Adding flexibility to multi-tenant networks

THÈSE N^{O} 8243 (2018)

PRÉSENTÉE LE 1ER FÉVRIER 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DES RÉSEAUX
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Georgios IOANNIDIS

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. A. Argyraki, directrice de thèse
Prof. G. Pierre, rapporteur
Prof. A. Wolf, rapporteur
Prof. E. Bugnion, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

Science bestowed immense new powers on man and at the same time created conditions which were largely beyond his comprehension and still more beyond his control.

Winston Churchill
address at MIT, 1949

To my parents and my sister.

For endlessly supplying me with courage and optimism since day #0.

Acknowledgments

No words may accurately portray my current thoughts, as I feel overwhelmed with mixed emotions, just if I were standing at a crossroads. A long and eventful journey is coming to an end, but a new, promising one, is lying ahead. As I am concluding my thesis, I would like to thank a few people who have, in their own way, contributed to this work.

First and foremost, this dissertation would not have been materialized without the continuous support of my thesis advisor, professor Katerina Argyraki. If I could summarize her core principles in one phrase, that would be: Katerina will vow to transform a graduate student into a fully fledged, independent researcher. She would always allow me room to expand and experiment with my own ideas, guide me on coming up with my proposals and interpreting the experimental results, as well as suggest considering alternative approaches when an experiment would run into an apparent dead-end. At the same time, Katerina had been repeatedly available whenever I would run into trouble. Ever heard of the “advisor’s aura”? This is exactly the feeling I would experience during my meetings with Katerina, as even five minutes would be enough to imbue me with inspiration and confidence to move on. Katerina, it has been an honor and a privilege to work with you, and I would like to wholeheartedly thank you for your guidance and for believing in this work.

Another significant moment in this journey was working with professor Guillaume Pierre in Rennes, France. Guillaume welcomed me and facilitated my integration within the Myriads team, where I had the opportunity to work for eight months. He would happily voice his views on my work, allowing me nevertheless to settle with my own, final decisions. Furthermore, he is a remarkably pleasant person to communicate with; we have shared during the past three years countless discussions on a broad spectrum of topics, ranging from scientific advancements to everyday stories. Guillaume, I would like to thank you very much for all your support.

I would also like to individually thank the rest of my jury committee members, professors Edouard Bugnion, Alexander L. Wolf and Willy Zwaenepoel. Ed was the first senior scientist I worked with, after joining Katerina’s lab, bringing valuable experience straight from the industry. I had the opportunity to collaborate with Alex during the HARNESS European project, where his guidance and comments facilitated the integration of my work within the big picture. Willy, the president of my jury committee, aided the smooth procedure of my oral examination by alleviating any anxiety that I would naturally have.

Furthermore, I would like to acknowledge two more collaborators: Dr. (José) Gabriel de Figueiredo Coutinho, and Dr. Mark Stillwell, both affiliated with Imperial College London.

Gabriel, Mark, I would like to express my true thanks for facilitating the deployment of real applications on the cloud, a process that was neither trivial nor, at times, straightforward. You were always available when issues would arise, promptly responding even late at night.
My colleagues from the Network Architecture Laboratory have also formed an influential element across my entire PhD journey. Ovidiu, we have shared the same office for the past six years, as well as our struggles and our endeavors, and I am very glad to have met you. Mihai, you tremendously facilitated my integration within the lab team and significantly aided my initial presentations with your valuable feedback and advice. Pavlo, you have significantly reduced my stress levels with your calm and positive energy. Dimitri, I am very happy to have shared all these interesting conversations with you, including the classic "`emacs` vs `vim`" arguments. Jonas, your experience was essential and fundamental during our work on real, Top-of-Rack switches. Luis and Georgia, you have tremendously helped me in managing my anxiety during the past months, as I was writing my thesis, with our diverse discussions.
Of course, I could not neglect my friends within the Greek community. A big thanks to all of you – you know who you are – for all the moments that we have shared. Although I cannot delve into details and individually address everyone, as this chapter would exceed the size of the rest of the thesis, I would like to specifically acknowledge one dear friend: Matt, you had been always around for anyone in need of your help or just to chat and your tremendous positive vibe is truly inspiring!
Even if it has been more than six years since I moved to Switzerland, I have to mention two people back in Greece, with whom I am maintaining very close ties: Lykourgos and Athanasia. Both played their own role in facilitating my first and last year, respectively, of my PhD journey. "Lyk," I feel exalted to have been maintaining a strong friendship with you for more than a decade and your encouragement was fundamental during my initial year in Switzerland. Athanasia, your continuous inspiration and support during the last months was indispensable, while I was writing my thesis and preparing my presentation, as you enormously galvanized me in enduring this long and arduous process. I owe you my sincere gratitude for all the optimism you have imbued me with.
Finally, none of this would had happened without the never-ending support and encouragement of my parents and my sister Anna. They were literally the first to believe in me and would steadily encourage me, in spite of any difficulties I would come across. Mother, Father and Anna, I would like to genuinely express to you my eternal indebtedness and affirm that I feel blessed to have you in my life.

Lausanne, December 2017 George Ioannidis

Abstract

Cloud computing has been experiencing sharp development over the last years, leading to an increased demand for application migration to the cloud. Cloud providers, in an effort to attract more customers and earn their confidence, offer to tenants the illusion of an isolated network, exposing familiar abstractions. At the same time, creating this illusion poses challenging problems for the providers, as one tenant's traffic may interfere with another's in complicated, unpredictable ways.

First, new challenges have arisen in administering access-control rules (ACLs). On the one hand, installing ACLs at the server is incompatible with bare-metal support and introduces unnecessary performance overhead. On the other hand, offloading the most popular ACLs on the limited hardware memory in Top-of-Rack (ToR) switches should not be conducted naïvely, as the existence of wildcard rules presents inter-rule dependencies that must be respected.

Second, tenants' demands have evolved beyond requesting hardware resources; for instance, tenants may require bandwidth provisions between their resources or optimized access to a specific cloud service, e.g., a Mail server or a Database. Cloud providers have not adequately adapted to these expanding demands, therefore elevating hardware resources to "first class citizens," as non-hardware constraints are not considered during resource allocation, instead they are applied afterwards.

In this thesis we propose two architectures that facilitate cloud providers in managing their shared network resources in a flexible way. First, we demonstrate *virtual flow tables*, a ToR architecture that handles ACLs using a two-level memory hierarchy. The most popular ACLs are stored in the limited hardware memory, respecting any dependencies between wildcard rules, while the ToR's supervisor engine maintains access to the entire ACL rule-set. Second, we present a *two-tiered architecture* for scheduling cloud resources, consisting of a resource-agnostic scheduling layer and a resource-specific enforcement layer. Network resources and constraints are taken into consideration during resource scheduling, instead of afterwards, while resource provisioning, as well as general network-management policies, are delegated to the resource-specific tier.

Key words: caching, cloud architecture, flexibility, network virtualization, scheduling, wildcard rules

Résumé

Le cloud computing a connu un fort développement au cours des dernières années, entraînant une demande accrue pour la migration d'applications vers le cloud. Les fournisseurs de services cloud, dans le but d'attirer plus de clients et gagner leur confiance, offrent aux locataires l'illusion d'un réseau isolé, exposant des abstractions familières. Dans le même temps, créer cette illusion pose des problèmes difficiles pour les fournisseurs de services cloud, car le trafic d'un locataire peut interférer avec celui d'un autre de manière compliquée et imprévisible.

Tout d'abord, de nouveaux défis sont apparus dans l'administration des règles de contrôle d'accès (ACLs). D'une part, l'installation des ACLs sur le serveur est incompatible avec le soutien de "bare-metal" et entraîne des pertes de performances inutiles. D'autre part, le déchargement des ACLs les plus populaires sur la mémoire matérielle limitée dans les commutateurs du Top-of-Rack (ToR) ne doit pas être effectué naïvement, car l'existence des règles génériques présente des dépendances entre règles qui doivent être respectées.

Deuxièmement, les demandes des locataires ont évolué au-delà de la simple demande de ressources matérielles ; par exemple, les locataires peuvent nécessiter des dispositions de bande passante entre leurs ressources ou un accès optimisé à un service de cloud spécifique, par exemple un serveur de messagerie électronique ou une base de données. Les fournisseurs de services cloud ne se sont pas adaptés de manière adéquate à ces exigences croissantes, élevant ainsi les ressources matérielles aux "citoyens de première classe", car les contraintes non matérielles ne sont pas prises en compte lors de la planification des ressources.

Dans cette thèse, nous proposons deux architectures qui facilitent les fournisseurs de services cloud dans la gestion de leurs ressources réseau partagées de manière flexible. Premièrement, nous démontrons des *tables des règles virtuel*, une architecture du ToR qui gère les ACLs en utilisant une hiérarchie de mémoire à deux niveaux. Les ACLs les plus populaires sont stockées dans la mémoire matérielle limitée, en respectant les dépendances entre les règles génériques, tandis que le moteur de supervision du ToR conserve l'accès à toutes des règles. Deuxièmement, nous présentons une architecture à deux niveaux pour la planification des ressources du cloud, constituée d'une couche de planification des ressources agnostique et d'une couche d'application des ressources spécifique. Les ressources réseau et les contraintes sont prises en compte lors de la planification des ressources, et non plus après, tandis que le provisionnement des ressources, ainsi que les politiques générales de gestion du réseau, sont déléguées au niveau spécifique à la ressource.

Mots clefs : architecture cloud, flexibilité, mise en cache, planification du placement, règles

génériques, virtualization des réseaux

Contents

Acknowledgments i

Abstract (English/Français) iii

Contents ix

List of Figures xi

List of Tables xiii

List of Algorithms xv

Listings xvii

1 Introduction 1
1.1 Managing Shared Flow Tables 1
1.2 Scheduling Shared Network Resources 2
1.3 Contributions 3

2 Related Work 5
2.1 Flow-Rule Forwarding in Software-Defined Networking 5
2.1.1 Packet Classification 5
2.1.2 Cache Management for Flow Rules 6
2.1.3 Our Approach in Facilitating Caching Overlapping Rules 7
2.2 Scheduling Cloud Resources 8
2.2.1 Bandwidth Provisioning 8
2.2.2 Resource Oversubscription 10
2.2.3 Enforcing Bandwidth Fairness 10
2.2.4 Our Approach in Facilitating Cloud Resource Scheduling 11

3 Managing Shared Flow Tables 13
3.1 Virtual Flow Tables 13
3.1.1 Abstraction and Implementation 13
3.1.2 The Challenge of Overlapping Rules 15
3.2 Cache Management for Overlapping Rules 19
3.2.1 The Software Table 19

3.2.2 Rule Replacement 25
3.2.3 Baseline Algorithm: Least Recently Used 25
3.2.4 An Implementable Algorithm: Most Heavily Used 26
3.3 Experimental Evaluation 29
3.3.1 Packet traces 30
3.3.2 Creating a synthetic overlapping rule set 30
3.3.3 Independent variables 31
3.3.4 Dependent variables 33
3.3.5 Results 34
3.4 Summary 44

4 Scheduling Shared Network Resources **47**
4.1 Two-Tiered Resource Scheduling 47
4.2 The Network Infrastructure Resource Manager 48
4.2.1 Modeling Network Links 49
4.2.2 Exposing Virtual Paths 50
4.2.3 Bandwidth Reservations 51
4.2.4 Latency Constraints 53
4.2.5 Oversubscription 54
4.2.6 Fair Sharing of the Network 56
4.3 Experimental Evaluation 56
4.3.1 Testbed 58
4.3.2 Applications 58
4.3.3 Scenario 1: Scheduling with Bandwidth Reservations 62
4.3.4 Scenario 2: Scheduling with Latency Constraints 66
4.3.5 Scenario 3: Scheduling with Oversubscription 68
4.3.6 Scenario 4: Enforcing Fairness 73
4.4 Summary 74

5 Conclusions **77**

A Flow Rule Definitions and Notations **79**
A.1 Flow Rule Concepts 79
A.2 Flow Rule Fields 80
A.3 Rule Matching 80
A.4 Flow Rule Hyperspace 80
A.5 Overlapping Flow Rules 81
A.5.1 Directly Overlapping Rules 81
A.5.2 Indirectly Overlapping Rules 82

B Packet Classifier Node Definition **83**

Bibliography **94**

Curriculum Vitae **95**

List of Figures

3.1 Virtual Flow Table abstraction (top) and underlying memory hierarchy (bottom) 14
3.2 System architecture 15
3.3 Ingress bytes handled by the software table versus the hardware table size . . . 36
3.4 Ingress bytes handled by the software table versus the maximum rules in a classifier partition 37
3.5 Ingress bytes handled by the software table versus the memory allocation factor of the partitioning algorithm 37
3.6 Ingress bytes handled by the software table versus the history factor of the MHU policy 38
3.7 Flow rule replacement rate versus the hardware table size 39
3.8 Flow rule replacement rate versus the maximum rules in a classifier partition . 40
3.9 Flow rule replacement rate versus the memory allocation factor of the partitioning algorithm 40
3.10 Flow rule replacement rate versus the history factor of the MHU policy 41
3.11 Individually cached flow rules versus the hardware table size 42
3.12 Individually cached flow rules versus the maximum rules in a classifier partition 43
3.13 Individually cached flow rules versus the memory allocation factor of the partitioning algorithm 43
3.14 Individually cached flow rules versus the history factor of the MHU policy . . . 44
3.15 Replication rate of flow rules 45

4.1 Two-tiered architecture to facilitate scheduling shared network resources . . . 48
4.2 Example of a simple “star” LAN topology 49
4.3 Grid’5000 network backbone 58
4.4 AdPredictor worker throughput over time 64
4.5 AdPredictor execution time over link capacity 64
4.6 AdPredictor execution time over cluster placement 65
4.7 WikiBench request miss rate, no latency constraints, based on originating cluster 67
4.8 WikiBench request Round-Trip-Time distribution, based on latency constraints 68
4.9 AdPredictor worker throughput over time; single tenant 70
4.10 AdPredictor “shuffle” traffic throughput over concurrently admitted tenants . . 70
4.11 AdPredictor execution time over concurrently admitted tenants 71
4.12 Link utilization over concurrently admitted tenants 71

4.13 AdPredictor "shuffle" throughput with bandwidth oversubscription, $N = 3$ tenants 72
4.14 Enforcing fairness on `iperf` applications 74

List of Tables

3.1 Characteristics of packet traces; original and accelerated versions 30
3.2 Independent variables used in virtual flow tables evaluation scenarios 33

4.1 Cluster specifications within Grid'5000 59
4.2 Round-Trip-Time of two batches of 100 requests to the wikipedia server 61
4.3 Size of Danish and English wikipedia dumps of 2016-08-01 62

A.1 Notations used in flow rule related context 79

List of Algorithms

3.1 Recursive partitioning of the flow rule hyperspace 21
3.2 Detect directly overlapping rules within a given leaf node 23
3.3 Detect indirectly overlapping rules within a given leaf node 24
3.4 “Least Recently Used” replacement policy 27
3.5 “Most Heavily Used” replacement policy 28
3.6 Evaluation of virtual flow tables 30
3.7 Generating a synthetic overlapping rule set from a packet trace 32

4.1 Exposing end-to-end bandwidth to the scheduler 52
4.2 Creating a new bandwidth reservation 53
4.3 Exposing end-to-end latency to the scheduler 54
4.4 Dynamic bandwidth oversubscription 56
4.5 Fair sharing of the network 57

Listings

4.1 Internal representation of network links in figure 4.2 49
4.2 Virtual paths exposed to the scheduler in figure 4.2 50
4.3 Scheduling with bandwidth resources example 51
4.4 Scheduling with latency constraints example 53
B.1 Partial definition of the `HiCutNode` class 83

1 Introduction

A key goal of modern cloud providers is to offer to each of their tenants the illusion of an isolated network that is under the tenant's control, e.g., by exposing to the tenants abstractions like layer-2 broadcast domains, IP subnets, or security groups. This illusion makes it easier for tenants to replicate their physical network organization in the cloud and manage it with the same familiar processes. At the same time, creating this illusion poses challenging problems for the cloud providers, as one tenant's traffic may interfere with another's in complicated, unpredictable ways.

In this thesis, we propose two mechanisms that are meant to help cloud providers manage shared network resources in ways that are compatible with the above goal. In particular, we propose: (i) a mechanism for managing shared flow tables in packet switches; and (ii) a mechanism for managing the throughput and latency experienced by different tenants sharing the same network links.

1.1 Managing Shared Flow Tables

Flow tables in packet switches are an ideal place for storing access-control rules (ACLs), i.e., rules that specify which source and destination addresses/ports are allowed to exchange traffic. ACLs are the main mechanism offered to cloud tenants for controlling their communications, and they enable tenants to enforce policies that in a physical network would be enforced by traditional stateful firewalls. Today, cloud providers typically install ACLs at the server, within the operating system (OS) that hosts the tenant virtual machines (VMs) or containers. This approach is convenient, because it does not require any changes to network devices, which are typically hard to reprogram. This convenience, however, comes at the cost of significant disadvantages:

- Incompatibility with bare-metal support (where the cloud provider offers tenants access to physical machines): For security reasons, ACLs must be installed at an entity outside the one being governed. For instance, if a rule specifies that tenant X must not send any

traffic to tenant Y, it does not make sense to let tenant X itself enforce this rule. Hence, when tenants are given access to servers, ACLs must be installed somewhere *outside* the server.

- Unnecessary performance overhead: Compute virtualization platforms are designed to avoid host-OS involvement as much as possible, and installing ACLs within the host OS violates this mentality. Bypassing the host OS with single-root I/O virtualization (SR-IOV) [45] and offloading the implementation of network abstractions elsewhere has tremendous potential for performance improvement [12, 44].

Instead, we propose installing ACLs at the Top-of-Rack switch (ToR). Others have taken similar approaches: FasTrak also installs ACLs at the ToR, but only for a few flows selected by the guest OS [44]. In contrast, our mechanism does it for all traffic and without any changes to the guest OS, which leads to very different technical challenges. Amazon and Microsoft also install ACLs outside the server, but at the network interface card (NIC) as opposed to the ToR [62, 21]. In contrast, our mechanism does not require proprietary hardware, but only software changes at the ToR.

The challenge we face is that a typical ToR lacks the amount of data-path memory necessary to store any significant number of ACLs. This is a fundamental limitation related to the ASIC manufacturing process, which is unlikely to disappear in the near future [41]. To address this challenge, we propose a ToR architecture that exports the abstraction of a *virtual flow table*, which fits orders of magnitude more ACLs than the ToR's data-path memory (the physical flow table). We provide this abstraction through a simple, two-level memory hierarchy, where the ToR's (fast but small) data-path memory acts as a cache for a much larger and slower backing store accessible from the ToR's supervisor engine. However, naïve caching in the presence of ACLs with wildcards leads to incorrect forwarding behavior. Hence, this thesis focuses on caching algorithms that support ACLs with wildcards while maintaining correct forwarding behavior.

1.2 Scheduling Shared Network Resources

Today, cloud providers do not typically provide their tenants with bandwidth or latency guarantees for their intra-cloud communications. For instance, today, a cloud tenant can request and obtain a certain number of virtual machines (VMs) or containers with certain processing/memory/storage resources; the same tenant, however, cannot request that the network paths between these compute nodes have certain bandwidth and/or latency properties.

The challenge we face is that cloud providers not only need to schedule many different resource types, but that these types keep evolving, e.g., cloud providers are now starting to offer processing on FPGAs, GPGPUs, scrubbing boxes, etc. Hence, we need a scheduling solution that seamlessly and flexibly integrates not only network resources, but also new resource types. To address this challenge, we propose a two-tiered scheduling architecture

that consists of two layers: (i) *resource-agnostic scheduling*, which makes scheduling decisions without having any semantic information about specific resource types; (ii) *resource-specific enforcement*, which consists of multiple *infrastructure resource managers*, each one handling a specific resource type, and which implements the scheduling decisions made by the resource-agnostic scheduler. This thesis focuses on the network infrastructure resource manager, which communicates to the resource-agnostic scheduler the availability of network resources and implements the scheduler's decisions that pertain to these resources.

1.3 Contributions

After presenting related work in Chapter 2, this thesis makes two contributions:

1. In Chapter 3, we present a switch architecture that exposes the abstraction of a *virtual flow table*, which fits orders of magnitude more rules than the switch's physical flow table. We provide this abstraction through a simple, two-level memory hierarchy, where the switch's (fast but small) data-path memory acts as a cache for a much larger and slower backing store accessible from the ToR's supervisor engine. We focus on the particular challenge of caching rules with wildcards, because naïve caching of such rules leads to incorrect forwarding behavior. We formulate the problem of choosing which wildcard rules to cache in the data-path and show that it is NP-hard. Then we propose a simple, practical caching algorithm, called *Most Heavily Used*, which favors the caching of rules that recently matched the most amount of traffic. We show—through extensive simulations—that, given realistic data-path memory sizes and realistic data-center traffic, our algorithm results in 5% miss rate (i.e., 95% of the traffic stays in the data-path).

2. In Chapter 4, we present a two-tiered architecture for scheduling cloud resources, consisting of a *resource-agnostic scheduling* layer, which makes scheduling decisions without any semantic information about specific resource types, and a *resource-specific enforcement* layer, which enforces the scheduler's decisions. We focus on the *network infrastructure resource manager*, which enforces scheduling decisions that pertain to network resources. We show—through extensive experiments on an educational cloud platform—that a cloud provider can use our mechanism to (a) provide bandwidth and latency guarantees to cloud tenants sharing the same network links; and (b) implement useful, general network-management policies, like oversubscription and fair network sharing.

We present our conclusions in Chapter 5.

2 Related Work

In this chapter, we present the fundamental background concepts on two research fields that have motivated our work, as well as notable related advancements and research proposals. In particular, section §2.1 illustrates current approaches on flow-rule forwarding in Software-Defined Networking, while section §2.2 exhibits the state-of-the-art practices on scheduling cloud resources.

These research fields have been a major focus of our work on adding flexibility in multi-tenant networks, as we will thoroughly show in chapters 3 and 4.

2.1 Flow-Rule Forwarding in Software-Defined Networking

Software-Defined Networking (SDN) [27] has recently enabled the dynamic management of a network by decoupling the control plane from the forwarding plane. Protocols such as OpenFlow [35] have popularized the use of flow-rule forwarding in Internet routers.

In this section, we highlight the related work that has been conducted on the following SDN-affiliated fields:

1. **Packet Classification** (section §2.1.1), algorithms matching ingress packets arriving at a switch with internally stored flow rules.
2. **Cache Management for Flow Rules** (section §2.1.2), the challenge of storing and maintaining the entire traffic flow rule-set, given the slower processing speed in the control-plane and the limited memory in the data-plane.

2.1.1 Packet Classification

Packet classification is the procedure of matching ingress packets at an Internet router with a flow rule. In this section, we present the fundamental background and related work in hardware- and software-based classification algorithms.

Hardware-based Classification

Packet classification within the hardware is typically conducted by storing flow rules within Ternary Content-Addressable Memory (TCAM), capable of conducting packet classification at Gigabit rate [69], as the lookup time is executed in amortized constant time (e.g., 1.25 ns [4]). Although packet classification within a TCAM offers significant advantages, fully replacing software-based classification with TCAM-based solutions may be cost-prohibitive, as a typical TCAM chip may cost 4 – 5 times more than a SRAM chip [34].

In this thesis, the terms “TCAM” and “hardware memory” will be used interchangeably.

Software-based Classification

Software-based packet classification, which has been extensively studied in the literature [24], is an elaborate problem, as two competing bottlenecks have to be simultaneously addressed: (i) worst-case time complexity, i.e., the required time to match an ingress packet to the appropriate rule; and (ii) worst-case storage complexity, i.e., the required memory to store all flow rules within the memory. The simplest packet classification algorithm is to conduct a linear search through the entire flow rule-set, a process which would only be feasible for a relatively small rule-set, as the lookup time complexity grows linearly with the number of installed traffic rules.

To address the aforementioned challenges in packet classification with bigger rule-sets, a plethora of *partitioning heuristics* have been proposed, such as HiCuts [25, 26], HyperCuts [54] and EffiCuts [60]. The core mechanism of this class of algorithms is the division of the entire flow rule space into smaller partitions, in order to facilitate quicker rule lookups.

2.1.2 Cache Management for Flow Rules

Cache management for flow rules is the problem of choosing an appropriate subset of the entire traffic rule-set to be cached in the limited hardware memory within the edge switches. In this section, we present the related work that has been conducted by the academic community on this field and highlight the potential challenges.

DevoFlow [10] exposes the scalability limitations of the OpenFlow flow setup rate. The authors propose utilizing wildcard rules at the edge switches to handle all ingress “microflows” by default and invoke the controller only when “elephant” flows are detected. Although this approach limits the number of specific, exact-match rules installed in egress switches, this work assumes enough available memory at the edge switches to install the necessary wildcard rules which will handle the ingress traffic, while preserving the network’s forwarding policy semantics.

vCRIB [42] and DIFANE [70] both partition the hyperspace of the traffic rules, respectively repli-

cating or partitioning rules as necessary. Dependencies between wildcard rules are resolved by treating the generated partitions as the atomic units, as they are proactively distributed across multiple devices in their entirety. In vCRIB, only ingress packets matching the installed partitions at a given edge switch are promptly handled, while the rest are redirected to other devices. In DIFANE, the first ingress packets of a network flow arriving at an edge switch are always redirected to the authoritative switch responsible for the partition containing the network flow, until the respective rules are cached at the edge switch. Nevertheless, both papers assume that there are enough switches available with adequate aggregate memory to store the entire rule-set.

CAB [67] also divides the rule-set hyperspace into smaller partitions, replicating rules as necessary, and initially keeping all generated partitions within the controller. When a new ingress packet arrives at an edge switch, the entire corresponding partition is installed, including the appropriate rule matching the ingress packet. Similarly, when a traffic rule expires, the entire corresponding partitions are removed. Since partitions are treated as atomic units, any dependency issues between rules are resolved by caching at the edge switch all wildcard rules within a given partition. Nevertheless, this approach is likely to bring in the TCAM mutually independent rule subsets, potentially polluting the limited hardware memory with flow rules that will apply to little or no traffic at all.

CacheFlow [29] offloads the entire rule-set on the guest OS within the switch, while updating periodically the limited switch TCAM with the subset of the "heaviest" rules, including high-priority dependent rules. To avoid polluting the hardware with numerous low-weight, high-priority rules that would handle little traffic, CacheFlow abstracts long chains of such dependent rules by introducing "cover" rules, redirecting microflows to the switch guest OS. Nonetheless, this approach assumes that it is feasible to truncate long dependency chains, replacing them with a few rules; this hypothesis may not be scalable in bigger data-centers or Internet Service Providers, where complex dependencies within a bigger traffic rule-set may arise.

2.1.3 Our Approach in Facilitating Caching Overlapping Rules

In this dissertation, we address the challenge of caching wildcard rules from a different perspective. In particular:

1. We do not assume that we have adequate aggregate hardware memory available within the data-plane switches to store the necessary traffic rules (DevoFlow, vCRIB, DIFANE). Instead, we concentrate on picking a rule subset to be cached in the switch TCAM.

2. We do not insert unnecessary rules into the limited hardware memory. This sets our work apart from CAB, where entire coarse-grained partitions are being inserted in the TCAM. Instead, we conduct an analysis to detect dependencies between wildcard flows, which are subsequently taken into consideration to maintain correct forwarding semantics. In

particular, when a rule R is inserted in the TCAM, we also insert all higher-priority rules that overlap with R; conversely, when a rule R is evicted from the TCAM, we also evict all lower-priority rules that overlap with R.

3. We do not make any assumptions on the dependency patterns between traffic rules (CacheFlow). Instead, we are utilizing software-based classification algorithms (§2.1.1) to break rule dependencies by partitioning the rule hyperspace.

