_size` in `azure.csv` exceeds the `max_vdc_size` threshold, the overflow VMs will be assigned to child VDC(s). A deployment VM is called an overflow VM if it arrives after the deployment size (`peak_deploy_size`) reaches the `max_vdc_size`. Child VDCs get UUIDs based on this formula:

$$child\_uuid = concat(parent\_uuid, concat\_str, child\_index)$$

We use `concat_str` = __, with two underscores, because deployment UUIDs in `azure.csv` do not contain this string. Using a unique string allows us to manually inspect the relationship between parent and child VDCs for debugging purposes. The child VDC UUID generation pseudocode is shown in `get_child_vdc_uuid()` function.

Deployments in `azure.csv` have 1-to-N relationship with VDCs, i.e., a de-

Algorithm 4 : VDC workload generation

```
max_vdc_size = 30                                        ▷ cap threshold
bpc = 1                                    ▷ bandwidth_per_core=1 Mbps
concat_str = '__'
vdc_child_index =                          ▷ {parent_vdc_uuid: 3, ...}
nticks = get_number_of_ticks(csv)            ▷ this is 8640 in azure.csv

procedure main()
 1:  ticked_workload = get_ticked_workload(csv)
 2:  batched_workload = batch_tick_vdcs(ticked_workload)
 3:  vm_peers, vm2vdc = chop_vdcs(batched_workload)
 4:  vdc_workload = add_network(batched_workload, vm_peers, vm2vdc)
 5:  output_to_json_file(vdc_workload)
end procedure
procedure get_ticked_workload(csv)
 6:  workload = {}
 7:  foreach tick in nticks:
 8:    workload[tick] = deque()         ▷ deque is the double ended queue
 9:  foreach line in csv
10:    vm_uuid, vdc_uuid = get_vm_uuid(line), get_vdc_uuid(line)
11:    cores, ram = get_vm_cores(line), get_vm_ram(line)
12:    create_tick = get_vm_create_time(line)
13:    delete_tick = get_vm_delete_time(line)
       ▷ add create event to the workload
14:    workload[create_tick].append(['create', vm_uuid, vdc_uuid, cores, ram])
       ▷ add delete event to the workload
15:    workload[delete_tick].appendleft(['delete', vm_uuid, vdc_uuid])
16:  return workload
end procedure
```

ployment can produce more than one VDC. The `child_index` variable captures the current number of VDC children and is used to derive the child VDC UUID. For example, a deployment with `deploy_uuid=abc` and `peak_deploy_size`=61 in `azure.csv` will produce three VDCs, one parent and two child VDCs, with the following VDC UUIDs and sizes, respectively:

