ST. PETERSBURG DEPARTMENT OF STEKLOV MATHEMATICAL
INSTITUTE OF RUSSIAN ACADEMY OF SCIENCES

Manuscript

Sergey Igorevich Nikolenko

Algorithms for Networking Problems and Their Theoretical Analysis

1.2.3. Theoretical computer science, cybernetics

Doctoral Dissertation
in Physics and Mathematics

St. Petersburg
2022

Dedicated to the memory of
Kirill Kogan

Contents

Chapter 1 Introduction **5**
1.1 Research objectives and overview 5
1.2 Buffer management 6
1.3 Packet classification 8
1.4 Control plane algorithms 10
1.5 Approbation of work: list of publications 13

Chapter 2 Buffer management algorithms and their analysis . . . **22**
2.1 Introduction to competitive analysis for buffer management and related work 22
2.2 Buffer management for packets with heterogeneous processing for a single queue 29
2.3 Buffer management for multiple queues separated by processing requirements 37
2.4 Buffer management for packets with heterogeneous processing and multiple output ports 42
2.5 Buffer management for packets with heterogeneous processing and varying size 53
2.6 Buffer management for packets with multiple characteristics . . . 56
2.7 Buffer management for packets with latency constraints 61
2.8 Choosing buffer management algorithms with machine learning . . 66

Chapter 3 Packet classification algorithms **73**
3.1 Introduction to packet classification and related work 73
3.2 Exploiting order independence for packet classification 76
3.3 Multigroup representations of packet classifiers 93
3.4 Packet classifier reduction on distributed platforms 99
3.5 Classifiers for policies reusing the same classes 116

3.6 Approximate packet classification 123

Chapter 4 Control plane algorithms: network simplification, cloud computing, and smart grids 128

4.1 Network simplification algorithms 128

4.2 Formalization and taxonomy of compute-aggregate problems for cloud computing applications 140

4.3 Efficient demand assignment in multi-connected microgrids 150

4.4 Robust counting of traffic flows 158

Chapter 5 Conclusion 165

5.1 Main results 165

5.1.1 Algorithms for buffer management and their analysis . . . 165

5.1.2 Packet classification algorithms 170

5.1.3 Control plane algorithms 173

5.2 Open questions for further research 177

Bibliography 179

Chapter 1 Introduction

1.1 Research objectives and overview

The Internet is built around a large variety of network elements, transferring information aggregated into packets that are forwarded towards their destinations. New economic models and services require networks to be more intelligent and expressive, which usually translates into more expensive network infrastructure. Reducing manageable network state and better reuse of underlying network infrastructure without compromising expressiveness allow to improve cost efficiency.

Therefore, **the object of the present research** includes network parameters, models, and algorithms for the data plane and control plane, and **the subject of the present research** includes their theoretical properties that can provide worst-case guarantees, properties of networking-related objects (e.g., packets in buffer management or packet classifiers) that lead to new algorithms, and optimization problems that arise from these properties coupled with economic objectives.

Research objectives include the development of new algorithms for various networking-related problems such as buffer management, packet classification, and others, and theoretical analysis that validates their performance. This work is of **theoretical nature**, its main results include new algorithms and the theoretical analysis of these algorithms, including theorems evaluating their worst-case performance with competitive analysis.

Methods. The ability to characterize traffic and understand its properties is critical for efficient and effective network operations. In particular, one has to determine service classes for various applications and different traffic patterns to facilitate accurate capacity planning and ensure that resources are used appropriately in support of implemented management policies. This yields new opportunities to justify and optimize vast investments involved in building networks, from

understanding network behavior to exploiting the underlying infrastructure.

Data plane services can be split into two major categories: (1) services that change properties of single packets (e.g, recoloring specific fields, changing encapsulations, access control lists) and (2) services that change inter-packet properties (e.g., rate-limiting, shaping). In this thesis, Chapter 2 is devoted to the latter category in the form of buffer management algorithms and their analysis, while Chapter 3 deals with the former via exploiting new structural properties of packet classifiers.

On the control plane, network infrastructure is an expensive resource that requires complex management. Network providers are often unable to fully leverage this huge investment. In Chapter 4 of this thesis, we discuss control plane abstractions and mechanisms that allow to better exploit network infrastructure.

In this dissertation, we consider both the data plane and the control plane, and below we present a brief overview and motivation of the main research directions of this thesis. For a more detailed motivational overview together with open questions posed for future research (some of which have been answered in this dissertation), see the work [33]. The **key results** presented for defense in this dissertation are summarized in Section 5.1.

1.2 Buffer management

Buffering architectures define how input and output ports of a network element are connected [109]. Their design and management directly impact performance and cost of each network element. If a burst of packets arrives, there are cases when it is impossible to transmit all packets immediately, and some of them must be buffered inside the network element (or dropped). A *buffer management policy* consists of three major parts: *admission control* (that decides which traffic should be admitted to the buffer), *processing policy* (defining a processing order of packets), and *packet scheduling* (that chooses which packet is transmitted next).

To analyze the performance of buffer management policies, the networking community has historically relied on stochastic models to evaluate performance. Worst-case counterparts of these problems were first introduced in the theoretical community in the early 2000s with the works [46, 132, 161] that applied competitive analysis [68] to this domain. Competitive analysis not only helps better

understand buffer management policies but also provides worst-case guarantees on performance, which is important for real life switches with unpredictable and highly variable traffic patterns. We identify two major directions in recent developments in this domain: incorporating new traffic characteristics into buffer management policies and studying the impact of new characteristics on final objectives.

Packet characteristics that have been studied in previous works include *value*, i.e., how much the packet contributes to the objective function when it is transmitted, *required processing*, i.e., how many processing cycles the packet has to go through before it can be transmitted (this is motivated by many different tasks of varying complexity that a modern network edge can apply to a packet), *size*, i.e., how much memory a packet takes up in the buffer, and others. Some works have considered packets that have multiple characteristics, e.g., value and required processing at the same time. A different kind of characteristic is the *slack* of a packet, i.e., the deadline before which this packet has to be transmitted (otherwise it does not contribute to the objective).

In Chapter 2, we consider several different settings where packets are endowed with various characteristics and combinations of them. In particular:

- Section 2.1 introduces the main definitions of competitive analysis and surveys prior work that applied it to buffer management;
- Section 2.2 is devoted to a single queue architecture with packets with heterogeneous required processing;
- Section 2.3 proceeds to buffers with multiple output queues, but considers the case when output queues have different characteristics (e.g., each output queue has only one processing requirement, but these requirements differ between different queues);
- Section 2.4 moves to the general case when multiple output queues can each receive packets with heterogeneous processing;
- starting from Section 2.5, we consider packets with multiple characteristics; Section 2.5 is devoted to packets that combine heterogeneous processing requirements and varying size;

- in Section 2.6, we consider buffer management for a single queue with packets with varying both requirement processing and value;
- Section 2.7 adds to this mix another characteristic, namely *latency*, i.e., a deadline for processing each packet;
- finally, Section 2.8 is devoted to a novel direction of research where buffer management algorithms can be chosen by machine learning models depending on traffic properties.

1.3 Packet classification

Usually, a service on the level of single packets is represented by a *packet classifier*. It is, in a nutshell, a program consisting of an ordered set of conditions. The first condition that matches an incoming packet defines which actions apply. Each condition in a classifier is expressed as a multi-field classification rule, with separate conditions applied to each field. Packet classifiers based on more than one field have become very common in modern networking.

If a classification rule looks for exact values for all fields, it can be represented by simply concatenating all fields; such classifiers can be implemented in content-addressable memory (CAM) or by a simple hash function in space linear in the number of rules. The problem becomes harder if a field is represented by a prefix (looking for specific values of only the most significant bits) or a range (confining values inside an interval) since a concatenation of prefixes or ranges is no longer a prefix or range. To represent a range in content-addressable memory, one has to apply range encoding schemes that represent each range with several entries, and if there are several range-based fields the complexity multiplies and may quickly get out of hand.

Many sophisticated software-based approaches have been proposed previously for packet classification [198]. Bounds derived from computational geometry imply that a software-based packet classifier with N rules and $k \geq 3$ fields have to use either $O(kN)$ space and $O(\log N)$ lookup time or $O(N)$ space and $O(\log^{k-1} N)$ time [112, 171]. Thus, software-based approaches will become either too slow or too memory-intensive even with only a few prefix- or range-based fields.

More advanced services require additional expressiveness on existing classifi-

cation fields or new ones. Given the above theoretical bounds, increasing expressiveness (adding new fields) and number of rules in a classifier can significantly impede efficiency of implementations. In Chapter 3 of this thesis, we introduce a novel approach and several directions that allow to address this fundamental tradeoff between scalability and expressiveness and improve efficiency of classifier representations. We study structural properties that allow to represent classifiers efficiently.

Previous research on the efficiency of packet classifiers has been mostly centered around semantically equivalent representations. Some of them introduce proprietary techniques, and most approaches are variations of Boolean minimization methods [51, 141, 195]. Boolean minimization approaches can reduce both classification width (number of bit identities involved in classification) and number of rules.

But what does one do if the optimized classifiers are irreducible but still do not reach desired efficiency metrics such as required memory and lookup time? Increasing classification width and number of rules significantly affects the efficiency of classifier representation, and hence feasibility; theoretical bounds show that there are no efficient representations without exponential memory and with feasible lookup time. The answer may come from structural properties of classifiers that lead to efficient implementations.

In Chapter 3, we introduce and study several structural properties that allow for more efficient representations and more efficient lookup algorithms: rule-order independence, action-order independence, relaxed semantic equivalence, prefix reorderability, and others. In particular:

- Section 3.1 presents the packet classification problem and surveys related work on packet classification;

- in Section 3.2 we introduce the main tool for our contributions to packet classification: the notion of order-independence and associated problems and results;

- in Section 3.3 we extend classifier reduction via order-independence by dividing the classifier into multiple groups of rules, introduce the associated optimization problems, present and analyze algorithms for solving them; in particular, here we show how the proposed approaches can help fit an IPv6

classifier into IPv4 or even MPLS hardware;

- Section 3.4 studies how packet classification can be done in a distributed way, and how classification tables can be efficiently divided between several network elements;

- Section 3.5 studies joint optimization of several classifiers that reuse the same subsets of rules (*classes*; this is a very common situation in large-scale real life classifiers);

- in Section 3.6, we introduce possibilities for even further reductions in case one is ready to relax some of the classification objectives, thus passing from exact to approximate packet classification.

1.4 Control plane algorithms

The software-defined networking (SDN) paradigm allows network operators to deploy network services through a centralized controller. Recent interest in SDN has fueled the implementation of a variety of network services on controllers written in different languages and supported by different organizations. Given the large number of network services and their increasing complexity, no single controller can provide all network services. Even if a controller provides all desired services, it is unlikely to have best-in-class implementations of all those services. Finding new abstractions can significantly simplify network management and improve flexibility of control planes.

Originally, it had been assumed that the standardization of south-bound API can satisfy SDN requirements. Later it became clear that a single virtual controller is not sufficient for network management. The decoupling of control and data planes and standardization of south-bound API only improve the resolution of management but do not address the problem in general since if one requires interoperability between several virtual controllers then the east-west API should be standardized as well. Before SDN, the standardization of east-west API was done with per protocol resolution, which is a complex task. We believe that standardization of east-west API should be protocol oblivious, and potentially the infrastructure of routing protocols can be exploited to achieve this goal.

To simplify network management, one can represent the original network with a simpler/cheaper network that still implements specific properties of the original, which results in a tradeoff between the simplicity of representations and efficient reuse of the underlying infrastructure. Preserving network capabilities allows services to operate on the simpler representation in a way transparent for the physical infrastructure, and understanding the constraints of a given network shows which resources might need additional investment. There have been attempts to virtualize specific network architectures [49,121], but there is no well-understood process to get a simplified representation of a network while preserving its "bandwidth" and "routing" capabilities.

In Section 4.1 we provide a first step in this direction, introducing new structural properties for networks and providing transformations that preserve these properties.

To address access bandwidth constraints to storage, data centers maintain data at different interconnected locations. Modern big data applications are highly distributed, and requests need to satisfy various objectives such as latency and cost efficiency [47, 74, 150]. *Compute-aggregate* problems, where several data chunks must be aggregated in a network sink, encompass an important class of big data applications implemented in modern data centers. To compute the final result, multiple data chunks must be aggregated in one place while satisfying combined latency constraints.

The works [88,89] studied compute-aggregate structures optimized to minimize latency. Unfortunately, the final result can heavily depend on the properties of underlying transports and utilization of network infrastructure. To address these limitations, in Section 4.2 we propose a model that constructs a schedule of aggregations under budget constraints that can be specified for each compute-aggregate task. Later, an underlying transport can redistribute the computed schedule in time to optimize desired objectives such as latency or throughput. We believe that this approach will allow to decouple optimization problems from underlying transports and provide better fine-grained control to exploit the network infrastructure.

Section 4.3 applies the methods of competitive analysis of online algorithms that we have developed in other contexts to efficient demand assignment in smart power grids. Rapid penetration of distributed generation resources, especially

diesel gensets and solar generation, have made it possible for groups of homes or businesses to meet much or all of their elastic demand through local generation. This usage pattern is especially common in developing countries such as India, which suffers with electricity deficit and as a result, millions of consumers are affected by inadequate power supply [61, 67]. There, customers use local generation to complement the central grid which is unreliable and limited in capacity. We anticipate that in the future geographically close microgrids will opportunistically form connections with each other to increase reliability, a natural recapitulation of the self-organizing process by which electricity grids were formed in the first place, before centralized generation essentially eliminated micro-generation a century ago. In Section 4.3, we concentrate on efficient demand satisfaction in the context of multi-connected microgrids, where a demand can be met by different generation resources: local, nearby, or on a regional grid that defines a *load balancing vector* for each demand that determines which generators are connected/available to satisfy this demand.

Challenges of scalable traffic monitoring include unrelenting traffic growth, device heterogeneity, and load unevenness. First, while traffic keeps growing in both volume and number of flows, processing and memory requirements for traffic monitoring in network elements grow as well. Second, networks comprise elements of increasing heterogeneity ranging from basic IoT (Internet of Things) access devices with greatly limited capabilities to high-end core routers that forward millions of concurrent flows. Third, the monitoring load on different network elements is uneven, and one element might get overloaded even while other elements have spare resources. Even a relatively simple task, such as computing the sizes of all flows at a network element, puts scarce data-plane resources under strain when the number of flows becomes large.

An alternative technique for addressing the scalability challenge in a single network element is to monitor traffic flows by utilizing resources in multiple network elements as a shared pool. If a monitoring task overloads an element, then the task execution can be shifted from the overloaded element along the flow path to another element that has ample resources. This kind of task relocation empowers the network to effectively leverage its global processing and memory resources and cope with local monitoring overloads, but at the same time exposes the monitoring to network noise. In Section 4.4, we introduce and study different

models of network noise, propose and study algorithms that execute a monitoring task in multiple network elements correctly despite network noise on the paths between the elements.

To sum up, the dissertation covers a wide array of algorithmic and analytic problems related to networking, introducing new algorithms, new techniques for the analysis of algorithms, and new theoretical results for networking algorithms on both data plane and control plane. Results presented for defense are listed in Section 5.1. Results of this work can be and already are being used for a wide variety of networking applications, including the design of switches and other network elements, packet classification, scheduling, network virtualization, packet counting in flows, and others.

1.5 Approbation of work: list of publications

Each of the key results presented in this thesis has been published in one of the peer-reviewed research papers listed below. Throughout the dissertation, publications are cited with arabic numbers (e.g., [1]), and works from this list are numbered from 1 to 42, while other works are numbered starting from 43.

Works published in top-tier journals (Q1 in Web of Science / Scopus)

1. V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Packet Classifiers With Controlled Accuracy. IEEE Transactions on Networking, vol. 29, no. 3, 2021, pp. 1141–1154.

   Contribution of the dissertation's author: Theorems 4 and 5 in collaboration with V.D., K.K.

2. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. IEEE Transactions on Networking, vol. 29, no. 1, 2021, pp. 275–288.

   Contribution of the dissertation's author: Theorem 3. Theorems 1, 5 in collaboration with K.K.

3. A. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Competitive buffer management for packets with latency constraints. Computer Networks, vol. 189, no. 22, Article ID 107942, 2021.

Contribution of the dissertation's author: Theorem 1. Theorems 2, 3, 4, 5 in collaboration with A.D.