We thoroughly present our work on this problem in chapter 3.

2.2 Scheduling Cloud Resources

Cloud Computing, the universal access to a shared pool of configurable computing resources [36], has been experiencing sharp development over the last years, leading to an increased demand for application migration to the cloud [50].

In this section we highlight the related work that has been conducted on the following fields affiliated with Cloud Computing:

1. **Bandwidth Provisioning** (section §2.2.1), offering minimum bandwidth guarantees to end-users, in addition to hardware-related performance guarantees.
2. **Resource Oversubscription** (section §2.2.2), the policy of allocating resources to end-users beyond the nominal capacity of the cloud infrastructure, in order to achieve higher resource utilization.
3. **Enforcing Bandwidth Fairness** (section §2.2.3), which ensures proper and "fair" bandwidth allocation across different tenants.

2.2.1 Bandwidth Provisioning

Recent growth in Cloud Computing has led cloud providers to offer increased resource and performance guarantees to potential tenants, such as CPUs, Virtual Machines or storage capacity, in an effort to attract more customers and earn their confidence. Nevertheless, cloud providers still fail to adequately provision network guarantees, such as assured bandwidth or tail latency thresholds. Notable state-of-the-art examples include:

- Amazon Web Services (AWS) provide "Enhanced Networking" [1, 2]. It does not directly address the tenants' needs, as tenants have to specify placement groups [3], thus indicating more constraints than the ones they really need. In a sense, tenants provide *themselves* the solution to the scheduler, whereas the scheduler should determine it on its own. Furthermore, there is no concrete quantification of the expected network

performance; instead, CPU performance is used to provide the end-user a performance estimation.

- Google Cloud Platform lists only the egress throughput caps that the tenant will experience [18], without providing any minimum guarantees during scheduling.
- Microsoft Azure offers a "Virtual Network" [40] to inter-connect scheduled resources, as well as bandwidth-optimization support [38, 39]. In spite of this, network provisioning is not taken into consideration during resource scheduling, but instead afterwards.
- The OpenStack "kilo" scheduler [46] may accept an `availability_zone` parameter, exposed to the tenant, so that the end-user may schedule their resources to a faster cluster. Notwithstanding the availability zones, users are essentially required to provide a hint to the scheduler, instead of simply stating their networking demands.

The absence of network resources or constraints during scheduling could seriously hinder the predictability of the infrastructure's performance, thus also the performance of tenant applications. For instance, concurrent batch applications, e.g., MapReduce applications, deployed by multiple tenants could introduce network congestion resulting in mutual underperformance. Furthermore, in such an environment, it would be infeasible to offer any latency guarantees to tenants deploying latency-sensitive applications, e.g., Memcached.

To address the aforementioned challenges, significant research has been conducted over the past years in an effort to provide minimum bandwidth guarantees and respect latency constraints.

OpenStack Neutron

OpenStack Neutron [47] provides "networking-as-a-service" to end-users, by presenting a networking management framework. Examples of services provided by Neutron include: (i) custom network topology or VLAN specification; (ii) floating IP address allocation, allowing dynamic resource reallocation across the cloud infrastructure in case of system failure; (iii) Quality of Service (QoS) functionalities aimed to fulfill Service-Level-Agreements (SLA) with end-users, such as egress bandwidth rate limiters; and (iv) exposing advanced features "as-a-service," such as firewalls or load balancers. Furthermore, Neutron offers a wide range of plugins to facilitate its deployment by seamlessly integrating in real and virtual switches, such as Open vSwitch [16] and Cisco Nexus 1000V [8].

Academic Research

The authors in [71] show that the decision of satisfying a heterogeneous bandwidth request in the cloud is a NP-complete problem and propose a Virtual Machine (VM) allocation heuristic. GARA [15] exposes both hardware and network resources to the scheduler as abstract resources. GateKeeper [51] proposes an admission control mechanism that provides minimum

bandwidth guarantees to tenants, albeit without enforcing bandwidth allocation on "bandwidth abusive" applications. SecondNet [23] implements a scalable bandwidth allocation algorithm by representing the scheduling problem as a weighted bipartite graph and executing min-cost network flow matching.

Proteus [66] provides predictable performance and cost guarantees by profiling the tenant applications and letting tenants choose their bandwidth caps based on the proposed cost models. Cicada [33] measures the application's traffic, provides feedback to the tenant and may accordingly update the resource placement. The authors in [17] approach the problem from the perspective of VM migration and propose a sequence planning heuristic that respects application bandwidth demands. CloudMirror [32] abstracts bandwidth reservation requests by allowing tenants to specify bandwidth requirements between different application components.

2.2.2 Resource Oversubscription

Resource oversubscription is the practice of reserving more resources on a cloud platform than nominally available, on the grounds that tenants are not expected to continuously utilize their requested resources to their full extent. Underutilized resources tend to incur higher maintenance costs within a given time interval, therefore cloud providers share a natural incentive to increase resource utilization, potentially beyond their nominal capacities.

Sponge [68] proposes a CPU oversubscription mechanism for Virtual Machines (VMs) with respect to minimizing the potential impact on application performance. The authors in [56] demonstrate that reserving more resources than nominally available does not necessarily violate the Service-Level-Agreement (SLA) with the end-users, while the authors in [59] present a thorough analysis on over-provisioning techniques of CPU and network resources.

2.2.3 Enforcing Bandwidth Fairness

Cloud providers, in an effort to achieve high network link utilization and to earn the end-users' confidence, are modeling TCP-like behavior in multi-tenant networks during bandwidth allocation, either at a virtual machine level or tenant level. Therefore, enforcing fairness has been extensively studied in the bibliography.

Seawall [53] proposes a hypervisor-deployed mechanism that imposes per-source fair bandwidth allocation. EyeQ [28] abstracts the data-center as an extensive switch, applying bandwidth sharing policies at an end-to-end level. FairCloud [48] enforces proportional sharing on congested links across different tenants, so that tenants may not take advantage of specific communication patterns. Spiderweb [30] redistributes bandwidth between tenants according to their demands and payments. ElasticSwitch [49] provides minimum bandwidth guarantees and dynamically adjusts VM-to-VM bandwidth throttling based on the congestion level of the network.

2.2.4 Our Approach in Facilitating Cloud Resource Scheduling

In this dissertation, we propose a *flexible cloud platform* to facilitate the implementation and the deployment of: (i) bandwidth provisioning; (ii) resource oversubscription; and (iii) enforcing bandwidth fairness. We have designed our platform based on two fundamental principles:

1. **Flexibility:** cloud administrators should be able to seamlessly integrate new resources and network-management mechanisms in our platform.
2. **Abstraction:** our architecture should conceal the low-level implementation details from the scheduler and the tenants. On the other hand, any features interacting with tenants should be exposed in a user-friendly way, oriented around the tenant demands.

We thoroughly present our approach in chapter 4.

3 Managing Shared Flow Tables

In this chapter, we present our solution for managing shared flow tables in multi-tenant environments. As stated in the introduction, we propose installing access control rules in packet switches—whereas today they are typically installed in servers, which is incompatible with bare metal support and yields unnecessary performance overhead. The place to store rules in packet switches is the data-path memory, which, however, is typically too small for storing all the rules necessary in a multi-tenant environment. We address this by using data-path memory as a cache for a larger, but slow state store. The technical challenge we focus on is that caching rules that include wildcards leads to incorrect forwarding behavior, if done naïvely. We show that the problem of identifying the best rules to cache is NP-hard and propose a simple, implementable algorithm to solve it. We evaluate our algorithm through extensive simulations and real traffic traces. All formal definitions and notations used in this chapter are summarized in appendix A.

3.1 Virtual Flow Tables

3.1.1 Abstraction and Implementation

Consider a Top-of-Rack (ToR) switch with a physical flow table that fits T access rules; we say that the switch exposes (to a network operator or SDN controller) the abstraction of a *virtual flow table* (VFT) if, from the network operator/controller's point of view, the switch can store $N \gg T$ access rules. The top picture in Figure 3.1 illustrates the abstraction.

We implement the VFT abstraction with a simple, two-layer memory hierarchy:

1. A *software table*, located outside the switch's data-path and accessible only by the switch's local control plane (typically a set of processes running on a CPU located in the same chassis as the data-path), which fits N access rules.
2. A *hardware table*, accessible by the switch's data-path (typically a switching ASIC), which fits $T \ll N$ access rules. This is the same as the switch's physical flow table; we use the

term “hardware” instead of “physical” to differentiate from the software table.

The software table is the authoritative table that stores all the access rules installed on the switch by the network operator or SDN controller. The hardware table acts as a cache for the software table, and we refer to access rules that are stored in the hardware table as “cached rules.” The bottom picture in Figure 3.1 illustrates our two-layer memory hierarchy.

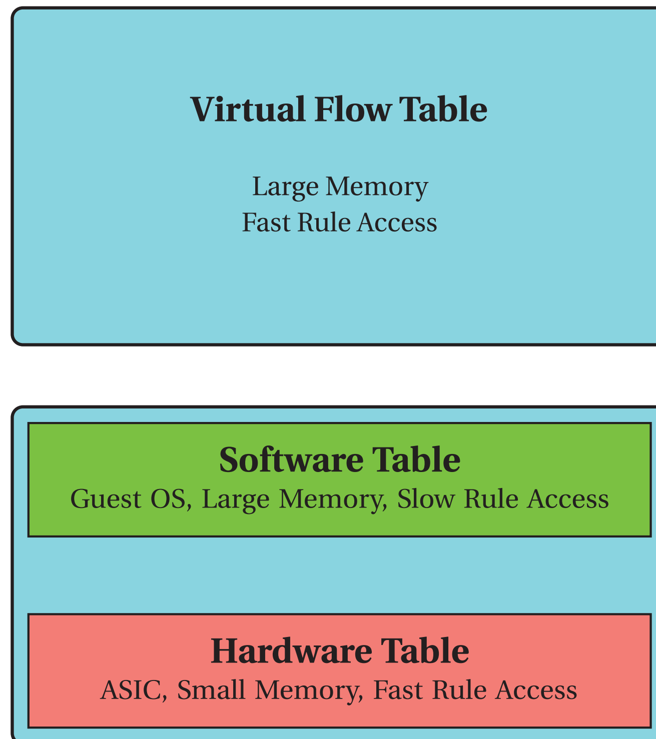


Figure 3.1 – Virtual Flow Table abstraction (top) and underlying memory hierarchy (bottom)

Caching is managed by the *cache manager*: a process running on the switch’s local control plane, which periodically polls the hardware table, computes a weight for each rule in that table that reflects the rule’s recent popularity, and replaces lighter rules with heavier rules.

The switch’s data-path handles all the traffic that can be served from the hardware table and passes the rest to the switch’s local control plane. More specifically: When a new packet arrives at the data-path, a lookup is performed in the hardware table. If the lookup returns an action other than “forward to the local control plane,” the packet stays in the data-path until it is dropped or forwarded. Otherwise, the packet is tagged with ingress port number and passed to the switch’s local control plane, where a lookup is performed in the software table. Since the software flow table is authoritative, every lookup returns either a “drop” or a “forward” action. If the latter, the packet is tagged with the proper egress port number and passed back to the switch’s data-path, which removes all tags and forwards the packet through the specified egress port. Figure 3.2 illustrates this interaction between data-path and local control plane.

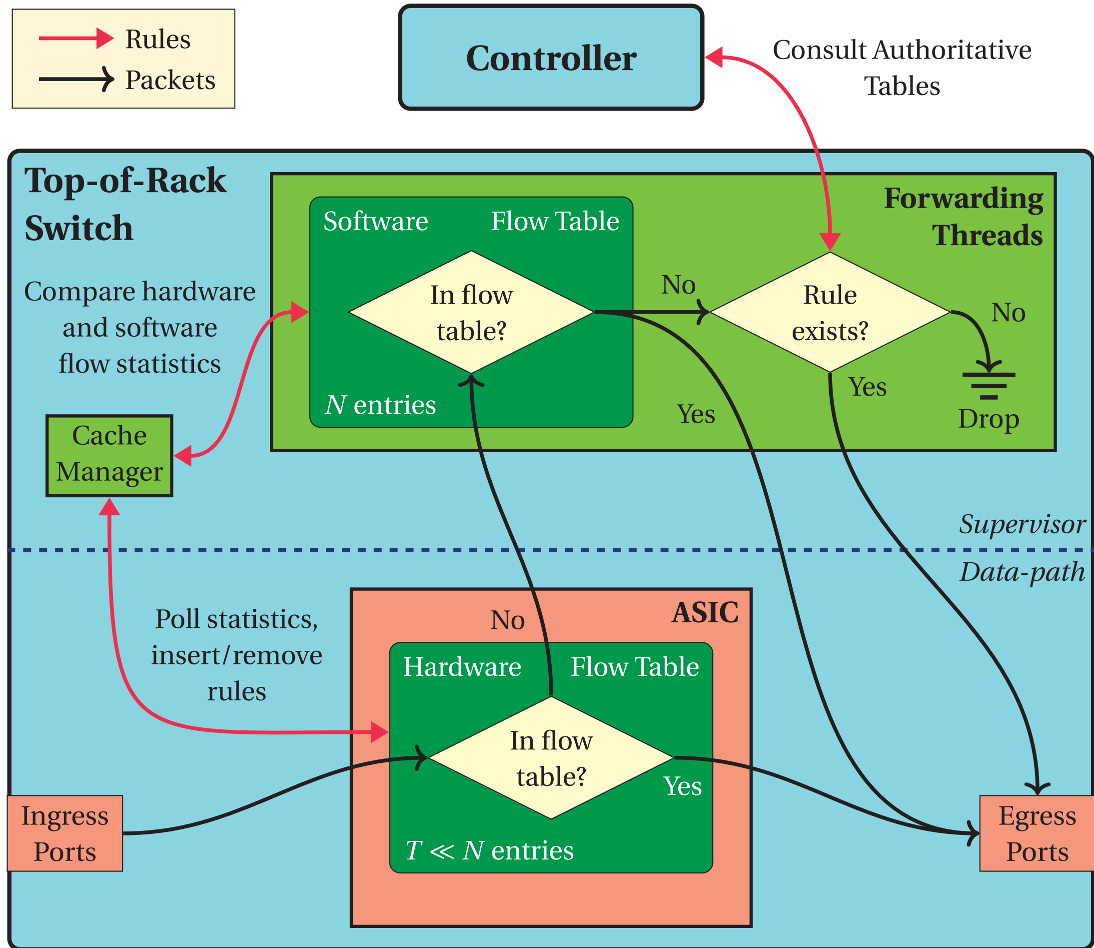


Figure 3.2 – System architecture

We have implemented a prototype that exports the VFT abstraction on a switch consisting of a Broadcom Trident+ switching ASIC coupled with a NetLogic XLP MIPS processor [14]. This dissertation, however, does not focus on the prototype implementation, but rather on the design of the VFT abstraction and, in particular, the challenge of handling overlapping rules, which is described next. The numbers we present in the rest of this chapter were obtained from a C++-based simulator, where both the hardware and software flow tables are implemented as Standard Template Library unordered (hash) tables.

3.1.2 The Challenge of Overlapping Rules

The presence of overlapping rules introduces profound challenges in managing cached rules with respect to maintaining correct forwarding behavior.

To demonstrate our argument, we start with a simple example of overlapping rules, which allow all traffic from `128.178.15.0/24` to `10.42.0.0/16` except ICMP traffic, which is dropped.

```
SrcIP: 128.178.15.0/24, DstIP: 10.42.0.0/16, Proto: any,  Action: allow
SrcIP: 128.178.15.0/24, DstIP: 10.42.0.0/16, Proto: icmp, Action: deny
```

These rules overlap in the sense that an ingress packet P may match both of them; this is why a *priority* is specified for each rule, indicating which rule should be preferred in case of multiple matches.

Throughout this thesis, we assume that two overlapping rules never share the same priority. Typically, more general rules are assigned lower priorities than a more specific rule. In the above example, the first rule (R_1), which allows all traffic between the specified prefixes, would be allocated a low priority than the second rule (R_2), which blocks all ICMP traffic between the same specified prefixes.

We say that "rule R_1 is lower-overlapping relative to R_2," or "R_2 is higher-overlapping relative to R_1," and we write $R_1 < R_2$, when R_1 and R_2 overlap and R_1 has lower priority than R_2. In this case, when a packet matches both rules, only the actions of R_2 are triggered.

Overlapping rules pose a challenge in our context, because careless caching of such rules leads to incorrect forwarding behavior. In the above example, if the first rule was cached in the hardware table but the second rule was not, an ICMP packet matching both rules would be allowed (whereas the correct behavior would have been to block it). Hence, if we insert the first rule in the cache, we must also insert the second one. Conversely, if we evict the second rule from the cache, we must also evict the second one.

In general, to maintain correct forwarding behavior, we must ensure that, for each cached rule R, there is no higher-overlapping rule that is not cached. This is equivalent to the following principles:

- When we insert a rule R in the hardware table, we must also insert all higher-overlapping rules $R_h > R$.
- When we evict a rule R from the hardware table, we must also evict all lower-overlapping rules $R_l < R$.

In case of "chain overlaps," we must follow these principles recursively. For example, consider the following 2-tuple rules:

```
SrcIP: 16.64.0.0/16,  DstIP: 32.0.0.0/8,    priority: high
SrcIP: 16.0.0.0/8,    DstIP: 32.128.0.0/16, priority: medium
SrcIP: 16.192.0.0/16, DstIP: 32.0.0.0/8,    priority: low
```

While $R_1 > R_2$ and $R_2 > R_3$, R_1 and R_3 do not overlap. Nevertheless, if we insert R_3 in the hardware table, we must also insert R_2 (so that packet `(16.192.1.1, 32.128.1.1)` triggers R_2, not R_3), and, recursively, we must also insert R_1 (so that packet `(16.64.1.1, 32.128.1.1)` triggers R_1, not R_2).

Problem Statement

Consider a set of N rules $\mathscr{R}_N = \{R_1, R_2, \ldots, R_N\}$ and a hardware table of size $T \leq N$. We assign to each rule R_i a benefit metric $b_i \in \mathbb{R}_{\geq 0}$. We want to choose a subset of the rules $\mathscr{R}_T \subseteq \mathscr{R}_N, s.t. |\mathscr{R}_T| \leq T$ to cache in the hardware table, such that we maximize the aggregate benefit of the cached rules and maintain correct forwarding behavior.

To formally state the problem, we represent the set of rules $\mathscr{R}_N$ as a directed acyclic graph $G = (V, A)$, where:

- For each rule R_i there is a vertex $u_i \in V$, and there are no other vertices in V.
- For each $R_i < R_j$ there is an arc $(i, j) \in A$, and there are no other arcs in A.

We must solve the following optimization problem:

$$\begin{aligned} \text{maximize:} \quad & \sum_{i=1}^{N} b_i x_i \\ \text{subject to:} \quad & \sum_{i=1}^{N} x_i \leq T \\ \text{and} \quad & x_i \in \{0, 1\} \\ \text{and} \quad & x_i \leq x_j \quad \forall (i, j) \in A \end{aligned}$$

In other words, we must choose at most T vertices from G, such that we maximize their aggregate benefit, on the condition that: if we choose a vertex u_i, we must also choose any vertex u_j that is reachable from u_i.

Complexity

We now show that our optimization problem is NP-hard, by reducing the NP-hard 0-1 knapsack problem [9] to an instance of our optimization problem.

We consider $M \in \mathbb{N}^+$ items. Each item y_i has weight $w_i \in \mathbb{N}^+$ and benefit value $b_i \in \mathbb{R}^+$. The 0-1 knapsack problem is the following optimization problem:

$$\begin{aligned} \text{maximize:} \quad & \sum_{i=1}^{M} b_i x_i \\ \text{subject to:} \quad & \sum_{i=1}^{M} w_i x_i \leq T \\ \text{and} \quad & x_i \in \{0, 1\} \end{aligned}$$

Furthermore, consider an instance of the original overlapping rules problem, with the follow-

ing constraints for a directed acyclic graph $G = (V, A)$:

C1. $b_j = 0 \; \forall u_j \in V : \exists(i, j) \in A$. Every vertex with an incoming arc has a zero benefit value. Vertices with *no* incoming arcs will be hereinafter referred to as “root vertices.”

C2. No vertex u_k exists that can be reached from two different vertices (u_i, u_j), $i \neq j$, with $b_i > 0$ and $b_j > 0$, i.e., from two different root vertices.

We may reduce the 0-1 knapsack problem to the aforementioned instance of our original problem as follows:

1. Each item y_i, $i = 1, 2, \ldots, M$ is mapped to a root vertex $u_j \in V$ with the same positive benefit value $b_i = b_j > 0$.

2. The weight w_i of item y_i is mapped to the $w_i - 1$ vertices $u_k \in V$ that are reachable from the root vertex u_j.

We may intuitively view the reduction as follows. Each item y_i of the knapsack problem with a benefit value $b_i > 0$ and weight $w_i > 0$ is mapped to a rule R_i so that: (i) rule R_i bears the same benefit value b_i; (ii) there are no lower-overlapping rules with respect to R_i; and (iii) if and only if R_i is inserted in the hardware flow table, this will trigger the insertion of $w_i - 1$ higher-overlapping rules with respect to R_i, bearing zero-benefit values.

Furthermore, there are no inherent dependencies within the knapsack problem parameters, due to the aforementioned constraints imposed on graph G. In particular:

- Constraint (C1) guarantees that no root vertex is reachable from another root vertex. This assures that the inclusion of a knapsack item y_i will not necessarily enforce the inclusion of a different item y_j, $j \neq i$.

- Constraint (C2) guarantees that no vertex will be “double-counted,” because two different root vertices have been included. This assures that the weights w of different knapsack items are independent of each other.

Conclusion

To summarize, overlapping rules pose a challenge in our context, because: (i) they introduce constraints as to which subset of rules we can insert in or evict from the hardware table in order to maintain correct forwarding behavior; and (ii) choosing the optimal subset of rules to cache in the hardware table is an NP-hard problem. In section §3.2, we present two heuristics that address this challenge.

3.2 Cache Management for Overlapping Rules

3.2.1 The Software Table

The software table provides the same lookup interface as the hardware table: lookup by multiple keys and multiple key ranges, e.g., "{IP source address 128.178.50.0/24, port number 0–1024}". However, unlike the hardware table, which is typically stored in ternary CAM (TCAM), hence benefits from hardware support that performs such lookup in $\mathcal{O}(1)$, the software table is stored in plain DRAM. Hence, special care must be taken to ensure good lookup performance.

We organize and search the software table as proposed in HiCuts [24], which we briefly introduced in §2.1.1: First, we partition the flow rule space. Second, we create the *packet classifier* structure, a directed acyclic graph where each leaf node corresponds to a generated partition. When a packet P is looked up, we retrieve the corresponding leaf node of the classifier, where a linear search is subsequently run to select the highest-priority flow rule that matches with P. At the same time, the software table embeds information that enables us to quickly detect rule overlaps. We provide further details in the following sub-sections.

To simplify description, we assume proactive rule management, where the software table is pre-populated with all the rules needed for correct forwarding behavior. Adapting our system to reactive rule management (where the software table is populated at run time by an SDN controller) would be straightforward.

We also assume that, given any rule R in the software table and the set of its higher-overlapping rules $\mathcal{R}_H$, the set $\{R\} \cup \mathcal{R}_H$ fits in the hardware table. If this is not the case, then the hardware table is too small to provide correct forwarding behavior.

Partitioning the Flow Rule Hyperspace

We recursively apply the HiCut algorithm [24] on the entire d-dimensional rule hyperspace $\mathbb{U}^d$, containing a set of M rules $\mathcal{R}_M = \{R_1, R_2, \ldots, R_M\}$, in order to limit the dependencies between overlapping rules. We call $\mathcal{R}_M$ the *original rule set.* Algorithm 3.1 illustrates the process in detail, where we provide:

1. Two arguments:
   (a) a d-dimensional hyperspace $\mathbb{S}^d \subseteq \mathbb{U}^d$.
   (b) a set $\mathcal{R}_S$ of $n \leq M$ rules $\{R_1, R_2, \ldots, R_n\}$, contained in $\mathbb{S}^d$.

   On the initial call of the algorithm we provide $\mathbb{S}^d = \mathbb{U}^d$ and $\mathcal{R}_S = \mathcal{R}_M$.

2. Two configurable parameters:
   (a) `binth`, the maximum rules that may be contained in a single partition. If a partition has at most `binth` rules, the algorithm stops.

(b) `spfac`, the "space factor," indicating how much memory should be allocated during partitioning.

The algorithm may be summarized as follows: First, we decide the dimension F_c of $\mathbb{S}^d$ that we should partition; we call F_c the "cut dimension." Second, we compute the number of partitions that should be generated. We subsequently distribute the rules in $\mathscr{R}_S$ to the respective partitions they *conflict* with; we say that a rule R conflicts with a partition covering a subspace $\mathbb{S}_P^d \subseteq \mathbb{S}^d$ if $R \in \mathbb{S}_P^d$. Rules that conflict with multiple partitions are promptly replicated. Finally, we apply recursively the partitioning algorithm on each generated partition.

If a partition is found to contain at most `binth` rules, it is no further partitioned. We call this partition a *leaf partition* or *leaf node* and store the rule set $\mathscr{R}_S$ in its corresponding data structure. We provide details on the data structure in subsequent paragraphs.

To find the "cut dimension" F_c, we utilize the following heuristic: First, we simulate a series of partitions across every dimension F_i of $\mathbb{S}^d$. We stop partitioning a dimension F_i when we have allocated at least `spfac` $\cdot\, n$ memory, in terms of the number of partitions and rules that would be generated. Second, for each dimension F_i we find the partition conflicting with the most rules; we define this number of conflicts as the "local maximum" of F_i. Finally, we choose the dimension F_c with the smallest "local maximum."

Since rules covering multiple partitions are replicated, the partitioning algorithm generates a set of $N \geq M$ rules $\mathscr{R}_N \supseteq \mathscr{R}_M$, which is defined as the union of the rule sets that stored in each leaf partition. The set $\mathscr{R}_N$ is called the *partitioned* rule set, which is the rule set that we store in the software table.

Creating the Packet Classifier

The *packet classifier* is a data structure which we create during the partitioning of the entire flow rule hyperspace $\mathbb{U}^d$, as we previously described. We represent the classifier as a directed acyclic graph (DAG) $G = (V, A)$, which is recursively constructed as follows:

1. A vertex $u_0 \in V$ is added to represent the entire flow rule hyperspace $\mathbb{U}^d$. Vertex u_0 is the "root" of G.
2. An instance of the partitioning algorithm is run on vertex $u = u_0$.
3. Each time the partitioning algorithm generates a new partition p, we add a new vertex $v \in V$ to represent p and a new arc $(u, v) \in A$. Let $V_P \subset V$ denote the set of all vertices v added during this step, in this instance of the partitioning algorithm.
4. If $V_P = \emptyset$, the algorithm stops. Otherwise, step (2) is recursively called $\forall u \in V_P$.