$$vdc\_uuid : peak\_vdc\_size \mapsto \{abc : 30, abc\_\_0 : 30, abc\_\_1 : 1\}$$

```
procedure batch_tick_vdcs(workload)
        ▷ batching ensures that all VMs in a VDC within the same tick appear
        ▷ in consecutive order as batched together, i.e., there is no VM of another
        ▷ VDC between the VMs that are being batched.
        ▷ Batching does not apply to VM delete events.
 1:     batched_workload = {}
 2:     foreach tick in workload
 3:       batched_workload[tick] = []
 4:     foreach tick, events in workload.items()
 5:       vdcs = {}
 6:       foreach event in events
 7:         if event.type == 'delete'
 8:           batched_workload[tick].append(event)
 9:           continue
            ▷ this is a create event, we just process it without appending to
            ▷ the batched_workload, yet
10:         if event.vdc_uuid in vdcs
11:           vdcs[event.vdc_uuid].append(event)
12:         else vdcs[event.vdc_uuid] = [event]
          ▷ append VDCs one-by-one to get the VDC VMs batched
13:       foreach vdc_uuid, events in vdcs.items()
14:         foreach event in events
15:           batched_workload[tick].append(event)
16:     return batched_workload
end procedure
```

```
procedure chop_vdcs(workload)
    ▷ Cap each VDC size to honour max_vdc_size
 1: vdc_uuid_map = {}        ▷ eg, {vdc_uuid_parent: vdc_uuid_current, ...}
 2: vdc_alive_vms = {}       ▷ eg, {vdc_uuid: [vm_uuid1, vm_uuid2, ...], ...}
 3: vm2vdc = {}              ▷ eg, {vm_uuid: vdc_uuid, ...}
 4: vm_peers = {}            ▷ eg, {vm_uuid: [peer_vm_uuid1, ...], ...}
 5: foreach tick, events in workload.items()
 6:   foreach event in events    ▷ 1st pass to build the VM-to-VDC membership
 7:     if event.type == 'delete'
          ▷ we must have already seen a create pair for this event
 8:       vdc_uuid_current = vm2vdc[event.vm_uuid]
 9:       assert(vdc_uuid_current in vdc_alive_vms)
10:       vdc_alive_vms[vdc_uuid_current].remove(event.vm_uuid)
11:       continue
        ▷ this is a create event: decide which VDC this VM gets assigned to
12:     if event.vdc_uuid not in vdc_uuid_map
          ▷ this is the first ever VM of this parent VDC
13:       vdc_uuid_map[event.vdc_uuid] = event.vdc_uuid
14:       vm2vdc[event.vm_uuid] = event.vdc_uuid
15:       vdc_alive_vms[event.vdc_uuid] = [event.vm_uuid]
16:       continue
17:     vdc_uuid_current = vdc_uuid_map[event.vdc_uuid]
        ▷ check peak_vdc_size and decide on child VDCs
        if len(vdc_alive_vms[vdc_uuid_current]) < max_vdc_size
18:       vdc_alive_vms[vdc_uuid_current].append(event.vm_uuid)
19:       vm2vdc[event.vm_uuid] = vdc_uuid_current
20:     else                     ▷ create a child VDC
21:       child_vdc_uuid = get_child_vdc_uuid(event.vdc_uuid)
22:       vdc_alive_vms[child_vdc_uuid] = [event.vm_uuid]
23:       vdc_uuid_map[event.vdc_uuid] = child_vdc_uuid
24:       vm2vdc[event.vm_uuid] = child_vdc_uuid
25:   foreach event in events    ▷ 2nd pass to build VM-to-peers dictionary
26:     if event.type == 'delete': continue
        ▷ This is a create event: add all alive VDC VM as peers. Important note:
        ▷ vdc_alive_vms operates on vdc_uuid_current, not on event.vdc_uuid.
27:     all_peers = vdc_alive_vms[vm2vdc[event.vm_uuid]]
28:     vm_peers[event.vm_uuid] = all_peers - event.vm_uuid
29: return (vm2vdc, vm_peers)
end procedure
```

```
procedure get_child_vdc_uuid(parent_uuid)
    if parent_uuid in vdc_child_index
        vdc_child_index[parent_uuid] += 1
    else vdc_child_index[parent_uuid] = 0
    return concat(parent_uuid, concat_str, vdc_child_index[parent_uuid])
end procedure
procedure add_network(workload, vm2vdc, vm_peers)
        ▷ the ‘workload’ in the parameter already has CPU/RAM fields;
        ▷ we add ‘net_conn_in_mbps’ field to the ‘create’ events
    foreach tick, events in workload.items()
        foreach event in events
            if event.type == delete: continue
            conn = {}
            for peer in vm_peers
                conn[peer] = bpc * min(event.cores, peer.cores)
            workload[‘net_conn_in_mbps’] = conn
    return workload
end procedure
```

Appendix C

Datacenters with Jupiter Topology

We first describe our reconstruction of the full Jupiter topology based on the description in the Jupiter paper [117], followed by our methodology to derive a subset 4-pod topology from the full topology.