4. P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalization and taxonomy of compute-aggregate problems for cloud computing applications. Computer Networks, vol. 189, no. 22, Article ID 107915, 2021.

   Contribution of the dissertation's author: Theorems 3, 4, 5, 7, 15, 16, 17, 18 in collaboration with P.C., A.D., K.K., A.S.

5. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. Towards Software-Defined Buffer Management. IEEE Transactions on Networking, vol. 28, no. 5, 2020, pp. 2337–2349.

   Contribution of the dissertation's author: Theorems 1, 2, 3, 4.

6. P. Chuprikov, S.I. Nikolenko, K. Kogan. Towards Declarative Self-Adapting Buffer Management. ACM SIGCOMM Computer Communication Review, vol. 50, no. 3, 2020, pp. 30–37.

   Contribution of the dissertation's author: application of multiarmed bandits to the problem; algorithms based on multiarmed bandits.

7. V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. IEEE Transactions on Networking, vol. 28, no. 3, 2020, pp. 1088–1101.

   Contribution of the dissertation's author: Theorems 8 and 9 in collaboration with P.C., K.K.

8. P. Chuprikov, S.I. Nikolenko, K. Kogan, A. Davydow. Priority Queueing for Packets with Two Characteristics. IEEE Transactions on Networking, vol. 26, no. 1, 2018, pp. 342–355.

   Contribution of the dissertation's author: Theorems 4, 5, 6, 13. Theorems 8, 9, 10 in collaboration with P.C., A.D.

9. P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Admission Control in Shared Memory Switches. Journal of Scheduling, vol. 21, no. 5, 2018, pp. 533–543.

Contribution of the dissertation's author: Theorems 1, 2, 3, 7. Theorem 5 in collaboration with P.E., A.K., K.K., A.S. Simulation study in collaboration with A.S.

10. K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. Efficient FIB Representations on Distributed Platforms. IEEE Transactions on Networking, vol. 25, no. 6, 2017, pp. 3309–3322.

    Contribution of the dissertation's author: Algorithms 1, 2, 3, 4, 5 in collaboration with K.K. Complexity analysis of the algorithms. Simulation study in collaboration with A.S.

11. P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Heterogeneous Packet Processing in Shared Memory Buffers. Journal of Parallel and Distributed Computing, vol. 99, 2017, pp. 1–12.

    Contribution of the dissertation's author: Theorems 1, 2, 3, 6, 7. Theorem 5 in collaboration with A.K., K.K., A.S. Simulation study in collaboration with A.S.

12. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. The Impact of Processing Order on Performance: A Taxonomy of Semi-FIFO Policies. Journal of Computer and System Sciences, vol. 88, 2017, pp. 220–235.

    Contribution of the dissertation's author: Theorems 5, 7, 8, 9, 10, 11, 14. Definitions of lazy policies and Theorems 1, 6, 13 in collaboration with K.K., A.L., A.S; simulation study in collaboration with A.S.

13. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub, M. Segal. Large profits or fast gains: A Dilemma in Maximizing Throughput with Applications to Network Processors. Journal of Network and Computer Applications, vol. 74, 2016, pp. 31–43.

    Contribution of the dissertation's author: Theorems 3, 4, 11, 13. Theorems 5, 8, 12 and Corollary 10 in collaboration with K.K., A.L., G.S., M.S.; Lemmas 1 and 2 in collaboration with K.K., A.L. Simulation study.

14. K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. Exploiting Order Independence for Scalable and Expressive Packet Classification. IEEE Transactions on Networking, vol. 24, no. 2, 2016, pp. 1251–1264.

Contribution of the dissertation's author: Theorems 5, 6. Theorems 1, 2 and the corresponding classifier reduction algorithms in collaboration with K.K. Simulation study.

Publications in proceedings of top tier conferences (A/A* by the CORE ranking)

15. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. 27th IEEE International Conference on Network Protocols (ICNP 2019), 2019, pp. 1–11.

    Contribution of the dissertation's author: same as [2] (this is the first conference paper combined in the journal version [2]).

16. V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Classifiers with Controlled Accuracy. 38th IEEE International Conference on Computer Communications (INFOCOM 2019), 2019, pp. 2044–2052.

    Contribution of the dissertation's author: same as [1] (this is its conference version).

17. V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. 26th IEEE International Conference on Network Protocols (ICNP 2018), 2018, pp. 121–131

    Contribution of the dissertation's author: same as [17] (this is its conference version).

18. S.I. Nikolenko, K. Kogan, A. Fernàndez Anta. Network Simplification Preserving Bandwidth and Routing Capabilities. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

    Contribution of the dissertation's author: Theorems 3, 4. Theorems 1, 2, 5, 6 in collaboration with K.K., A.F.A. Braess paradox construction in Section IV. Evaluation study.

19. A.P. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Throughput Optimization with Latency Constraints. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

Contribution of the dissertation's author: same as [3] (this is its conference version).

20. P. Chuprikov, K. Kogan, S.I. Nikolenko. General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Contribution of the dissertation's author: problem settings, approaches to reducing classifier width (Section VI).

21. P. Chuprikov, A.P. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Planning in Compute-Aggregate Problems as Optimization Problems on Graphs. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Contribution of the dissertation's author: same as [4] (this is the first conference paper combined in the journal version [4]).

22. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. A Programmable Buffer Management Platform. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Contribution of the dissertation's author: same as [5] (this is the second conference paper combined in the journal version [5]).

23. P. Chuprikov, S.I. Nikolenko, K. Kogan. On Demand Elastic Capacity Planning for Service Auto-scaling. Proceedings of 2016 IEEE International Conference on Computer Communications (INFOCOM 2016), 2016.

    Contribution of the dissertation's author: problem settings, introducing the model, algorithm NRAP in collaboration with K.K.

24. K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. FIB Efficiency in Distributed Platforms. Proc. 24th IEEE International Conference on Network Protocols (ICNP 2016), 2016.

    Contribution of the dissertation's author: same as [10] (this is its conference version).

25. P. Chuprikov, K. Kogan, S.I. Nikolenko. Priority Queueing with Multiple Packet Characteristics. Proceedings of IEEE INFOCOM 2015, pp. 1418–1426.

Contribution of the dissertation's author: same as [8] (this is its conference version).

26. K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). ACM SIGCOMM 2014, ACM Press, 2014, pp. 15–26.

    Contribution of the dissertation's author: same as [14] (this is its conference version).

27. P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Shared-Memory Buffer Management for Heterogeneous Packet Processing. 34th International Conference on Distributed Computing Systems (ICDCS 2014).

    Contribution of the dissertation's author: same as [11] (this is its conference version).

Publications in other venues

28. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Online Scheduling FIFO Policies with Admission and Push-Out. Theory of Computing Systems, vol. 58, no. 2, 2016, pp. 322-344.

    Contribution of the dissertation's author: Theorems 1, 2, 3, 4, 6. Theorems 5 and 7 in collaboration with K.K., A.L., A.S. Experimental study.

29. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Distributed Counting along Lossy Paths Without Feedback. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), Lecture Notes in Computer Science, vol. 11685, 2018, pp. 30–33 (CORE rank B).

    Contribution of the dissertation's author: same as [2] (this is the second conference paper combined in the journal version [2]).

30. P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalizing Compute-Aggregate Problems in Cloud Computing. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), 2018, pp. 377–391 (CORE rank B).

Contribution of the dissertation's author: same as [4] (this is the second conference paper combined in the journal version [4]).

31. P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Essential Traffic Parameters for Shared Memory Switch Performance. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015), Lecture Notes in Computer Science, vol. 9439, Springer, 2015, pp. 61–75. (CORE rank B).

    Contribution of the dissertation's author: same as [9] (this is its conference version).

32. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. A Taxonomy of Semi-FIFO Policies. Proc. 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), 2012, pp. 295–304. (CORE rank B).

    Contribution of the dissertation's author: same as [12] (this is its conference version).

33. K. Kogan, S.I. Nikolenko, V. Demianiuk, P. Chuprikov, A.P. Davydow. Personal Insights on Three Research Directions in Networked Systems. Proc. 10th International Conference on Communication Systems and Networks (COMSNETS 2018), IEEE, 2018.

    Contribution of the dissertation's author: majority of the paper's text (this is a problem setting publication with no novel results presented).

34. P. Chuprikov, K. Kogan, S.I. Nikolenko. How to implement complex policies on existing network infrastructure. Proc. 4th Symposium on SDN Research (SOSR 2018), ACM, 2018, pp. 1–8.

    Contribution of the dissertation's author: Boolean minimization heuristics and incomparability results (Sections 4.3, 4.4).

35. S.I. Nikolenko, K. Kogan, G. Retvari, E.R. Berczi-Kovacs, A. Shalimov. How to Represent IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes. Proc. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): GI 2016: 9th IEEE Global Internet Symposium (IEEE GI 2016), 2016.

Contribution of the dissertation's author: Algorithms 1, 2, 3, 4, 5, 6, in collaboration with K.K., G.R., E.B.K. Simulation study, in collaboration with A.S.

36. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, P. Eugster. BASEL (Buffering Architecture SpEcification Language). Proc. 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2016), 2016, pp. 69–74.

    Contribution of the dissertation's author: same as [5] (this is the first conference paper combined in the journal version [5]).

37. K. Kogan, S.I. Nikolenko, P. Eugster, E. Ruan. Strategies for Mitigating TCAM Space Bottlenecks. 22nd IEEE Annual Symposium on High-Performance Interconnects (HOTI 2014), IEEE, 2014, pp. 25–32.

    Contribution of the dissertation's author: Boolean heuristics for classifier optimization (Section VII), simulation study.

38. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub. Balancing Work and Size with Bounded Buffers. Proc. 6th International Conference on Communication Systems and Networks (COMSNETS 2014), 2014, pp. 1–8.

    Contribution of the dissertation's author: same as [13] (this is its conference version).

39. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Multi-Queued Network Processors for Packets with Heterogeneous Processing Requirements. Proc. 5th International Conference on Communication Systems and Networks (COMSNETS 2013), 2013, pp. 1–10.

    Contribution of the dissertation's author: Theorems 1, 2, 3, 4, 6, 8, 9, 10, 11; Theorems 5, 7 in collaboration with K.K. and A.L.; simulation study in collaboration with A.S.

40. K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient Demand Assignment in Multi-Connected Microgrids with a Shared Central Grid. Proc. 3rd IFIP Conference on Sustainable Internet and ICT for Sustainability 2013 (SustainIT 2013), 2013.

Contribution of the dissertation's author: bounds on the approximation ratio shown in Table 1, in collaboration with K.K., A.L.

41. K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient demand assignment in multi-connected microgrids. Proc. 4th Energy-Efficient Computing and Networking (e-Energy 2013), pp. 277–278.

    Contribution of the dissertation's author: Theorems 1, 2, 3, 4. Theorems 5, 6 in collaboration with K.K., A.L.

42. K. Kogan, S.I. Nikolenko, W. Culhane, P. Eugster, E. Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013), pp. 153–154.

    Contribution of the dissertation's author: applications of MinDNF and complexity analysis of TCAM minimization (Section 2.2).

Chapter 2 Buffer management algorithms and their analysis

2.1 Introduction to competitive analysis for buffer management and related work

Let us begin with an introduction and general definitions of competitive analysis and a brief overview of the main results in competitive analysis for buffer management algorithms that precede our work. The literature overview in this section follows the work [167].

Network switches perform two major tasks:

(1) changing properties of individual packets (recoloring, encapsulation);

(2) changing properties of packet streams (rate-limiting, shaping); the latter may require buffers.

Design and management of a buffering architecture is a fundamental problem in network switch design. To accommodate for network size, number of transport requests, and efficient reuse of network infrastructure, the Internet currently implements "best-effort" servicing that prioritizes some types of traffic to optimize desired objectives (e.g., economic properties of the network). Unfortunately, the current network is mostly built for fairness whereas network operators are looking for methods to deploy other economic models (e.g., throughput or profit). Most admission control policies are based on a simple characteristic (e.g., buffer occupancy), while traffic has additional important characteristics (e.g., processing requirements or values) that are not taken into account. Incorporating new characteristics in admission decisions beyond fairness leads to new challenges in performance and implementation for switch architectures.

A network edge with a limited buffer performs three main tasks, namely:

(1) *buffer management*, i.e., admission control of newly arrived packets and push-out of currently stored packets (if push-out capability is supported);

(2) *processing*, i.e., deciding which of the currently stored packets will be processed;

(3) *transmission*, i.e., deciding if already processed packets should be transmitted and transmitting those that should.

We assume discrete slotted time. A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles.

Each packet may have the following characteristics:

(1) *required processing*, i.e., how many processing cycles the packet has to go through before it can be transmitted;

(2) *value*, i.e., how much the packet contributes to the objective function when it is transmitted;

(3) *output port*, i.e., where the packet is headed; in settings with multiple output ports it is usually assumed that processing occurs independently at each port, so it becomes advantageous to have more busy output ports at a time;

(4) *size*, i.e., how many slots (bytes) a packet occupies in the buffer.

The objective of a buffer management policy is to maximize the total value of transmitted packets. Different settings may assume that some characteristics are uniform. For instance, in this section we do not consider size, assuming that all packets take up exactly one slot in the buffer of size B, but packets with varying size will be considered in Section 2.5.

Competitive analysis provides a uniform throughput guarantee for online algorithms across all traffic patterns.

Definition 2.1.1. *An online algorithm* ALG *is said to be* α-competitive *with respect to some objective function* f *for some* $\alpha \geq 1$*, which is called the* competitive ratio*, if for any arrival sequence* σ *the objective function value on the result of* ALG *is at least* $1/\alpha$ *times the objective function value on the solution obtained by an offline clairvoyant algorithm, denoted* OPT*.*

Note that there are three main kinds of results in competitive analysis:

(1) a *lower bound* on the competitive ratio for a specific algorithm can be proven with a specific hard example; this is the easiest kind of results;

(2) a *general lower bound* on the competitive ratio sometimes can be proven for all deterministic online algorithms or for some specific subclass of algorithms (e.g., for all priority queues); in this case, a lower bound has to be adversarial, with the hard instance depending on the choices an online algorithm might make;

(3) finally, an *upper bound* on the competitive ratio represents a general statement that should hold over all possible inputs, so proving an upper bound is usually a much harder mathematical problem.

Throughout this chapter, we present a number of lower, general lower, and upper bounds on the competitive ratio of many different buffer management algorithms in various settings.

Problem 2.1.1 (competitive ratio)**.** *For a given switch architecture, packet characteristics (e.g., required processing, value, size etc.), and an online algorithm* ALG *in a given setting, prove lower and upper bounds on its competitive ratio with respect to weighted throughput (total value of packets transmitted by an algorithm).*

Problem 2.1.2 (competitive policies)**.** *For a given switch architecture and packet characteristics (e.g., required processing, value, size etc.), find online algorithms that have the best (smallest) possible upper bounds on their competitive ratios.*

In the rest of this section, policies and lower and upper bounds on their competitive ratios are outlined according to problem settings; the latter differ in which packet characteristics they assume to be uniform and which are allowed to vary, sometimes with additional restrictions imposed on admission, processing and/or transmission order, and admissible packet characteristics. Sometimes also the so-called *two-valued case* is distinguished, when a characteristic can take one of two possible values, usually assumed to be 1 and $x > 1$.

Uniform processing, uniform value, shared memory switch. Since all packets are identical, so the problem for a single queue with one output port is trivial. We consider an $M \times N$ shared memory switch that can hold B packets, with a separate processor on each output port. All packets require a single processing

cycle and have equal value; the goal is to maximize the number of transmitted packets. Each packet is labeled with an output port where it has to be processed and transmitted.

Non-push-out policies. Kesselman and Mansour [136] show an adversarial logarithmic lower bound: no non-push-out policy can achieve competitive ratio better than $d/2$ for $d = \log_d N$. On the positive side, they present the Harmonic policy that allocates approximately $1/i$ of the buffer to the ith largest queue, and for its variant, the Parametric Harmonic policy, show an upper bound of $c \log_c N + 1$.

Push-out policies. The best known policy is Longest Queue Drop (LQD): accept packets greedily if the buffer is not full; if it is, accept the new packet and then drop a packet from the longest queue (destined to the output port with the most packets assigned to it). Aiello et al. [44, 114] show that the competitive ratio of LQD is between $\sqrt{2}$ and 2; they also provide non-constant lower bounds for other popular policies and a general adversarial lower bound of $\frac{4}{3}$ on the competitive ratio of any online algorithm.