We utilize the aforementioned process to create the linked data structure, where each *node* of

Algorithm 3.1: Recursive partitioning of the flow rule hyperspace

Data: d-dimensional hyperspace $\mathbb{S}^d$
Data: Set of rules $\mathcal{R}_S$ contained in $\mathbb{S}^d$
Data: binth: maximum number of rules to store in a partition
Data: spfac: space factor
Result: Recursively partition $\mathbb{S}^d$ until each leaf node has at most binth rules

```
1  Function Partition(S^d, R_S):
2      n ← |R_S|
3      if n ≤ binth then                                              /* this is a leaf node */
4          store R_S in this partition
5          return
6      forall dimensions of S^d do                                    /* simulated partitioning */
7          nump ← min(4, √n)                                          /* initial number of partitions */
8          range ← interval [a, b) of current dimension
9          done ← false
10         while not done do
11             cut range in nump equal intervals
12             memory ← nump                                          /* memory overhead */
13             forall generated partitions do
14                 conflicts ← number of rules in R_S that conflict with this partition
15                 memory ← memory + conflicts
16             if memory < spmf(n) then nump ← 2·nump                 /* continue partitioning */
17             else done ← true
18         local_max ← max{conflicts}
19     dim ← dimension with the min{local_max}                        /* cut dimension */
20     partition dim, committing the respective simulation, generating nump partitions
21     forall generated partitions do
22         S_P^d ← sub-space of this partition, with "narrower" dim   /* S_P^d ⊂ S^d */
23         R_P ← replicate all rules {R ∈ S^d | R ∈ S_P^d}            /* conflicting rule set */
24         Partition(S_P^d, R_P)                                      /* recursive partitioning */
25     return
26 Function spmf(n) : int:
27     return spfac · n                                               /* space measure function */
```

the classifier corresponds to a vertex of the generated DAG. We present the definition of the node class in listing B.1, which stores the following information:

1. `space_box`, representing the d-dimensional hyperspace $\mathbb{S}^d$ covered by the node. Each dimension is represented by a left-closed, right-open interval $[a, b)$.
2. `cut_dimension`, representing the dimension along which the non-leaf node has been partitioned based on the HiCut algorithm. It is set to `KEY_FIELD_NONE` in leaf nodes.
3. `children_map`, a map holding the pointers to the child nodes, if any. The key k of the map corresponds to the left bound of that child's interval $[a_c, b_c)$, across dimension `cut_dimension`; therefore, $k = a_c$.
4. `entries`, a vector maintaining pointers to all flow rules contained in that particular node's hyperspace. This container may be optionally left empty in non-leaf nodes to preserve memory.

We illustrate a detailed example of our data structure in appendix B.

Detecting Direct and Indirect Rule Overlaps

After the packet classifier structure has been created, we detect all rule overlaps within each leaf node of the classifier. As stated above, each leaf node stores:

- The subspace $\mathbb{S}^d$ of the rule set it covers. Since leaf nodes represent the final partitioning of the entire rule set hyperspace $\mathbb{U}^d \supseteq \mathbb{S}^d$, the subspace intersection of any two leaf nodes is always the empty set.
- A vector with pointers[1] to the flow rules contained in the leaf node's hyperspace $\mathbb{S}^d$. Flow rules covering multiple partitions have been already replicated across the respective leaf nodes.

As a result, each leaf node maintains the sole ownership of its contained flow rules and no cross-leaf node dependencies exist between rules. Therefore, we may execute our pre-processing algorithms to detect rule dependencies independently on each leaf node, containing at most `binth` $\ll N$ rules.

We subsequently detect the potential dependencies between rules independently within each leaf node. Each flow rule R_i holds two vectors, called `high` and `low`, which are initially empty. We will populate `high` with pointers[2] to all other flows R_j that are directly or indirectly higher-overlapping[3] with R_i. In a similar way, we will populate `low` with pointers to all lower-priority directly or indirectly overlapping flows.

[1] C++14 `std::shared_ptr` to indicate "permanent" ownership.
[2] C++14 `std::weak_ptr` to indicate "temporal" ownership, acquired only when necessary.
[3] Refer to appendix A.5 for the formal definitions of "directly overlapping" and "indirectly overlapping."

Direct Overlaps We initially detect the direct overlaps with a simple algorithm (Algorithm 3.2), running[4] in $\mathcal{O}\left((\texttt{binth})^2 d\right)$. We consider every pair of flow rules and intersect their respective hyperspaces to determine whether they overlap or not.

Algorithm 3.2: Detect directly overlapping rules within a given leaf node

Data: vector $\mathsf{rule}[n]$ of n rules
Result: vector $\mathsf{high}[i]$ with directly higher-overlapping rules relative to $\mathsf{rule}[i]$, $\forall i \in [1, n]$
Result: vector $\mathsf{low}[i]$ with directly lower-overlapping rules relative to $\mathsf{rule}[i]$, $\forall i \in [1, n]$

```
1  for i = 1 to n do
2  |  R_i ← rule[i]
3  |  for j = i + 1 to n do
4  |  |  R_j ← rule[j]
5  |  |  if R_1 ∩ R_2 ≠ ∅ then          /* direct overlap; see appendix A.5.1 */
6  |  |  |  if R_i > R_j then                     /* R_i has higher priority */
7  |  |  |  |  append R_i to high[j]
8  |  |  |  |  append R_j to low[i]
9  |  |  |  else                                 /* R_j has higher priority */
10 |  |  |  |  append R_i to low[j]
11 |  |  |  |  append R_j to high[i]
```

Indirect Overlaps We subsequently compute the indirectly overlapping rules relative to each rule within a given leaf node (Algorithm 3.3).

First, we create a directed acyclic graph $G = (V, A)$, where:

- For each rule R_i in the given leaf node there is a vertex $u_i \in V$ and there are no other vertices in V.
- For each $R_i < R_j$ there is an arc $(i, j) \in A$, and there are no other arcs in A.

Second, we execute the following steps for every vertex $u_i \in V$:

1. We run the Depth-first search (DFS) algorithm with u_i as the start vertex.
2. For all visited vertices $u_j \in V$, the respective rules R_j are added to vector `high` of R_i, if not already present. As a result, vector `high` will contain all directly or indirectly higher-overlapping rules with respect to R_i.

We follow a similar approach to populate the `low` vector of each rule R_i with all indirectly lower-overlapping rules relatively to R_i. To do that, we inverse the arcs of G and execute DFS

[4] Subsequent Algorithm 3.3 runs in $\mathcal{O}\left(\texttt{binth}^2\right)$, thus we did not optimize the time complexity of Algorithm 3.2.

on every vertex, as previously. As a result, the time complexity of detecting indirect overlaps within a single leaf node is $\mathcal{O}\left(\texttt{binth}^2\right)$.

Algorithm 3.3: Detect indirectly overlapping rules within a given leaf node

Data: vector $\mathsf{rule}[n]$ of n rules
Data: vectors high and low, already populated from Algorithm 3.2.
Result: append to $\mathsf{high}[i]$ all indirectly higher-overlapping rules relative to $\mathsf{rule}[i]$, $\forall i \in [1, n]$
Result: append to $\mathsf{low}[i]$ all indirectly lower-overlapping rules relative to $\mathsf{rule}[i]$, $\forall i \in [1, n]$

```
1  G(V, A) ← empty directed graph
2  for i = 1 to n do add u_i ∈ V                                /* add vertices */
3  for i = 1 to n do                                                /* add arcs */
4      forall rules R_j ∈ high[i] do add (i, j) ∈ A
5  foreach u_i ∈ V do                       /* find indirectly higher-overlapping */
6      run DFS on u_i
7      forall visited vertices u_j ∈ V do
8          add rule R_j in high[i], if not already present
9      mark all vertices as not visited     /* to not interfere with subsequent DFS */
10 invert every arc a ∈ A
11 foreach u_i ∈ V do                        /* find indirectly lower-overlapping */
12     run DFS on u_i
13     forall visited vertices u_j ∈ V do
14         add rule R_j in low[i], if not already present
15     mark all vertices as not visited
```

Packet Lookup on the Software Table

When a packet P is looked up at the software table, we first locate the corresponding leaf node of the classifier and then find the highest-priority flow rule that matches P.

Leaf node search By definition, each leaf node covers a specific partition of the entire d-dimensional flow rule hyperspace. Therefore, a packet P will be always contained in the subspace of exactly one leaf node, regardless whether a matching rule exists or not. As we have already stated, each non-leaf node stores: (i) the dimension it has been partitioned along (`cut_dimension`); and (ii) its child nodes in an ordered map (`children_map`). As a result, when a packet is looked up on a non-leaf node, a binary search[5] matches the packet header corresponding to `cut_dimension` with the child node which also contains P. This process is repeated recursively until we reach a leaf node and runs in $\mathcal{O}(d)$ [26].

[5] C++14 `std::map::upper_bound`

Flow rule search After P is matched with a leaf node, we conduct a linear search on all flow rules contained in the leaf node, stored in vector `entries`. Let set $\mathscr{R}_m$ denote all rules that match[6] with P. If $\mathscr{R}_m \neq \emptyset$, then we trigger the forwarding actions of the rule with the highest priority in $\mathscr{R}_m$. This process runs in $\mathscr{O}(\texttt{binth})$.

Otherwise, if $\mathscr{R}_m = \emptyset$, no matching rule exists and a default, policy-specific action is triggered, e.g., to drop P or to forward P to the controller. We assume in this dissertation that a matching rule always exists for every packet P that is looked up on the software table.

3.2.2 Rule Replacement

Let T and N be the number of rules that fit in the hardware and the software table, respectively, with $T \ll N$, $\mathscr{R}_T$ the set of rules that are cached in the hardware table at a given time, with $|\mathscr{R}_T| \leq T$, and $\mathscr{R}_N \supseteq \mathscr{R}_T$ the entire rule-set, with $|\mathscr{R}_N| \leq N$, which is always stored in the software table.

The cache manager periodically runs a *rule replacement* algorithm that performs two tasks:

1. it picks a set of rules $\mathscr{R}_I \subseteq \mathscr{R}_N$ to insert in the hardware table, where $|\mathscr{R}_I| \leq T$ and $\mathscr{R}_I \cap \mathscr{R}_T = \emptyset$ (no rules in $\mathscr{R}_I$ are currently cached);
2. if the hardware table is full, it picks a set of rules $\mathscr{R}_E \subseteq \mathscr{R}_T$ to evict from the hardware table, where $\mathscr{R}_E \cap \mathscr{R}_I = \emptyset$ (no point in evicting a rule that will be inserted), so that $|(\mathscr{R}_T \setminus \mathscr{R}_E) \cup \mathscr{R}_I| = |\mathscr{R}_T| - |\mathscr{R}_E| + |\mathscr{R}_I| \leq T$.

The goal of the rule replacement algorithm is to maximize the number of bytes that will be handled by the data-path (i.e., the number of bytes contained in packets that will match a cached rule) over the next update interval. There are two reasons why the algorithm is not optimal: First, to be optimal, it would require advance knowledge of all the new packets that will arrive until the next run of the algorithm. Second, even if it did have such knowledge, it would need to solve an NP-hard problem, as we have already discussed in section §3.1.2.

Hence, we have developed two algorithms, a non-implementable one that acts as our baseline (§3.2.3) and an implementable one (§3.2.4). Both algorithms make educated guesses on the ideal sets $\mathscr{R}_I$ and $\mathscr{R}_E$ by leveraging the anticipated packet locality based on recent history.

3.2.3 Baseline Algorithm: Least Recently Used

Our baseline algorithm (Algorithm 3.4) follows a "least recently used" (LRU) replacement policy. It runs every time a new packet P arrives. In the following description,

- R is the rule that matches P, which is found either in the hardware or the software table

[6] appendix A.3

as part of the forwarding process;

- $\mathbb{L}$ is a linked list of all the cached rules and R_{LRU} is the rule at the back of $\mathbb{L}$.

The algorithm performs the following tasks:

1. Identifies the set of rules $\mathscr{R}_H$ that are higher-overlapping relative to R.
2. If $R \notin \mathbb{L}$ (the matched rule is not already cached), the algorithm needs to insert R and all the rules in $\mathscr{R}_H \setminus \mathbb{L}$ in the hardware table.

   If there is not enough space in the hardware table, the algorithm evicts enough cached rules to make space, as follows:

   (a) Moves all the rules in $\mathscr{R}_H \cap \mathbb{L}$ to the front of $\mathbb{L}$.

   (b) Identifies the set of rules $\mathscr{R}_L \subseteq \mathbb{L}$ that are lower-priority overlapping relative to R_{LRU}. Then it evicts R_{LRU}[7] and all the rules in $\mathscr{R}_L$, both from the hardware table and the $\mathbb{L}$.

   (c) Repeats step (2b) as many times as necessary.

   After enough space has been made, the algorithm inserts R and all the rules in $\mathscr{R}_H \setminus \mathbb{L}$ in the hardware table and the $\mathbb{L}$.
3. Moves R and all the rules in $\mathscr{R}_H$ to the front of $\mathbb{L}$.

The reason this algorithm is not implementable is that it potentially triggers hardware-table evictions and insertions every time a new packet arrives. In state-of-the-art switches, evicting and inserting rules from the hardware table are costly operations that can take several microseconds, whereas new packets arrive at the granularity of nanoseconds.

Still, we implemented this algorithm because LRU is arguably the most successful cache replacement policy, and we need to compare to it any implementable algorithm we propose.

3.2.4 An Implementable Algorithm: Most Heavily Used

Our implementable algorithm (Algorithm 3.5) follows a "most heavily used" (MHU) policy. It runs every t_u time units, where t_u is a configurable parameter called the *update interval.* In the following description,

- $b \in \mathbb{R}_{\geq 0}$ is a *benefit metric* assigned to a given rule R, initialized at $b = 0$;
- `traffic` is the aggregate size, in bytes, of all packets that have been matched with a given rule R over the last update interval, in either the hardware or the software table;

[7] Due to step (2a), $R_{LRU} \notin \mathscr{R}_H$. We assume $|\{R\} \cup \mathscr{R}_H| \leq T$ (see §3.2.1).

Algorithm 3.4: "Least Recently Used" replacement policy

Data: Size T of hardware table
Data: Set $\mathscr{R}_T$ of cached rules in the hardware table
Data: Linked list $\mathbb{L}$ of all cached rules $\mathscr{R}_T$
Data: Packet P with a matching rule R
Result: Insert in $\mathscr{R}_T$ rule R and higher-overlapping rules relative to R

```
1  R_H ← higher-overlapping rules relative to R
2  forall rules in R_H ∩ L do move rule to front of L
3  if R ∉ L then R_I ← {R} ∪ (R_H \ L)            /* R not cached in hardware table */
4  else R_I ← ∅
5  while |L| + |R_I| > T do                        /* not enough space to insert R_I */
6      R_LRU ← rule at the back of L
7      R_L ← lower-overlapping rules in L with respect to R_LRU
8      evict all the rules in {R_LRU} ∪ R_L from L and R_T
9  insert all the rules in R_I at the front of L and in R_T
```

- $hf \in [0,1)$ is the *history factor* of the MHU policy, representing the weight allocated on the previous value of b when computing its new value.

The algorithm performs the following tasks:

1. Updates the benefit value of all the rules in $\mathscr{R}_N$ as follows:

$$b_{new} = hf \cdot b_{old} + (1 - hf) \cdot \texttt{traffic}$$

2. Sorts[8] all rules in $\mathscr{R}_N$ by decreasing benefit value b.
3. Constructs the set of "chosen" rules $\mathscr{R}_C$, which represents the next "snapshot" of the entire hardware table, as follows:
   (a) Determines the rule $R \in (\mathscr{R}_N \setminus \mathscr{R}_C)$ with the highest benefit value $b > 0$.
   (b) Identifies the set of rules $\mathscr{R}_H$ that are higher-overlapping relative to R.
   (c) If this set fits in the remaining hardware memory, i.e., if $|\{R\} \cup \mathscr{R}_H \cup \mathscr{R}_C| \leq T$, adds R and all the rules in $\mathscr{R}_H \setminus \mathscr{R}_C$ to $\mathscr{R}_C$. Otherwise, R is no longer considered for insertion.
   (d) Repeats steps (3a) through (3c) as many times as necessary, until $|\mathscr{R}_C| = T$ or no other rules in $\mathscr{R}_N \setminus \mathscr{R}_C$ have a positive benefit value.
4. Determines the set of rules $\mathscr{R}_I = \mathscr{R}_C \setminus \mathscr{R}_T$ that will be inserted in the hardware table. The remaining rules in $\mathscr{R}_C \cap \mathscr{R}_T$ are already present in the hardware table and will be protected from potential eviction.

[8] C++14 `std::sort`

5. If there is not enough space in the hardware table, the algorithm evicts[9] enough cached rules to make space, as follows:

   (a) A random rule $R \in \mathscr{R}_T \setminus \mathscr{R}_C$ is chosen.

   (b) Identifies the set of rules $\mathscr{R}_L$ that are lower-overlapping relative to R.

   (c) Then it evicts R and all the rules in $\mathscr{R}_L$ from the hardware table (rule set $\mathscr{R}_T$).

   (d) Repeats steps (5a) through (5c) as many times as necessary.

6. After enough space has been made, inserts all the rules in $\mathscr{R}_I$ in the hardware table.

Algorithm 3.5: "Most Heavily Used" replacement policy

Data: Size T of hardware table
Data: Set $\mathscr{R}_T$ of cached rules in the hardware table
Data: Set $\mathscr{R}_N$ of rules stored in the software table
Data: Benefit value b of every rule
Data: traffic bytes handled by every rule in last update interval t_u
Data: MHU history factor $hf \in [0, 1)$
Result: Update $\mathscr{R}_T$ with "most heavily used" rules in $\mathscr{R}_N$

```
1  forall rules in ℛ_N do  b ← hf · b + (1 − hf) · traffic
2  sort all the rules in ℛ_N by decreasing value b
3  ℛ_C ← ∅                                                  /* ‘‘chosen’’ rules */
4  for R = front of ℛ_N to back of ℛ_N do
5      if |ℛ_C| = T then break                        /* no more hardware memory */
6      else if b = 0 then break              /* rest of the rules have zero benefit */
7      else if R ∉ ℛ_C then                       /* rule has not been chosen yet */
8          ℛ_H ← rules overlapping with R, with a higher priority
9          ℛ_temp ← {R} ∪ (ℛ_H \ ℛ_C)        /* R and non-chosen higher-overlapping */
10         if |ℛ_temp| + |ℛ_C| ≤ T then          /* we have space to insert ℛ_temp */
11             ℛ_C ← ℛ_temp ∪ ℛ_C
12 ℛ_I ← ℛ_C \ ℛ_T                                     /* chosen and non-cached */
13 while |ℛ_T| + |ℛ_I| > T do                  /* not enough memory to insert ℛ_I */
14     R ← random rule in ℛ_T \ ℛ_C                     /* cached, but not chosen */
15     ℛ_L ← lower-overlapping rules in ℛ_T, with respect to R
16     evict all the rules in {R} ∪ ℛ_L from ℛ_T
17 insert all the rules in ℛ_I to ℛ_T
```

[9] Initially, we would straightforwardly evict all the rules in $\mathscr{R}_T \setminus \mathscr{R}_C$ from the hardware table. However, in experimental scenarios where $\mathscr{R}_C$ was consistently smaller than the entire hardware table ($|\mathscr{R}_C| < T$), multiple rules were unnecessarily being evicted, inflating the rule replacement rate. This resulted from the good locality of our packet traces (§3.3.1), therefore less than T rules would have a positive benefit value b. The replacement rate of the LRU policy would outperform the MHU replacement rate in these scenarios.

3.3 Experimental Evaluation

In this section we present the experimental evaluation of our virtual flow tables architecture, which we conducted according to the following methodology:

1. We obtained two data center packet traces T_1 and T_2 for the purposes of simulating the behavior of our platform in a real environment. We present these traces in section §3.3.1.
2. For each of the aforementioned packet traces, we generated a synthetic overlapping rule set $\mathscr{R}_M$ of M rules, which we call the *original* rule set. We describe this procedure in section §3.3.2.
3. We selected a set $\mathbb{X}$ of independent variables that we varied in our benchmarks and the set $\mathbb{Y}$ of dependent variables that we measured to evaluate the performance of our platform. We present these variables in sections §3.3.3 and §3.3.4, respectively.
4. For each distinctive independent variable set $\mathbb{X}$, we followed the following steps (Algorithm 3.6):
   (a) We selected the $T_i = T_1$ packet trace and the respective original rule set $\mathscr{R}_M$ we had generated in step (2).
   (b) We populated the software table with the original rule set $\mathscr{R}_M$.
   (c) We structured the software table as we have described in §3.2.1, by partitioning the $\mathbb{U}^2$ space and creating the packet classifier data structure.
   (d) The partitioning algorithm generated a new rule set $\mathscr{R}_N \supseteq \mathscr{R}_M$ of $N \geq M$ rules, which we call the *partitioned* rule set. The software table was at this point populated with $\mathscr{R}_N$.
   (e) We selected the LRU replacement policy to be executed concurrently with step (4f).
   (f) We executed a benchmark by feeding the packet trace T_i to the hardware table. Every ingress packet was looked up at the hardware table and, if no matching rule was found, was also looked up at the software table.
   (g) All dependent variables in $\mathbb{Y}$ were measured during the execution of the benchmark, in step (4f).
   (h) Steps (4e) through (4g) were executed again, using the MHU policy.
   (i) Steps (4a) through (4h) were executed again, using $T_i = T_2$ as the packet trace.

We illustrate our experimental results in section §3.3.5. We conducted all benchmarks on a single, isolated Dell Poweredge C6220, bearing an Intel Xeon E5-2660 CPU.

For the purposes of this evaluation, we worked within the 2-dimensional space $\mathbb{U}^2$ = {source IP address, destination IP address}. Adapting our evaluation to higher dimensions would be

straightforward. Furthermore, we assume that there are no "rule timeouts" and the update interval of the MHU policy is always configured at $t_u = 1s$.

Algorithm 3.6: Evaluation of virtual flow tables

Data: 2-dimensional space $\mathbb{U}^2$ = {source IP address, destination IP address}
Data: Set of packet traces $\mathscr{T} = \{T_1,\ T_2\}$
Data: Set of replacement policies $\mathscr{P}$ = {LRU, MHU}
Data: Set of independent variables $\mathbb{X}$
Result: Set of dependent variables $\mathbb{Y}$

```
1 foreach T_i ∈ 𝒯 do
2     generate original overlapping rule set ℛ_M = Generate(T_i)      /* Algorithm 3.7 */
3     populate software table with ℛ_M
4     configure partitioning algorithm Partition with 𝕏              /* Algorithm 3.1 */
5     run partitioning algorithm Partition(𝕌², ℛ_M)
6     for policy ∈ 𝒫 do
7         feed T_i to the hardware table
8         obtain measurements for each y ∈ 𝕐
```

3.3.1 Packet traces

For the purposes of our evaluation, we have chosen to deploy data-sets from two university campus data centers that have been presented and analyzed in [7]. In particular, we have selected the packet traces *EDU1* and *EDU2*, due to the dense locality characteristics they exhibit, as the number of active flows within a one second interval does not exceed 500 in 90% of the time. These packet traces are publicly available at [6], and will hereinafter be referred to as *UNI1* and *UNI2*, respectively. For the purposes of our evaluation scenarios we have accelerated both traces by a factor of 100. We illustrate the characteristics of the original and the accelerated packet packets in Table 3.1

Characteristic	Original		Accelerated	
	UNI1	UNI2	UNI1	UNI2
Packets per second	4688	10428	$4.7 \cdot 10^5$	$1.04 \cdot 10^6$
Throughput (Mbit/sec)	26.2	63.8	$2.6 \cdot 10^3$	$6.4 \cdot 10^3$
Duration (sec)	3914	9480	39.1	94.8

Table 3.1 – Characteristics of packet traces; original and accelerated versions

3.3.2 Creating a synthetic overlapping rule set

We have generated a synthetic set $\mathscr{R}_M$ of overlapping rules for each of the two packet traces UNI1 and UNI2, by executing the following steps (Algorithm 3.7):

1. Each packet generates a single rule R with the same source and destination IP address pair, hereinafter denoted as $\{\text{srcIP}, \text{dstIP}\}$
2. Each generated rule has a 33% chance of being:
   (a) An exact match rule $\{\text{srcIP}/32, \text{dstIP}/32\}$.
   (b) A wildcard rule $\{\text{srcIP}/24, \text{dstIP}/24\}$.
   (c) A wildcard rule $\{\text{srcIP}/16, \text{dstIP}/16\}$.
3. A priority is assigned to each generated flow rule. Exact match rules are assigned the highest priorities, followed by the /24 wildcard rules and subsequently by the /16 wildcard rules, which are assigned the lowest priorities.
4. Finally, any duplicate rules are purged.

This methodology ensures that:

- Every packet will be matched to at least one rule, since a rule R is generated for every packet, unless R already exists.
- The dense locality characteristics of the traces will result in generating overlapping rules, e.g., packets $P_1 : 1.1.1.1 \rightarrow 2.2.2.2$ and $P_2 : 1.1.5.5 \rightarrow 2.2.6.6$ may result in generating rules $R_1 : 1.1.1.1/32 \rightarrow 2.2.2.2/32$ and $R_2 : 1.1.0.0/16 \rightarrow 2.2.0.0/16$, which are overlapping rules.
- The more specific rules will bear a higher preference than the more generic rules, thus avoiding redundancies, e.g., rule $R_1 : 1.1.1.1/32 \rightarrow 2.2.2.2/32$ will have a higher preference than $R_2 : 1.1.0.0/16 \rightarrow 2.2.0.0/16$, otherwise no ingress packet would ever match with R_1.

We generated two synthetic rule-sets $\mathscr{R}_M$ from traces UNI1 and UNI2, with $8,994$ and $11,210$ rules, respectively.

3.3.3 Independent variables

We have selected the following independent variables, hereinafter referred to as *parameters*, to evaluate our platform:

1. The size T of the hardware table, hereinafter referred to as `tcam`. This represents the total rule space in the TCAM memory.
2. The `binth` parameter of the partitioning Algorithm 3.1, i.e., the maximum number of flow rules in a classifier leaf node.
3. The `spfac` parameter of the partitioning Algorithm 3.1, i.e., the memory allocation factor.

Algorithm 3.7: Generating a synthetic overlapping rule set from a packet trace

Data: A packet trace T

Result: Set of overlapping rules $\mathscr{R}_N$

```
1  open T for reading
2  while not at the end of T do
3      read next packet P ∈ T
4      extract ip_pair = (srcIP, dstIP) from P
5      if ip_pair has not been processed yet then     /* skip if we have processed it */
6          generate random integer k ∈ [1,3]
7          switch k do                                 /* 33% probability to create... */
8              case 1 do mask ← 32                     /* ... an exact match rule */
9              case 2 do mask ← 24                     /* ... a wildcard /24 rule */
10             case 3 do mask ← 16                     /* ... or a wildcard /16 rule */
11         generate R with ip_pair and mask
12         if R ∉ R_N then add R to R_N   /* two packets may generate the same R */
13 close T
14 priority ← 0                                  /* lower value denotes higher priority */
15 foreach m_i ∈ {32, 24, 16} do                 /* exact rules first, wildcards last */
16     foreach R ∈ R_N with mask = m_i do
17         assign priority to R
18         priority ← priority + 1
```

4. The hf parameter of the Most Heavily Used replacement policy (Algorithm 3.5), i.e., the history factor, hereinafter referred to as `history`. As anticipated, we have benchmarked only the Most Heavily Used policy when varying this independent variable.

As we have already stated, set $\mathbb{X} = \{\texttt{tcam}, \texttt{binth}, \texttt{spfac}, \texttt{history}\}$ is provided as an argument to Algorithm 3.6. We have executed this Algorithm under four use-cases, each time varying a different parameter $x_i \in \mathbb{X}$, while keeping constant the value of the other parameters $x_j \in \mathbb{X}, j \neq i$ (Table 3.2). In detail:

- The constant value of `tcam` is being set to 1024, which is a realistic configuration given the memory capacity of contemporary TCAMs (§2.1.1).
- The constant value of `binth` is set to 10, as it creates relatively few dependencies while at the same time demanding a memory footprint that our hardware can provide. Nevertheless, when the `spfac` parameter is being varied, for high values of `spfac` we would run out of memory, thus `binth` is set to 50 in these experiments.
- The constant value of `spfac` is set to 2. We have empirically found, as presented in §3.3.5, to be a near-optimal trade-off between memory allocation and execution time.

- The constant value of `history` is set to 0.5 to equally weight both the observed bytes in the last update interval and the previous benefit value of each flow rule.