C.1 Full Jupiter Topology

Figure C.1 shows the full Jupiter topology with a sample pod. A single pod has 32 racks, each with 48 servers, for a total 1536 servers. Each server has 1x40 Gbps uplink to a top-of-rack (ToR) switch made of a Centauri chassis. A Centauri chassis hosts four chips, each with 16x40G (we shorten "Gbps" to "G" for brevity) bandwidth that can offer 16x40G or 64x10G, since each port can operate either in 1x40G or in 4x10G mode. A chip's bandwidth is split into a 3:1 downlink:uplink oversubscription ratio (also called south:north), i.e., 48x10G to the rack servers and the remaining 16x10=8x2x10G to the Middle Blocks (MB). Thus, four chips in a ToR offer 4x48x10=48x40G downlink and 4x16x10=16x40G uplink. The 16x40G uplink chip capacity is consumed by 8 MBs, 2x10G uplinks to each MB. Figure C.1 shows an aggregate ToR-to-MB capacity, which is 4x2x10G for four chips in a ToR. The 48x40G downlink capacity is consumed by 48 servers, 1x40G downlink to each server. Thus, full Jupiter topology has 3:1 downlink:uplink oversubscrip-

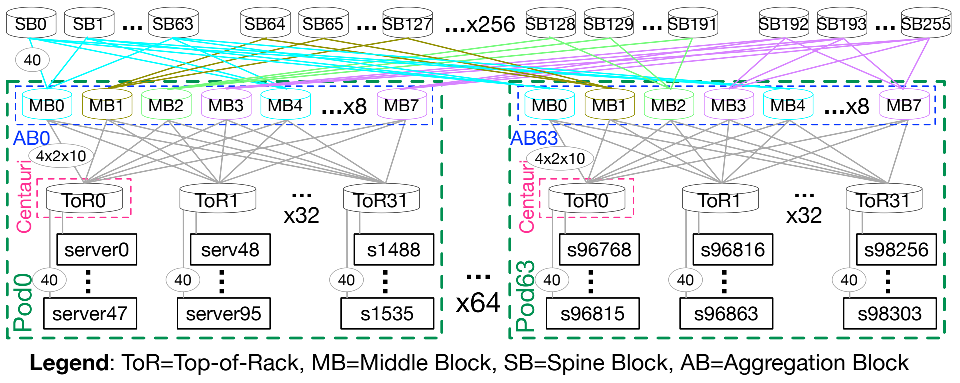


Figure C.1: Full Jupiter Datacenter Topology.

Table C.1: Node Connectivity in Full Jupiter Datacenter Topology. We show wiring between Aggregation Blocks (AB) and Spine Blocks (SB).

AB0MB0 - SB0	AB0MB2 - SB128 ...	AB1MB0 - SB0 ...
AB0MB0 - SB1	AB0MB2 - SB191	AB1MB7 - SB255 ...
...		
AB0MB0 - SB63	AB0MB3 - SB192 ...	AB2MB0 - SB0 ...
	AB0MB3 - SB255	AB2MB7 - SB255 ...
AB0MB1 - SB64 ...	AB0MB4 - SB0 ...	...
AB0MB1 - SB127	...	AB63MB0 - SB0 ...
	AB0MB7 - SB255	AB63MB7 - SB255

tion ratio in the ToR layer.

An Aggregation Block (AB) has 8 MBs, each with 8 chips, for a total of 64 chips. Figure C.1 shows AB in the blue dashed box. We refer to AB as a *pod* for readability, i.e., all servers under an AB belong to a single pod. Each MB chip offers 32x10G downlinks that is an aggregate 8x32x10=256x10G for 8 chips in the MB. An MB chip accepts one 2x10G connection from each of 32 ToRs for a total of 32x2x10=64x10G downlink bandwidth. That is 8x64x10=256x10G downlink for 8 chips in the MB. There is no oversubscription in the MB layer. Thus, with 8 MBs in an AB, the aggregate AB uplink bandwidth is 8x256x10G.