Uniform processing, uniform value, multiple separated queues. In an $N \times 1$ switch where each of N input queues has a separate independent buffer of size B, a policy must select which input queue to take a packet from and set admission policies for input queues. Azar and Richter [59] prove that any work-conserving policy (a policy that keeps processing while input queues are nonempty) is at most 2-competitive; this covers both push-out and non-push-out case since with uniform values push-out is unnecessary. They also show a lower bound of $2 - \frac{1}{N}$ for $B = 1$ and $1.366 - \Theta(1/N)$ for arbitrary B [59]. Albers and Schmidt [48] show that the greedy policy that takes a packet from the longest queue is at least $\left(2 - \frac{1}{B}\right)$-competitive and design a new SemiGreedy policy which they prove to be 1.889-competitive for any B and 1.857-competitive for $B = 2$, which is a tight bound. Finally, Azar and Litichevskey [58] closed this problem, designing a deterministic policy whose competitive ratio tends to $\frac{e}{e-1} \approx 1.582$ for arbitrary B, with a matching lower bound already shown in [59].

Uniform processing, variable values, single queue. Here, there is only one output port (a single queue), and each packet is fully processed in one cycle; however, packets have different values, making it desirable to drop packets with

smaller value and process packets of larger value. It is easy to show that the Priority Queue (PQ) policy that sorts packets with respect to values and pushes out smaller values for larger ones is optimal. Research has concentrated on models with additional constraints: non-push-out policies that are not allowed to push admitted packets out and the FIFO model where packets have to be transmitted in order of arrival. Another important special case considers two possible values, 1 and $V > 1$.

Non-push-out policies. Aiello et al. [52] consider five online policies for the two-valued case, considering the specific cases of $V = 1$, $V = 2$, and $V = \infty$. Andelman, Mansour, and Zhu provide a deterministic policy (Ratio Partition) that achieves optimal $\left(2 - \frac{1}{V}\right)$-competitiveness [212]. In the case of arbitrary values between 1 and $V > 1$, they show that the optimal competitive ratio is $\ln V$, proving tightly matching bounds of $1 + \ln V$ and $2 + \ln V + O(\ln^2 V/B)$ [52, 212].

Push-out policies. In the push-out case, we have already mentioned that PQ able to rearrange packets is optimal. In the FIFO model, there has been a line of adversarial lower bounds culminating in the lower bound of 1.419 shown by Kesselman, Mansour, and van Stee [138] that applies to all algorithms, with a stronger bound of 1.434 for $B = 2$ [52, 212]. As for upper bounds, in this simple model the FIFO greedy push-out policy (accept every packet to end of queue, then push out the packet with smallest value if buffer has overflown) has been shown by Kesselman et al. to be 2-competitive [133]; in the two-valued case, they provide an adversarial lower bound of 1.282, and a long line of improvements for the upper bound has led through a tight upper bound of $\left(2 - \frac{2}{V+1}\right)$ for the greedy push-out policy [133], loss-bounded analysis of Kesselman and Mansour [135], and a policy of Lotker and Patt-Shamir with upper bound $\left(1 + \frac{1}{\sqrt{V}} + O(\frac{1}{V})\right)$ [159], to the optimal Account Strategy policy of Englert and Westermann [102]. They show an adversarial lower bound of $r = \frac{1}{2}(\sqrt{13} - 1) \approx 1.303$ for any $B \geq 2$ and $r_\infty = \sqrt{2} - \frac{1}{2}(\sqrt{5 + 4\sqrt{2}} - 3) \approx 1.282$ for $B \to \infty$ and show that Account Strategy achieves competitive ratio r for arbitrary B and r_∞ for $B \to \infty$. Thus, in the push-out two-valued case the gap between lower and upper bounds has been closed completely.

Uniform processing, variable values, multiple separated queues. Kawahara et al. [124] consider an $N \times 1$ switch with N separated queues, each of

which has a distinct buffer of size B and has a value α_j associated with it, $1 = \alpha_1 \leq \ldots \leq \alpha_N = \alpha$. A policy selects one of N queues, maximizing total transmitted value; the work [124] provides matching lower and upper bounds for the PQ policy as $1 + \frac{\sum_{j=1}^{n'} \alpha_j}{\sum_{j=1}^{n'+1} \alpha_j}$, where $n' = \arg\max_n \frac{\sum_{j=1}^{n} \alpha_j}{\sum_{j=1}^{n+1} \alpha_j}$, and an adversarial lower bound $1 + \frac{\alpha^3+\alpha^2+\alpha}{\alpha^4+4\alpha^3+3\alpha^2+4\alpha+1}$ for any online algorithm. Azar and Richter [59] show that any r-competitive policy for a FIFO queue with variable values yields a $2r$-competitive policy for multiple queues. Kobayashi et al. [145] show that an r-competitive policy for unit values and multiple queues yields a $\min\left\{Vr, \frac{Vr(2-r)+r^2-2r+2}{V(2-r)+r-1}\right\}$-competitive policy for the two-valued case.

Uniform processing, CIOQ switches. A critical aspect of the switching architecture is the placement of buffers. In the output queuing (OQ) architecture, packets arriving from the input lines immediately cross the switching fabric, and join a queue at the switch output port. Thus, the OQ architecture both allows to maximize throughput and permits accurate control of packet latency. However, in order to avoid contention, the internal speed of an OQ switch must be equal to the sum of all the input line rates. Recent developments in networking technology have resulted in dramatic growth in line rates, which has made the internal speedup requirements of OQ switches difficult to meet. This has in turn generated great interest in the input queuing (IQ) switch architecture, where packets arriving from the input lines are queued at the input ports. The packets are then extracted from the input queues to cross the switching fabric and to be forwarded to the output ports. It is well known that IQ architecture may lead to low throughput, and it does not allow for controlling the latency through the switch. The main problem of the IQ architecture is head-of-line (HOL) blocking, which occurs when packets at the head of various input queues contend on a specific output port of the switch. To alleviate the problem of HOL blocking, one can maintain at each input a separate queue for each output. This technique is known as virtual output queuing (VOQ). Another way to get the delay guarantees of an IQ switch closer to that of an OQ switch is to increase the *speedup* S of the switching fabric. A switch is said to have a speedup S, if the switching fabric runs S times faster than each of the input or the output ports. Hence, an OQ switch has a speedup of N (where N is the number of input/output ports), whereas an IQ switch has a speedup of 1. For values of S between 1 and N, packets need to be buffered at the

inputs before switching as well as at the outputs after switching. This architecture is called Combined-Input-and-Output Queued (CIOQ) switch.

Uniform values. Consider an $N \times N$ CIOQ switch with speedup S. Packets of equal size arrive at input ports, each labeled with the output port where it has to leave the switch. Each packet is placed in the input queue corresponding to its output port; when it crosses the switch fabric, it is placed in the output queue and resides there until it is sent on the output link. For unit-valued packets, Kesselman and Rosén [139] proposed a non-push-out policy which is 3-competitive for any S and 2-competitive for $S = 1$.

Variable values. For up to m packet values in $[1, V]$, Kesselman and Rosén [139] show two push-out policies to be $4S$- and $8 \min\{m, 2 \log V\}$-competitive. Azar and Richter [60] propose a push-out policy β-PG with parameter β; Kesselman et al. [131] show that the competitive ratio of β-PG is at most 7.5 for $\beta = 3$ and at most 7.47 for $\beta = 2.8$. Kesselman and Rosén [140] consider CIOQ switches with PQ buffers (transmit the highest value packet) and show that this policy is 6-competitive for any S.

Uniform processing, crossbar switches. In the buffered crossbar switch architecture, a small buffer is placed on each crosspoint in addition to input and output queues, which greatly simplifies the scheduling process. For packets with unit length and value, Kesselman et al. [131] introduce a greedy switch policy with competitive ratio between $\frac{3}{2}$ and 4 and show a general lower bound of $\frac{3}{2}$ for unit size buffers. For variable values and PQ buffers, they propose a push-out greedy switch policy with preemption factor β with competitive ratio between $(2\beta - 1)/(\beta - 1)$ (3.87 for $\beta = 1.53$) and $(\beta + 2)^2 + 2/(\beta - 1)$ (16.24 for $\beta = 1.53$). For variable values and FIFO buffers, they propose a β-push-out greedy switching policy with competitive ratio $6 + 4\beta + \beta^2 + 3/(\beta - 1)$ (19.95 for $\beta = 1.67$) [130].

Uniform values, variable processing, single queue. In this setting, each packet contributes one unit to the objective function, but different packets have different processing requirements, i.e., they spend a different number of time slots at the processor. We denote maximal possible required processing by k.

Non-push-out policies. For a single queue and packets with heterogeneous processing, non-push-out policies had not been considered in any detail before the

results shown below in Section 2.2.

Push-out policies. Keslassy et al. [128] showed that again, for a single queue PQ (Priority Queue) that sorts packets with respect to required processing (smallest first) is optimal; research has concentrated on the FIFO case, where packets have to be transmitted in order of arrival. This area was significantly advanced by our results shown in Section 2.2.

Copying cost. An important generalization of the heterogeneous processing model was introduced by Keslassy et al. [128]. They attach a penalty α called copying cost to admitting a packet in the queue; thus, the objective function is now $T - \alpha A$, where T is the number of transmitted packets and A is the number of accepted ones, and it becomes less advantageous to push packets out. To deal with copying cost, the authors propose to use β-push-out policies that push a packet out only if its required processing is at least $\beta > 1$ times less than the required processing of a packet which is being pushed out. Keslassy et al. [128] consider the PQ_β policy (Priority Queue with β-Push-Out) and show that it is at most $\frac{1}{1-\alpha \log_\beta k}\left(1 + \log_{\frac{\beta}{\beta-1}} \frac{k}{2} + 2\log_\beta k)(1-\alpha)\right)$-competitive. Throughout this section, we show a number of results for model variations with nontrivial copying cost.

2.2 Buffer management for packets with heterogeneous processing for a single queue

The modern network edge is required to perform tasks with heterogeneous complexity, including features such as advanced VPNs services, deep packet inspection, firewall, and intrusion detection, to list just a few. Each of these features may require a different amount of processing at the network processor and may introduce new challenges for traditional architectures, posing implementation and performance issues. All such features directly affect processing delay. As a result, the processing order of packets and the way how these packets are processed can have a significant impact not only on the delay and throughput but also can cause reordering of transmitted packets.

In this section, we study the single queue setting for packets with heterogeneous processing. We denote a packet with required processing r by $\boxed{r}$, and a sequence of n packets with the same required processing r by $n \times \boxed{r}$. If a packet is

dropped prior to being transmitted (while it still has a positive number of required processing cycles), it is lost. A packet may be dropped either upon arrival or due to a push-out decision while it is stored in the buffer. A packet contributes one unit to the objective function only upon being successfully transmitted. The goal is to devise buffer management algorithms that maximize the overall throughput, i.e., the total number of packets transmitted out of the queue.

The queue performs three main tasks, namely:

(1) *buffer management*, i.e., admission control of newly arrived packets and push-out of currently stored packets (if push-out capability is supported);

(2) *processing*, i.e., deciding which of the currently stored packets will be processed;

(3) *transmission*, i.e., deciding if already processed packets should be transmitted and transmitting those that should.

A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles. Our framework assumes a multi-core environment with C processors (cores), so at most C packets may be chosen for processing at any given time (two cores are not allowed to process the same packet simultaneously). We assume discrete slotted time, where each time slot consists of three phases (see Fig. 2.1 for an example):

(1) *arrival*: new packets arrive, and admission control decides if a packet should be dropped or, possibly, should push out an already admitted packet;

(2) *processing*: C packets are selected for processing by the scheduling unit;

(3) *transmission*: at most one fully processed packet is selected for transmission and leaves the queue.

For an algorithm ALG and time slot t, we denote by IB_t^{ALG} the set of packets stored in ALG's buffer at time slot t after arrival but before processing (i.e., the buffer state shown in the second row of Fig. 2.1). The number of *processing cycles* of a packet is key to our algorithms. Formally, for every time slot t and every packet p currently stored in the queue, its number of *residual processing cycles*, denoted $r_t(p)$, is defined to be the number of processing cycles it requires before

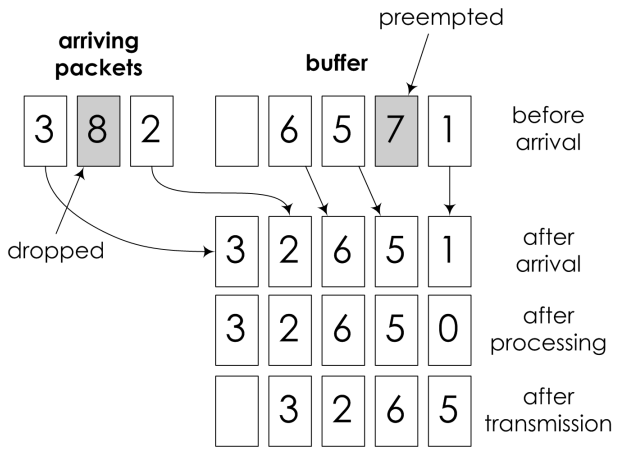


Figure 2.1: Zoom in on one time slot of a FIFO push-out policy. The incoming packet with $r = 8$ is dropped (not admitted), and the packet with $r = 7$ is pushed out of the queue to make space for the packet with $r = 3$.

it can be successfully transmitted. We define a *greedy* buffer management policy as a policy that accepts all arrivals if there is available buffer space in the queue. A policy is *work-conserving* if it always processes whenever its buffer is nonempty. We say that an arriving packet p *pushes out* a packet q that has already been accepted into the buffer when q is dropped in order to free buffer space for p, and p is admitted to the buffer instead in FIFO order. A buffer management policy is called a *push-out* policy whenever it allows packets to push out currently stored packets. Figure 2.1 shows a sample time slot in our model (for the greedy push-out case).

We begin with results about three algorithms. The Non-Push-Out algorithm NPO is a simple greedy work-conserving policy that accepts a packet if there is enough available buffer space. The Push-Out algorithm PO is also greedy and work-conserving, but now, when an arriving packet p requires less processing cycles than at least one packet in its buffer, PO pushes out the first packet with the maximal number of processing cycles in its buffer and accepts p according to FIFO order. For processing, PO always selects the first packet in the queue. We introduce a new Lazy-Push-Out algorithm LPO that mimics the behaviour of PO with two important differences:

(1) LPO does not transmit a head of line packet with a single processing cycle if its buffer contains at least one packet with more than one residual processing cycle, until the buffer contains only packets with a single residual processing cycle;

(2) once all packets in LPO's buffer (say there are m packets there) have a single processing cycle remaining, LPO transmits them over the next m processing cycles.

LPO has push-out similar to PO: if an arriving packet p requires less processing than the first packet q with maximal number of processing cycles in LPO's buffer, p pushes out q. Intuitively, LPO is a weakened version of PO since PO tends to empty its buffer faster.

We have performed detailed competitive analysis for these algorithms, obtaining the following results (proofs and more details are given in [28] and [147]).

Theorem 2.2.1 (incomparability of PO and LPO in the worst case)**.**

1. *There exists a sequence of inputs on which* PO *processes* $\geq \frac{3}{2}$ *times more packets than* LPO*.*

2. *There exists a sequence of inputs on which* LPO *processes* $\geq \frac{5}{4}$ *times more packets than* PO*.*

Theorem 2.2.2 (tight lower and upper bounds on the competitiveness of NPO)**.** *For a sufficiently long arrival sequence, the competitiveness of* NPO *is at least* k *and at most* k*.*

Theorem 2.2.3 (lower bounds for LPO and PO)**.** *The competitive ratio of both* LPO *and* PO *is at least* $2\left(1-\frac{1}{B}\right)$ *for* $k \geq B$*. The competitive ratio for* $k < B$ *is at least* $\frac{2k}{k+1}$ *for* PO *and at least* $\frac{2k-1}{k}$ *for* LPO*.*

Theorem 2.2.4 (logarithmic lower bound for large k)**.** *The competitive ratio of both* PO *and* LPO *is at least* $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$*.*

Our first main result is an upper bound on the competitiveness of LPO.