Varied parameter	Fixed parameters			
	`tcam`	`binth`	`spfac`	`history`
`tcam`	–	10	2	0.5
`binth`	1024	–	2	0.5
`spfac`	1024	50	–	0.5
`history`	1024	10	2	–

Table 3.2 – Independent variables used in virtual flow tables evaluation scenarios

3.3.4 Dependent variables

We have evaluated our platform in terms of:

I. **Efficiency**: how much data is handled within the data-path and how many operations are needed, on average.

II. **Memory consumption**: how the memory demands of our platform are affected based on the configuration of the partition Algorithm 3.1.

To quantify the efficiency and the memory consumption of our platform, we have measured the following dependent variables, hereinafter referred to as *metrics*:

1. **Byte Miss Rate:** the percentage of the ingress bytes that has been handled by the software pipeline, calculated as:

$$\text{Byte Miss Rate} = \frac{\text{Ingress bytes handled by software table}}{\text{Total ingress bytes}} \cdot 100\%$$

   An efficient rule replacement policy is expected to produce a low byte miss rate, as a larger proportion of the ingress bytes ends up being matched with a rule stored in the hardware table and, therefore, forwarded at line rate.

2. **Replacement Rate:** the number of rules that have been evicted from the hardware table, on average, over the *nominal duration* of the packet trace, as dictated by the replacement policy that is being employed, calculated as:

$$\text{Replacement Rate} = \frac{\text{Number of evicted rules from the hardware table}}{\text{Nominal duration of the packet trace}}$$

   The *nominal duration* of a packet trace (Table 3.1) is defined as:

$$\text{Nominal duration} = \text{Timestamp of last packet} - \text{Timestamp of first packet}$$

A replacement policy producing a formidable replacement rate would be unpractical to be deployed on a real switch due to hardware limitations and energy consumption issues. Therefore, a lower replacement rate is desirable.

3. **Individual Caching:** the number of distinct rules from the partitioned rule set $\mathscr{R}_N$ that have been inserted in the hardware table at least once, at any time during the experiment's execution, as a fraction of the number of original rules in $\mathscr{R}_M$, calculated as:

$$\text{Individual Caching} = \frac{\left|\left\{R \in \mathscr{R}_N \mid R \text{ has been cached at least one time in the past}\right\}\right|}{|\mathscr{R}_M|}$$

4. **Replication Rate:** the number of rules in the partitioned rule set $\mathscr{R}_N$ stored in the software table, over the number of rules in the original rule set $\mathscr{R}_M$, calculated as:

$$\text{Replication Rate} = \frac{|\mathscr{R}_N|}{|\mathscr{R}_M|}$$

Metrics 1 – 2 evaluate our platform in terms of efficiency, while metrics 3 – 4 evaluate our platform in terms of memory consumption.

3.3.5 Results

In this section we present the results of our evaluation benchmarks, organized by the measured metrics $y \in \mathbb{Y}$.

Byte Miss Rate

We present the byte miss rate when measured against:

- the `tcam` parameter, in Figure 3.3;
- the `binth` parameter, in Figure 3.4;
- the `spfac` parameter, in Figure 3.5;
- the MHU `history` parameter, in Figure 3.6.

`tcam` **parameter** The miss rate exhibits the anticipated behavior. Fewer bytes are handled by the software table as we increase the value of `tcam`, due to having more available memory in the hardware table. As a result, more rules are stored concurrently in the hardware table, resulting in fewer packets being redirected to the software table.

`binth` **parameter** The performance of the MHU policy is directly dependent on the `binth` parameter. In particular, higher `binth` values lead to coarse-grained partitioning of the $\mathbb{U}^2$ plane, resulting in more rules per leaf partition and thus more dependencies between rules. Therefore, a rule that is being brought into the hardware table will tend to bring more higher-overlapping rules with it, effectively increasing its memory footprint. Consequently, less rules with a high benefit value may be stored, increasing the byte miss rate, as more packets are being redirected to the software table.

On the other hand, the LRU policy is virtually unaffected by the `binth` parameter. This behavior is anticipated, as the LRU policy is triggered on every ingress packet, therefore always inserting in the hardware table the same sequence of flow rules. Although the cached rule sets may be bigger, the relatively big TCAM size (Table 3.2) prevents the eviction of active flow rules.

`spfac` **parameter** The memory factor only affects the internal structure and the depth of the packet classifier tree, but does not influence how cross-rule dependencies are mitigated or how rules are replaced in the hardware table. Therefore, it does not affect our metric, which we have also verified empirically.

`history` **parameter** We have classified the miss rate into three types, depending on the status of the matching rule in the software table, at the time of the packet lookup:

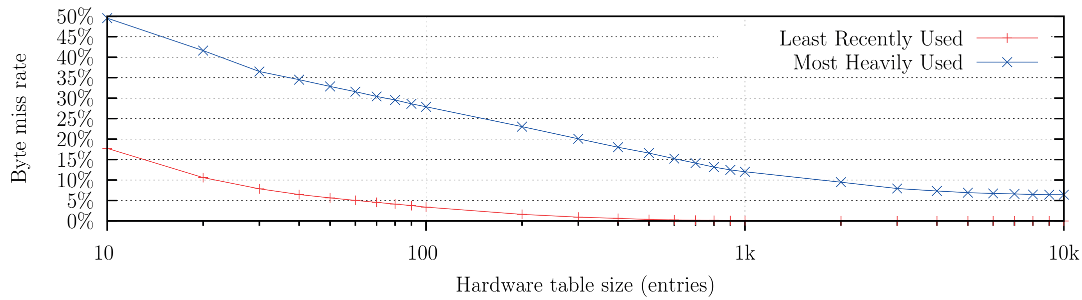
1. *Mandatory bytes*: the matching flow rule has never been inserted in the hardware table at any point in the past and this is the first time a packet matches this flow rule. This type is not shown in our results, as it was less than 1% and thus not meaningful.

2. *Cold bytes*: the matching flow rule has never been inserted in the hardware table at any point in the past, but has been matched by at least one packet. The LRU policy never produces this type, as every flow matched with an ingress packet is immediately cached in the hardware table.

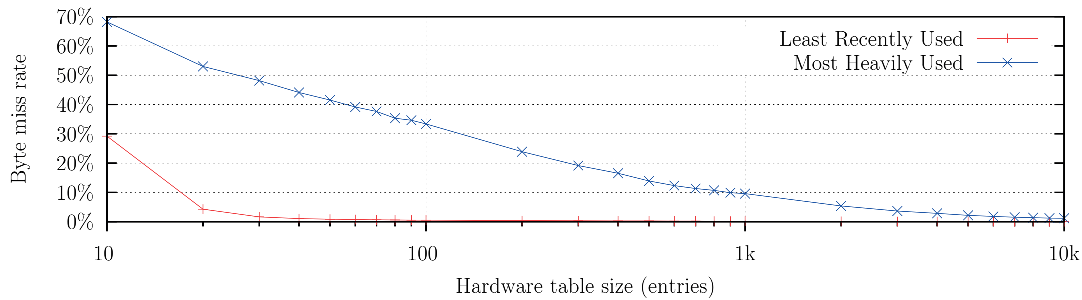
3. *Evicted bytes*: the matching flow rule had been inserted in the hardware table at some point in the past, but was eventually evicted.

The total byte miss rate tends to decrease as we increase the MHU history factor. In particular, by allocating more weight to the previous benefit value of a flow rule at the expense of the observed traffic, flow rules tend to stay cached for longer intervals in the hardware table. Consequently, our traces, which exhibit strong locality characteristics, take advantage of the prolonged preservation of flow rules within the hardware table.

Regarding the specific miss types, we observe that as we increase the history factor:



(a) "UNI1" packet trace



(b) "UNI2" packet trace

Figure 3.3 – Byte miss rate versus `tcam`

- there are less *evicted* miss bytes, since flows tend to stay longer in the hardware table;
- there are more *cold* miss bytes, since flows need to stay active for a longer time to get eventually inserted in the hardware table.

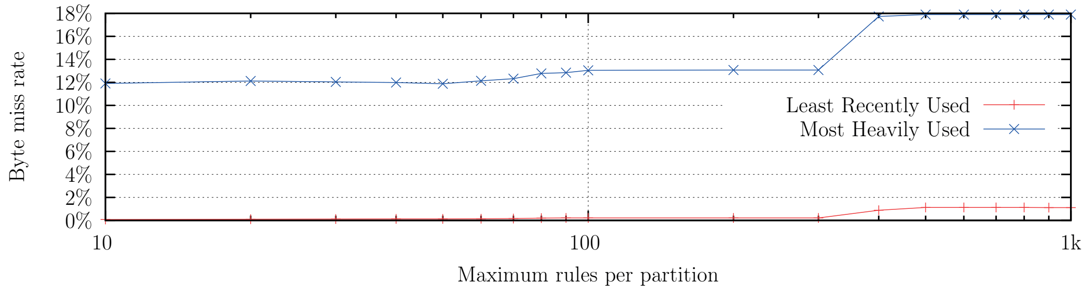
Overall, the impact of the decrease in *evicted* miss bytes is much more substantial, leading to an overall decrease in the byte miss rate.

Replacement Rate

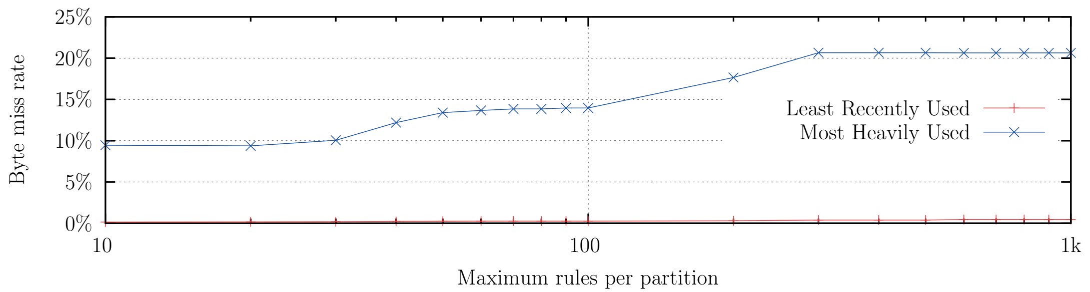
We present the rule replacement rate when measured against:

- the `tcam` parameter, in Figure 3.7;
- the `binth` parameter, in Figure 3.8;
- the `spfac` parameter, in Figure 3.9;
- the MHU `history` parameter, in Figure 3.10.

General observations We observe that the replacement rate of the LRU policy is substantially higher than the respective replacement rate of the MHU policy, even up to 5 orders

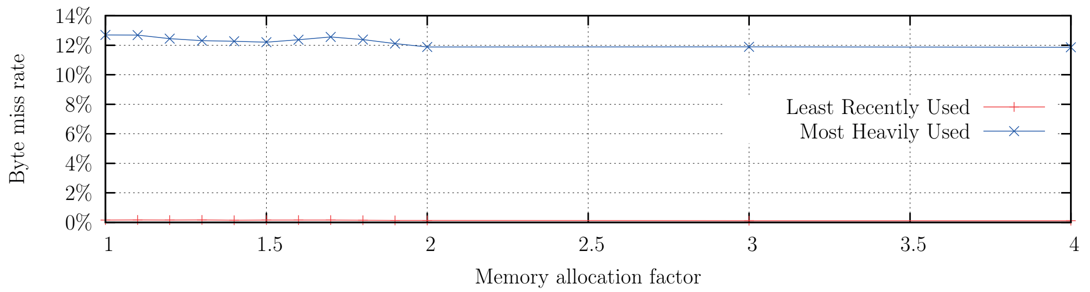


(a) “UNI1” packet trace

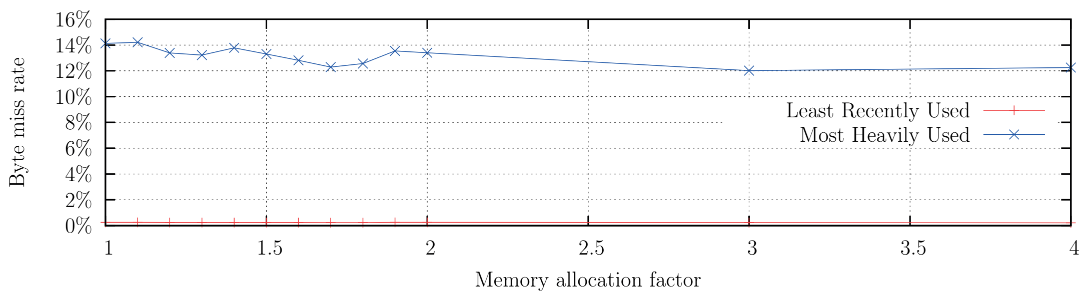


(b) “UNI2” packet trace

Figure 3.4 – Byte miss rate versus `binth`

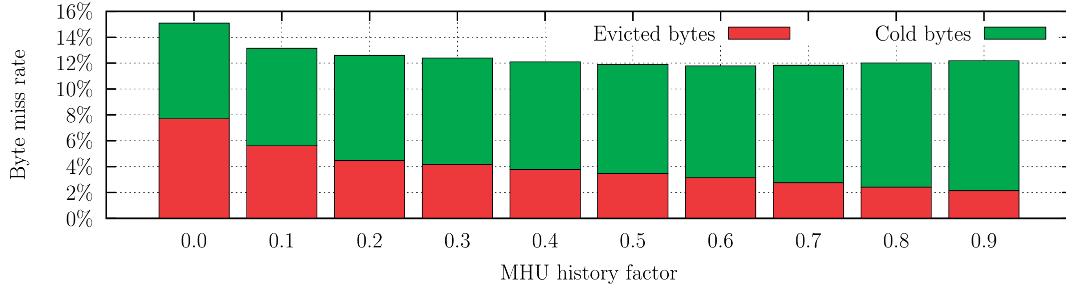


(a) “UNI1” packet trace

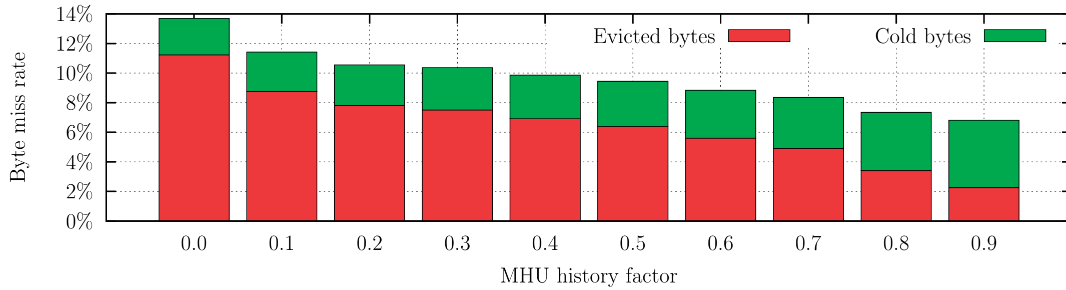


(b) “UNI2” packet trace

Figure 3.5 – Byte miss rate versus `spfac`



(a) "UNI1" packet trace



(b) "UNI2" packet trace

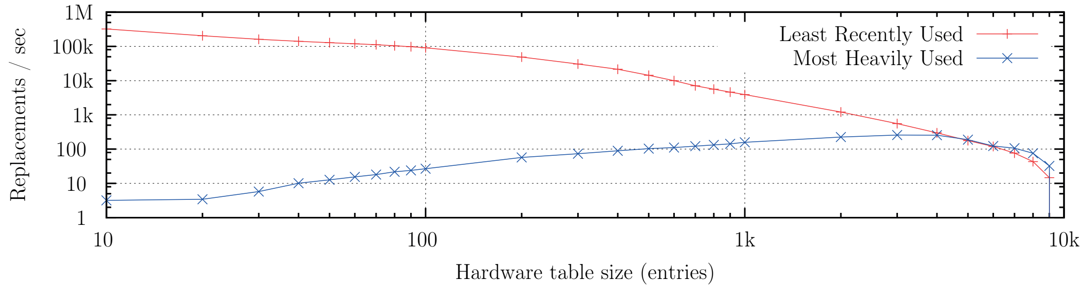
Figure 3.6 – Byte miss rate versus `history`

of magnitude for small sizes of the hardware table. This behavior is attributed to the fact that the LRU policy is triggered on every packet miss, which may produce more than $100k$ (theoretical) updates per second. For comparison, when evaluating our earlier prototype[10] we had measured the average latency between ASIC updates at $300\mu s$ [14], translating to approximately $3,333$ updates per second; the LRU policy would not be feasible.

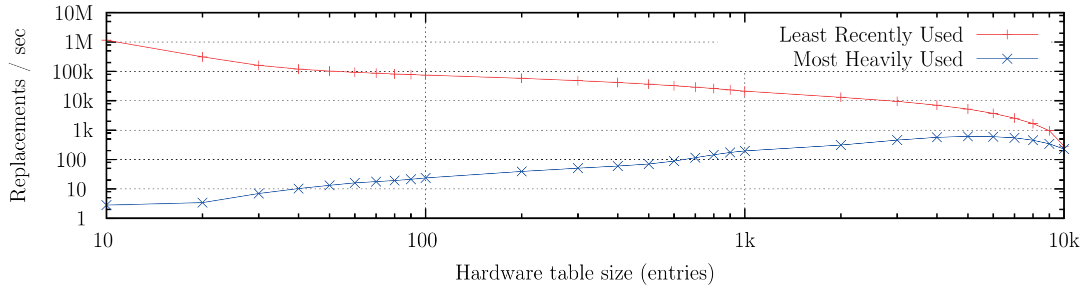
`tcam` **parameter** We observe in Figure 3.7 that when we employ the LRU replacement policy on a relatively small hardware table, e.g., with `tcam` = 10 entries, the algorithm is constantly updating the hardware table in order to keep the traffic within the data-path. The replacement rate tends to drop as we increase the value of `tcam`, due to having more available memory in the hardware table, resulting in less lookup misses; as a result, fewer rules are evicted.

On the other hand, the MHU policy is triggered every $t_u = 1s$, therefore no more than $M =$ `tcam` entries may be replaced within a second, regardless of the packet lookup rate at the hardware table. We observe an initial increase of the replacement rate as we allocate more memory to the hardware table; this is a direct consequence of our replacement policy, which replaces approximately the entire hardware table when limited hardware memory is available. Nevertheless, the replacement rate tends to eventually stabilize, due to having abundant memory for the current "working set" in a given update interval t_u.

[10] We have briefly described the relation between this prototype and our current work in section §3.1.1.



(a) "UNI1" packet trace



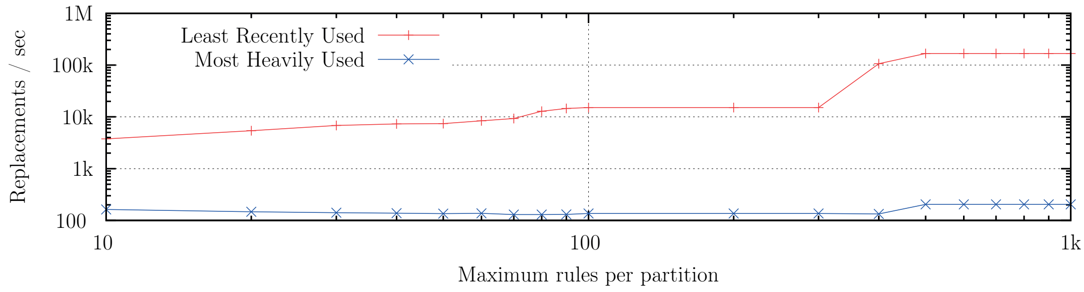
(b) "UNI2" packet trace

Figure 3.7 – Replacement rate versus `tcam`

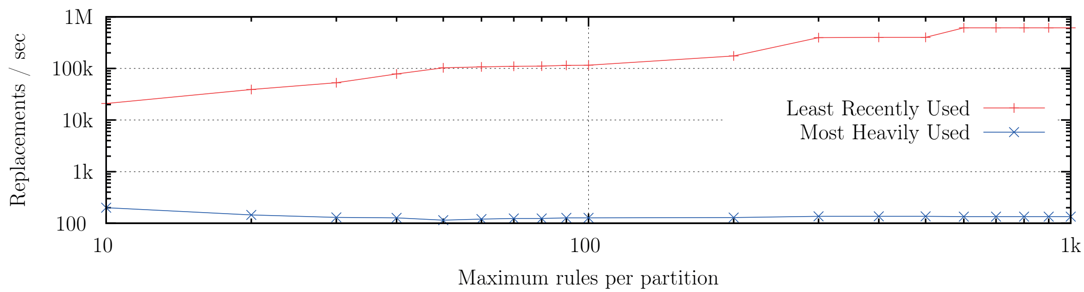
`binth` **parameter** Lower values of the `binth` parameter tend to produce more fine-grained partitions of the $\mathbb{U}^2$ space, with less overlaps between traffic rules. On the other hand, higher values tend to produce more coarse-grained partitions with more complex dependencies. This directly affects the LRU algorithm, which brings into the hardware table a larger overlapping rule set, on average. On the other hand, the MHU policy is virtually unaffected, as it is primarily constrained by the size T of the hardware table.

`spfac` **parameter** The memory factor only affects the internal structure and the depth of the packet classifier tree, but does not influence how cross-rule dependencies are mitigated or how rules are replaced in the hardware table. Therefore, it does not affect our metric, which we have also verified empirically.

`history` **parameter** The replacement rate tends to decrease linearly as we increase the MHU history factor. This is the anticipated behavior, as a higher history factor allocates more weight to the previous benefit value of a flow rule at the expense of the observed traffic in the last update interval t_u. As a result, there is a lower incentive to evict existing flows from the hardware table, unless a flow remains "inactive" for a longer period of time. Furthermore, newer flows must exhibit substantially higher handled traffic within the same time interval to be assigned a higher benefit value and, thus, qualify to be cached in the hardware table.

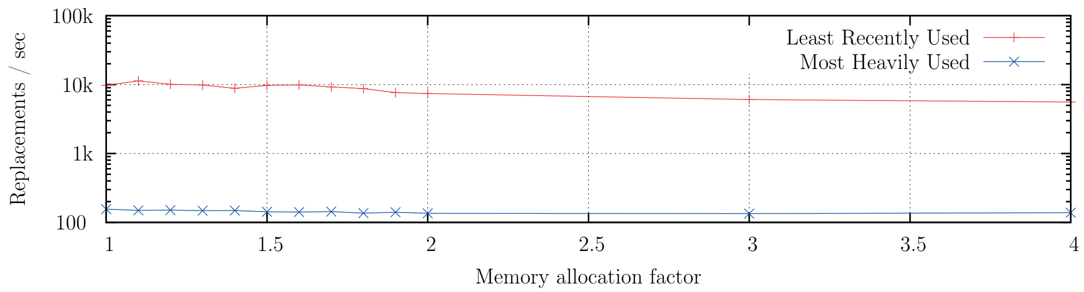


(a) “UNI1” packet trace

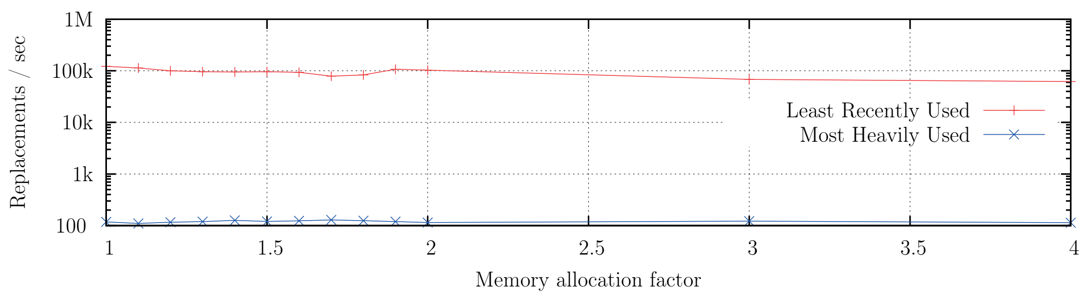


(b) “UNI2” packet trace

Figure 3.8 – Replacement rate versus `binth`



(a) “UNI1” packet trace



(b) “UNI2” packet trace

Figure 3.9 – Replacement rate versus `spfac`

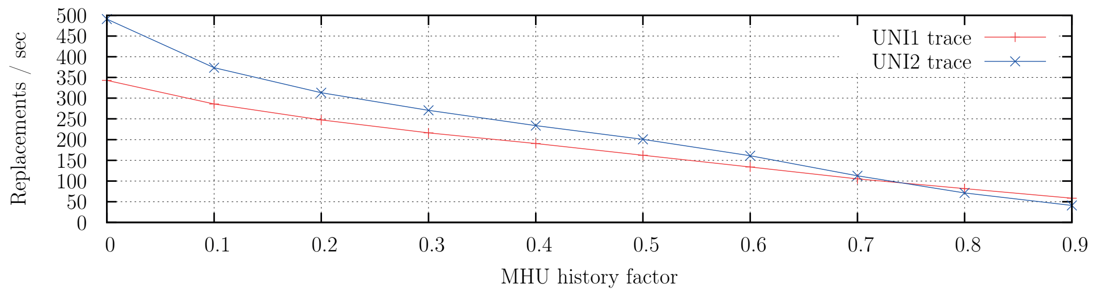


Figure 3.10 – Replacement rate versus `history`

Individually Cached Flow Entries

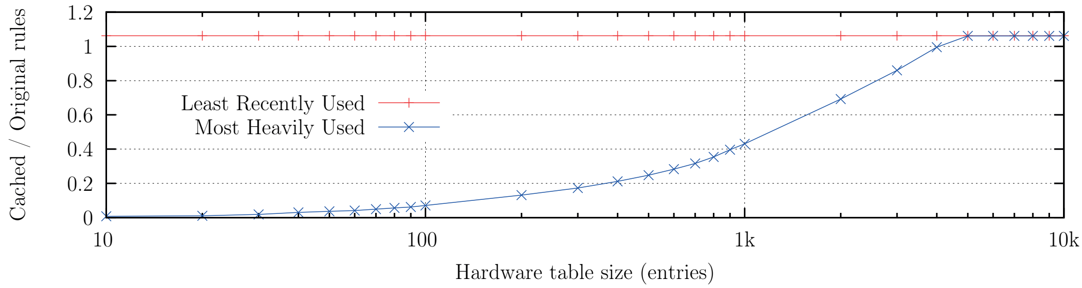
We present the fraction of individually cached rules when measured against:

- the `tcam` parameter, in Figure 3.11;
- the `binth` parameter, in Figure 3.12;
- the `spfac` parameter, in Figure 3.13;
- the MHU `history` parameter, in Figure 3.14.

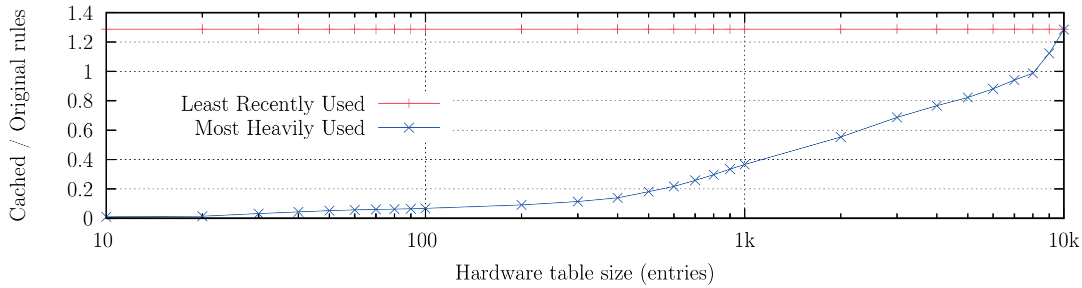
General observations We observe that the LRU policy almost always caches the same number of individual rules. This is the expected behavior, as the LRU always brings in the hardware table the corresponding rule R that matches an ingress packet. Any minor variations occur from the resulting dependencies between rules, as the LRU also brings any higher-overlapping rules relative to R.