There are 256 Spine Blocks (SB) to support 64 ABs. An SB has 16 chips that provide an aggregate 16x8x40G downlink capacity. There is only one dual-redundant 40G connection from an AB to SB. This means that each AB connects to two SBs, 1x40G each. For example, only MB0 and MB4 (in an AB) connect to

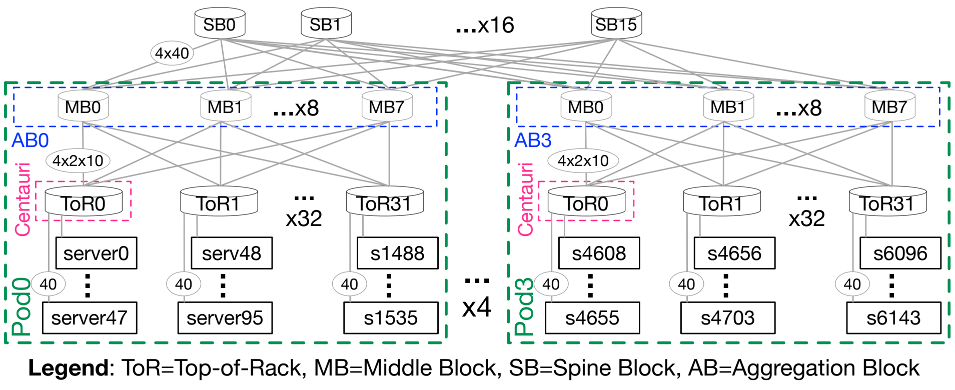


Figure C.2: Four-pod Jupiter Datacenter Topology.

SB0. Table C.1 demonstrates AB-to-SB connectivity, which can be captured with this formula (note: (y `mod` 4) expression provides the dual redundancy):

- AB(x)MB(y) connects to SB(64*(y `mod` 4)+i) where $0 \leq x, i < 64$ and $0 \leq y < 8$; "i" is the MB-to-SB repeat index.

The wiring layout in Table C.1 satisfies the MB uplink and SB downlink radix. Recall that each MB chip has 8 uplinks that are 40G each (8x40G). This gives 8x8x40G uplinks for 8 chips in an MB. As the table and formula show, the repeat index "i" ranges from 0 to 63 for each MB, giving 64 uplinks per MB. For example, AB0MB0 has uplinks to SB0-to-SB63. Each SB chip also has 8 downlinks that are 40G each (8x40G). This gives 16x8x40=128x40G downlinks for 16 chips in an SB. As the table and formula show, an SB connects to every 4th MB of an AB, which is called "striping in a superblock" size of 4 MBs. With two superblocks (8 MBs) in each AB and 64 ABs in the Jupiter cluster, an SB has 2x64x40=128x40G downlinks. For example, SB0 has downlinks to (AB0MB0, AB0MB4, AB1MB0, AB1MB4, ..., AB63MB0, AB63MB4).

C.2 Four-pod Jupiter Topology

The 4-pod Jupiter datacenter has 16 times fewer pods compared to the full Jupiter datacenter. Thus, we keep the first 4 pods and omit the remaining 60 pods. However, omitting 60 pods leaves many Spine Block (SB) ports idle, i.e., the SB port

Table C.2: Node Connectivity in Four-pod Jupiter Datacenter Topology. We show wiring between Aggregation Blocks (AB) and Spine Blocks (SB).

AB0MB0 - SB0	AB1MB0 - SB0 ...	AB2MB0 - SB0 ...	AB3MB0 - SB0 ...
AB0MB0 - SB15	AB1MB0 - SB15 ...	AB2MB0 - SB15 ...	AB3MB0 - SB15 ...
AB0MB1 - SB0 ...	...	...	...
AB0MB1 - SB15	AB1MB7 - SB0 ...	AB2MB7 - SB0 ...	AB3MB7 - SB0 ...
...	AB1MB7 - SB15	AB2MB7 - SB15	AB3MB7 - SB15
AB0MB7 - SB0 ...			
AB0MB7 - SB15			

radix would differ between 4-pod and 64-pod datacenters. We avoid this by dividing the number of SBs by 16 times such that no SB port is left idle in the 4-pod datacenter. Table C.2 shows AB-to-SB connectivity where each connection's bandwidth has changed from 1x40G in the full topology to 4x40G in the 4-pod datacenter. This happens because each MB in an AB in the 4-pod datacenter connects to 16 SBs, instead of 64 SBs in the full topology. The 4x40G connection can be achieved by using 4 ports in MBs and SBs. We represent it with a single link with 4x40G bandwidth.