Theorem 2.2.5 (upper bound for LPO)**.** LPO *is at most* $(\max\{1, \ln k\} + 2 + o(1))$*-competitive.*

We have also studied a special two-valued case when there are only two possible required processing values, 1 and k, so there are only two kinds of packets, $\boxed{1}$ and $\boxed{k}$. This case often occurs in practice; for instance, $\boxed{1}$ may represent "commodity" packets that are processed to completion while $\boxed{k}$ corresponds to packets requiring advanced processing of some kind, e.g., DPI (deep packet inspection).

Theorem 2.2.6 (lower bounds for LPO and PO in the two-valued case)**.**

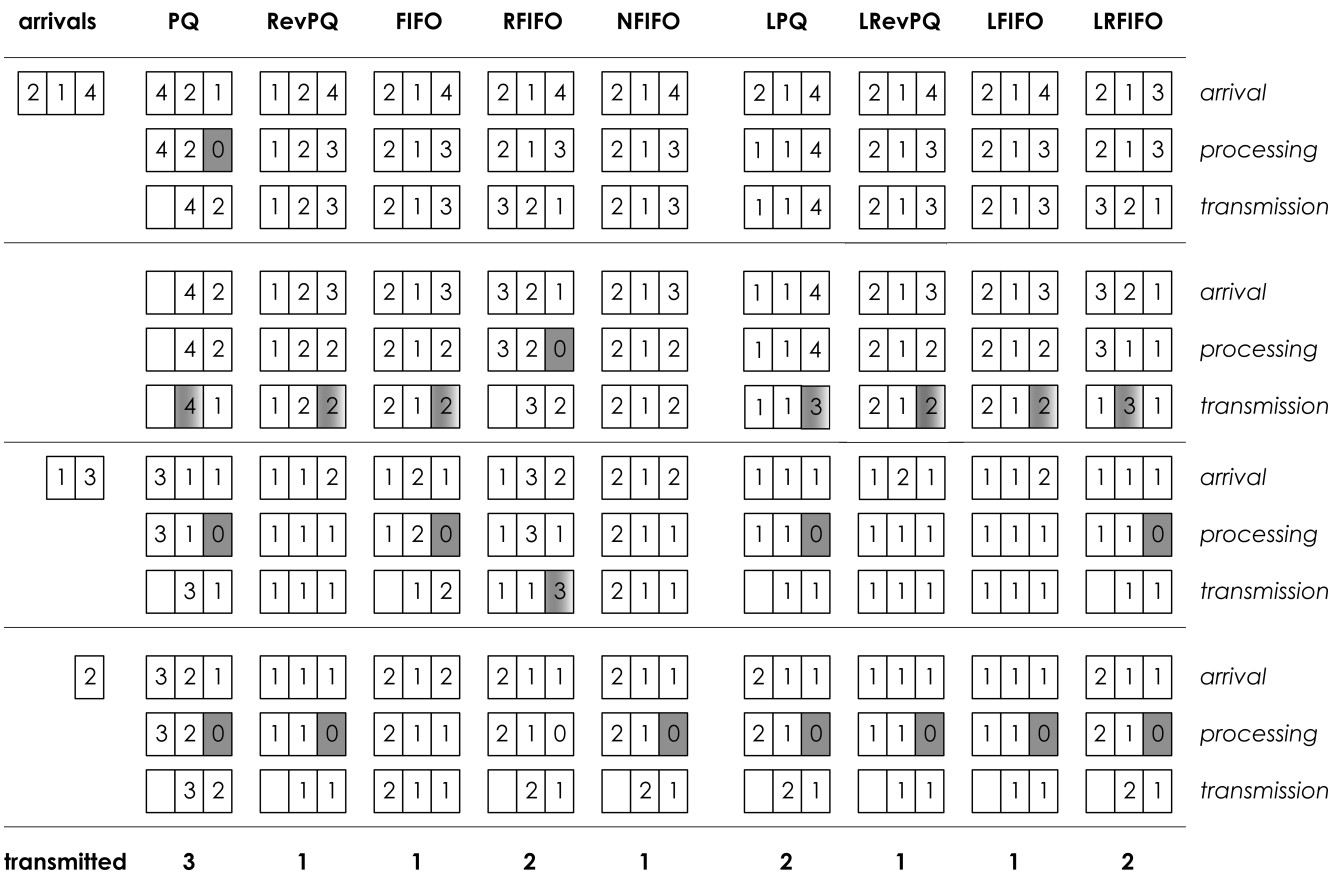


Figure 2.2: Sample operation of the algorithms. On the left, regular policies; on the right, lazy Semi-FIFO policies. Time flows from top to bottom, HOL packets are shown on the right of each buffer, all algorithms start with an empty buffer and have $B = 3$. Shaded rectangles denote transmitted packets; rectangles with gradient fill, packets that will be pushed out (for a packet with $r(p) = 1$ that arrives on the third time slot or for a packet with $r(p) = 2$ on the fourth time slot). In each time slot, three buffer states are shown: after arrival, processing, and transmission respectively.

1. The competitive ratio of LPO *in the two-valued case is at least* $2 - \frac{1}{k}$ *for* $B \geq k$.

2. The competitive ratio of PO *in the two-valued case is at least* $\frac{2}{1+\frac{1}{k}}$ *for* $B \geq k$.

In this case, it becomes possible to prove a constant upper bound for LPO.

Theorem 2.2.7 (upper bound for LPO in the two-valued case)**.** *In the two-valued case,* LPO *is at most* $\left(2 + \frac{1}{B}\right)$*-competitive.*

Next, we consider the family of *Semi-FIFO* policies that may process packets in different orders but transmit them in the order of arrivals to avoid reordering. A policy is usually defined by a subset of the above properties and its processing order. In what follows, we consider the following processing orders:

(1) *priority queueing* (PQ), where a packet with minimal residual work is pro-

cessed first;

(2) *reversed priority queueing* (RevPQ), where a packet with maximal residual work is processed first;

(3) *FIFO*, a regular first-in-first-out processing order;

(4) *FIFO with recycles* (RFIFO), where non-fully processed packets Tare recycled to the back of the queue after each processing step.

In case that the order of arrivals and processing coincide for any input, the algorithm name is defined by the name of processing order (e.g. PQ algorithm processes and transmits packets in PQ order). Figure 2.2 shows a sample operation for the algorithms we consider below; the five algorithms on the left correspond to these four processing orders together with the non-push-out algorithm.

We introduce a more general notion of lazy algorithms.

Definition 2.2.1. *A buffer processing policy* LA *is called* lazy *if it satisfies the following conditions:*

(1) LA *is greedy, i.e., accepts packets if its buffer is not full;*

(2) LA *pushes out the first packet with maximal number of processing cycles in case of congestion;*

(3) LA *does not process and transmit packets with a single processing cycle if its buffer contains at least one packet with more than one processing cycle left;*

(4) once all packets in LA*'s buffer* (*say* m *packets*) *have a single processing cycle remaining,* LA *transmits them in arrival order over the next* m *time slots, even if additional packets arrive* (*and are accepted*) *during that time.*

Definition 2.2.1 allows for a well-defined accounting infrastructure. In particular, it lets us define an *iteration* over which we can count the number of packets transmitted by the optimal algorithm and compare it to the contents of LA's buffer. The first iteration begins with the first arrival. An iteration ends when all packets in the LA buffer have a single processing pass left. Each subsequent iteration starts after LA has transmitted all packets from the previous iteration. Since all packets at the end of an iteration have a single processing cycle, these

packets can be transmitted in any order (in our case, FIFO). On Fig. 2.2, LPQ and LRFIFO fill their buffer with $\boxed{1}$s and begin transmission on the third time slot; LRevPQ and LRFIFO, on the fourth. Definition 2.2.1 covers a wide array of policies because it does not specify the processing order during an iteration. The processing order matters since it determines how much processing is "wasted" on pushed out packets. However, lazy policies are only a subclass of Semi-FIFO policies.

The second main result of this part is a general upper bound on the competitive ratio of all lazy processing policies.

Theorem 2.2.8 (general upper bound for LA)**.** *The competitive ratio of* LA *is at most*

$$\frac{1}{B}\log_{B/(B-1)} k + 3.$$

Note that asymptotically, for $k, B \to \infty$, this is a logarithmic bound in k:

$$\frac{1}{B}\log_{B/(B-1)} k = \frac{\ln k}{B \ln\left(1 + \frac{1}{B} + o(\frac{1}{B})\right)} = \ln k + o(\ln k).$$

There exist processing orders for which this bound is tight.

Theorem 2.2.9 (lower bound for LRFIFO, LRevPQ, RFIFO)**.** *The competitive ratio of* LRFIFO*,* LRevPQ*, and* RFIFO *is at least* $(1 + \frac{1}{B}\log_{B/(B-1)} k)$*.*

A natural question arises: is there any processing order with competitive ratio much better than this? To show that the processing order during an iteration has significant impact on the resulting behaviour, we consider the lazy Semi-FIFO policy with PQ processing order during iteration (LPQ), showing a constant upper bound.

Theorem 2.2.10 (matching lower and upper bounds for LPQ)**.**

1. LPQ *is at most* 2*-competitive.*
2. LPQ *is at least* $\left(2 - \frac{1}{B}\left\lceil\frac{B}{k}\right\rceil\right)$*-competitive.*

In fact, we can use the construction from Theorem 2.2.10(2) to obtain a general lower bound for every Semi-FIFO policy.

Theorem 2.2.11. *Any greedy Semi-FIFO policy is at least* $\left(1 + \frac{m-1}{B}\right)$*-competitive for* $m = \min\{k, B\}$*.*

We have also shown incomparability results between several different semi-FIFO policies.

Theorem 2.2.12 (incomparability among semi-FIFO policies)**.** *Any lazy policy* LA *(including* LRevPQ*) is incomparable with either* FIFO *or* RFIFO *in the worst case for every* $k > 2$ *and* $B > 2$.

For non-push-out policies, linear lower bounds can be proven.

Theorem 2.2.13 (general lower bounds for non-push-out policies)**.** *Any greedy non-push-out Semi-FIFO policy* NPO *is at least* $\frac{k+1}{2}$*-competitive. Any lazy greedy non-push-out policy* NLPO *is at least* $(k-1)$*-competitive.*

In some systems we want to control the number of pushed out packets. This is done for at least two reasons: each admitted packet incurs some *copying cost* α, and pushing a packet out is not a simple operation. The addition of a copying cost α covers both cases. To this end, Keslassy et al. [127] introduced the notion of copying cost in the performance of transmission algorithms. Namely, if for an input sequence σ an online algorithm accepts A packets and transmits T packets, its transmitted value is $\max(0, T - \alpha A)$.

Next we show the third main result of this section, a general upper bound for any lazy β-push-out Semi-FIFO policy — one more application of the powerful accounting infrastructure provided by lazy algorithms.

Theorem 2.2.14 (upper bound for β-push-out LA)**.** *For copying cost* α *satisfying* $0 < \alpha < \frac{1}{\log_\beta k}$*, the competitive ratio of* LA_β *is at most*

$$\left(3 + \frac{1}{B} \log_{\frac{\beta B}{\beta B - 1}} k\right) \frac{1-\alpha}{1 - \alpha \log_\beta k}.$$

It is also possible to prove a general lower bound in this case.

Theorem 2.2.15 (lower bound for β-push-out LA)**.** *For copying cost* α *satisfying* $0 < \alpha < \frac{1}{\log_\beta k}$*, the competitive ratio of any* β*-push-out policy is at least*

$$\frac{1-\alpha}{1 - \alpha \log_\beta k}.$$

We have also extended results for the two-valued case, discussed above for LPO and PO, to the more general case of lazy policies, including the case of β-push-out policies.

Theorem 2.2.16 (upper bound for LA in the two-valued case)**.** *In the two-valued case,* LA *is at most* $\left(2+\frac{1}{B}\right)$*-competitive.*

Theorem 2.2.17 (upper bound for β-push-out LA in the two-valued case)**.** *For copying cost* α *satisfying* $0<\alpha<\frac{1}{2}$*, the competitive ratio of* LA_β *for* $\beta<k$ *in the two-valued case (when only* $\boxed{1}$ *and* $\boxed{k}$ *packets can enter the system) is at most*

$$\left(2+\frac{1}{B}\right)\frac{1-\alpha}{1-2\alpha}.$$

Theorem 2.2.18 (lower bound for β-push-out LA in the two-valued case)**.** *For copying cost* α *satisfying* $0<\alpha<\frac{1}{2}$*, the competitive ratio of* LA_β *for* $\beta<k$ *in the two-valued case is at least*

$$\frac{(2B-2)(1-\alpha)}{(B-1)(1-2\alpha)+(1-\alpha)}.$$

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms, suggest specific use cases and situations suitable for different algorithms, and open up possibilities for practical implementations. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [12, 28, 32].

2.3 Buffer management for multiple queues separated by processing requirements

In Section 2.2, we have considered a single queue that accepts packets with heterogeneous processing requirements. In such environments, a policy that can change processing order of packets and that implements a push-out mechanism (i.e., has an ability to push out already admitted packets) is required to optimize throughput; in the case of a single queue, a greedy push-out policy PQ that processes a packet with minimal required processing first has been shown to be optimal with respect to throughput [127]. To understand the impact of push-out on throughput performance, in Section 2.2 we showed that a greedy push-out policy that implements FIFO or FIFO-with-recycles processing order is at least $\Omega(\log k)$-competitive versus the optimal clairvoyant algorithm, where k is the maximal possible required processing per packet, and at the same time non-push-out poli-

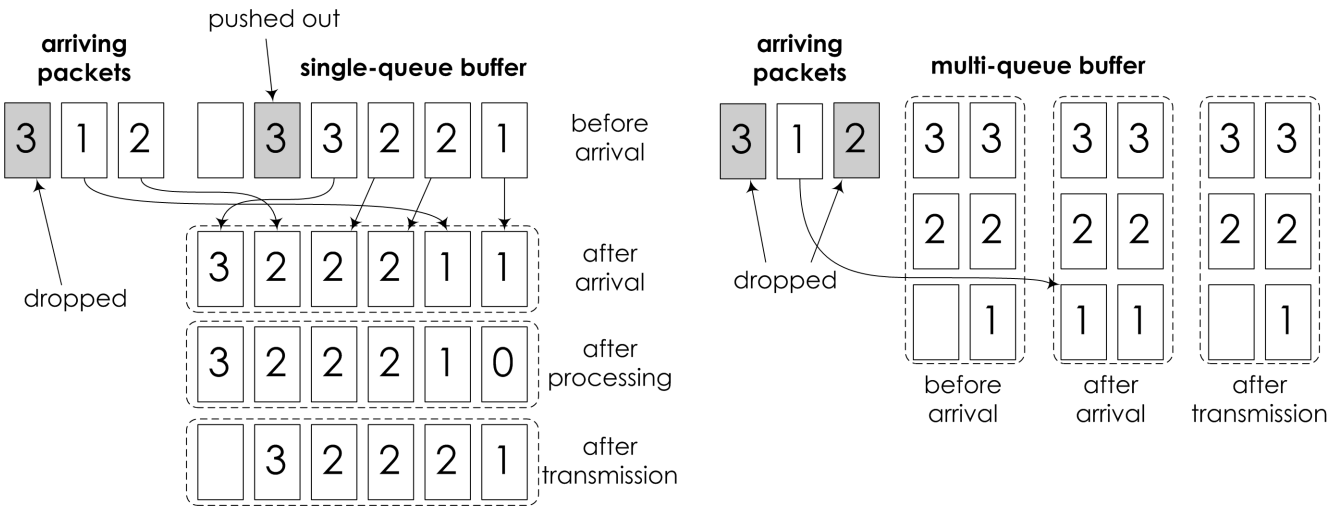


Figure 2.3: A sample time slot. On the left: a single priority queue with buffer of size $B = 6$; on the right: a multi-queued switch with three queues (here $k = 3$) and buffer of size $B = 2$ each. Dashed lines enclose buffers. Note that the sets of packets in the buffers at the end of the time slot are different.

cies with the same processing orders have poor ($\Omega(k)$-competitive) performance in the worst case.