`tcam` **parameter** The MHU policy gradually caches more individual rules as we allocate more memory in the hardware table, until the hardware table is big enough to match the respective metric reported by the LRU. As a result, the MHU avoids polluting the limited hardware memory with numerous low-benefit rules with few matching ingress packets. On the other hand, the LRU does not apply such optimizations and always inserts the same number of rules.

`binth` **parameter** Increasing the `binth` value generates more coarse-grained partitions, producing larger dependent rule sets. As a result, a rule R tends to bring more higher-overlapping rules in the hardware table (Figure 3.8). Nevertheless, coarse-grained partitions reduce rule replication, therefore fewer individual rules end up being inserted in the hardware table.



(a) "UNI1" packet trace



(b) "UNI2" packet trace

Figure 3.11 – Cached rules versus `tcam`

`spfac` **parameter** The memory factor only affects the internal structure and the depth of the packet classifier tree, but does not influence rule replication, how cross-rule dependencies are mitigated or how rules are replaced in the hardware table. Therefore, it does not affect our metric, which we have also verified empirically.

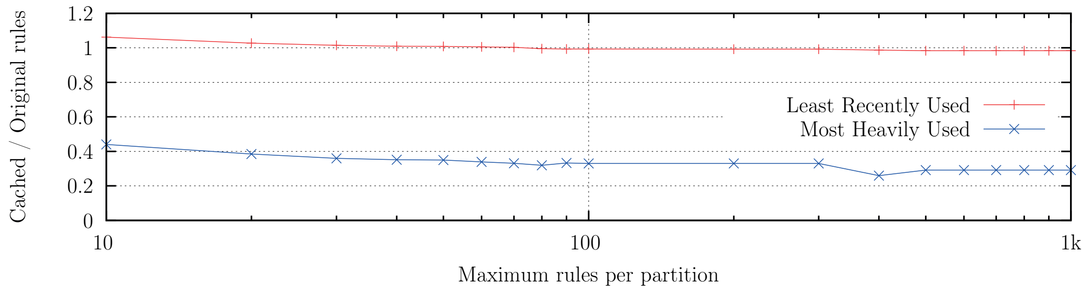
`history` **parameter** Our results are consistent with our previous findings in Figure 3.10. When increasing the history factor, flow rules tend to stay longer in the hardware table, therefore less individual rules are cached.

Flow Rule Replication Rate

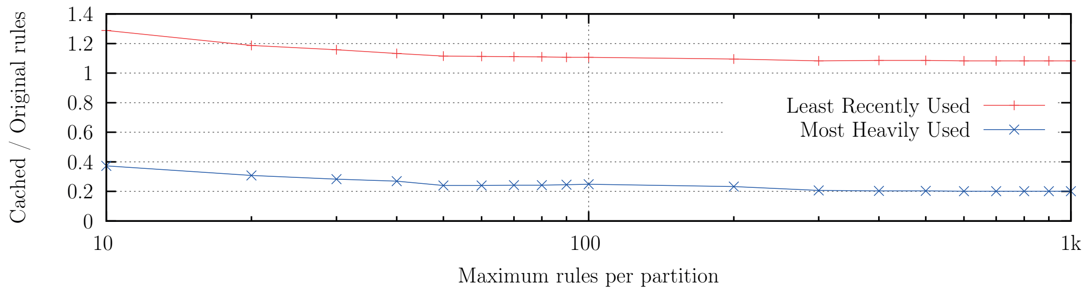
We present the rule replication rate when measured against:

- the `binth` parameter, in Figure 3.15a;
- the `spfac` parameter, in Figure 3.15b.

`binth` **parameter** Increasing the `binth` value generates more coarse-grained partitions, therefore reducing the number of rules that conflict with multiple partitions. The number of

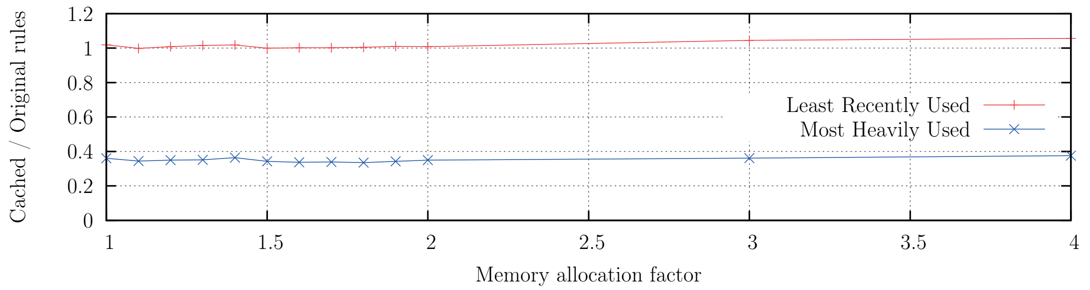


(a) “UNI1” packet trace

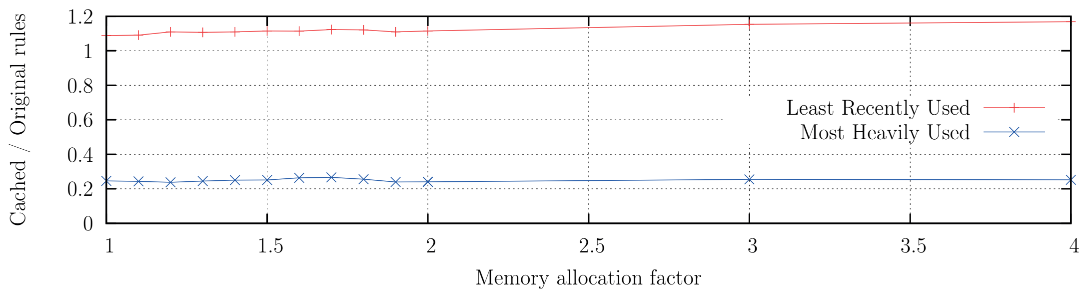


(b) “UNI2” packet trace

Figure 3.12 – Cached rules versus `binth`



(a) “UNI1” packet trace



(b) “UNI2” packet trace

Figure 3.13 – Cached rules versus `spfac`

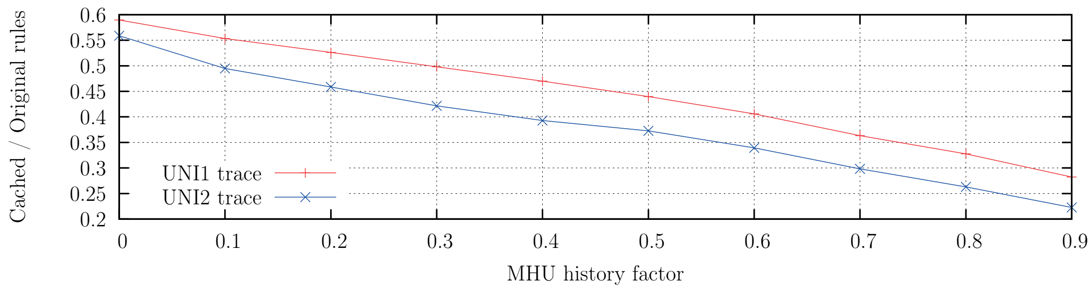


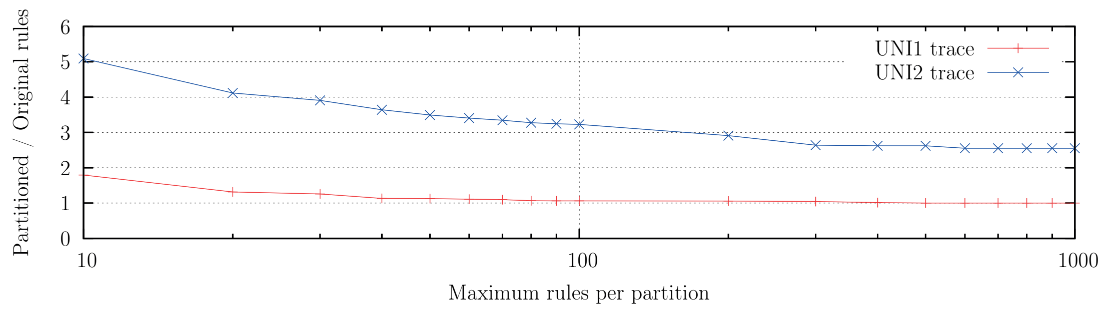
Figure 3.14 – Cached rules versus `history`

leaf partitions tends to remain constant for `binth` ≥ 100.

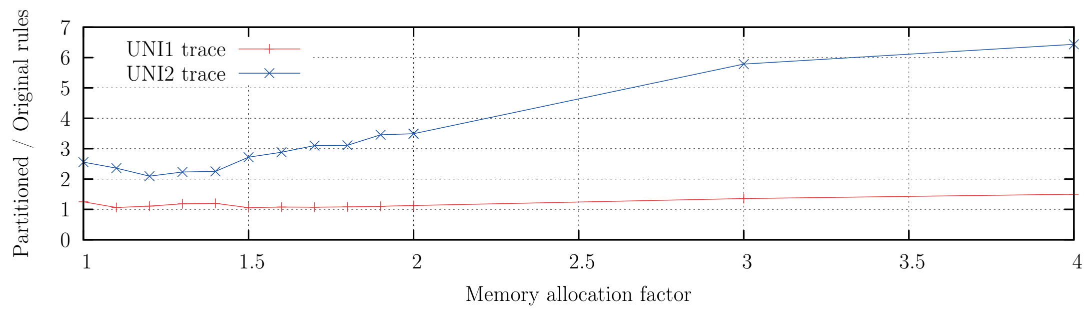
`spfac` **parameter** Increasing the memory allocation factor results in creating a tree structure with longer "width" and shorter "height." As a result, the partition algorithm on a given node tends to create more child partitions, therefore wildcard rules will conflict with more partitions. Since we are producing `/16` and `/24` wildcard rules, the effect of the `spfac` parameter is observed when partitioning a relatively large rule set, such as the rule-set we created from the UNI2 packet trace.

3.4 Summary

In this chapter, we proposed a switch architecture that exposes the abstraction of a virtual flow table, which fits significantly more rules than the switch's physical flow table. We implemented this abstraction with a simple, two-layer memory hierarchy: a software table, located off the data-path and accessible by the switch's supervisor engine; and a much smaller but faster hardware table, located on the data-path, and acting as a cache for the software table. The challenge we focused on is the caching of overlapping rules, e.g., "admit all traffic from X" and "drop all ICMP traffic from X"; if we cache the former but not the latter, then the switch will admit all traffic from X, which is incorrect forwarding behavior. Even if we know exactly what traffic will arrive at the switch in the future, identifying which rules to cache such that we maintain correct forwarding behavior *and* maximize the amount of traffic that stays inside the data-path is NP-hard; we showed it by mapping a simpler version of this problem to the 0-1 knapsack problem. On the positive side, we showed that there exists at least one simple heuristic that works well in practice: an algorithm that caches groups of overlapping rules, favoring the caching of groups that recently matches large amounts of traffic. Given realistic data-path memory sizes and realistic traffic, our algorithm achieved hit rate 95% in terms of the fraction of received traffic that stayed within the data-path.



(a) versus `binth`



(b) versus `spfac`

Figure 3.15 – Replication rate of flow rules

4 Scheduling Shared Network Resources

In this chapter, we present our solution for scheduling shared network resources. As stated in the introduction, we seek a solution that integrates seamlessly with the scheduling of other resources and enables cloud providers to easily incorporate new resource types in their infrastructure. We address this by making the computation of scheduling decisions resource-agnostic and relegating it to a general-purpose constraint solver, and the enforcement of scheduling decisions resource-specific and relegating it to special managers—one per resource type. We focus on the manager for network resources and propose simple ways to enforce bandwidth reservations and latency constraints. We evaluate our solution through extensive experiments on an educational cloud platform.

4.1 Two-Tiered Resource Scheduling

We propose a two-tiered resource-scheduling architecture that consists of the following layers:

1. *Resource-agnostic scheduling* takes as input (i) the set of available resources (e.g., number of available processing cores, available bandwidth between two servers) and (ii) the set of resource requests made by users for these resources, and it makes a "scheduling decision," i.e., decides if, and to what extent, it can satisfy each request. This layer has no semantic information about specific resource types.

2. *Resource-specific enforcement* takes as input the scheduling decision made by the resource-agnostic scheduler and tries to implement it; it also communicates back to the resource-agnostic scheduler the set of available resources, abstracting away resource-specific details (e.g., where a processing core is physically located, or which network links carry traffic between two servers). This layer consists of multiple *infrastructure resource managers* (IRMs), each one handling a specific resource type.

Figure 4.1 illustrates this architecture.

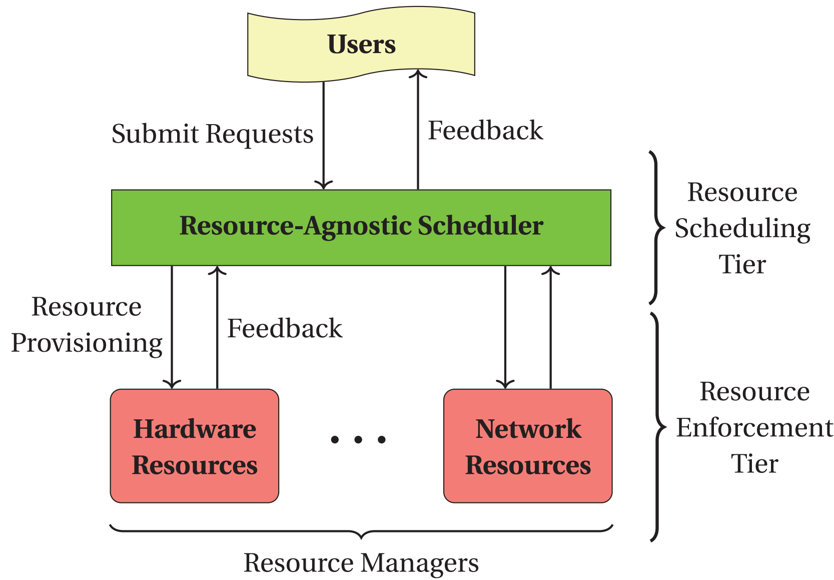


Figure 4.1 – Two-tiered architecture to facilitate scheduling shared network resources

This thesis focuses on resource-specific enforcement and, in particular, the design and implementation of the *network infrastructure resource manager* (IRM-NET), which handles network resources. For evaluation purposes, we deployed our IRM-NET implementation together with the resource-agnostic scheduler and other IRMs implemented in the context of the HARNESS European project [11]—an effort in which the author of this thesis participated actively.

4.2 The Network Infrastructure Resource Manager

IRM-NET plays two roles:

1. It communicates with the resource-agnostic scheduler: IRM-NET tells the scheduler what is the available bandwidth (§4.2.3) and latency (§4.2.4) between each pair of compute nodes, and it enforces the scheduler's bandwidth-reservation decisions.

2. It implements the cloud provider's general network-management policies, e.g., oversubscription of network links/paths when tenants are not anticipated to fully use their bandwidth reservations concurrently (§4.2.5), or fair sharing of the network between different tenants (§4.2.6).

We assume a single path between each pair of compute nodes (and, consequently, represent network topology as an undirected tree). However, our design and implementation can be easily extended to the more realistic scenario of multiple paths.

4.2.1 Modeling Network Links

IRM-NET maintains a complete picture of the cloud network topology, including all network links, their endpoints, and attributes. Each network link is modeled as a *compound resource*, represented with a `json` syntax, where the following information is maintained: (i) the link's endpoints, defined as the identifiers of the corresponding hardware components, such as a compute node or a switch; (ii) the maximum capacity of the link, in Mbits/second; (iii) the available bandwidth of the link, defined as the difference between the link's capacity and the bandwidth being currently consumed, in Mbits per second; and (iv) the latency between the two end-points, in milliseconds.

As an example, consider the simple "star" LAN, illustrated in figure 4.2. Three compute nodes are connected through a common switch via 1 Gbit Ethernet links and three bandwidth-intensive applications are running between the three nodes, consuming 425, 475 and 375 Mbit/sec, respectively.

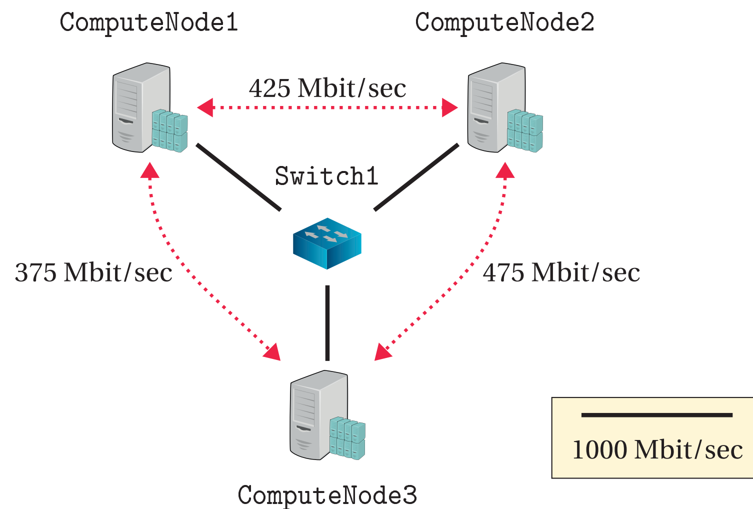


Figure 4.2 – Example of a simple "star" LAN topology

Listing 4.1 illustrates how the network links in figure 4.2 are modeled. We may identify for each link: (i) the `Capacity` attribute, indicating the maximum bandwidth capacity of the link; and (ii) the `Bandwidth` attribute, reporting how much available bandwidth may be scheduled through the link. For example, there are two flows over the link `ComputeNode1` – `Switch1`, thus the available bandwidth is 1000 – 425 – 375 = 200 Mbit/sec.

Listing 4.1 – Internal representation of network links in figure 4.2

```
1 LinkList = [
2     {
3         "Type": "Link",
4         "Source": "ComputeNode1",
5         "Target": "Switch1",
```

```
6           "Attributes": {
7               "Latency": 0.22,
8               "Capacity": 1000,
9               "Bandwidth": 200
10          }
11      },
12      {
13          "Type": "Link",
14          "Source": "ComputeNode2",
15          "Target": "Switch1",
16          "Attributes": {
17              "Latency": 0.15,
18              "Capacity": 1000,
19              "Bandwidth": 100
20          }
21      },
22      {
23          "Type": "Link",
24          "Source": "ComputeNode3",
25          "Target": "Switch1",
26          "Attributes": {
27              "Latency": 0.11,
28              "Capacity": 1000,
29              "Bandwidth": 150
30          }
31      }
32  ]
```

4.2.2 Exposing Virtual Paths

IRM-NET hides the network topology from the resource-agnostic scheduler, exposing only the *virtual end-to-end paths* between each pair of compute nodes. We use a similar `json` syntax to represent virtual paths, where the `Bandwidth` attribute corresponds to the maximum available bandwidth that may be scheduled on this path and the `Latency` attribute represents the corresponding end-to-end latency.

For example, listing 4.2 illustrates the end-to-end Virtual Paths exposed to the scheduler in figure 4.2.

Listing 4.2 – Virtual paths exposed to the scheduler in figure 4.2

```
1   PathList = [
2       {
3           "Type": "Path",
4           "Source": "ComputeNode1",
5           "Target": "ComputeNode2",
6           "Attributes": {
```

```
7               "Latency": 0.37,
8               "Bandwidth": 100
9           }
10      },
11      {
12          "Type": "Path",
13          "Source": "ComputeNode1",
14          "Target": "ComputeNode3",
15          "Attributes": {
16              "Latency": 0.33,
17              "Bandwidth": 100
18          }
19      },
20      {
21          "Type": "Path",
22          "Source": "ComputeNode2",
23          "Target": "ComputeNode3",
24          "Attributes": {
25              "Latency": 0.26,
26              "Bandwidth": 150
27          }
28      }
29  ]
```

4.2.3 Bandwidth Reservations

In this section, we present how tenants may schedule shared network resources with bandwidth reservations in the cloud through our proposed two-tiered platform. In particular, our platform supports heterogeneous resource requests, and tenants may submit their requests for network resources concurrently with their hardware resource requests.

Listing 4.3 illustrates an example scheduling with bandwidth reservations. A tenant requests two containers with 1 core per container and 100 Mbit/sec available bandwidth between the two containers.

Listing 4.3 – Scheduling with bandwidth resources example

```
1   { "Allocation": [
2       {
3           "Group": "g0",
4           "Type": "Machine",
5           "Attributes": {
6               "Cores": 1
7           }
8       },
9       {
10          "Group": "g1",
```

```
11            "Type": "Machine",
12            "Attributes": {
13                "Cores": 1
14            }
15        },
16        {
17            "Type": "Link",
18            "Attributes": {
19                "Source": "g0",
20                "Target": "g1",
21                "Bandwidth": 100
22            }
23        }
24  ]}
```

As we have stated in §4.2.2, IRM-NET hides the network topology from the scheduler, exposing only the end-to-end virtual paths between compute nodes. Algorithm 4.1 specifies how IRM-NET computes the `Bandwidth` attribute of each virtual path—in summary, by monitoring all the links that compose the path and picking the least available bandwidth reported by any one of them. IRM-NET then fills a matrix with the available bandwidth of all the virtual paths and provides it to the scheduler.

Algorithm 4.1: Exposing end-to-end bandwidth to the scheduler

Data: Set of virtual paths $\mathbb{P} = \{P_1, P_2, \ldots\}$ between each pair of compute nodes
Result: Expose the available end-to-end bandwidth of each path in $\mathbb{P}$ to the scheduler

1 $\mathsf{min} \leftarrow \infty$ `/* bandwidth of bottleneck link */`
2 **foreach** *path* $P \in \mathbb{P}$ **do**
3   **foreach** *link* $l \in P$ **do**
4         $b \leftarrow$ available bandwidth of link l
5   **if** $b < \mathsf{min}$ **then** `/* l is the potential bottleneck link of P */`
6             $\mathsf{min} \leftarrow b$
7   **if** $\mathsf{min} < 0$ **then** $\mathsf{min} \leftarrow 0$ `/* redundant; needed for mechanism in §4.2.5 */`
8   expose to the scheduler min available bandwidth for path P

When the scheduler decides on a new bandwidth reservation of b Mbit/sec between two containers C_1 and C_2, IRM-NET iterates over the links composing the path between C_1 and C_2 and deducts b from their respective `Bandwidth` attributes, as illustrated in algorithm 4.2. The inverse process, i.e., adding b, is applied when the scheduler releases a bandwidth reservation. IRM-NET executes algorithm 4.1 after every new or released bandwidth reservation, as any reservation/release may affect the available bandwidth of multiple paths. To enforce the bandwidth reservation, our implementation uses the linux `tc` tool to throttle traffic egressing C_1 and destined for C_2 to b Mbit/sec and vice-versa.

Algorithm 4.2: Creating a new bandwidth reservation

Data: Container C_1 scheduled on compute node N_1
Data: Container C_2 scheduled on compute node $N_2 \neq N_1$
Data: Path P between compute nodes N_1 and N_2
Data: Bandwidth reservation b
Result: Reserve bandwidth b on path P

1 **forall** *links* $l \in P$ **do**
2   reduce available bandwidth of l by b
3 run algorithm 4.1
4 throttle egress traffic $C_1 \rightarrow C_2$ on container C_1 to b
5 throttle egress traffic $C_2 \rightarrow C_1$ on container C_2 to b

By default, the egress traffic on the allocated containers is throttled to a default minimum of 10 Mbit/sec, by virtue of a default, low-priority rule installed by the `tc` tool. Therefore, basic connectivity is allowed even in the absence of bandwidth reservations and higher-priority rules installed through algorithm 4.2 may allow a higher throughput. Nevertheless, this bandwidth is not guaranteed, as no reservation is being registered and the aforementioned algorithms are not executed.

4.2.4 Latency Constraints

In this section, we present how tenants may schedule shared network resources with latency constraints in the cloud through our proposed two-tiered platform. Listing 4.4 illustrates an example of scheduling with latency constraints. A tenant requests two containers with 1 core per container, with the constraint that the end-to-end latency between the two containers must not exceed 2 milliseconds. Latency constraints are specified in a `Constraints` list, which is separate from the `Allocation` list, as latency is not a resource that can be reserved, but a constraint that needs to be honored.

Listing 4.4 – Scheduling with latency constraints example

```
1  { "Allocation": [
2      {
3          "Group": "g0",
4          "Type": "Machine",
5          "Attributes": {
6              "Cores": 1
7          }
8      },
9      {
10         "Group": "g1",
11         "Type": "Machine",
12         "Attributes": {
13             "Cores": 1
```

```
14            }
15        }
16 ],
17 "Constraints": [
18        {
19            "Source": "g0",
20            "Target": "g1",
21            "ConstraintType": "<=",
22            "Latency": 2.0
23        }
24 ]}
```

As we have stated in §4.2.2, IRM-NET communicates to the scheduler the end-to-end latency of each virtual path through the `Latency` attribute. Algorithm 4.3 specifies how IRM-NET computes this attribute—in summary by adding the latencies of the links that compose the given path. IRM-NET then fills a proximity matrix with the latencies of all the virtual paths and provides it to the scheduler.

Algorithm 4.3: Exposing end-to-end latency to the scheduler

Data: Set of virtual paths $\mathbb{P} = \{P_1, P_2, \ldots\}$ between each pair of compute nodes
Result: Expose the end-to-end latency of each path in $\mathbb{P}$ to the scheduler

```
1 foreach path P ∈ ℙ do
2     sum ← 0                                         /* path latency */
3     foreach link l ∈ P do
4         d ← end-to-end latency of link l
5         sum ← sum + d
6     expose to the scheduler sum end-to-end latency for path P
```

4.2.5 Oversubscription

In this section, we describe a general network-management policy supported by IRM-NET, bandwidth oversubscription.

Static Oversubscription

One approach to oversubscription is to introduce a *global oversubscription factor* $\alpha \in (0,1]$ and substitute line 2 in algorithm 4.2 with “reduce available bandwidth of link l by $\alpha \cdot b$”. This approach ensures that: (i) more bandwidth reservations may be scheduled over the same network link, by reserving less bandwidth than actually requested; (ii) tenants are always guaranteed a fraction of their requested bandwidth; and (iii) tenants may still use the entire bandwidth they requested, if it is available.

This approach is simple to implement, but has an important shortcoming: a single oversubscription factor is unlikely to fit all applications; for instance, a smaller factor (more tenants) would be more suitable in a cluster with low-throughput applications, whereas a higher oversubscription factor (less tenants) would be more suitable in a different cluster with high-throughput applications.

Dynamic Oversubscription

An alternative approach is to measure the bandwidth that is actually consumed on each link and report that to the scheduler—as opposed to reporting the bandwidth that is theoretically available based on the active reservations. Algorithm 4.4 specifies how this works. In summary:

1. Bandwidth reservations are conducted exactly as described in §4.2.3; no oversubscription factor is used.

2. A new, periodical bandwidth-measurement mechanism is added that is repeated every 5 seconds and described in the next steps.

3. For every bandwidth resource B that has been allocated between any two containers, the actual consumed bandwidth M is measured over a period of 3 seconds.

4. A minimum threshold of $T = 10$ Mbit/sec is always guaranteed, if the consumed bandwidth is measured lower than T.

5. The difference $D = B - M$ is calculated between the allocated (nominal) bandwidth B and the measured (actual) bandwidth M.

6. This difference D is added to every link within the corresponding virtual path, representing that more bandwidth is now available for reservation. If $D < 0$, it will be subtracted, instead, representing that less bandwidth is available for reservation.