Figure C.2 shows the 4-pod Jupiter topology. Note that the pod-internal connectivity is identical between the 4-pod and the full topology. The only difference between 4-pod and full topology is MB-to-SB connectivity, which is caused by reducing the number of SBs as mentioned earlier. Thus, the wiring layout in the 4-pod topology has all-to-all connectivity between every MB and every SB, as shown in Table C.2. We can capture this layout with the following formula:

- AB(x)MB(y) connects to SB(i) where $0 \leq x < 4$, $0 \leq y < 8$, and $0 \leq i < 16$; "i" is the MB-to-SB repeat index.

The wiring layout in Table C.2 also satisfies MB uplink and SB downlink port radixes. Each MB chip has 8 uplinks that are 40G each (8x40G). This gives 8x8x40=64x40G uplinks for 8 chips in an MB. As the Table C.2 and the above formula show, the repeat index "i" ranges from 0 to 16 for each MB, giving 16x4x40=64x40G uplinks per MB. For example, AB0MB0 has 4x40G uplinks to SB0-to-SB15. Each SB chip also has 8 downlinks that are 40G each (8x40G). This gives 16x8x40=128x40G downlinks for 16 chips in an SB. Again, as Table C.2

and formula show, an SB connects to every MB with 4x40G links. With 4 ABs, 8 MBs per AB, an SB has 4x8x4x40=128x40G downlinks. For example, SB0 has downlinks to (AB0MB0, AB0MB1, AB0MB2, ..., AB0MB7, ..., AB4MB7).

Appendix D

VM Allocation Failures in Practice

The revenue gain metric fundamentally depends on VMs not getting allocated, or rejected, due to insufficient datacenter network bandwidth. One might ask a legitimate question: *Do VM rejections happen in practice?* Note that this is a general question about all VM resources, not limited to network bandwidth guarantees. We answer this question in the context of the most basic VM resource: compute. We survey the prior work to answer the following question: *Do cloud providers reject a VM when a datacenter has insufficient compute capacity to host the VM?* The Resource Central paper [40], gives a negative answer. Although the Resource Central authors do not directly state that VMs get allocated without sufficient compute capacity available in the datacenter, they do state that server utilization can exceed 100% and this is considered a "VM scheduling failure" for the VM(s) hosted in that server [40].

The VM scheduling failures, per the Resource Central paper [40], happen when a VM gets successfully allocated but it operates, perhaps intermittently, with lower compute capacity than what is "promised" in its flavor. For example, a VM gets allocated with 12 cores but might operate with 11 cores for the duration when the host server's utilization is above 100%. We call this duration the *under-capacity* operation time.[1] The Azure cloud operators use the fact that tenant VMs do not use

[1] Tenants can read a VM's "steal time" CPU counter to check if it is operating under-capacity [47].

all of the compute resources they request. The operators monitor the VMs' CPU utilization levels at runtime and adjust the server CPU oversubscription levels to minimize the amount of time that VMs execute with CPU resources below their requested level. In other words, the VM scheduler objective in the Resource Central paper is to minimize the number of VM scheduling failures.

This dissertation is about scheduling datacenter network bandwidth for VDCs. Given that our VDC workloads lack information about VDC VMs' runtime bandwidth utilization levels, i.e., the VDC workload contains only the *requested* bandwidth, the VDC scheduler's objective is to minimize the number of failed VMs solely based on the request. The revenue gain metric directly captures the value of failed VMs by omitting their revenue. Similar to how Resource Central monitors VM CPU consumption at runtime, one can monitor VM network bandwidth consumption at runtime and apply the oversubscription concept to datacenter networking. It will then be possible to develop a VDC scheduler whose objective is to minimize the bandwidth under-capacity at runtime. We leave this for future work.