The question that we study in this section is to find a simplified architecture that has, on the one hand, comparable throughput guarantees to a single queue architecture that implements PQ order but, on the other hand, does not require one to implement a push-out mechanism and advanced processing order. As an example of such an alternative, we consider a multi-queued system where each queue accepts packets with a given processing requirement, and the total size of all queue buffers is the same as the buffer size in the single queue architecture (see Fig. 2.3). One major advantage of such a system is a significantly simpler implementation: since each queue holds packets with the same processing requirement, there is no need for a push-out mechanism, and a given queue simply processes packets in FIFO order. This separation provides an additional opportunity to allocate different processing bandwidths to packets from different queues in order to implement more sophisticated QoS mechanisms; admission control can be applied to each queue separately, in a fully distributed manner. Moreover, if queues are implemented as separate buffers then total memory bandwidth increases proportionally to the number of queues. An obvious drawback of this architecture is the preconfigured limit on the size of each queue, so that buffers are not shared between packets with different processing requirements. However, in many applications this is not a limitation but rather an additional advantage since it is

desirable to limit the number of packets from the same class. In this section, we explore the effect of such system simplifications on performance.

First, we concentrate on the buffer occupancy characteristic and propose the Longest-Queue-First (LQF) online policy that proactively processes the HOL packet of a currently longest queue to accommodate with the future potential incoming burst.

Theorem 2.3.1 (lower bound for LQF)**.** LQF *is at least* $\frac{m}{2}$*-competitive for* $m = \min\{k, B\}$.

Next, we introduce a different policy that also uses the buffer occupancy characteristic but processes a packet from a currently shortest queue. We call this policy Shortest-Queue-First (SQF).

Theorem 2.3.2 (lower bound for SQF)**.** SQF *is at least* k*-competitive.*

Theorems 2.3.1 and 2.3.2 demonstrate that there is little difference in the worst case between policies based on the buffer occupancy characteristic and a policy that processes any available packet first (note that any reasonable policy has a trivial upper bound of k since as long as packets are available, it is able to process at least one packet per k time slots). Next, we consider the second characteristic and introduce another policy that, in contrast to the previous ones, does not care about the current buffer occupancy and processes a packet from a non-empty queue with minimal required processing. We call it Minimal-Queue-First (MQF). For the lower bound, note that MQF may spend substantial effort processing a queue which will never be congested.

Theorem 2.3.3 (lower bound for MQF)**.** MQF *is at least* $\left(1 + \frac{k-1}{2k}\right)$*-competitive.*

Theorem 2.3.3 demonstrates that processing requirement potentially has a greater impact on the throughput than buffer occupancy. The opposite of MQF, a policy MaxQF that processes maximal valued queue first, is obviously the worst policy among those we consider in this section.

Theorem 2.3.4 (lower bound for MaxQF)**.** MaxQF *is at least* k*-competitive.*

The main result of this section is the following constant upper bound.

Theorem 2.3.5 (upper bound for MQF)**.** MQF *is at most* 2*-competitive.*

We have also studied the special case when there are only two types of packets, $\boxed{a}$ and $\boxed{b}$, corresponding to only two queues in the system. For this special case, we show exactly matching lower and upper bounds.

Theorem 2.3.6 (lower bound for MQF in the two-valued case)**.** *The competitiveness of* MQF *with two queues is at least*

$$\left(1 + \frac{1 + \left\lfloor \frac{aB-1}{b} \right\rfloor}{B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil}\right).$$

Theorem 2.3.7 (upper bound for MQF in the two-valued case)**.** *The competitiveness of* MQF *with two queues is at most*

$$\left(1 + \frac{1 + \left\lfloor \frac{aB-1}{b} \right\rfloor}{B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil}\right).$$

Next, we evaluate the impact of implementation of *fairness* on performance of throughput. We consider fairness for both packet and processing cycle levels. Since LQF is at least $\frac{m}{2}$-competitive, where $m = \min\{k, B\}$, we do not expect to get better throughput performance for policies that implement the fairness property on any level. First, we try to implement fairness on the level of a single packet. The *Packet-Round-Robin* policy (PRR) implements a round-robin scheme between currently active queues on the packet level by switching to the next active queue after processing the HOL packet in the current queue.

Theorem 2.3.8 (lower bound for PRR)**.** PRR *is at least* $\frac{3k(k+2)}{4k+16}$*-competitive.*

Although the result is predictable, it is interesting that PRR can outperform SQF. Next, consider the Cycle-Round-Robin policy (CRR) with a round-robin scheme between active queues that switches to the next active queue after each processing cycle.

Theorem 2.3.9 (lower bound for CRR)**.** CRR *is at least* $\frac{k}{H(k)}$*-competitive for* $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$.

Finally, we prove several worst-case incomparabilities between the considered policies.

Theorem 2.3.10. MQF *is incomparable with* SQF*,* LQF*,* CRR*, and* PRR*.*

Theorem 2.3.11.

1. PRR *is incomparable with* SQF*,* LQF*, and* CRR*;*

2. CRR *is incomparable with* SQF *and* LQF*;*

3. SQF *and* LQF *are incomparable.*

Apart from these results, we have also studied the possibilities for simulations between multi-queue and single-queue settings. This is an important question for how complex should the queue abstraction and implementation in software-defined buffer management. Consider first a single queue buffering architecture of size B, uniformly sized packets with individual values, and the objective of maximizing the total transmitted value (weighted throughput); this is the setting of Section 2.2. In this case, we already know from Section 2.2 that there exists no deterministic online optimal algorithm for the single queue architecture, weighted throughput objective, and FIFO processing. Moreover, we know that PQ is optimal in this architecture, and can be separated from (shown to be strictly better than) any FIFO algorithm.

Theorem 2.3.12 (strict optimality of PQ)**.** *In the single queue architecture, PQ works at least as well, than any online deterministic policy with FIFO processing on any input, and there exist inputs where it works strictly better.*

Consider next a multiple queue architecture with the same buffer size as the single queue architecture above, where each queue has FIFO processing and is dedicated to packets with the same value (possibly the same value for several queues). This setting simplifies processing and moves the complexity to scheduling decisions. Now the algorithm that pushes out the least valuable and schedules the most valuable packet transmits the same total value as PQ.

Theorem 2.3.13 (multiple queue architecture simulating a single queue)**.** *There exists an online deterministic algorithm for the multi-queue architecture with FIFO processing at every queue that works as well as PQ on the single queue with the same arrivals and hence is better than any deterministic online single queue algorithm with FIFO processing.*

It turns out that for the same buffer size B these settings are equally expressive.

Theorem 2.3.14 (single queue architecture simulating multiple queues)**.** *For any deterministic (or probabilistic) multi-queue policy ALG, there exists a single queue policy that for any input sequence transmits exactly the same set of packets (or a set of packets with the same distribution) as ALG.*

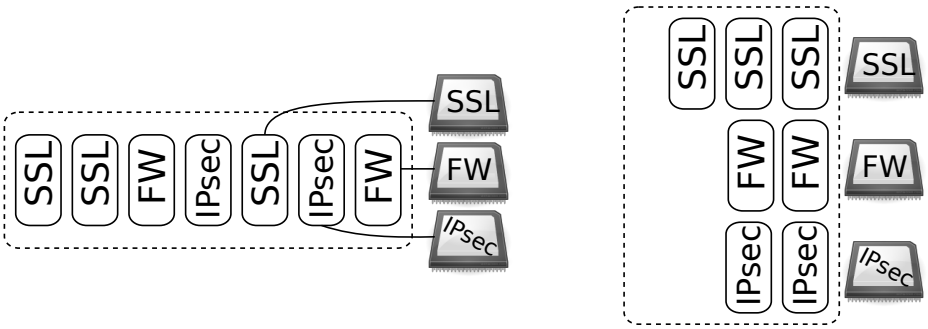


Figure 2.4: Two different buffer management models. Left: each core can process any traffic type, IPsec, SSL, or firewall (FW); right: each core processes a specific traffic type from its own queue, but all queues share the same buffer.

Theorem 2.3.14 shows that in theory, our single queue architecture is as expressive as the multi-queue architecture for any objective and any combination of packet characteristics for both deterministic and stochastic policies. However, additional constraints and parameters such as the time complexity of operations may arise that can make one buffering architecture preferable.

Theorem 2.3.15 (complexity separation between single queue and multi-queue architectures)**.** *In the worst case, the multi-queue architecture with m queues has time complexity of operations at least $O(m/\log B)$ times better than the single queue architecture simulating the same order.*

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [5, 22, 36].

2.4 Buffer management for packets with heterogeneous processing and multiple output ports

So far, we have considered only buffer management in a single queue, i.e., with a single possible destination for incoming packets; even the multi-queue architecture considered in Section 2.2 is merely an auxiliary construction intended to simplify buffer management algorithms, and the packets still all have the same destination.

Indeed, the best way to accommodate traffic burstiness is to adopt a single queue architecture, where the whole buffer is shared among all types of traffic and every core is able to process any type of traffic (see the left side of Fig 2.4). In

this architecture, an online greedy policy that implements priority queuing (PQ), where packets are ordered in non-decreasing order of their processing requirement and can be pushed out upon arrival of new packets, has optimal throughput [126].

However, this simple approach has important drawbacks. First, PQ order is required: we have already seen that with a simpler to implement FIFO processing order the competitive ratio degrades to $\Omega(\log k)$ with respect to the optimal clairvoyant algorithm (Section 2.3), where k is the maximal possible processing per packet. Second, it can cause starvation of packets with higher processing requirements, which implicitly leads to setting priorities among traffic types and thus services so that they are rigged to the inverse of the processing requirements. Third, handling several traffic types on individual cores leads to increased complexity of the programs loaded onto these cores, as these must implement several services in a combined manner.

We thus consider a shared memory switch where the buffer is shared among all types of traffic but in contrast to the previous single queue model each core processes only one type of traffic. The single queue buffer architecture is in fact a special case of a shared memory switch (see the right-hand side of Fig. 2.4). Since all arriving packets with the same output port label have the same traffic type, the implementation of advanced processing order is not required (regular FIFO suffices). Furthermore, there is no starvation of any specific type of traffic. We consider unit-sized packets labeled with an output port and a processing requirement (i.e., packets with heterogeneous processing), maximizing the number of transmitted packets, analyzing the performance of buffer management policies under various characteristics via competitive analysis.

We consider an $l \times n$ shared memory switch with a buffer of size B; l and n represent the number of input and output ports, respectively. We assume that $B \geq n$. Each ith output port manages a single output queue Q_i, $1 \leq i \leq n$; the number of packets in Q_i is denoted by $|Q_i|$. Each Q_i implements first-in-first-out (FIFO) processing order. Each packet $p(d, w)$ arriving at an input port is labeled with the output port number d and its required work w in processing cycles ($1 \leq d \leq n$ and $1 \leq w \leq k$), where k denotes the global upper bound on required work per packet. In what follows, we denote by $\boxed{w}$ a packet with required work w; by $h \times \boxed{w}$, a burst of h packets with required work w each.

We call the *sequential case* the special case of our model when each queue Q_i

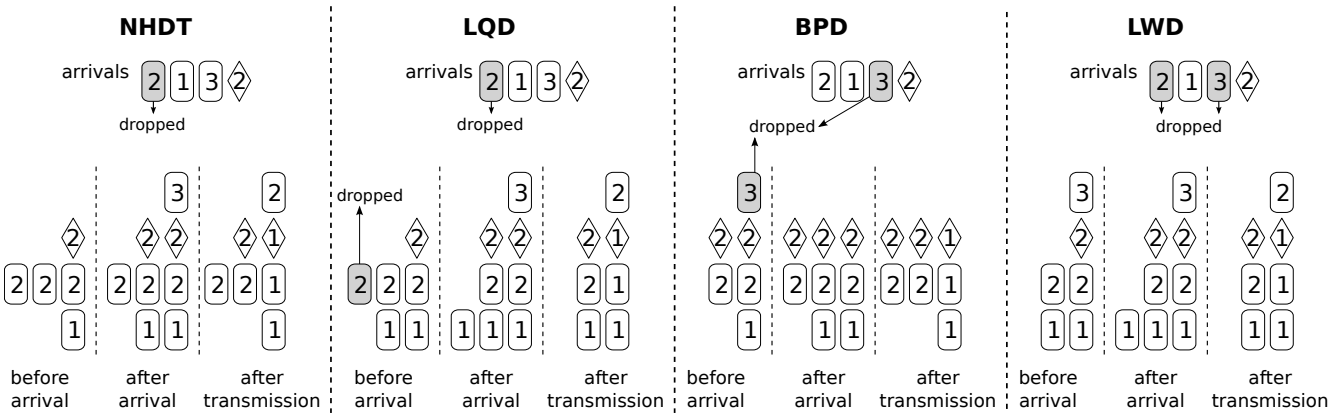


Figure 2.5: A sample time slot of Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT), Longest-Queue-Drop (LQD), Biggest-Packet-Drop (BPD), and Longest-Work-Drop (LWD) policies with maximal processing $k = 3$, 4 output ports (there are two different ports with the same processing requirement 2, denoted by rectangular and rhomboidal packets respectively), and a shared buffer of size $B = 8$.

contains packets with required processing i, i.e., each value of required processing occurs in exactly one queue; this is a special case but many large lower bounds on competitive ratios hold in this case already.

Time is slotted; we divide each time slot into two phases (see Fig. 2.5). During the *arrival phase* a burst of new packets arrives at each input port, and the (*buffer management*) *policy* decides which ones should be admitted. We assume that during the arrival phase input ports are served in a fixed order from input port 1 to input port l with no restriction on the burst size (at the same time slot more than l arrivals are allowed). The arrivals are adversarial and do not assume any specific traffic distribution except a single constraint: all packets arriving with output port label i have the same required processing w_i, $1 \le w_i \le k$. Note that two different output queues Q_i and Q_j, $1 \le i, j \le n$ can still accept packets with the same processing requirement, $w_i = w_j$. Without loss of generality, we assume that queues are sorted in the order of their processing requirement: if $i < j$ then $w_i \le w_j$. An accepted packet can be later dropped from the buffer when another packet is accepted instead; in this case we say that a packet p is *pushed out* by another packet q, and a policy that allows this is called a *push-out* policy. During the *transmission phase*, required work of the head-of-line packet in FIFO order at each non-empty queue is reduced by one, and every packet with zero residual work is transmitted.

We begin with non-push-out algorithms. In the case of a single queue, we have seen in Section 2.3 that non-push-out greedy policies perform poorly (they are

k-competitive) on packets with heterogeneous processing regardless of processing order. In this model, we begin with two simple cases of greedy non-push-out buffer management policies with static thresholds. The first policy sets static thresholds on each queue inversely proportional to required processing, while the second simply uses identical thresholds for all queues. Interestingly, the second policy has a better competitive ratio. We denote $Z = \sum_{i=1}^{n} \frac{1}{r_i}$ (sum of inverse values of required processing for each port); also, in what follows we omit $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ for clarity, assuming that B divides all we need it to divide.

Non-Push-Out-Harmonic-Static-Threshold (NHST): during the arrival of a packet p with output port i, if $|Q_i| < \frac{B}{r_i Z}$, accept p into Q_i; else drop p.

Theorem 2.4.1 (upper bound for NHST)**.** *NHST is* $(kZ + o(kZ))$*-competitive.*

Non-Push-Out-Equal-Static-Threshold (NEST): during the arrival of a p with output port i, if $|Q_i| < B/n$ packets then accept p into Q_i, else drop p.

Theorem 2.4.2 (upper bound for NEST)**.** *NEST is* $(n + o(n))$*-competitive.*

Next we consider the Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT) policy, previously considered in [137]. In the NHDT policy, thresholds for queues are dynamic and depend on the number of packets in the queues. The idea is that for each $1 \le m \le k$, m queues with most packets in them should contain at most $\frac{B}{H_k}\left(1 + \frac{1}{2} + \ldots + \frac{1}{m}\right)$ packets, where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ is the k-th harmonic number.

Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT): during the arrival of a packet p with output port i, denote by $j_1, j_2, \ldots, j_m = i$ the queues for which $|Q_{j_s}| \ge |Q_i|$; if

$$\sum_{s=1}^{m} |Q_{j_s}| < \frac{B}{H_k}\left(1 + \frac{1}{2} + \ldots + \frac{1}{m}\right)$$

then accept p into Q_i, else drop p.

The NHDT policy is illustrated with an example in Fig. 2.5; it is the first policy on the left.

For packets with homogeneous processing, NHDT was shown to be at most $O(\log n)$-competitive for n output ports [137] (in our model, there are k output ports). It turns out that for heterogeneous processing requirements, NHDT is still lacking: although we have not been able to prove a linear bound similar to

Theorem 2.4.1 for NHST, the following lower bound still grows too fast.