7. Bandwidth throttling on the scheduled containers is never affected at any point.

A few notes regarding the aforementioned algorithm:

- The difference $D = B - M$ between the nominal (B) and the measured (M) bandwidth of a bandwidth reservation is expected to be positive during the first iteration, i.e., to consume less than the requested bandwidth.

- The `Bandwidth` attribute of a network link may report a negative value. This may occur if, for example, multiple tenants have been oversubscribed on a link and an application exhibits a sudden bandwidth burst. A negative `Bandwidth` value on a link indicates that bandwidth oversubscription is currently taking place on that link.

Algorithm 4.4: Dynamic bandwidth oversubscription

Data: Set of virtual paths $\mathbb{P} = \{P_1, P_2, \ldots\}$ between each pair of compute nodes
Data: Threshold T of minimum guaranteed bandwidth
Result: Measure bandwidth consumptions and update available bandwidth

```
1 forall reserved bandwidth resources B do                    /* B: nominal bandwidth */
2     (C1, C2) ← container end-points of bandwidth reservation
3     P ← virtual path ∈ ℙ between C1 ↔ C2
4     M ← measured bandwidth between C1 ↔ C2
5     if M < T then M ← T                       /* at least T bandwidth is guaranteed */
6     D ← B − M                            /* difference between measured and nominal */
7     foreach link l ∈ P do
8         bandwidth ← bandwidth + D                               /* D may be negative */
9 run algorithm 4.1 to update the end-to-end available bandwidth of all paths in ℙ
```

- The end-to-end available bandwidth exposed to the scheduler is always non-negative, even if a network link reports negative available bandwidth, as already illustrated in algorithm 4.1.

4.2.6 Fair Sharing of the Network

In this section, we describe another general network-management policy supported by IRM-NET, fair sharing of the network.

To demonstrate flexibility in supporting different network-management policies, IRM-NET supports proportional bandwidth sharing at the link level, as proposed in FairCloud [48]. When a cloud provider offers proportional bandwidth sharing, tenants do not make bandwidth requests; IRM-NET allocates bandwidth to tenants as specified in Algorithm 4.5.

This is a simple algorithm—just a proof-of-concept—but it can be extended relatively easily to provide more sophisticated policies, such as: (i) Tenants requesting a desired minimum bandwidth between their resources; if a path had surplus bandwidth, then less bandwidth would be allocated to a virtual path, otherwise either the current scheme would be enforced or the request would be rejected, depending on the policy of the cloud provider. (ii) Tenants specifying their own weights W_X and W_Y to indicate the importance of each requested path; line 6 in algorithm 4.5 would change to $\frac{W_X}{N_X} + \frac{W_Y}{N_Y}$.

4.3 Experimental Evaluation

In this section, we report four cloud deployment scenarios using shared network resources, which are currently not supported by traditional cloud computing platforms:

Algorithm 4.5: Fair sharing of the network

Data: Network links of cloud platform

Data: Set of virtual paths $\mathbb{P} = \{P_1, P_2, \ldots\}$ between each pair of compute nodes

Result: Determine bandwidth to be allocated to each virtual path

```
1  forall network links l do
2  |   sum_l ← 0                                         /* running sum of path weights */
3  |   foreach path in {P_{X−Y} ∈ ℙ | l ∈ P_{X−Y}} do   /* all paths on link l */
4  |   |   N_X ← number of containers X is communicating with over l
5  |   |   N_Y ← number of containers Y is communicating with over l
6  |   |   W_{P_{X−Y},l} ← 1/N_X + 1/N_Y                 /* weight of the path on link l */
7  |   |_  sum_l ← sum_l + W_{P_{X−Y},l}
   |_
8  forall virtual paths P_{X−Y} do
9  |   min ← ∞                                           /* bandwidth of bottleneck link */
10 |   foreach link l ∈ P_{X−Y} do
11 |   |   C ← capacity of link l
12 |   |   B_l ← C · W_{P_{X−Y},l} / sum_l               /* bandwidth allocated to P_{X−Y} on link l */
13 |   |   if B_l < min then                             /* l is the potential bottleneck link of P_{X−Y} */
14 |   |   |_ min ← B
   |   |_
15 |   throttle egress traffic X → Y on container X to min
16 |_  throttle egress traffic Y → X on container Y to min
```

1. **Scheduling with Bandwidth Reservations:** (section §4.3.3) This scenario exhibits the advantages of resource-agnostic scheduling when deploying distributed applications on the cloud. In this case, tenants may reserve bandwidth during resource scheduling, which impacts where the requested containers will be assigned.

2. **Scheduling with Latency Constraints:** (section §4.3.4) This scenario demonstrates the benefits of resource-agnostic scheduling when deploying latency-sensitive applications on the cloud. In this scenario, tenants may designate end-to-end latency constraints during resource scheduling, which influences where jobs will be deployed.

3. **Scheduling with Oversubscription:** (section §4.3.5) This scenario illustrates the advantages of bandwidth oversubscription when deploying distributed applications on a multi-tenant cloud platform. In this scenario, multiple tenants are concurrently requesting bandwidth on a congested platform, which will affect how many tenants will be admitted, as well as the performance experienced by the end-users.

4. **Enforcing Fairness:** (section §4.3.6) This scenario presents the benefit of added flexibility by integrating a bandwidth management mechanism to enforce proper bandwidth allocation across different tenants. In this scenario, two tenants are deploying a

bandwidth-intensive application over a common network link and their performances will be benchmarked in the presence and in the absence of the mechanism described in section §4.2.6.

4.3.1 Testbed

We have deployed our platform on *Grid'5000*[1], which provides:

- A large-scale research testbed with over 1000 nodes, allowing us to experiment with realistic multi-tenant scenarios.
- Node grouping in homogeneous, isolated clusters spread out across 9 different sites within France, as shown in Figure 4.3 [22].
- A core network interconnecting the various clusters together, used concurrently by multiple tenants.

Therefore, Grid'5000 enables us to experiment with variable network conditions (inter- and intra-site connections respectively) without needing to simulate them.

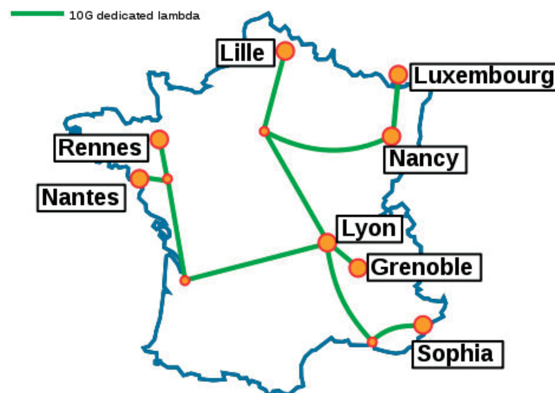


Figure 4.3 – Grid'5000 network backbone

Table 4.1 lists the hardware specifications of the clusters which we have been used to conduct our experiments in this chapter.

4.3.2 Applications

In this section we present the applications which we have used in our evaluation scenarios to benchmark the performance and the benefits of our platform:

1. **AdPredictor**, used in the scenarios presented in sections §4.3.3 and §4.3.5.

[1] http://www.grid5000.fr

Location	Cluster	Model	Nodes	CPU	Cores	Threads	RAM	Network
Rennes	paravance	Dell Poweredge R630	72	Intel Xeon E5-2630v3	16	32	128 GiB	10 Gigabit Ethernet
Rennes	paranoia	Dell Poweredge C6220	8	Intel Xeon E5-2660v2	20	40	128 GiB	10 Gigabit Ethernet
Nantes	econome	Dell Poweredge C6220	22	Intel Xeon E5-2660	16	32	64 GiB	10 Gigabit Ethernet

Table 4.1 – Cluster specifications within Grid'5000

2. **WikiBench**, used in the scenario presented in §4.3.4.
3. **iperf**, used in the scenario presented in §4.3.6.

AdPredictor

AdPredictor [20] is a machine learning algorithm from a class of contemporary industrial-oriented applications, commonly known as *recommender systems*, addressed to end-users utilizing on-line services. For instance, one such service is Last.fm [31], a free online music catalog that recommends related music and events based on user interests. Similar such services include Google [19] and Bing [37], two commercial on-line search engines that produce commercial product recommendations to end-users, based on the end users' queries. In recommender systems, items or services are paired with end-users and, due to the constant, overwhelming data generation, such computations are usually deployed in large, distributed data-centers.

For the purposes of our evaluation, we have used a hadoop-based version of the AdPredictor algorithm to serve as an example of a large-scale distributed application that may be deployed on a cloud platform. AdPredictor processes session logs from commercial search engines to predict the click-through rate of commercial advertisements. We have deployed this implementation using hadoop version `1.2.1`.

Dependent variables We have evaluated the performance of AdPredictor by measuring the following dependent variables: (hereinafter: *metrics*)

1. *Execution time:* the total time needed to execute the application, not measuring the required time needed to upload the data-set to the Hadoop Distributed File System (HDFS).
2. *Throughput:* the throughput of a worker container, over time, measured every 100 milliseconds (§4.3.3) or every 1 second (§4.3.5). This traffic may be categorized either as: (i) *HDFS traffic*, resulting from I/O operations between the map-reduce tasks and the

local hard disk; or (ii) *shuffle traffic*, resulting from data being transferred between map and reduce tasks. This metric will also be referred to as the *instantaneous throughput*.

Data-set We have utilized session logs of a web search engine [55] to evaluate the performance of AdPredictor. This data-set contains approximately 150 million training instances (9.9 GiB), derived from interactions between end-users and the search engine.

WikiBench

WikiBench [61] is a framework designed to benchmark web-hosting systems, such as application servers or cloud computing platforms. This framework is deployed on client-server architectures and operates as follows:

- The system being evaluated, hereinafter called the *wikipedia server*, is hosting MediaWiki [64], a real major web application.
- The wikipedia server is modeling a real data center server, as an actual database dump of a wikipedia website [63] has been deployed on the MediaWiki application.
- A real data-set is utilized to benchmark the system, by replaying traces addressed to `wikipedia.org` [57]. The WikiBench framework is deployed on each *client* and is tasked to replay the access trace by sending each request to the designated wikipedia server and evaluate the system performance, as described further below.

Dependent variables The WikiBench framework evaluates the performance on the system by measuring the following dependent variables (hereinafter: *metrics*) on the *client* side:

1. *Miss rate:* the percentage of the requests that were skipped by the benchmark. The WikiBench benchmark is observing the timestamp differences between consecutive requests, skipping intermediate requests as necessary.

   Example: let R_1, R_2 and R_3 denote three consecutive requests of a single client, with respective original timestamps $t_1 < t_2 < t_3$. Let Δt_1 denote the time required to complete the request R_1. If $t_2 - t_1 < \Delta t_1 < t_3 - t_1$, then the request R_2 will be skipped and the benchmark shall proceed directly to request R_3. In order to respect the timestamp differences, the benchmark shall stall for Δt_s seconds, so that $\Delta t_1 + \Delta t_s = t_3 - t_1$.

2. *RTT:* the Round-Trip-Time between issuing a request and receiving a successful response, including the payload. Only responses returning `HTTP 200 OK` are filtered, as they usually bear a noticeable payload compared to responses such as `HTTP 404 Not Found` or `HTTP 301 Moved Permanently`, therefore exhibiting a higher RTT due to the increased transmission delay on the wikipedia server side.

Bottleneck challenges To successfully evaluate our system, we had to overcome a series of challenges to verify that the performance bottleneck would be the network, instead of I/O operations. In particular, we identified an I/O bottleneck when issuing a page request to our wikipedia server for the first time, where the Round-Trip-Time (RTT) was measured 1 – 2 orders of magnitude higher than subsequent requests to the same page. This indicated that a bottleneck was existing during an access request to the MySQL database, which was lifted when the page was subsequently cached by the database.

In order to properly identify this bottleneck, we set up the English wikipedia dump on our server and issued, from a single client, requests addressed to various pages hosted the server. Two batches of 100 requests each were issued for each page using the linux `ab` benchmarking tool, with a maximum of 10 requests running concurrently within a single batch. The requests' RTTs were measured during the process. The second batch of 100 requests was issued after the first one had been fully completed to measure whether the MySQL caching had an impact on the RTT.

Batch	RTT (msec)			
	50%	80%	90%	100%
First	160	328	18,169	19,893
Second	159	315	342	373

Table 4.2 – Round-Trip-Time of two batches of 100 requests to the wikipedia server

Table 4.2 presents our measurements on the requests issued to retrieve page `/index.php/42`. During the first batch, the 90% and 100% RTTs[2] were measured about 55 – 60 times higher than the corresponding 80% RTT. Nevertheless, there was no such discrepancy during the second batch, where the maximum RTT was comparable to the 80% RTT of the first batch. This clearly indicates that during the second batch, the requested page had been cached within the MySQL database, which significantly improved the RTT time.

Only, after applying a series of MySQL optimizations were we able to shift the bottleneck from the MySQL database to the network. These optimizations included: (i) file caching, which stores the rendered HTML pages to files on the local disk; (ii) page compression, so that the requested page may be sent faster to the network; (iii) disabling page-view counters and running MediaWiki on "miser" mode. Under these circumstances, the throughput rate was measured as high as 909 Mbit/sec, on a 1 Gbit link, indicating that the requested pages were being transferred essentially at line rate. Therefore, we decided to address this challenge by permanently caching all pages on the local hard disk, which effectively resulted in completely bypassing the MySQL database.

[2] we are referring to the corresponding data-points of the cumulative distribution function

Server configuration We have hosted the Danish wikipedia dump of August 01, 2016 [65] on MediaWiki `1.26.2`, running on our wikipedia server. We have chosen the Danish dump due to its considerable smaller size compared to the respective English dump. Table 4.3 lists the size of the aforementioned wikipedia dump files. As previously mentioned, the MySQL database is effectively bypassed, in order to eliminate any I/O bottlenecks, by caching all articles directly on the hard disk.

Language	Compressed	Uncompressed	Articles
Danish	243 MiB	1.03 GiB	$\approx 210,000$
English	12.2 GiB	54.2 GiB	$\approx 5,200,000$

Table 4.3 – Size of `pages-articles.xml.bz2`, containing the 2016-08-01 wikipedia dumps

Client data-set We have selected the September 2007 wikipedia access traces [58] to benchmark our system. These traces contain approximately 2.4 billion requests, corresponding to 10% of all user requests addressed to wikipedia during that month. From these requests we have isolated the 311,013 requests addressed specifically to the Danish wikipedia articles, under `da.wikipedia.org`, but not other pages such as talk pages. These requests will be collectively hereinafter referred to as the *wikipedia trace*, which will be replayed from the clients, addressed to our wikipedia server.

In addition, we have "sped up" our wikipedia trace in order to simulate a realistic workload on the server side. In particular, the original timestamps of the isolated requests to the Danish wikipedia span across a period of 12 calendar days (from 2007-09-19 to 2007-09-30), generating an average workload of 0.3 requests per second. We would obtain no meaningful measurements under this rate, since the WikiBench framework, which would send the client requests, observes the timestamp difference between two consecutive requests. Therefore, we have mapped the timestamps of the danish requests to the timestamps of the first 311,013 requests in the packet trace, which span over 10.3 minutes, effectively "speeding up" our trace to an average of 502.6 requests per second.

iperf

`iperf` [13] is a cross-platform benchmarking tool used to measure the maximum attainable bandwidth on an IP network. When an `iperf` application is deployed, a single container is be utilized as the *server*, hosting the `iperf` daemon, while the rest are be used as *clients*, opening and maintaining TCP `iperf` sessions issued to the `iperf` server.

4.3.3 Scenario 1: Scheduling with Bandwidth Reservations

In this section, we are exploring the scenario where a single tenant is deploying a bandwidth-sensitive distributed application on the cloud.

Experimental setup Our setup consists of 8 physical compute nodes; 4 nodes located in the *paranoia* cluster, and 4 nodes in the *econome* cluster. We have already presented the hardware specifications in Table 4.1. We have selected these two clusters as: (i) the physical compute nodes possess virtually identical hardware characteristics; (ii) the two clusters are located in different sites (Rennes and Nantes), which introduces a natural inter-cluster bandwidth bottleneck of approximately 200 Mbps. While both clusters offer gigabit connectivity, we have emulated bandwidth congestion within the *econome* cluster, using the linux `tc` tool, in order to evaluate the impact of bandwidth reservations on resource scheduling. Therefore, we have imposed a bandwidth threshold of 500 Mbit/sec on every physical machine within the aforementioned cluster. We assume that our platform is aware of the network conditions on each cluster.

Tenant request A single tenant is submitting an AdPredictor application request to the cloud platform with the following characteristics:

- Three containers are being requested, each one will host a single hadoop DataNode and a single TaskTracker. These containers will be referred to as the *worker* containers.
- Each worker container will host 10 concurrent map tasks and 2 reduce tasks. Therefore, 12 cores are requested for each worker. This ensures that no two workers will be scheduled on the same physical compute node, as there is no node with 24 cores.
- An extra container is also requested, which will host a single hadoop NameNode and a single JobTracker. This container will be referred to as the *master* container, for which a single core is requested. The position of the master container is not expected to affect the execution of the job, therefore we will assume for the rest of the scenario that it is always allocated on an available node in the *paranoia* cluster.

Data-set The entire data-set, as described in §4.3.2, has been used in this scenario.

Worker throughput In order to evaluate the bandwidth requirements of the AdPredictor application, we first measured the instantaneous throughput between two worker containers during its execution, while deployed on the *paranoia* cluster, i.e., in a cluster offering gigabit connectivity. Figure 4.4 reports our measurements on one of the three worker containers. We may observe a constant throughput of approximately 200 Mbit/sec to the hard disk at the conclusion of the experiment, as well as 1 Gbit/sec traffic bursts throughout the experiment as a result of the data being “shuffled” between the map and the reduce taks.

Link capacity Following our observations in figure 4.4, we would expect to observe degradation on the execution time, should AdPredictor run on a link with a lower capacity than

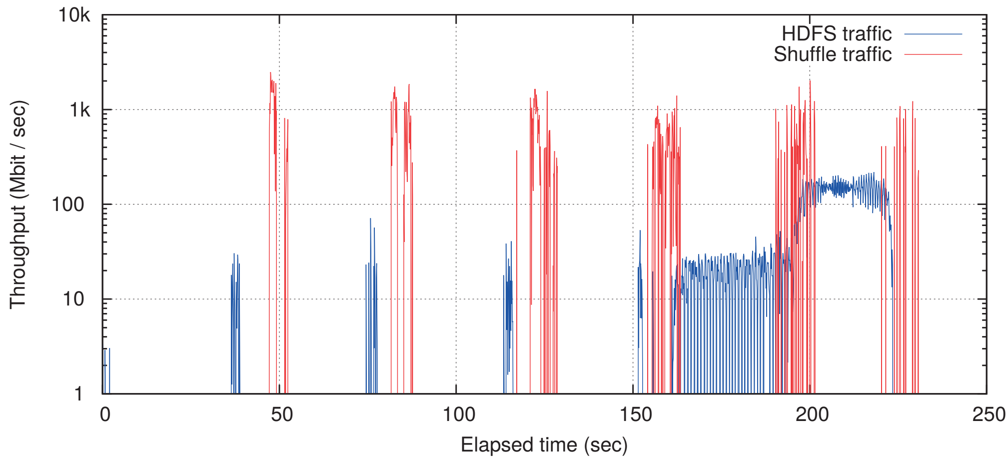


Figure 4.4 – AdPredictor worker throughput over time

200 Mbit/sec. To validate our intuition, we subsequently measured the execution time while varying the underlying link capacity between all worker containers. To achieve that, we deployed AdPredictor on the *paranoia* cluster and used the linux `tc` tool to simulate a lower link capacity between all worker containers. Figure 4.5 illustrates the results of our measurements, where we have detected a "knee" point at a link capacity of ~ 80 – 200 Mbit/sec. All executions running on links slower than 80 Mbit/sec showed a severe increase in their execution time, being consistent with our measurements in figure 4.4. On the contrary, no execution running on links faster than 200 Mbit/sec exhibited any improvement compared to the execution running 200 Mbit/sec links.

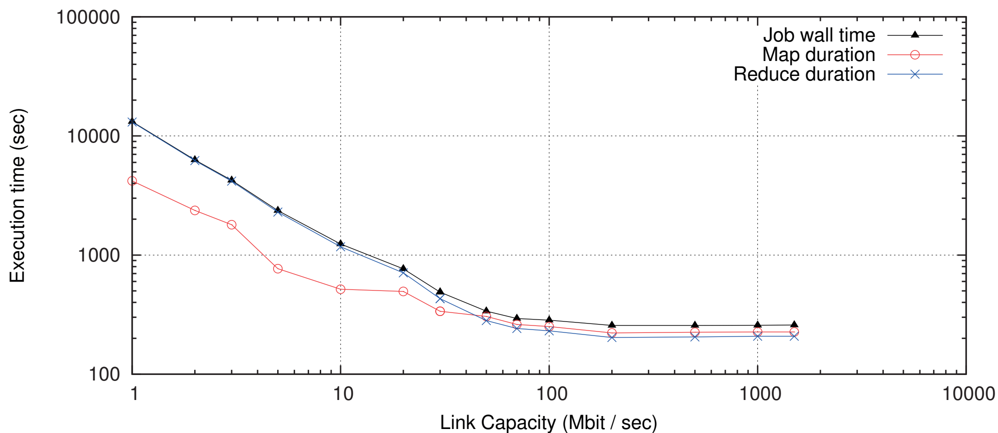


Figure 4.5 – AdPredictor execution time over link capacity

Scheduling without bandwidth reservations Our main evaluation scenario consists of a tenant submitting a job request for 3 container workers, as previously described, without providing any bandwidth reservations. This experiment has been repeated 20 times. Figure 4.6

reports the measured execution time as a function of the final placement of the container workers. There are 4 different placements of the 3 worker containers, labeled as *A – B*, with *A* denoting the number of containers scheduled in the *paranoia* cluster and *B* denoting the number of containers scheduled in the *econome* cluster. Over 20 experiments, there was an approximately equal probability for a container to be scheduled in either cluster. Also visible, the duration of the individual map and reduce phases, as well as the standard deviation of all iterations that resulted in the given placement.

It is evident that scheduling without bandwidth reservations may affect the performance of bandwidth-sensitive distributed applications. The execution time of AdPredictor exhibits a gradual improvement (lower duration) as more container workers are being scheduled in the "faster" *paranoia* cluster, with label *3 – 0* denoting the optimal placement where all containers have been scheduled on that cluster. Nevertheless, there is approximately only 7% probability[3] that the cloud scheduler will choose the optimal placement without bandwidth reservations. Therefore, end-users will inevitably experience variability in their performance, due to the unpredictability of the container placement.

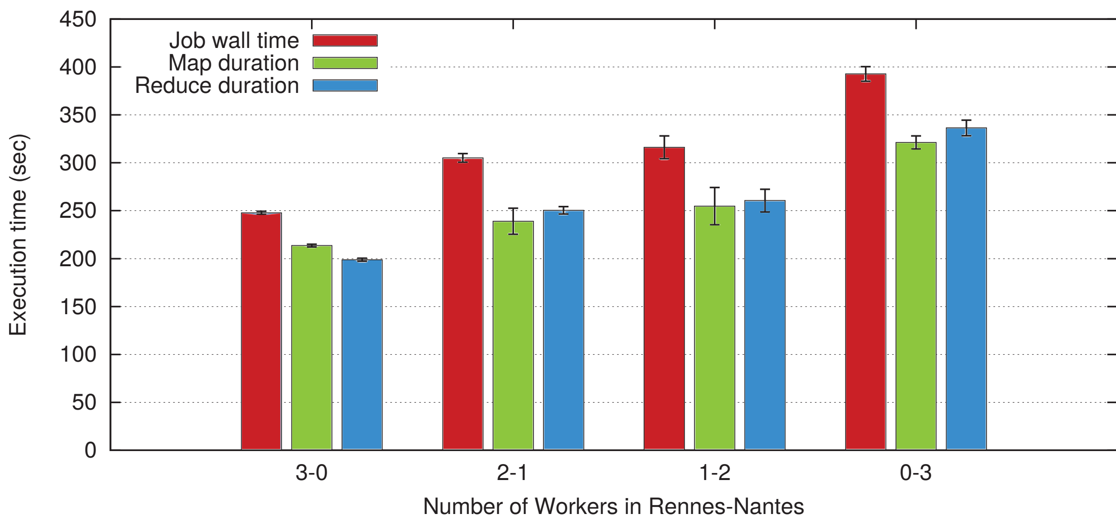


Figure 4.6 – AdPredictor execution time over cluster placement

Scheduling with bandwidth reservations In order to eliminate all suboptimal placements presented in figure 4.6, tenants have to specify bandwidth reservations during resource scheduling. In this case, we add three additional requests during job submission: 300 Mbit/sec between any two worker containers. Therefore, a container may only be scheduled on a compute node that has at least 600 Mbit/sec available bandwidth, as it would require two reservations of 300 Mbit/sec with each other container. Consequently, by including bandwidth reservations in our requests, all containers were eventually scheduled in the fast *paranoia* cluster, since there was not adequate bandwidth available in the slower *econome* cluster.

[3] the theoretical probability of choosing 3 nodes in the *paranoia* cluster is $\frac{C(4,3)}{C(8,3)} = \frac{1}{14}$

4.3.4 Scenario 2: Scheduling with Latency Constraints

In this section, we are exploring the scenario where multiple latency-sensitive applications are deployed on the cloud.

Experimental setup Our setup consists of 40 physical machines; 20 out of them located in the *paravance* cluster, situated in Rennes, and 20 located in the *econome* cluster, situated in Nantes. The wikipedia server is running on a separate physical machine within the *paravance* cluster. The RTT between two compute nodes on different clusters is, in this scenario, at least $1.2ms$.

Data-set The entire data-set, as described in §4.3.2, has been used in this scenario. A uniform distribution has been applied on the wikipedia trace to split the trace into 20 segments of approximately equal size; all together reconstruct the original trace. This will simulate the entire workload originating from multiple clients.

Tenant request A single tenant is submitting an AdPredictor application request to the cloud platform with the following characteristics:

- 20 *client* containers are being requested, which will host a WikiBench benchmark.
- As in section §4.3.3, 12 cores are requested for each container, to ensure that they will be scheduled on different physical compute nodes. Despite running a single fetcher on WikiBench, we assume that the tenant is requesting 12 cores to execute post-execution computations.
- After scheduling, each of the 20 data-set segments will be loaded on a different container, along with the WikiBench framework.
- The WikiBench framework will commence its execution simultaneously on all 20 containers, replaying the entire data-set by sending the requests to the wikipedia server. Each container utilizes a single thread to send requests in sequence, respecting the requests' timestamp differences.

This experiment was executed 100 times, each time resulting in a different placement of the requested 20 containers across the two clusters.

Scheduling without latency constraints We initially scheduled without latency constraints, where there was an approximately equal probability for a requested container to be scheduled in either cluster. A timeout threshold of 100 milliseconds has been set on the WikiBench framework: if the Round-Trip-Time (RTT) of a request is longer than 100 milliseconds, the

request will be immediately dropped and a "miss" is registered. We elected this threshold, as 100 milliseconds is the point that a request feels "instant" to the human user and 1 second where the end-user will notice a delay [43].

Figure 4.7 reports the miss rate distribution across all experiments, based on the originating cluster. The box plots extend until they encompass the 95% of the data-points, with any outliers also being depicted. We observe that only 2% of requests originating from the paravance cluster were dropped, whereas the respective miss rate of requests originating from the econome cluster was 16%.

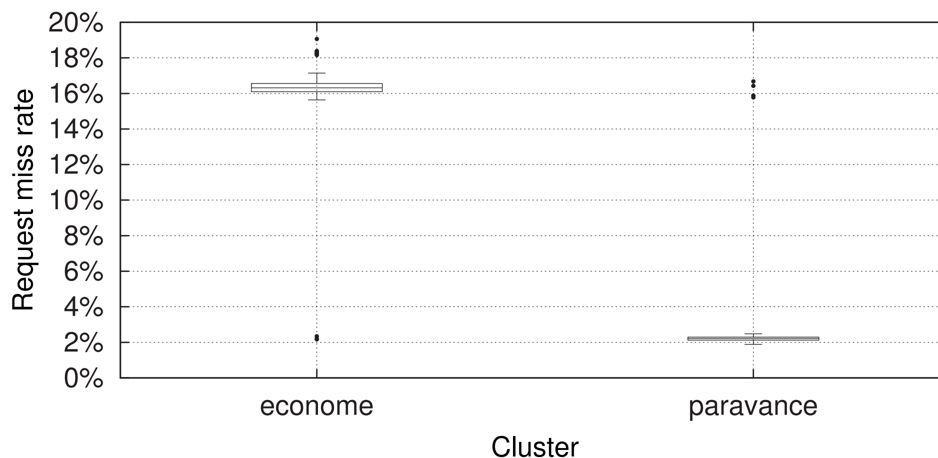


Figure 4.7 – WikiBench request miss rate, no latency constraints, based on originating cluster

Figure 4.8 reports the distribution of the Round-Trip-Time of every request, across all experiments, based on the originating cluster. The box plots extend until they encompass the 95% of the data-points, with any outliers omitted from the figure; The following scenarios are depicted: (a) scheduling with *no constraints*, requests originating from either cluster; (b) scheduling with *no constraints*, requests originating from the *econome* cluster; (c) scheduling with *no constraints*, requests originating from the *paravance* cluster; (d) scheduling *with latency constraints*, requests originating from either cluster. We obtained the distribution (d) when we subsequently executed all our experiments with latency constraints, as we describe below.

We observe that the RTT of requests originating from the *econome* cluster was about an order of magnitude longer than the corresponding RTT of requests originating from the *paravance* cluster.

Scheduling with latency constraints Based on our previous measurements, we repeated all experiments by providing maximum latency constraints between the requested containers and the wikipedia server in the *paravance* server. In particular, an end-to-end latency of 0.5 milliseconds (1 millisecond RTT) was provided; based on our observations in figure 4.8, this

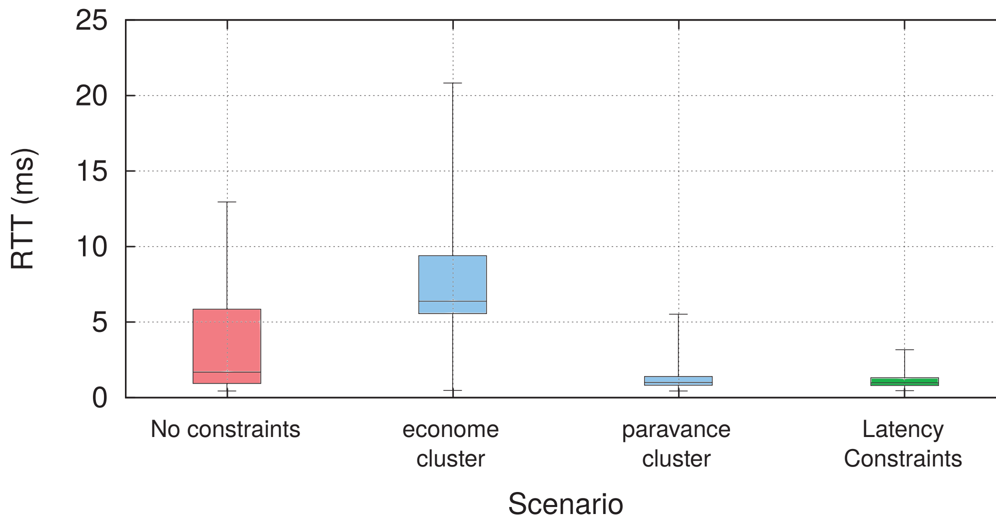


Figure 4.8 – WikiBench request Round-Trip-Time distribution, based on latency constraints

was adequate to exclude the *econome* cluster from consideration. This resulted in directly affecting the placement of the requested resources, which where all placed in the *paravance* cluster. The RTT distribution of all requests is illustrated in figure 4.8, while the miss rate was virtually identical to the behavior observed in figure 4.7.

4.3.5 Scenario 3: Scheduling with Oversubscription

In this section, we are exploring the scenario where multiple tenants are deploying a distributed application on a cloud platform with insufficient bandwidth to accommodate all requests.

Experimental setup Our setup consists of 2 physical compute nodes, both located in the *paravance* cluster, which may be reviewed in table 4.1. We have emulated congestion on the network link between the two compute nodes using the linux `tc` tool, limiting its capacity to 100 Mbit/sec. We assume that our platform is aware of the link's capacity.

Tenant requests Multiple tenants are submitting the same AdPredictor application request to the cloud platform with the following characteristics:

- Two *worker* containers, as defined in section §4.3.3 are being requested.
- Each worker container will host 1 map task and 1 reduce task. Therefore, 2 cores are requested for each worker.
- We will also assume that the two worker containers cannot be allocated on the same physical compute node. This may be implemented through, e.g., a request for a specific

hardware resource. This assumption is undertaken to evaluate the performance of the job when data is being transferred over an over-subscribed network link.

- A bandwidth constraint of 100 Mbit/sec between the two worker containers is also requested, which coincides with the total capacity of the link.

- As in §4.3.3, a *master* container is also requested, which we will not examine in this scenario, as it does not affect the execution of the job.

Data-set We have chosen to limit the data-set to 10 million entries (680 MiB), as previously described, on the grounds that it offered the most suitable trade-off between: (i) producing an impactful throughput in the order of magnitude of at least 100 Mbit/sec, which required a bigger data-set; and (ii) being able to concurrently store multiple copies of the data-set, the intermediate output of the map tasks and the final output in memory, as we will discuss further below, which required a smaller data-set. This reduced data-set produces a behavior similar to the one reported in figure 4.4. The peak *shuffle* bandwidth has been measured within 100 – 120 Mbit/sec.

Scheduling without oversubscription The cloud platform will only accept a single tenant request, since a single request will reserve the entire capacity of the network link between the two compute nodes. Consequently, any further job submissions are rejected. Nevertheless, as we have already illustrated in figure 4.4, although the throughput between two workers may reach, or exceed, 100 Mbit/sec, there is no continuous data transfer between two worker containers. This leads to the link being severely under-utilized; as we may observe in figure 4.9, link utilization is only 14%, defined as:

$$\text{utilization} = \frac{\text{data transfered}}{\text{capacity} \cdot \text{time}} \cdot 100\%$$

Scheduling with oversubscription In order to boost the utilization of the network link, we have scheduled the aforementioned requests with bandwidth oversubscription. We have deployed the static oversubscription algorithm presented in section §4.2.5.

A maximum of 8 requests may be concurrently accepted, thus 8 worker containers per compute node, as 2 cores have been requested per container and each compute node has 16 cores. We have therefore executed 8 different experiments, each time decreasing the oversubscription factor, until all 8 tenants have been concurrently scheduled on the cloud. Since the entire link capacity is requested by every tenant, it is evident that only 1 tenant may be scheduled with $\alpha = 1$, 2 tenants with $\alpha = \frac{1}{2}$ and 8 tenants with $\alpha = \frac{1}{8}$. Each experiment for a given value of the oversubscription factor α was repeated 20 times.

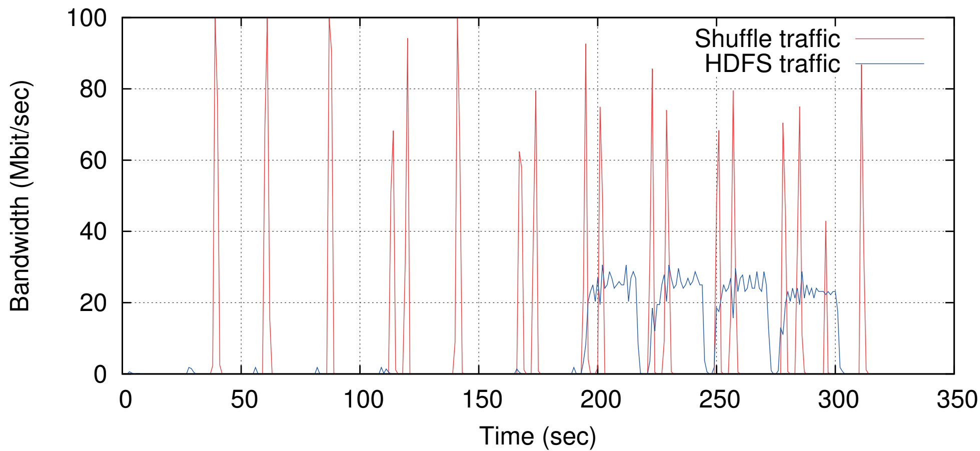


Figure 4.9 – AdPredictor worker throughput over time; single tenant

Figure 4.10 illustrates the distribution of the instantaneous throughput of every worker container, for all executions of a given tenant admission threshold. The box plots extend until they encompass the 95% of the data-points, with any outliers omitted from the figure; therefore the top box bar corresponds to the 97.5$^{\text{th}}$ percentile. We may observe that the median throughput is, as expected, degrading as we are scheduling more concurrent tenants, but up to $N = 3$ tenants the degradation does not exceed 5.9% compared to the $N = 1$ case. A similar degradation may be observed for the 75$^{\text{th}}$ throughput percentile and the 97.5$^{\text{th}}$ percentile, which do not exceed 8.9% and 5%, respectively. We may therefore conclude that, if the cloud provider accepts 3 concurrent tenants, the performance degradation will not be noticed by the end-users.

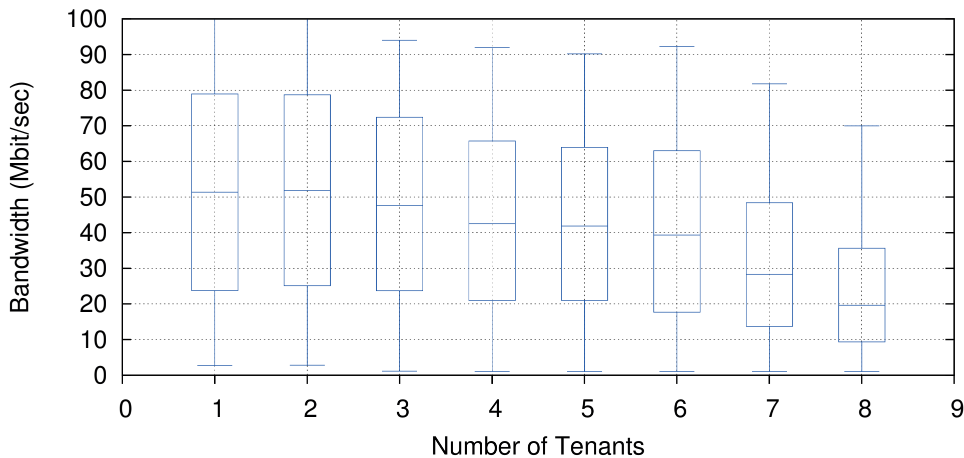


Figure 4.10 – AdPredictor "shuffle" traffic throughput over concurrently admitted tenants

On the other hand, as the number of concurrent tenants increases, the end-users will unavoidably notice the performance degradation, as the median, the 75$^{\text{th}}$ and the 97.5$^{\text{th}}$ bandwidth

percentiles experience a 61%, 58.2% and 30% decline, respectively, when $N = 8$ users are admitted, in comparison to the single-tenant admission.

Figure 4.11 illustrates the distribution of the execution time of every AdPredictor deployment, for a single execution of a given tenant admission threshold. For $N = 3$ tenants, the median execution time increases by 9.09%, compared to the single-tenant case, while for $N = 8$ tenants, the same increase is 32.9%.

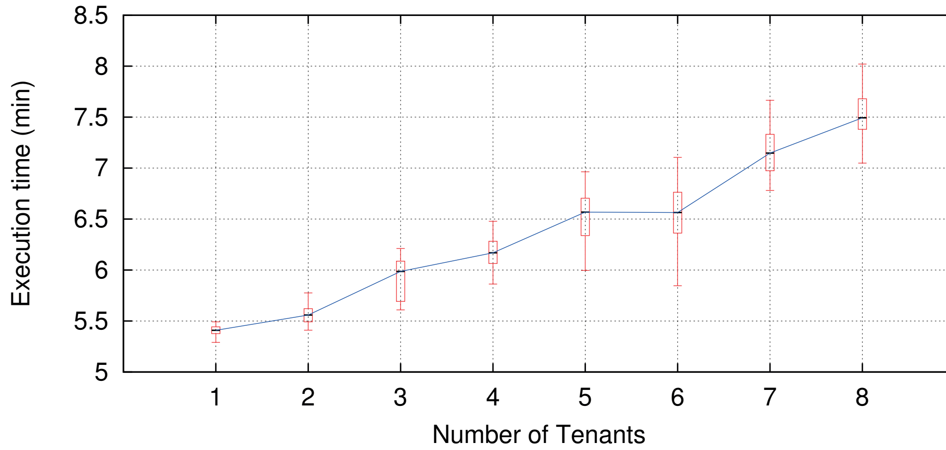


Figure 4.11 – AdPredictor execution time over concurrently admitted tenants

Figure 4.12 reports the link utilization for a single execution of a given tenant admission threshold. It is evident that, while the individual tenant performance exhibits degradation, the link utilization is enhanced as more tenants are being concurrently admitted to the cloud platform.

Figure 4.12 – Link utilization over concurrently admitted tenants

Figure 4.13 illustrates the instantaneous "shuffle" bandwidth, during a single execution, when $N = 3$ tenants have been scheduled concurrently due to bandwidth oversubscription. This figure validates our measurements in figure 4.12, as, on average, more data is transferred over the network link within a 1 second interval.

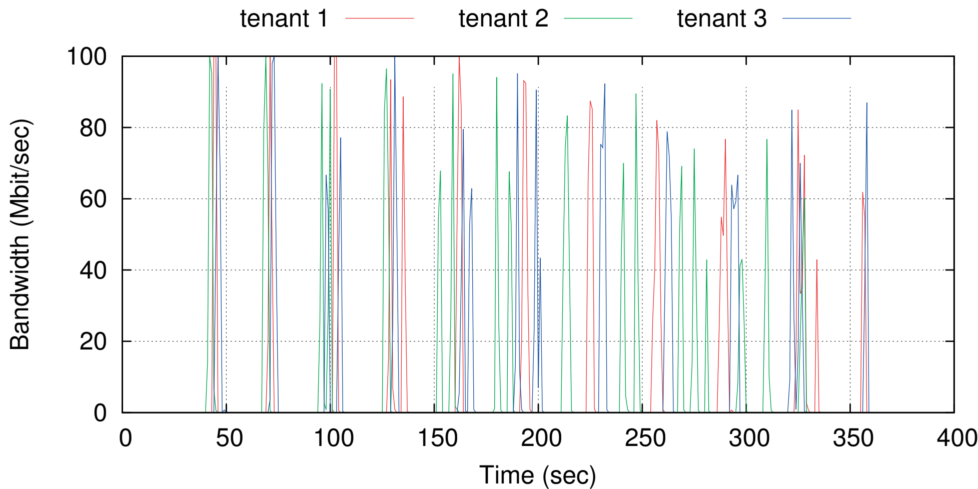


Figure 4.13 – AdPredictor "shuffle" throughput with bandwidth oversubscription, $N = 3$ tenants

To conclude, the cloud provider has a strong incentive to oversubscribe at least $N = 3$ tenants in this scenario, as it will double the link utilization with a relatively low impact on the individual performance of the tenants' jobs. The exact policy to be used to determine the trade-off between link utilization and tenant performance may be determined by the cloud provider.

Eliminating non-network-related bottlenecks Due to the concurrent executions of multiple distributed applications on the same physical compute node, we had run into a number of non-network-related bottlenecks that we had to address:

- *Hard disk:* we observed that hadoop is writing to the local hard disk not only the final output of the reduce tasks, but also the intermediate output of the map tasks. Preliminary measurements with multiple tenants indicated an I/O bottleneck, which affected our measurements. To address this issue, we decided to mount the entire Hadoop Distributed File System (HDFS) on the memory, including the data-set and the final output.
- *CPU affinity:* we observed that hadoop does not enforce any core affinity on the running map or reduce tasks, a mechanism left to the linux scheduler. Therefore, it was commonplace to have the running map or reduce tasks being moved across multiple cores, resulting in the loss of the instruction pipeline, as well as the warm L1/L2 caches,

producing eventually inconclusive results. To address this issue, we developed a simple scheduler utilizing the linux `taskset` tool, to "pin" each map and reduce task on an unused CPU.

An alternative way to approach this issue would be to use Apache Hadoop YARN [5], a subsequent Hadoop version, as the design goal of YARN is to improve resource utilization by replacing map-reduce slots with containers [52].

4.3.6 Scenario 4: Enforcing Fairness

In this section, we are investigating the scenario where multiple tenants are contesting over limited bandwidth resources to deploy their bandwidth-intensive applications. The cloud provider is regulating the bandwidth provisioning between the tenants by operating the mechanism described in §4.2.6. We have conducted a preliminary evaluation to demonstrate the proof-of-concept of our proposal.

Experimental setup For the purposes of our evaluation, we have replicated the experimental setup demonstrated in §4.3.5. In this scenario, we have limited the capacity of the network link between the two compute nodes to $1,000$ Mbit/sec.

Tenant requests Two tenants, referred to as "Alice and Bob," are submitting the same `iperf` application request to the cloud platform with the following characteristics:

- Two containers are being requested; we will assume that the two containers cannot be allocated on the same physical compute node.
- An `iperf` server and the respective client will be deployed on the two containers, respectively.
- There are no specific bandwidth requests; the cloud provider will allocate as much bandwidth as possible based on the bandwidth allocation scheme.

Results Figure 4.14 illustrates the results of our preliminary evaluation, presenting the average bandwidth over the entire duration of each experiment. We are reporting the following use-cases:

1. Alice is the only tenant with allocated resources on the testbed, running her `iperf` application over the network link.
2. Both Alice and Bob have their resources allocated on the testbed, running their respective `iperf` applications over the same network link. No bandwidth management

mechanism is employed, thus the TCP congestion control will administer the bandwidth allocation between Alice and Bob.

3. Replicating use-case (2), with the aforementioned fairness mechanism in operation.

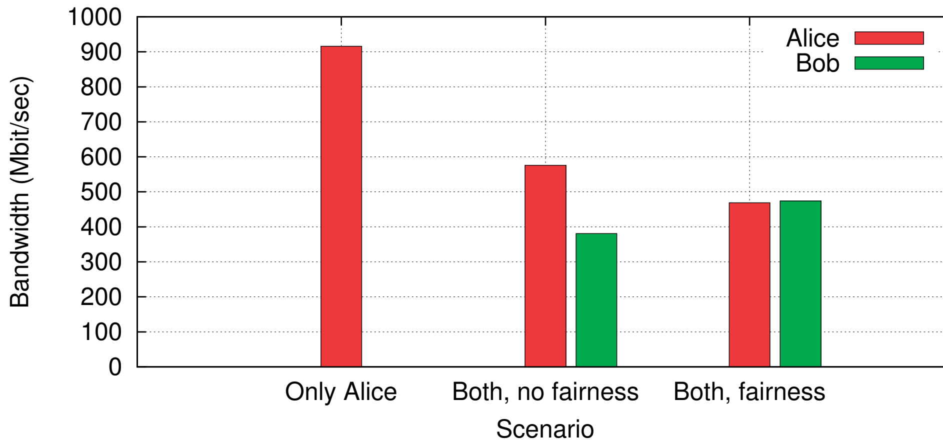


Figure 4.14 – Enforcing fairness on `iperf` applications

As expected, when Alice is the lone scheduled tenant, she receives the entire bandwidth of the network link. However, when both tenants are deployed, the TCP congestion control allocates more bandwidth to Alice's session, which had started at an earlier time than Bob's respective session. When we subsequently enable the fairness mechanism that both tenants are allocated approximately 500 Mbit/sec on the shared network link. This is the expected behavior, as both connections are allocated an equal weight $W_A = W_B = \frac{1}{N_X} + \frac{1}{N_Y} = \frac{1}{1} + \frac{1}{1} = 2$; therefore, half of the link's capacity is allocated to each connection. We conclude that in the absence of a "fair" bandwidth allocation mechanism, the TCP congestion control mechanism will tend to favor the existing connections over the newer ones, whereas a simple fairness enforcement will result in a proper bandwidth allocation between different tenants.

4.4 Summary

In this chapter, we proposed a two-tiered approach to scheduling cloud resources: first, a resource-agnostic layer makes scheduling decisions without relying on any resource-specific semantics; second, a resource-specific layer enforces these scheduling decisions through special infrastructure resource managers (IRMs), one per resource type. We focused on the IRM responsible for network resources (IRM-NET), which plays two roles: (a) it informs the resource-agnostic scheduler of the available bandwidth and latency of end-to-end paths between compute nodes and implements the scheduler's decisions; (b) it implements the cloud provider's general network-management policies, e.g., oversubscription or fair network sharing. To evaluate IRM-NET, we used both benchmarking tools and realistic applications. We experimented with scenarios where different tenants request bandwidth resources or

latency constraints, and with scenarios where the cloud provider supports oversubscription or fair network sharing. We showed that our approach enables cloud providers to (a) provide bandwidth and latency guarantees to tenants sharing the same network links, and to (b) implement useful general network-management policies.

5 Conclusions

In this thesis, we have studied the problem of managing shared network resources on multi-tenant cloud networks, in a way that exposes to tenants the illusion of an isolated network exclusively under the tenant's control.

Our work has been motivated by two challenges in resource management, stemming from the increased demand for application migration to the cloud in recent years. First, storing access-control rules (ACLs) within the operating system (OS) that hosts the tenant virtual machines (VMs) or containers, despite being convenient, introduces unnecessary performance overhead and security issues. Second, offering bandwidth and latency guarantees to end-users in the cloud poses profound challenges, as cloud providers are allowing increased heterogeneous resource support.

To address the challenge of storing ACLs in the cloud, we have developed a Top-of-Rack (ToR) switch architecture that exposes the abstraction of a two-layered virtual flow table, composed of: (i) a fast, but small data-path memory, acting as a flow rule cache; and (ii) a large, but slow backing store accessible from the switch's supervisor engine, which fits orders of magnitude more rules than the switch's physical flow table. We have showed that by deploying a simple, practical heuristic to store in the limited cache memory the rules that recently matched the most amount of traffic, we may attain fast packet-forwarding performance at line-rate by keeping up to 95% of the traffic in the data-path. Our approach maintains the correct forwarding behavior by resolving any challenges that rise from caching wildcard rules.

Furthermore, we have also proposed a two-tiered approach to scheduling cloud resources, consisting of: (i) a resource-agnostic scheduling layer; and (ii) a resource-specific layer enforcing the scheduling decisions. We have demonstrated that our architectures facilitates cloud providers in providing bandwidth and latency guarantees to tenants sharing the same network links, and in implementing useful general network-management policies, such as bandwidth oversubscription or fair network sharing.

We consider the outcome of our extensive evaluations to be a favorable development for all the

growing efforts in cloud computing: we conclude that when designing resource management abstractions, flexibility does not necessarily preclude, but may facilitate performance.

A Flow Rule Definitions and Notations

In this appendix we present the definitions and notations that we utilize throughout chapter 3. All notations that we introduce are briefly illustrated Table A.1.

Notation	Representation
R	Flow Rule
d	Number of fields in Flow Rule
F_i	i-th field of Flow Rule, $i \in [0, d]$
$\mathbb{U}^d$	the entire d-dimensional flow rule space
$\mathbb{S}_R^d$	d-dimensional space of Flow Rule R
$R_1 \cap R_2 \neq \emptyset$	Rules R_1 and R_2 overlap directly
P	Ingress Packet
$P \in R$	Packet P matches rule R

Table A.1 – Notations used in flow rule related context

A.1 Flow Rule Concepts

A *flow rule* is a set of actions applied to ingress packets. Flow rules are usually determined by the network provider's or administrator's policy and may include forwarding a packet to a designated egress port or dropping it. Rules are applied on *flows*, defined as a set of headers characterizing a wide range of ingress packets. Flows are usually described by a tuple of *header fields*, such as the 5-tuple: {source IP address, destination IP address, IP protocol, Transport source port, Transport destination port}.

We formally define a *flow rule* R as a data structure composed of:

- A number of $d \in \mathbb{N}^+$ *fields*, used to characterize the range of ingress packets that will match with rule R. Rule fields are typically ranges of packet headers, such as the source IP address or the source port of the transport protocol.

- A series of $k \in \mathbb{N}^+$ *actions* to be applied on the matching ingress packets. Examples may include "forward the packet to egress port p", "drop the packet" or "forward the packet to the controller."

We will interchangeably refer to flow rules as *traffic rules*. In this dissertation, we will not focus further on rule actions, being outside of the scope of our work.

A.2 Flow Rule Fields

A flow rule R is composed of d fields $F_1, F_2, \dots, F_d$, corresponding to d different packet headers, such as the source IP address. Each rule field F_i is defined as a closed interval $F_i = [a, b]$, representing the range of the respective ingress packet header that should match with field F_i.

> **Example:** a rule applying on all ingress packets originating from the network 10.0.42.0/24 has its respective source IP field set to $F_{srcIP} = [10.0.42.0, 10.0.42.255]$. ■

A.3 Rule Matching

We say that a packet P with headers $f_1, f_2, \dots, f_d$ *matches* a rule R with respective fields $F_1, F_2, \dots, F_d$, denoting it as $P \in R$, if and only if:

$$P \in R \iff f_i \in F_i, \quad \forall i \in [0, d]$$

Equivalently, we also say that a rule R *is matched* with a packet P or *matches* a packet P, if and only if P matches R.

A.4 Flow Rule Hyperspace

The d-dimensional *hyperspace* $\mathbb{S}^d$ (or simply *space*) of a flow rule R denotes the entire range of packets that may be matched with this rule and is defined as the Cartesian product of its d fields:

$$\mathbb{S}^d = F_1 \times F_2 \times \cdots \times F_d$$

Throughout this document, we assume that all flow rules within the same context are composed of the same d fields.

> **Example:** assuming $d = 2$ and that rules are defined within the plane {source IP address, destination IP address}, a rule defined simply as $F_1 = [10.0.0.0, 10.255.255.255]$, i.e., defined to match all ingress packets originating from 10.0.0.0/8, will have its destination IP address field implicitly set to $F_2 = [0.0.0.0, 255.255.255.255]$. Any other packet headers,

such as the source MAC address, are ignored in this context. ■

We say that R is contained within the d-dimensional *universe* $\mathbb{U}^d \supseteq \mathbb{S}^d$, defined as the hyperspace representing the entire range of possible values of the d-tuple $(F_1, F_2, \ldots, F_d)$.

A.5 Overlapping Flow Rules

In this section, we will introduce the concept of overlapping flow rules, which have constituted a considerable focus of our work.

A.5.1 Directly Overlapping Rules

We say that two flow rules R_x and R_y, with respective d-dimensional spaces $\mathbb{S}_x^d$ and $\mathbb{S}_y^d$, *overlap directly*, denoting it as $R_x \cap R_y \neq \emptyset$, if and only if:

$$R_x \cap R_y \neq \emptyset \iff \mathbb{S}_x^d \cap \mathbb{S}_y^d \neq \emptyset \iff F_{x,i} \cap F_{y,i} \neq \emptyset, \quad \forall i \in [0, d]$$

Rules that overlap directly will also be referred as *directly overlapping rules*.

Intuitively, we may visualize directly overlapping rules in the 2-dimensional space as two rectangles on the 2D plane that overlap with each other, i.e., share a common surface. The same concept applies on higher dimensions, where two rules overlap directly if they share a common subspace. Ingress packets with headers contained within this shared subspace may potentially match with either rule, in the absence of other priorities.