Theorem 2.4.3 (lower bound for NHDT)**.** *If B is asymptotically greater than k, the NHDT policy is at least $\left(\frac{1}{2}\sqrt{k \ln k} - o(\sqrt{k \ln k})\right)$-competitive in the sequential case.*

We now proceed to push-out policies, starting with the well-known Longest-Queue-Drop that overlooks required processing and only considers queue sizes.

Longest-Queue-Drop (LQD): during the arrival of a packet p with output port i, denote by $j^* = \arg\max_j\{|Q_j| + [i = j]\}$ where $[i = j] = 1$ if $i = j$ and 0 otherwise (i.e., Q_{j^*} is the longest queue once we virtually add p to Q_i; we choose one with largest required processing if there are several); then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and $i \neq j^*$, push out last packet from Q_{j^*} and accept p into Q_i; else drop p.

LQD is illustrated in Fig. 2.5; it is the second policy on the left. In case of homogeneous processing, LQD is at least $\sqrt{2}$- and at most 2-competitive [45]. For heterogeneous required processing, the situation is worse.

Theorem 2.4.4 (lower bound for LQD)**.** *For sufficiently large B, LQD is at least $(\sqrt{k} - o(\sqrt{k}))$-competitive in the sequential case.*

Next we propose a policy that aims to minimize total required processing in case of congestion by pushing out packets with maximal processing requirements.

Biggest-Packet-Drop (BPD): during the arrival of a packet p with output port i, denote by Q_j the nonempty queue with largest processing requirement; then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and $i \leq j$, push out last packet from Q_j and accept p into Q_i;

(3) if the buffer is full and $i > j$, drop p.

Figure 2.5 shows a sample time slot of the BPD policy; it is the second policy on the right.

Theorem 2.4.5 (lower bound for BPD)**.** *For $B \geq \frac{k(k+1)}{2}$, BPD is at least $(\ln k+\gamma)$-competitive in the sequential case, where $\gamma = 0.5772\ldots$ is the Euler–Mascheroni constant.*

As an improvement over LQD, we propose a policy that pushes packets out of the queue with most required work; we denote the total work in queue Q_i (sum of remaining work for packets in Q_i) by W_i.

Longest-Work-Drop (LWD): during the arrival of a packet p with output port i, let $j^* = \arg\max_j\{W_j + [i = j]r_i\}$ (i.e., W_{j^*} is maximal once we virtually add p to Q_i; we choose maximal among those queues if there are several); then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and $i \neq j^*$, push out last packet from Q_{j^*} and accept p into Q_i; else drop p.

LWD is the rightmost policy illustrated in Fig. 2.5.

Theorem 2.4.6 (lower bound for LWD)**.** *If $k \geq 6$, LWD is at least $\left(\frac{4}{3} - \frac{6}{B}\right)$-competitive in the sequential case.*

Since LWD is equivalent to LQD when packets in all queues have the same processing requirement, LWD is at least $\sqrt{2}$-competitive in this setting [45]. Note that our lower bound for LWD in the sequential case is less than the $\sqrt{2}$ lower bound of LQD in the model with uniform processing requirements. LWD optimizes a "local" state by dropping packets from queues with maximal latency. On the other hand, OPT can leave these packets and keep these ports active when OPT knows that there will be no later arrivals to high-latency queues, whereas the corresponding ports of LWD will be idle; a similar phenomenon explains the non-optimality of LQD in the model with uniform processing. But in the sequential case of the generalized model it is significantly harder for OPT to win much in this way since LWD will drop less packets from queues with the highest latencies (proportional to the required processing).

The main result of this section is a constant upper bound on the competitive ratio of LWD. Note that although this bound is equal to the upper bound on LQD in the homogeneous processing model [45], the mapping routine is actually very different.

Theorem 2.4.7 (upper bound for LWD)**.** *LWD is at most* 2*-competitive.*

As the second setting, we consider the model that allows for packets with heterogeneous processing requirements to be admitted to the same queue. This generalization has significant impact on the efficiency of considered policies and is applicable to a number of real-world scenarios. The model above suggests to allocate a separate queue per processing requirement per output port, but in this case constraints on the maximal number of supported queues can significantly limit applicability once k and n are growing.

Again, we consider an $1 \times n$ shared memory switch with one input and n output ports and a buffer of size B, that is, the total length of all queues is bounded by B. We assume that $B \geq n$. The number of input ports have no significance for the model: we process all arriving packets at the same time, as a single burst. Each output port, on the other hand, manages a separate output queue, denoted Q_i for port i, $1 \leq i \leq n$; the number of packets in Q_i is denoted by $|Q_i|$. Each packet arriving at the input port is labeled with the output port number d and its required work w in processing cycles ($1 \leq d \leq n$ and $1 \leq w \leq k$), where k denotes the global upper bound on required work per packet. Each Q_i implements either

(1) priority queueing (PQ) processing order, where packets are processed in non-decreasing order of required processing, or

(2) first-in-first-out (FIFO) processing order, where packets are processed in order of arrival.

In what follows, we denote by $\boxed{w \mid i}$ a packet with required work w intended for output port i; by $h \times \boxed{w \mid i}$, a burst of h $\boxed{w \mid i}$ packets arriving at the same time. We also denote by $r_t(p)$ the remaining required processing of a packet p at time unit t.

We consider several different policies in this setting, as illustrated in Fig. 2.6. Below we define these policies and show the results of our theoretical analysis.

Longest-Queue-Drop (LQD): during the arrival of a packet p with output port i, denote by $j^* = \arg\max_j \{|Q_j| + [i = j]\}$ where $[i = j] = 1$ if $i = j$ and 0 otherwise (i.e., Q_{j^*} is the longest queue once we virtually add p to Q_i; we choose one with largest total required processing if there are several); then do the following:

(1) if the buffer is not full, accept p into Q_i;

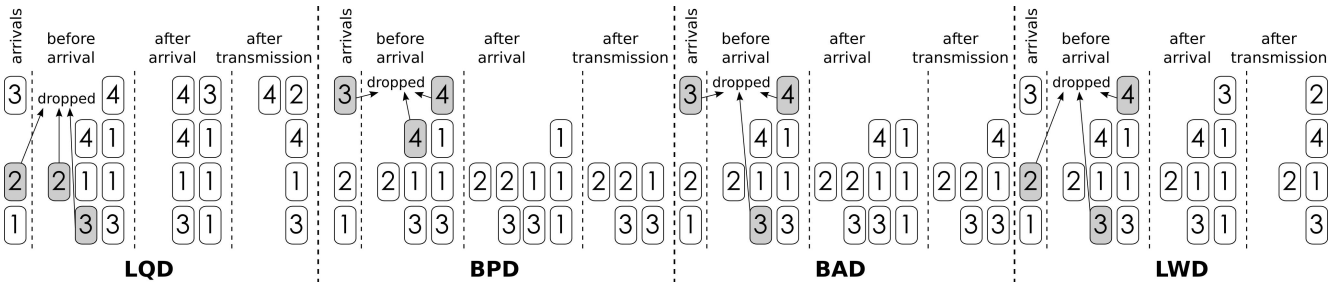


Figure 2.6: A sample time slot of Longest Queue Drop (LQD), Biggest Packet Drop (BPD), Biggest Average Drop (BAD), and Largest Work Drop (LWD) policies with maximal processing $k = 4$, $n = 4$ output ports, and a shared buffer of size $B = 8$. Queues for each output port are shown horizontally. Shaded packets are dropped during arrival.

(2) if the buffer is full, accept p into Q_i and push out last packet from Q_{j^*}; else drop p.

Note that in case (2), if $i = j^*$ and p is the largest packet in Q_{j^*}, p will be "pushed out" itself, i.e., dropped.

Theorem 2.4.8 (lower bound for LQD)**.** *For $k \geq n(n-1)$ and $B \geq k+n$, LQD is at least $(n/2 - o(n))$-competitive.*

Biggest-Packet-Drop (BPD): during the arrival of a packet p with required work w and output port i, denote by Q_j the nonempty queue that contains a packet $p_{\max}$ with the largest processing requirement $w_{\max}$; then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and $w < w_{\max}$, push out $p_{\max}$ from Q_j and accept p into Q_i;

(3) if the buffer is full and $w > w_{\max}$, drop p.

Biggest-Average-Drop (BAD): during the arrival of a packet p with required work w and output port i, denote by Q_j the nonempty queue with largest average processing requirement $\bar{w}_{\max}$; then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and $w < \bar{w}_{\max}$, push out packet with maximal work from Q_j and accept p into Q_i;

(3) if the buffer is full and $w > \bar{w}_{\max}$, drop p.

Theorem 2.4.9 (lower bound for BPD and BAD)**.** *BPD and BAD are both at least* $(n+1)/2$*-competitive.*

Largest-Work-Drop (LWD): during the arrival of a packet p with output port i and required processing w, denote by $j^* = \arg\max_j\{W_j + [i=j]w\}$ where $[i=j] = 1$ if $i = j$ and 0 otherwise, and W_j is the total required processing of all packets in queue Q_j (i.e., Q_{j^*} is the queue with the largest total required processing once we virtually add p to Q_i; we choose the one with the largest single packet if there are several queues with largest work). Then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and w is smaller than the required processing of at least one packet in Q_{j^*}, push out the largest packet from Q_{j^*} and accept p into Q_i; else drop p.

Theorem 2.4.10 (lower bound for LWD)**.** *LWD with FIFO processing order is at least* $(\log_{B/n} k)(1 - n/B) + 1$*-competitive.*

The main theoretical result in this setting is given by the following upper bound on the competitive ratio of LWD.

Theorem 2.4.11 (upper bound for LWD)**.** *For a shared memory* $n \times n$ *switch with a buffer* B*, LWD is at most* $1 + \frac{B}{T+B}$*-competitive, where* T *is the minimal number of packets transmitted between any two consecutive iterations.*

Third, we consider another generalization, posed as an open problem in [109]. In this model, each unit-sized packet with required work 1 has two parameters: an output port label and an intrinsic *value*; the objective is to maximize the total transmitted value. The model is otherwise the same, only each packet now requires exactly one processing cycle but contributes differently to the objective. Note that for unit values this reduces to the model of [45], so the 4/3 lower bound on any online policy shown there holds in this case as well.

Even if the processing order is fixed, it is unclear which traffic should be admitted; there may be a tradeoff between traffic that opens new active ports and traffic that increases the total admitted value. We do not consider non-push-out policies since it is straightforward that a greedy non-push-out policy that accepts a packet if there is available space is at least k-competitive (fill the buffer with $\boxed{1}$s, then send in the $\boxed{k}$s). We define several push-out scheduling policies that greedily

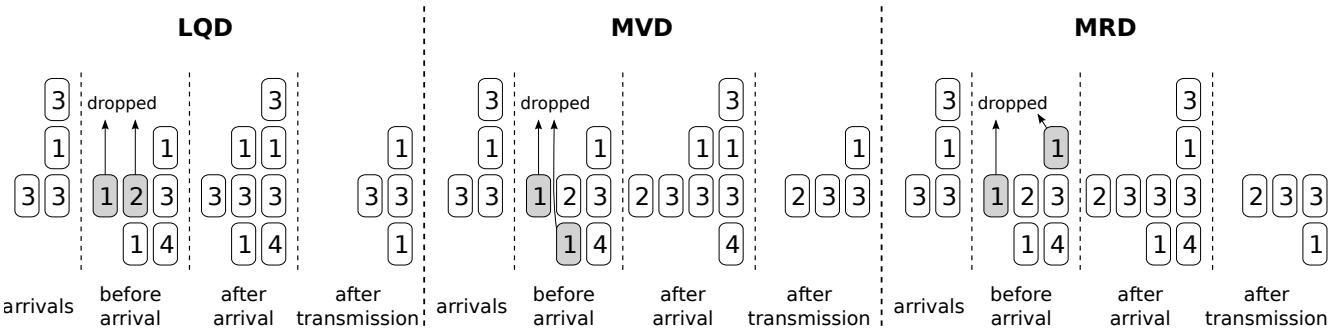


Figure 2.7: A sample time slot of Longest-Queue-Drop (LQD), Minimal-Value-Drop (MVD), and Maximal-Ratio-Drop (MRD) with maximal value $k = 4$, 4 output ports, and a shared buffer of size $B = 8$.

optimize the number of active ports, total admitted value, or a combination of both. The algorithms are illustrated in Fig. 2.7.

First, we consider the Longest-Queue-Drop (LQD) heuristic again; in this model, LQD drops the last (lowest value) packet from the longest queue when a buffer is congested, balancing the queue sizes. It is illustrated on the left of Fig. 2.7. However, in the value-based model LQD again fails to reach constant competitiveness.

Theorem 2.4.12 (lower bound for LQD)**.** *The competitive ratio of LQD is at least* $\left(\sqrt[3]{k} - o\left(\sqrt[3]{k}\right)\right)$.

The next policy greedily maximizes total admitted value by pushing out packets with minimal current value.

Minimal-Value-Drop (MVD): during the arrival of an $\boxed{m}$-packet p that is destined to the i-th port, denote by Q_j the nonempty queue that contains a packet with minimal value (if there are several such queues, choose the longest among them); then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and the minimal admitted value is less than $\boxed{m}$, push out last packet from Q_j and accept p into Q_i;

(3) if the buffer is full and the minimal admitted value is at most $\boxed{m}$, drop p.

MVD is the middle policy shown in Fig. 2.7. The MVD algorithm prioritizes packets of maximal value. Unfortunately, MVD has a linear lower bound on competitiveness in the worst case.

Theorem 2.4.13 (lower bound for MVD)**.** MVD *is at least* $\frac{m-1}{2}$*-competitive for* $m = \min\{k, B\}$.

As a potentially better policy we propose a combination of these two characteristics, an idea similar to the LWD policy in the previous model that we have shown to be 2-competitive in Theorem 2.4.7. A natural idea would be to try to combine these two heuristics, prioritizing packets with high value only if they come from shorter queues. One implementation of such an idea is given in the Maximal Ratio Drop algorithm.

Maximal-Ratio-Drop (MRD): during the arrival of an $\boxed{m}$-packet p that is destined to the i-th port, denote by Q_j the nonempty queue with a maximal value of $\frac{|Q_j|}{a_j}$, where a_j is an average value in Q_j (if there are several such queues, choose as Q_j a queue that contains a packet with a smaller value) then do the following:

(1) if the buffer is not full, accept p into Q_i;

(2) if the buffer is full and has admitted values smaller than $\boxed{m}$, push out last packet from Q_j and accept p to Q_i;

(3) if the buffer is full and minimal admitted value in the buffer is bigger than $\boxed{m}$, drop p.

A sample MRD time slot shown on the right of Fig. 2.7, Note that MRD emulates LQD in case all packets have unit values, and the lower bound $\sqrt{2}$ shown in [45] applies. We can show a constant lower bound for MRD in the special case when each packet's value is equal to its output port label.

Theorem 2.4.14 (lower bound for MRD with homogeneous values in queues)**.** MRD *is at least* $\frac{4}{3}$*-competitive in case each packet's value is equal to its output port label.*

Unfortunately, in the general case, at least for large n, MRD is not competitive either.

Theorem 2.4.15 (lower bound for MRD for large n)**.** *The Maximal-Ratio-Drop (MRD) algorithm is at least* V*-competitive if* $n \geq B - V^2 + 1$.

We have also supported all results shown in this section with comprehensive simulation studies that complement and extend our theoretical results. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [9, 11, 27, 31].

2.5 Buffer management for packets with heterogeneous processing and varying size

In Sections 2.2, 2.3, and 2.4 we have considered buffer management for packets with heterogeneous processing in different queueing architectures. In this section, we focus our attention on a general model for the problem where a network processor is required to manage the admission control and scheduling modules of a single bounded size queue, where arriving traffic consists of *packets*, such that each packet has some *size* (e.g., in bytes), and a *processing requirement* (in processor cycles). A packet is successfully *transmitted* once the scheduling module has scheduled the packet for processing for at least its required number of cycles, while the packet resides in the buffer. If a packet is dropped from the buffer, either upon arrival due to admission control policies, or after being admitted and possibly partially, but not fully, processed (e.g., in scenarios where push-out is allowed), then the packet is irrevocably lost. We focus our attention on maximizing the *throughput* of the queue, measured by the overall number of bytes of packets that are successfully transmitted by the queue.

The above simple model highlights a natural tension between two (potentially) conflicting goals: on the one hand it seems beneficial to focus on packets which require few processing cycles, since these can contribute to the throughput relatively fast. On the other hand, one is also enticed to favor large packets since these imply a significant boost to the throughput once they are completed. Our work addresses this dilemma, and presents algorithms with provable performance guarantees in such settings.