Example: consider the 2-dimensional plane {source IP address, destination IP address} and two rules defined within it:

$$\begin{aligned} R_x: \quad & F_{x,1} = [10.0.0.0, 10.255.255.255], \qquad & F_{x,2} = [0.0.0.0, 255.255.255.255] \\ R_y: \quad & F_{y,1} = [0.0.0.0, 255.255.255.255], \qquad & F_{y,2} = [42.0.0.0, 42.255.255.255] \end{aligned}$$

In other words, R_x is applied on all packets originating from network 10.0.0.0/8 and R_y is applied on all packets destined to network 42.0.0.0/8. These rules are directly overlapping, as the intersection of their respective hyperspaces is non-empty:

$$\mathbb{S}_x^2 \cap \mathbb{S}_y^2 = [10.0.0.0, 10.255.255.255] \times [42.0.0.0, 42.255.255.255]$$

Packets originating from network 10.0.0.0/8 and destined to network 42.0.0.0/8 may match with either rule R_x or R_y. ■

Conversely, $R_x \cap R_y = \emptyset$ denotes that rules R_x and R_y do not overlap directly, i.e.:

$$R_x \cap R_y = \emptyset \iff \mathbb{S}_x^d \cap \mathbb{S}_y^d = \emptyset \iff \exists i \in [0, d] : F_{x,i} \cap F_{y,i} = \emptyset$$

A.5.2 Indirectly Overlapping Rules

We say that two flow rules R_x and R_y *overlap indirectly* if and only if $m \in \mathbb{N}^+$ other rules R_i exist, such that:

$$\begin{aligned} \exists R_1, R_2, \ldots, R_m : \quad & R_x \cap R_1 \neq \emptyset \\ \text{and} \quad & R_i \cap R_{i+1} \neq \emptyset, \qquad i = 1, 2, \ldots, m-1, \text{ if } m > 1 \\ \text{and} \quad & R_m \cap R_y \neq \emptyset \end{aligned}$$

Rules that overlap indirectly will also be referred as *indirectly overlapping rules*. The trivial case where $m = 0$ corresponds to R_x and R_y being directly overlapping rules. We do not consider the trivial case in this dissertation; therefore, when we say that two rules overlap indirectly, we assume that $m > 0$.

Intuitively, we may visualize two indirectly overlapping rules as a "chain" of directly overlapping rules. For instance, rules R_x and R_y may not directly overlap with each other, but overlap with a common "intermediate" rule R_z.

Example: consider the 2-dimensional plane {source transport port, destination transport port} and three rules defined within it:

$$\begin{aligned} R_x : \quad & F_{x,1} = [0, 1023], & F_{x,2} = [0, 65535] \\ R_y : \quad & F_{y,1} = [8080, 9000], & F_{y,2} = [0, 65535] \\ R_z : \quad & F_{z,1} = [0, 65535], & F_{z,2} = [2000, 3000] \end{aligned}$$

We may see that $R_x \cap R_y = \emptyset$, but $R_x \cap R_z \neq \emptyset$ and $R_z \cap R_y \neq \emptyset$, therefore R_x and R_y overlap indirectly, but not directly. ■

B Packet Classifier Node Definition

In this appendix we present in detail the definition of the tree node, developed for the software table described in section §3.2.1, which we partition by executing the HiCut algorithm [24]. We call the entire tree the *Packet Classifier*. In particular, we illustrate:

1. A partial definition of the `HiCutNode` class in listing B.1, developed in C++14, including the relevant types and private members.
2. A detailed example of a root node and the contents of its private members.

Listing B.1 – Partial definition of the `HiCutNode` class

```
1  #include <map>
2  #include <memory>
3  #include <vector>
4  #include "flow_entry.hh"     // defines class ``FlowEntry''
5
6  class HiCutNode {
7
8  public:
9      /** @brief Packet Header fields.
10      * An enumeration listing the packet fields.
11      * Each field may be used to define a dimension
12      * of the entire flow hyperspace.
13      */
14     enum Field {
15         FIELD_NONE = 0,
16         FIELD_NW_SRC,    // src. IP address
17         FIELD_NW_DST,    // dst. IP address
18         FIELD_NW_PROTO, // IP protocol
19         FIELD_TP_SRC,    // src. transport port
```

```
20            FIELD_TP_DST ,    // dst. transport port
21            // more may be specified , e.g., src./dst. MAC address
22        };
23        using Dimension = Field;  // alias
24
25        /** @brief Single-dimension boundaries.
26         * A pair defining the boundaries of a single dimension.
27         * First  element: left  bound (closed)
28         * Second element: right bound (open)
29         * Range: [left , right)
30         */
31        using Boundary = std::pair< uint64_t , uint64_t >;
32
33        /** @brief Node hyperspace.
34         * A map representing the hyperspace of the node.
35         * Key   = Dimension
36         * Value = [left , right)
37         *
38         * NOTE: We have utilized an ordered map,
39         * instead of an unordered map,
40         * as insertions are not common
41         * and it's useful to have the dimensions ordered.
42         */
43        using SpaceBox = std::map< Dimension , Boundary >;
44
45        // Constructors and public methods omitted for simplicity
46
47    private:
48        /** @brief The Child Map type.
49         * A map holding shared_ptr to this node's children.
50         * If this node if further partitioned across a dimension
51         * within its space box, defined by @cut_dimension ,
52         * then children are created representing the partitions.
53         *  K = The left bound of the partition.
54         *  V = Pointer to the child representing this partition.
55         * We are using a map, so that lookups occur
56         * in logarithmic time.
57         */
58        using ChildMap = std::map<
59            uint64_t , std::shared_ptr< HiCutNode >
60            >;
61
```

```
62      /** @brief Box hyperspace space.
63       * A dimension hash map, encoding the hyperspace space
64       * represented by this node.
65       */
66      const SpaceBox space_box;
67
68      /** @brief Cut dimension.
69       * Dimension to cut across the hyperspace space
70       * represented by this node.
71       * Is set to Dimension::dim_none if and only if
72       * this is a leaf node.
73       */
74      Dimension cut_dimension;
75
76      /** @brief The Child Map member.
77       * The actual @ChildMap member.
78       */
79      ChildMap children_map;
80
81      /** @brief Node Entry List.
82       * Container with all flow rules in this node's
          hyperspace.
83       * The root node holds the entire rule-set.
84       * Each node holds the subset of its parent's rules
85       * that conflicts with its own "space box".
86       * This list may be cleared on non-leaf nodes.
87       */
88       std::vector< FlowEntry > entries;
89  };
```

Example: consider a root node covering the entire 2-dimensional hyperspace "{IP source address, IP destination address}". The HiCut algorithm has partitioned the root node in 4 segments, along the destination IP addresses 64.0.0.0, 128.0.0.0 and 192.0.0.0. The root node will hold the following data:

```
1  space_box = {
2      { FIELD_NW_SRC, { 0, (2 << 32) } },
3      { FIELD_NW_DST, { 0, (2 << 32) } },
4  };
5  cut_dimension = FIELD_NW_DST;
6  children_map = {
7      { 0x00000000, pointer_to_child_0 },
8      { 0x40000000, pointer_to_child_1 },
```

```
9       { 0x80000000, pointer_to_child_2 },
10      { 0xC0000000, pointer_to_child_3 },
11  }
```

The children nodes will hold the following respective data in their `space_box` variables:

```
1   space_box_1 = {
2       { FIELD_NW_SRC, { 0 , (2 << 32) } },
3       { FIELD_NW_DST, { 0x00000000, 0x40000000 } },
4   };
5   space_box_2 = {
6       { FIELD_NW_SRC, { 0 , (2 << 32) } },
7       { FIELD_NW_DST, { 0x40000000, 0x80000000 } },
8   };
9   space_box_3 = {
10      { FIELD_NW_SRC, { 0 , (2 << 32) } },
11      { FIELD_NW_DST, { 0x80000000, 0xC0000000 } },
12  };
13  space_box_4 = {
14      { FIELD_NW_SRC, { 0 , (2 << 32) } },
15      { FIELD_NW_DST, { 0xC0000000, (2 << 32) } },
16  };
```

■

Bibliography

[1] Amazon Web Services. EC2 Instance Types, 2017. URL http://aws.amazon.com/ec2/instance-types. [Online; accessed 28-October-2017].

[2] Amazon Web Services. Enhanced Networking on Linux, 2017. URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html. [Online; accessed 28-October-2017].

[3] Amazon Web Services. Placement Groups, 2017. URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html. [Online; accessed 28-October-2017].

[4] Analog Bits Inc. 4096 x 128 ternary CAM datasheet (28nm), 2011. URL http://www.analogbits.com/pdf/28nm_TCAM_Product_Brief.pdf. [Online; accessed 9-November-2017].

[5] Apache Software Foundation. Apache Hadoop YARN, 2017. URL https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. [Online; accessed 26-October-2017].

[6] Theophilus Benson. Data Set for IMC 2010 Data Center Measurement, 2010. URL http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html. [Online; accessed 13-October-2017].

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. doi: 10.1145/1879141.1879175. URL http://doi.acm.org/10.1145/1879141.1879175.

[8] Cisco Systems Inc. Cisco Nexus 1000V Switch for KVM, 2016. URL https://www.cisco.com/c/en/us/products/switches/nexus-1000v-kvm/index.html. [Online; accessed 12-November-2017].

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[10] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0797-0. doi: 10.1145/2018436.2018466. URL http://doi.acm.org/10.1145/2018436.2018466.

[11] Jose Gabriel de Figueiredo Coutinho, Mark Lee Stillwell, Katerina Argyraki, George Ioannidis, Anca Iordache, Christoph Kleineweber, Alexandros Koliousis, John McGlone, Guillaume Pierre, Carmelo Ragusa, Peter Sanders, Thorsten Schütt, Teng Yu, and Alexander L. Wolf. The HARNESS Platform: A Hardware- and Network-Enhanced Software System for Cloud Computing. *Software Architecture for Big Data and the Cloud*, 2017. URL https://hal.inria.fr/hal-01507344/file/harness.pdf.

[12] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.*, 72 (11):1471–1480, 2012. doi: 10.1016/j.jpdc.2012.01.020. URL https://doi.org/10.1016/j.jpdc.2012.01.020.

[13] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iPerf – the TCP, UDP and SCTP network bandwidth measurement tool. URL https://iperf.fr. [Online; accessed 24-October-2017].

[14] Jonas Fietz, Sam Whitlock, George Ioannidis, Katerina Argyraki, and Edouard Bugnion. Vntor: Network virtualization at the top-of-rack switch. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 428–441, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987582. URL http://doi.acm.org/10.1145/2987550.2987582.

[15] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, pages 27–36, 1999. doi: 10.1109/IWQOS.1999.766475.

[16] Linux Foundation. Open vSwitch: Production quality, multilayer open virtual switch, 2016. URL http://openvswitch.org. [Online; accessed 12-November-2017].

[17] Soudeh Ghorbani and Matthew Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 67–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1477-0. doi: 10.1145/2342441.2342455. URL http://doi.acm.org/10.1145/2342441.2342455.

[18] Google Cloud Platform. Egress throughput caps, 2017. URL https://cloud.google.com/compute/docs/networks-and-firewalls#egress_throughput_caps. [Online; accessed 28-October-2017].

[19] Google Inc. Google web search engine. URL https://www.google.com. [Online; accessed 24-October-2017].

[20] Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, pages 13–20. Omnipress, 2010. URL http://dblp.uni-trier.de/db/conf/icml/icml2010.html#GraepelCBH10;http://www.icml2010.org/papers/901.pdf;http://www.bibsonomy.org/bibtex/2b008aa80a83b88a6e5fee59caa9b6493/dblp.

[21] Albert Greenberg. SDN for the cloud, 2015. URL http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf. [Online; accessed 20-November-2017].

[22] Grid'5000 development team. Grid'5000 Network Backbone, 2016. URL https://www.grid5000.fr/mediawiki/index.php/Grid5000:Network. [Online; accessed 15-October-2017].

[23] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0448-1. doi: 10.1145/1921168.1921188. URL http://doi.acm.org/10.1145/1921168.1921188.

[24] Pankaj Gupta and Nick Mckeown. Packet classification using hierarchical intelligent cuttings. In *in Hot Interconnects VII*, pages 34–41, August 1999.

[25] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, January 2000. ISSN 0272-1732. doi: 10.1109/40.820051. URL http://dx.doi.org/10.1109/40.820051.

[26] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15 (2):24–32, March 2001. ISSN 0890-8044. doi: 10.1109/65.912717.

[27] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, Jamal Hadi Salim, David Meyer, and Odysseas Koufopavlou. Software-Defined Networking (SDN): Layers and Architecture Terminology. RFC 7426, January 2015. URL https://rfc-editor.org/rfc/rfc7426.txt.

[28] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud. In Rodrigo Fonseca and David A. Maltz, editors, *HotCloud*. USENIX Association, 2012. URL http://dblp.uni-trier.de/db/conf/hotcloud/hotcloud2012.html#JeyakumarAMPK12;https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/jeyakumar;http://www.bibsonomy.org/bibtex/26f7b77222e16d115b5ded5d2e0d8b7b1/dblp.

[29] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. CacheFlow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the*

Symposium on SDN Research, SOSR '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4211-7. doi: 10.1145/2890955.2890969. URL http://doi.acm.org/10.1145/2890955.2890969.

[30] G. Kim and W. Lee. Cannot take my allocation: Enforcing fairness by considering demand and payment in clouds. In *2013 Fourth International Conference on the Network of the Future (NoF)*, pages 1–5, Oct 2013. doi: 10.1109/NOF.2013.6724514.

[31] Last.fm. Online music catalog & music recommendation system. URL https://www.last.fm. [Online; accessed 24-October-2017].

[32] Jeongkeun Lee, Myungjin Lee, Lucian Popa, Yoshio Turner, Sujata Banerjee, Puneet Sharma, and Bryan Stephenson. CloudMirror: Application-Aware Bandwidth Reservations in the Cloud. In Dilma Da Silva and George Porter, editors, *HotCloud*. USENIX Association, 2013. URL http://dblp.uni-trier.de/db/conf/hotcloud/hotcloud2013.html#LeeLPTBSS13.

[33] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064015. URL http://doi.acm.org/10.1145/3035918.3064015.

[34] Layong Luo, Gaogang Xie, Steve Uhlig, Laurent Mathy, Kavé Salamatian, and Yingke Xie. Towards tcam-based scalable virtual routers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 73–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1775-7. doi: 10.1145/2413176.2413186. URL http://doi.acm.org/10.1145/2413176.2413186.

[35] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. URL http://doi.acm.org/10.1145/1355734.1355746.

[36] Peter M. Mell and Timothy Grance. SP 800-145. The NIST definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011. URL http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[37] Microsoft Corporation. Bing web search engine. URL https://www.bing.com. [Online; accessed 24-October-2017].

[38] Microsoft Corporation. Optimize network throughput for Azure virtual machines, 2017. URL https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-optimize-network-bandwidth. [Online; accessed 28-October-2017].

[39] Microsoft Corporation. Bandwidth/throughput testing (NTTTCP), 2017. URL https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-bandwidth-testing. [Online; accessed 28-October-2017].

[40] Microsoft Corporation. Azure virtual network, 2017. URL https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview. [Online; accessed 28-October-2017].

[41] Nitin Mohan and Manoj Sachdev. Low-leakage storage cells for ternary content addressable memories. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(5):604–612, May 2009. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2006040. URL http://dx.doi.org/10.1109/TVLSI.2008.2006040.

[42] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vcrib: Virtualized rule management in the cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342763.2342786.

[43] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050.

[44] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. FasTrak: Enabling express lanes in multi-tenant data centers. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 139–150, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2101-3. doi: 10.1145/2535372.2535386. URL http://doi.acm.org/10.1145/2535372.2535386.

[45] OpenStack "kilo" documentation. SR-IOV-Passthrough-For-Networking, 2015. URL https://wiki.openstack.org/w/index.php?title=SR-IOV-Passthrough-For-Networking&oldid=93943. [Online; accessed 20-November-2017].

[46] OpenStack "kilo" documentation. Scheduling, 2016. URL https://docs.openstack.org/kilo/config-reference/content/section_compute-scheduler.html. [Online; accessed 28-October-2017].

[47] OpenStack "kilo" documentation. OpenStack Neutron, 2017. URL https://docs.openstack.org/neutron/latest. [Online; accessed 12-November-2017].

[48] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the network in cloud computing. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *SIGCOMM*, pages 187–198. ACM, 2012. ISBN 978-1-4503-1419-0. URL http://doi.acm.org/10.1145/2342356.2342396.

[49] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for

Cloud Computing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 351–362, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486027. URL http://doi.acm.org/10.1145/2486001.2486027.

[50] RightScale, Inc. Cloud computing trends: 2017 state of the cloud survey, 2017. URL https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-cloud-survey. [Online; accessed 12-November-2017].

[51] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In Sanjay Kumar, Himanshu Raj, and Karsten Schwan, editors, *WIOV*. USENIX Association, 2011. URL http://dblp.uni-trier.de/db/conf/usenix/wiov2011.html#RodriguesSTSG11;https://www.usenix.org/conference/wiov11/gatekeeper-supporting-bandwidth-guarantees-multi-tenant-datacenter-networks;http://www.bibsonomy.org/bibtex/21f7e01a21cd0590d77760e40adac714f/dblp.

[52] Sandy Ryza. Improvements in the hadoop YARN fair scheduler, 2013. URL https://blog.cloudera.com/blog/2013/06/improvements-in-the-hadoop-yarn-fair-scheduler. [Online; accessed 26-October-2017].

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863104.

[54] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4. doi: 10.1145/863955.863980. URL http://doi.acm.org/10.1145/863955.863980.

[55] Tencent Holdings Ltd. Session logs of `soso.com` search engine [data-set]. Kaggle KDD Cup 2012, Track 2 competition, 2012. URL https://www.kaggle.com/c/kddcup2012-track2. [Online; accessed 24-October-2017].

[56] Luis Tomás and Johan Tordsson. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 5:1–5:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2172-3. doi: 10.1145/2494621.2494627. URL http://doi.acm.org/10.1145/2494621.2494627.

[57] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. URL http://www.globule.org/publi/WWADH_comnet2009.html.

[58] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. September 2007 Wikipedia access traces, 2014. URL http://carambolier.irisa.fr/wiki/2007-09. [Online; accessed 22-October-2017].

[59] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239–254, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844151. URL http://doi.acm.org/10.1145/844128.844151.

[60] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 207–218, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851208. URL http://doi.acm.org/10.1145/1851182.1851208.

[61] Erik-Jan van Baaren. WikiBench: A distributed, Wikipedia based web application benchmark. Master's thesis, VU University Amsterdam, May 2009. URL http://www.wikibench.eu/wp-content/uploads/2010/10/van-baaren-thesis.pdf.

[62] Werner Vogels. 10 lessons from 10 years of Amazon web services, 2016. URL http://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html. [Online; accessed 20-November-2017].

[63] Wikimedia Foundation. Wikimedia Downloads. URL https://dumps.wikimedia.org. [Online; accessed 22-October-2017].

[64] Wikimedia Foundation. Mediawiki, the free wiki engine, 2016. URL https://www.mediawiki.org/w/index.php?title=MediaWiki&oldid=2301969. [Online; accessed 22-October-2017].

[65] Wikimedia Foundation. Wikimedia database dump of the Danish Wikipedia on August 01, 2016, 2016. URL https://archive.org/details/dawiki-20160801. [Online; accessed 22-October-2017].

[66] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 199–210, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1419-0. doi: 10.1145/2342356.2342397. URL http://doi.acm.org/10.1145/2342356.2342397.

[67] Bo Yan, Yang Xu, Hongya Xing, Kang Xi, and H. Jonathan Chao. CAB: A reactive wildcard rule caching system for software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 163–168, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2989-7. doi: 10.1145/2620728.2620732. URL http://doi.acm.org/10.1145/2620728.2620732.

[68] L. Ying. Sponge: an oversubscription strategy supporting performance interference management in cloud. *China Communications*, 12(11):1–14, November 2015. ISSN 1673-5447. doi: 10.1109/CC.2015.7366242.

[69] Fang Yu, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, ICNP '04, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2161-4. URL http://dl.acm.org/citation.cfm?id=1025124.1025890.

[70] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851224. URL http://doi.acm.org/10.1145/1851182.1851224.

[71] Jing Zhu, Dan Li, Jianping Wu, Hongnan Liu, Ying Zhang, and Jingcheng Zhang. Towards bandwidth guarantee in multi-tenancy cloud computing networks. In *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ICNP '12, pages 1–10, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-2445-8. doi: 10.1109/ICNP.2012.6459986. URL http://dx.doi.org/10.1109/ICNP.2012.6459986.

George Ioannidis

PhD in Computer & Communication Sciences

ioanngio@gmail.com
github.com/gioannidis

Education

2011–2017 **PhD in Computer & Communication Sciences**, *EPFL*, Switzerland.
School of Computer and Communication Sciences.
In addition, completed semester projects and courses worth of 30 ECTS credits.

2006–2011 **Diploma in Electrical & Computer Engineering**, *National Technical University of Athens*, Greece, *GPA: 9.58/10.00*.

- Major: Computer Systems, Computer Software
- Minor: Communications & Computer Networks, Electronics – Circuits – Materials

Dissertations

PhD Thesis

title *Adding Flexibility to Multi-Tenant Networks*

supervisor Prof. Katerina Argyraki, EPFL, Switzerland

description We propose two mechanisms to facilitate cloud providers in managing shared resources: (i) a virtual flow table abstraction for storing access-control rules (ACLs) in Top-of-Rack (ToR) switches; and (ii) a two-tiered resource scheduling architecture that provides bandwidth reservations and latency constraints to cloud tenants.

Diploma Thesis

title *Cone-Beam C-arm Radiography System Modeling for assisting in Closed Intramedullary Nailing of Long-Bone Surgeries*

supervisor Prof. Elias Koukoutsis, National Technical University of Athens, Greece

description C-arm radiography system modeling for assisting surgeons in the placement of locking intramedullary nails in long-bone orthopedic surgeries.

Journal Publications

2017 Figueiredo Coutinho, Jose Gabriel de, Mark Lee Stillwell, Katerina Argyraki, George Ioannidis, Anca Iordache, Christoph Kleineweber, Alexandros Koliousis, John McGlone, Guillaume Pierre, Carmelo Ragusa, Peter Sanders, Thorsten Schütt, Teng Yu, and Alexander L. Wolf. "The HARNESS Platform: A Hardware- and Network-Enhanced Software System for Cloud Computing". In: *Software Architecture for Big Data and the Cloud*. Ed. by Ivan Mistrik, Nour Ali, Rami Bahsoon, Maritta Heisel, and Bruce R. Maxim.

Conference Publications

2016 — Fietz, Jonas, Sam Whitlock, George Ioannidis, Katerina Argyraki, and Edouard Bugnion. “VNToR: Network Virtualization at the Top-of-Rack Switch”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC ’16. Santa Clara, CA, USA: ACM, pp. 428–441.

Projects

2014–2017 — **HARNESS Cloud Platform**, *EPFL*, Switzerland.
Hardware- and Network-Enhanced Software Systems for Cloud Computing. Heterogeneous hardware and network technology integration into data center platforms, achieving performance improvement and reduction in energy consumption and cost profiles for important and high-value cloud applications such as real-time business analytics and the geosciences. In collaboration with Imperial College London, Université de Rennes 1, SAP Belfast, Maxeler Technologies and Konrad-Zuse-Zentrum für Informationstechnik Berlin. Website: `www.harness-project.eu`

2011–2017 — **Virtual Data Plane**, *EPFL*, Switzerland.
A programmable, high-performance packet-processing platform. We developed a platform on a commodity hardware router, to address real-time resource management issues. In cooperation with Broadcom Corporation, who had provided us with the hardware router to deploy our prototype within our data center.

Achievements & Awards

2013 — **Outstanding Teaching Assistant Award**, *School of Computer and Communication Sciences, EPFL*, Switzerland.

2011 — **1st Year Fellowship**, *Computer and Communication Sciences Doctoral School, EPFL*, Lausanne, Switzerland.

2011 — **Top 2% Graduate**, *National Technical University of Athens*, Greece.
Ranked 5th among 325 graduates.

2008 — **Zosima Scholarship for Undergraduate Studies**, *Zosima Bros. Foundation*, Greece.

2007 — **Papakyriakopoulos Award for Excellence in Mathematics**, *National Technical University of Athens*, Greece.
Awarded to the second-year students with the highest grades in Mathematics courses.

2006 — **Deligiannis Scholarship for Undergraduate Studies**, *EKO ABEE, subsidiary of Hellenic Petroleum S.A.*, Greece.
Awarded to the first-year university students who attained the highest grades in the Greek national examinations.

2005 — **15th Greek National Physics Olympiad Award**, *3rd place.*
Organized by the Greek National Physicists Union at a nation-wide level.

Languages

Greek	**Native**	*Mother Tongue*
English	**Fluent**	*Certificates of Proficiency (Cambridge/Michigan), TOEFL 109/120*
French	**Very Good**	*Daily practice, can handle everyday communication*
German	**Good**	*Zertifikat Deutsch (B1), Goethe Institut*

Experience

Academic Experience

2011–2017 **Research Assistant**, *Network Architecture Laboratory at EPFL*, Lausanne, Switzerland.
Extensive research on facilitating cloud providers in managing shared resources.

2014–2015 **Research Assistant**, *Université de Rennes 1*, Rennes, France.
MYRIADS team.
Eight-month internship. Worked on the *HARNESS* project.

Teaching Experience

2012–2016 **Teaching Assistant**, *Computer Networks*, EPFL, Switzerland.
Prof. Katerina Argyraki, IC faculty.
Fall Semesters 2012-2013, 2013-2014, 2016-2017, COM-208.

2016 **Teaching Assistant**, *Programming II*, EPFL, Switzerland.
Prof. Ronan Boulic, IC faculty.
Spring Semester 2015-2016, CS-112(c).

2014 **Teaching Assistant**, *Programming*, EPFL, Switzerland.
Prof. Thomas Lochmatter, ENAC faculty.
Spring Semester 2013-2014, CS-111(b).

2013 **Teaching Assistant**, *Computer-Aided Engineering*, EPFL, Switzerland.
Dr. David Lindelöf, ENAC faculty.
Spring Semester 2012-2013, CS-110(e).

2007–2011 **Laboratory Assistant**, *Introduction to Programming*, NTUA, Greece.
Prof. Stathis Zachos, ECE faculty.
Fall Semesters 2007-2008, 2008-2009, 2009-2010, 2010-2011, course 3.4.01.1.

Professional Experience

2007–2010 **Junior IT Engineer**, *Hellenic Petroleum S.A. Group*, Marousi, Greece.
IT department.
Four summer internships. Main Duties:

- Provide technical assistance to everyday issues.
- Upgrade the desktop infrastructure.
- Relocate the IT infrastructure from a subsidiary to the Group's new Headquarters.
- Develop team work skills.
- Take responsibilities and initiatives in a real industrial environment.

Skills

Coding	extensive C++; experienced in C, Matlab, Octave, Python	**Platforms**	Apache Hadoop, Click Modular Router, OpenStack, Wireshark
Operating Systems	Linux, Microsoft Windows	**Productivity Software**	Microsoft Office, OpenOffice, LaTeX

References

Katerina Argyraki
Professor
Network Architecture Laboratory
EPFL
Lausanne, Switzerland
✉ katerina.argyraki@epfl.ch
people.epfl.ch/katerina.argyraki
☎ +41 21 69 38132

Guillaume Pierre
Professor
Myriads Team
Université de Rennes 1
Rennes, France
✉ guillaume.pierre@irisa.fr
globule.org/~gpierre
☎ +33 2 99 84 25 20