We consider a buffer with a bounded capacity of B bytes, handling the arrival of a sequence of packets. Each arriving packet p has some size $\ell(p) \in \{1, \ldots, L\}$ (in bytes) and some number of required processing cycles $r(p) \in \{1, \ldots, k\}$. Both of these quantities are known for every arriving packet. We focus our attention on priority-based buffer management and scheduling, in both *push-out* settings, where admitted packets are allowed to be pushed out of the queue prior to having its processing completed (in which case the packet does not contribute to the system's throughput), and *non-push-out* settings, in which case buffer management decisions are limited to admission control.

Specifically, we consider the following priority queueing regimes:

(1) Shortest Remaining Processing Time (SRPT) first, common in job scheduling environments,

(2) Longest-Packet (LP) first, and

(3) Most-Effective-Packet (MEP) first, which prioritizes packets by their residual processing requirement to size ratio.

We present competitive buffer management algorithms for these schemes, and further present lower bounds on the performance of algorithms for such priority queues. We show that the competitive ratio obtained by our algorithms depends on two fundamental parameters of the problem, namely

(1) the maximum size of a packet and

(2) the maximum number of processing cycles required by a packet.

We note that none of the algorithms below require knowledge of the maximum number of processing cycles in advance (although some of our algorithms need prior knowledge of the maximum allowed packet size). Figure 2.8 provides a graphic depiction of our general model (disregarding at this point the specific priority scheme employed by the system).

To facilitate our proofs, we make use of the following properties of ordered (multi-)sets that we have proven. We consider multi-sets of real numbers, where we assume each multi-set is ordered in non-decreasing order. We will refer to such multi-sets as *ordered sets*. For every $1 \leq i \leq |A|$, we will further refer to element $a_i \in A$ or to $A[i]$ as the i-th element in the set A, as induced by the order. Given two ordered sets A, B, we say $A \geq B$, if for every i for which both a_i and b_i exist, $a_i \geq b_i$.

Theorem 2.5.1. *For any two ordered sets A, B satisfying $A \geq B$, and any two real numbers a, b such that $a \geq b$, if either $b \leq b_{|B|}$ or $|A| \leq |B|$ then the ordered sets $A' = A \cup \{a\}$, $B' = B \cup \{b\}$ satisfy $A' \geq B'$.*

Theorem 2.5.2. *For any two ordered sets A, B satisfying $A \geq B$, and any real number b, if either $b \leq b_{|B|}$ or $|A| \leq |B|$ then the ordered set $B' = B \cup \{b\}$ satisfies $A \geq B'$.*

We begin with the performance of the NPO greedy algorithm, where we are able to prove tightly matching lower and upper bounds.

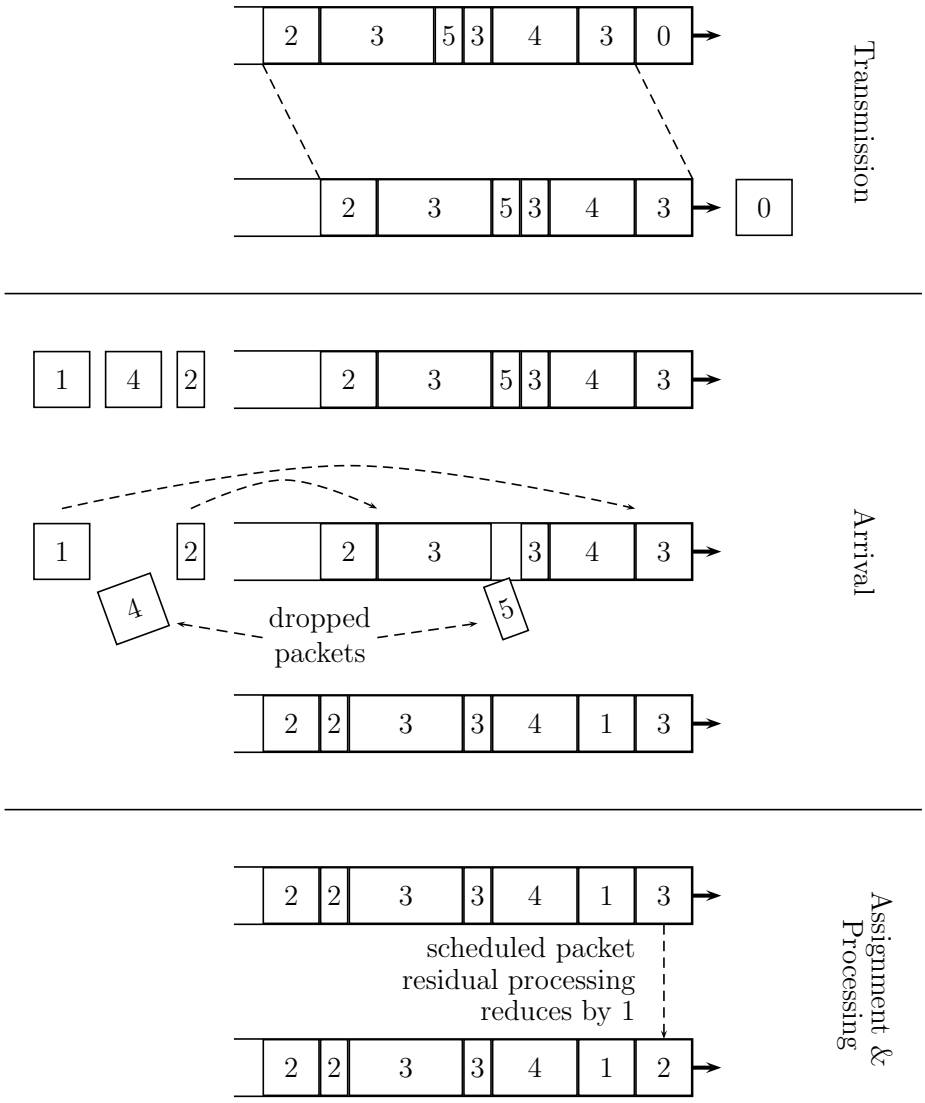


Figure 2.8: An outline of the model. The top subfigure shows the transmission phase, the middle subfigure shows the arrival phase where packets might be discarded, and the bottom subfigure shows the assignment and processing phase. The length of a packet represents its size, and the number stamped on the packet represents the number of its (residual) required processing cycles.

Theorem 2.5.3 (lower bound for NPO)**.** NPO *has competitive ratio at least* kL *for SRPT-based priorities.*

Theorem 2.5.4 (upper bound for NPO)**.** NPO *is at most* $kL\frac{B}{B-L}$*-competitive for SRPT-based priorities.*

For the PO algorithm, the situation with size is worse than it was in previous sections.

Theorem 2.5.5 (lower bound for PO)**.** PO *has competitive ratio at least* L *for SRPT-based priorities, for* $B \geq 2L$*.*

However, we are also able provide a first upper bound for PO, the first main

result of this section. Specifically, we prove the following.

Theorem 2.5.6 (upper bound for PO when $B > 2L$)**.** *If $B > 2L$, then* PO *is at most $\frac{4L-2}{L_a}$-competitive for SRPT-based priorities, where L_a is an average length of packets transmitted by* PO*.*

For sufficiently large buffers, the mapping used in Theorem 2.5.6 can be generalized.

Theorem 2.5.7 (upper bound for PO)**.** PO *is at most $\frac{(2L-1)(N+1)}{NL_a}$-competitive for SRPT-based priorities, where L_a is an average length between all transmitted packets by* PO *and $N = \lceil \frac{B-2L+1}{2L-1} \rceil$.*

The next theorem shows that it is possible to improve the upper bound on the competitive ratio of PO for large B using this generalization.

Theorem 2.5.8 (upper bound for PO for large B)**.** PO *is at most $(k + 3)$-competitive for LP-based priorities on sufficiently big buffers.*

Next, we study the performance of a priority queue (PQ), where priorities are set in accordance with the non increasing order of processing cycles divided by packet length. This priority is dubbed the *Most Effective Packet* first priority (MEP), or the *effective-ratio* priority. Recall that our objective here is to maximize the amount of sent bytes. We concentrate our attention to preemptive case since non-preemptive has similar results as NPO. The next theorem is the second main result in this section.

Theorem 2.5.9 (upper bound for PQ)**.** PQ *is $\Omega(\log k)$-competitive.*

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [13, 38, 39].

2.6 Buffer management for packets with multiple characteristics

As we have already discussed in this chapter, incorporation of new characteristics (e.g., required processing per packet or value) in admission decisions and implementation of additional objectives beyond fairness lead to new challenges in design and implementation of traditional network elements. In this section, we

again consider a single-queue switch where a buffer of size B is shared among all types of traffic. However, the difference in this setting is that now we are studying the impact of *both* packet values and required processing on weighted throughput.

Namely, we assume that each arriving packet p is branded with:

(1) the number of required processing cycles (required work, or weight) $w(p) \in \{1, \ldots, W\}$;

(2) its processing value $v(p) \in \{1, \ldots, V\}$.

Although the values of W and V will play a fundamental role in our analysis, our algorithms will not need to know W or V in advance. Note that for $W = 1$ the model degenerates into a single queue of uniform packets with nonuinform value, as considered in, e.g., [52, 212], while for $V = 1$ it becomes a single queue of unit-valued packets with different required processing, as considered in Section 2.2. We will denote a packet with required processing w and value v by $\boxed{w \mid v}$, and a sequence of n packets with the same parameters w and v by $n \times \boxed{w \mid v}$.

The ultimate goal of this entire line of research is to choose the right buffer management policy in every problem setting, i.e., for each possible switch configuration and every objective. For instance, previous works (and previous sections of this chapter) have studied in detail the interrelations between policies with and without *push-out*, the capability to drop previously accepted packets from the buffer; while non-push-out policies are simpler to implement in practice, they often turn out to be *non-competitive* in terms of weighted throughput with lower bounds on the competitive ratio linear in problem parameters such as buffer size B, maximal possible value V, or maximal required processing W. Note that even this is not quite the whole story: although non-push-out policies are usually clearly inferior with respect to (weighted) throughput, they can still come out ahead for other objectives, e.g., minimizing the total power consumption (push-out might be a costly procedure).

Still, in this section we concentrate on the weighted throughput objective, and consider a setting where worst-case lower bounds, i.e., hard examples, are relatively easy to come by: we have two characteristics to play with for every packet, value and required processing, instead of just a single one as in most previous works. For this reason, we concentrate on studying the best class of natural algorithms available for the single queue setting, and previous research

indicates that *priority queues with push-out* are the best tools that often lead to good results. In particular, Keslassy et al. showed that a single priority queue with push-out is optimal for packets with varying required processing and unit value [128].

Note, however, that although in previous work a single priority queue was usually the best algorithm, sometimes simply optimal, and the goal often was to try and achieve comparable throughput under additional constraints such as FIFO transmission order or multiple separate queues, in the setting with two different characteristics it is not even clear what a priority queue sorts its packets on: if one packet has less value but also less required processing than another, which one should we prefer? To capture different possible orderings in a priority queue, we introduce the following definition.

Definition 2.6.1. *Let f be a function of packets, $f(w, v) \in \mathbb{R}$, with the intuition that better packets have larger values of f. Then the* PQ_f *processing policy is defined as follows:*

- PQ_f *is greedy, i.e., it accepts incoming packets as long as there is space in its buffer;*
- PQ_f *is work-conserving, i.e., it processes a packet as long as its buffer is not empty;*
- PQ_f *orders and processes packets in its queue in the order of decreasing values of f;*
- PQ_f *pushes out a packet p and adds a new packet p' to the queue at time slot t if the buffer is full, p is currently the worst packet in the buffer and p' is better than p: $f(p) = \min_{q \in \mathrm{IB}^{\mathrm{PQ}_f}} f(q)$, and $f(p') > f(p)$. Here* $\mathrm{IB}^{\mathrm{PQ}_f}$ *is* $\mathrm{IB}^{\mathrm{PQ}_f}(t)$ *for the current time slot t.*

In other words, PQ_f orders and processes packets according to the function f.

In particular, we consider three specific priority queues (here w denotes the *current* residual work and v denotes the packet's value):

(1) $\mathrm{PQ}_{-w,v} = \mathrm{PQ}_{-w+v/(V+1)}$ orders packets in the increasing order of their required processing, breaking ties by value;

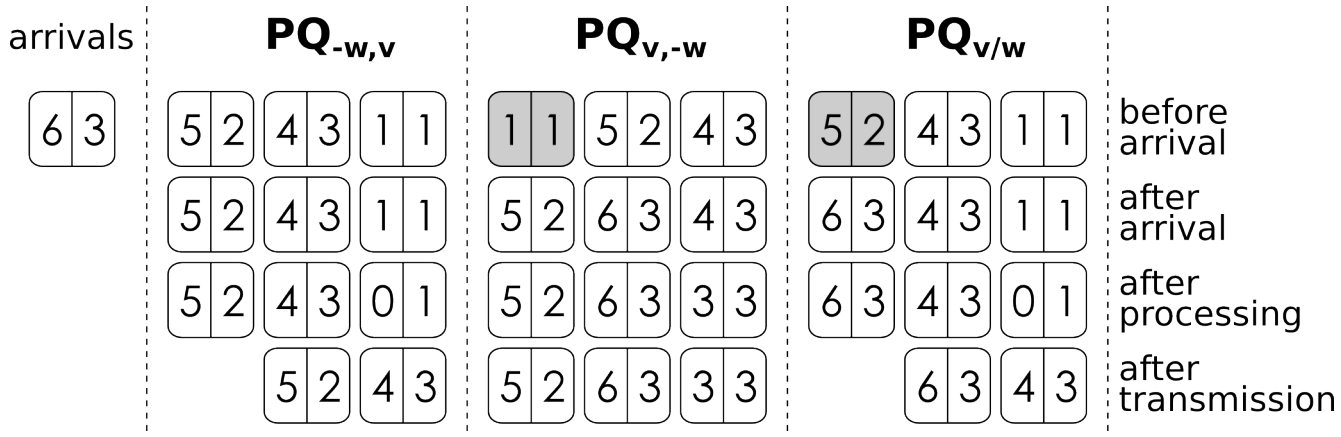


Figure 2.9: A sample time slot of $\mathrm{PQ}_{-w,v}$, $\mathrm{PQ}_{v,-w}$, and $\mathrm{PQ}_{v/w}$.

(2) $\mathrm{PQ}_{v,-w} = \mathrm{PQ}_{v-w/(W+1)}$ orders packets in the decreasing order of their value, breaking ties by required processing;

(3) $\mathrm{PQ}_{v/w}$ orders packets in the decreasing order of their value-to-work ratio, i.e., it prioritizes packets that yield the best value per one time slot of processing. Note that here we also have two possibilities for breaking ties, $\mathrm{PQ}_{v/w,-w} = \mathrm{PQ}_{v/w-w/(W^2+1)}$ and $\mathrm{PQ}_{v/w,v} = \mathrm{PQ}_{v/w+v/(WV+1)}$, but in this case the tie-breakers will be irrelevant for all our statements, so we unite them under the same notation.

Figure 2.9 shows a sample time slot of these priority queues.

The first set of our results deals with general lower bounds in this problem setting. They are proven in an *adversarial* way: we construct a sequence of inputs where further inputs depend on the choices an online algorithm makes, so in the end we find a "bad" input for every possible choice. We establish strong bounds for several special cases and a weaker general lower bound for an arbitrary priority queue.

Theorem 2.6.1 (general lower bound for $B = 1$)**.** *For $B = 1$ and $V > \sqrt{W}$, any online algorithm ALG is at least $\sqrt{W}$-competitive. Further, in the two-valued case or if $V \le \sqrt{W}$ any online algorithm ALG is at least $\min\{V, W/V\}$-competitive for $B = 1$.*

Theorem 2.6.2 (general lower bound for FIFO order)**.** *For arbitrary B and $V > \sqrt{W}$, any online algorithm ALG that preserves FIFO processing and transmission order is at least $\left(\frac{\sqrt{W}}{B} + 1 - \frac{1}{B}\right)$-competitive. In the two-valued case or if $V \le \sqrt{W}$, any online algorithm ALG with FIFO processing and transmission or-*

der is at least $\frac{V+B-1}{W+B-1}$*-competitive.*

Theorem 2.6.3 (arbitrary PQ)**.** *For* $V > \sqrt{W}$ *and any priority function* f *such that* $f(w,v) \in \mathbb{R}$*, the algorithm* PQ_f *is at least* $\sqrt{W}$*-competitive. In the two-valued case or if* $V \leq \sqrt{W}$*, algorithm* PQ_f *is at least* $\min\{V, W/V\}$*-competitive.*

The main result in this setting shown in the works [8, 25] is an upper bound on the competitiveness of $\mathrm{PQ}_{v,-w}$ for the two-valued case, where required processing can take arbitrary values from 1 to W but there are only two possible values, 1 and V. Other results include a number of lower bounds on the competitiveness of various priority queues. These results are not part of this dissertation, but we also proved a generalization of several lower bounds to the β-push-out case. In this case, the priority queues are defined as follows.

Definition 2.6.2. *Let* f *be a function of packets,* $f(w,v) \in \mathbb{R}$*, with better packets corresponding to larger values of* f*. The* PQ_f^β *processing policy for* $\beta > 1$ *is defined as* PQ_f *with the following difference:* PQ_f^β *can push out a packet* p *and add a new packet* p' *to the queue at time slot* t *if* p *is currently the worst packet in the buffer and* p' *is better than* p *at least by a factor of* β*:* $f(p) = \min_{q \in \mathrm{IB}^{\mathrm{PQ}_f}} f(q)$*, and* $f(p') > \beta f(p)$*.*

Theorem 2.6.4. *Consider a buffer of size* B *with maximal required processing* W *and maximal packet value* V*. Then:*

(1) $\mathrm{PQ}_{-w,v}^\beta$ *is at least* V*-competitive both in the case of arbitrary packet values and in the two-valued case;*

(2) $\mathrm{PQ}_{v/w}^\beta$ *is at least* $\min\{V, W\}$*-competitive in the case of arbitrary packet values;*

(3) in the two-valued case, if $\beta W \geq V$ *then* $\mathrm{PQ}_{v/w}^\beta$ *is at least* V*-competitive;*

(4) $\mathrm{PQ}_{v,-w}^\beta$ *is at least* $\left(\frac{(V-1)}{V}W - o(1)\right)$*-competitive in the case of arbitrary packet values and at least* $\left(\frac{W}{V} + o(1)\right)$*-competitive in the two-valued case.*

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [8, 25].

2.7 Buffer management for packets with latency constraints

Modern datacenters are increasingly required to deal with latency-sensitive applications. Low latency is critical for interactive applications, and achieving desired latency is a challenging task. Modern applications are highly distributed; moreover, in order to complete a final result application-level operations consisting of many queries often should satisfy latency constraints [47, 74, 150]. Latency is certainly an important factor, but there are two primary ways to represent latency requirements for optimization problems that arise in the design of network elements. One approach is to include latency in the final objectives by actually optimizing for it; for example, a popular way to represent latency in the final objective is to minimize *average flow completion time* [49, 107]. This, however, requires prior knowledge about flow duration which may be too restrictive, and still leaves the problem of how to balance latency with other desired objectives such as throughput, fairness, and others.

Instead of optimizing latency explicitly, a different way to look at the problem is to satisfy latency constraints on the level of individual packets; this constraint-based approach allows to optimize additional objectives and does not require prior knowledge of a flow's duration. Originally, weighted throughput optimization for packets with associated intrinsic value and slack was introduced in [133] and called a *bounded-delay model*. In this model, the *slack* defines a latency constraint, i.e., the offset from the arrival time when a packet must be transmitted or is lost otherwise, and every successfully transmitted packet contributes its *intrinsic value* to the weighted throughput. However, the model of [133] does not include another important characteristic: processing requirements per packet, a characteristic that we have considered throughout this chapter.

Incorporation of multiple traffic characteristics (e.g., packet values and required processing requirements) significantly increases the complexity of buffer management policies. In this context two major questions arise: how to represent the latency in desired objectives and how to provide guarantees for buffer management policies that would hold across a wide variety of traffic patterns. In this section, we consider a single queue buffering architecture, where every incoming packet is prepended with intrinsic value, required processing, and slack

(offset from the arrival time during which this packet should be transmitted); the buffer size is implicitly bounded by slack values. Our goal is to maximize a total transmitted value (weighted throughput). In these settings, we study worst-case performance guarantees of the proposed online algorithms by means of competitive analysis whose effectiveness is compared versus an optimal clairvoyant offline algorithm.

Formally, we consider a single-queue switch where a buffer is shared among all types of traffic, and we assume that each arriving packet p is branded with three characteristics:

(1) the number of required processing cycles (*work*) $w(p) \in \{1, \ldots, W\}$,

(2) its processing *value* $v(p) \in \{1, \ldots, V\}$,

(3) *slack* $s(p) \in \{1, \ldots, S\}$ that defines how long from the arrival time a packet should be transmitted before it is lost without any gain to the final objective (weighted throughtput).

We denote by $[\![w \mid v \mid s]\!]$ a packet with work w, value v, and slack s; a sequence of n packets with the same parameters, by $n \times [\![w \mid v \mid s]\!]$. Similar to [133], we assume unbounded queue size; note that the maximal slack value S effectively bounds it. In the notation above, slack s of a packet decreases on every time step, i.e., we are talking about "current", effective slack. A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles. Although the maximal work W, maximal value V, and maximal slack S will play a fundamental role in our analysis, the proposed online algorithms do not need to know them in advance. Note that for $W = 1$ the model degenerates into a single queue of unit-sized packets with heterogeneous intrinsic values, as in [133].

We again consider various priority queues. Let f be a function of packets, $f(w, v) \in \mathbb{R}$, with the intuition that better packets have larger values of f. Then the PQ_f processing policy (*priority queue*) is defined as the policy that processes packets in its queue in the decreasing order of the values of f on these packets. In particular, we consider the following specific priority queues; below w denotes the current residual work, v denotes the value, and s denotes slack:

(1) $\mathrm{PQ}_{-w} = \mathrm{PQ}_{-w+v/(V+1)}$ orders packets in the increasing order of their required processing, breaking ties by value;

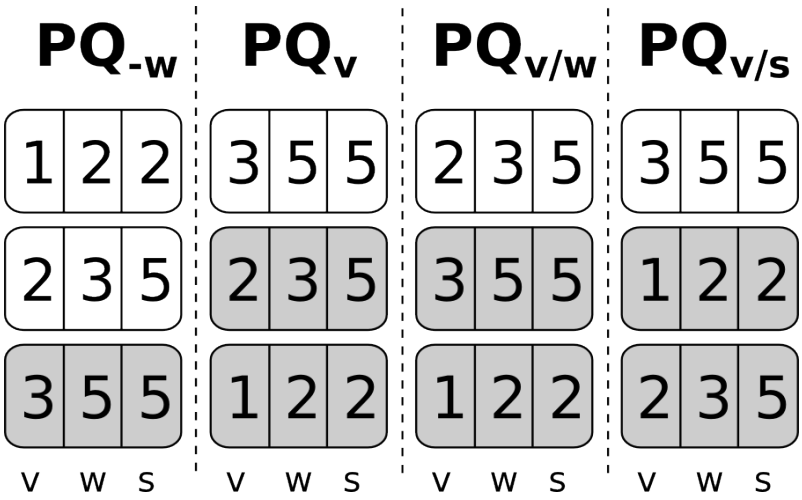


Figure 2.10: A sample time slot for PQ_{-w}, PQ_v, $PQ_{v/w}$, and $\mathrm{PQ}_{v/s}$. The head of line packet is at the top. Shaded packets are those that will not be transmitted due to lack of time.

(2) $\mathrm{PQ}_v = \mathrm{PQ}_{v-w/(W+1)}$ orders packets in the decreasing order of their value, breaking ties by required processing;

(3) $\mathrm{PQ}_{v/w}$ orders packets in the decreasing order of their value-to-work ratio, i.e., it prioritizes packets that yield the best value per one time slot of processing; one can break ties either by work or by value, all statements below will remain valid, so we unite these variations;

(4) PQ_{v/w_0} orders packets in the decreasing order of their value-to-original-work ratio, i.e., in difference from $\mathrm{PQ}_{v/w}$ the priority of a packet does not increase as it gets processed more and its work decreases;

(5) $\mathrm{PQ}_{v/s}$ and PQ_{v/s_0} order packets in the decreasing order of their value-to-slack and value-to-original-slack ratio respectively; this idea is known in scheduling, where it has been shown to work well as a variation on the earliest due date (EDD) heuristic [160].

Figure 2.10 illustrates a sample time slot with four different processing orders; the queues are shown vertically, with head of the queue on top. Shaded packets are those that the corresponding algorithm will not have time to process; this illustrates the differences between four processing orders.

We begin by showing a number of *lower* bounds for the competitiveness of online algorithms, i.e., results that show how bad a given online algorithm is in the worst case when compared to a clairvoyant offline algorithm. We first concen-

trate on *general* lower bounds, i.e., bounds that hold for all deterministic online algorithms.

Theorem 2.7.1. *The competitiveness of any deterministic online algorithm A cannot be bounded from above by a constant.*

Unfortunately, even if we try to separate the slacks from required processing values by some *additive* constant c (i.e., $s(p) \geq w(p) + c$ for all packets), there are no deterministic online algorithms with constant competitiveness.

Theorem 2.7.2. *The competitiveness of any deterministic online algorithm* ALG *cannot be bounded from above by a constant even if there is a predefined constant c such that for every packet $s(p) \geq w(p) + c$.*

However, in our main result below we will see that if we require the slack s to be separated from w by some constant multiplicative factor (i.e., $s(p) > cw(p)$ for some constant c and for every packet p), then we will be able to achieve a constant competitive ratio.

Theorem 2.7.2 already shows that if "real-time" packets are allowed no online algorithm is competitive. Still, it turns out that baseline priority queues are non-competitive even with large slacks (by large slack we will denote a slack which is always greater than the work by some constant c).

Theorem 2.7.3. *If w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw for some constant c, the competitiveness of priority queue algorithms* PQ_v, $\mathrm{PQ}_{v/w}$, PQ_{v/w_0}, $\mathrm{PQ}_{s/w}$, PQ_{s/w_0} *is at least*

$$\frac{1 - (\frac{w-2}{w})^{\frac{1}{w}\left\lfloor \log_{\frac{w}{w-2}} v \right\rfloor}}{1 - \frac{w-2}{w}} \frac{1}{c},$$

which is non-constant in v and w.

On the other hand, if we consider non-preemptive algorithms the situation is even worse.

Theorem 2.7.4. *If w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw, then if $w > c$ the competitiveness of any non-preemptive algorithm is at least V.*

However, in the worst case α-preemption also does not really help; we illustrate it below for $\mathrm{PQ}_{v/w}$, and similar examples can be constructed for other priority queues.

Theorem 2.7.5. *The* $\mathrm{PQ}_{v/w}$ *algorithm with* α*-preemption has a lower bound on its competitiveness of* $\Omega(w)$ *for arbitrarily large slacks.*

The main result of this section is a new upper bound for the model with three characteristics, achieved by introducing a new algorithm. The main idea of the algorithm is to enhance the α-preemptive $\mathrm{PQ}_{v/w}$ algorithm with two new ideas designed to solve the problems highlighted in Theorem 2.7.5:

- special priority assignment rules that reduce the value of new packets and/or increase the value of packets that have already been partially processed (to avoid wasted effort due to too much preemption);
- an additional rule (we call it β-pessimism) which restricts the algorithm from working with packets with expiring deadlines, so it can never corner itself in "just-in-time" situations.

The resulting algorithm can be thought of as a priority queue with two new modifications:

- every packet has a flag that shows whether the algorithm has already begun processing it;
- if a packet has not begun processing yet (it is a "new" packet), then for the purposes of preemption (i.e., the order of the priority queue) its value is reduced by a factor of α (this is the α-preemption property);
- moreover, in order for new packets to begin processing they need to have an extra "cushion" on the slack: the algorithm makes a provision to spend share β of the new packet's processing time on other packets due to preemption, so work on the new packet can begin only if its slack is at least $(1+\beta)w$ (we call this property β-*pessimism*).

The new algorithm is simply a priority queue $PQ_{v/w}$ with the two modifications above.

The main result of [3, 19] (which is not part of this dissertation so we do not formulate it as a theorem) states that if $s \geq cw$ for every packet at time of arrival, the SPQ algorithm defined above has competitive ratio at most

$$\left(1 + \frac{1}{\beta(\alpha-1)-1}\right)\left(1 + \alpha + \frac{\alpha c}{(c-\beta-1)}\right).$$

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [3, 19].

2.8 Choosing buffer management algorithms with machine learning

In this section, we study the design principles of a self-adjusting queueing module (QM), which is an essential building block in every network element. Traditional network management allows only to deploy a fixed set of buffer management policies optimizing predefined objectives and incorporating specific traffic parameters. Finding the "perfect" policy settings is a complicated task that requires high qualifications of network operators. Moreover, well-chosen parameters reflect only the current understanding about traffic patterns: usually, there is no *closed loop* between network operators and the QM, and as a result the performance of even well-chosen policy parameters will deteriorate when traffic patterns change over time.

Instead of implementing a complex closed loop, we propose a declarative approach, where the QM is provided with a set of "implementable" policy instances and metrics to compare their performance. Each policy instance is a "black box", and network operators can supply to the QM multiple "reasonable" policy instances. The QM infrastructure exploits machine learning techniques and deploys currently best suitable policy candidate. This approach is based on the multiarmed bandits model, where the best policy (arm) has to be chosen based on incomplete information, minimizing the total cost of learning (regret). However, to make multiarmed bandits useful in real settings, one has to choose which model is best suited: different objectives and assumptions lead to different bandit algorithms. One also has to solve problems related to discretization and representation of performance indicators, e.g., choose the measurement interval; in this section, we discuss these issues, and our results have been successfully validated in practice. Moreover, various programming abstractions have recently defined atomic primitives that can be used to compose buffer management policies. In this case, a network operator can supply to QM instances of such atomic primitives (instead of policy instances) and the QM infrastructure will build a set of candidate

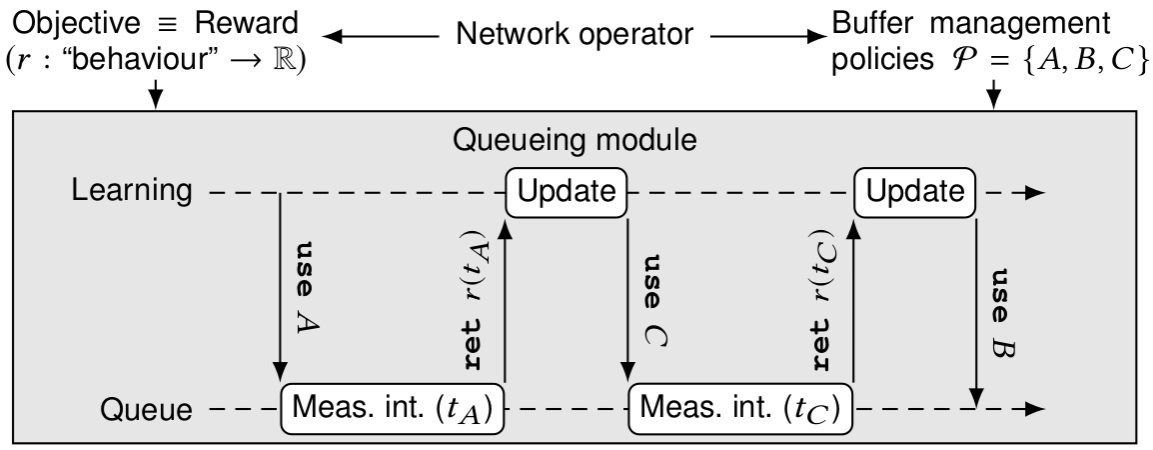


Figure 2.11: System structure: the network operator defines an objective and a set of policies. The learning algorithm interacts with the queue to identify the best policy.

policies by making the QM even less dependent on network operator skills in the design of new policies.

We assume that a network operator configures the QM in a *declarative* way, by specifying a set of *candidate policies* $\mathcal{P}$ and a *reward function* r, $0 \leq r(t) \leq 1$, that expresses a desired objective (more is better) and allows to evaluate QM performance over any time interval t; sample candidate policies might include *RED* [105] or *CoDel* [123] with various parameters, and r could be, for instance, the network power [203], average packet delay, or utilization over respective time interval. Each policy instance is an implemented algorithm with fixed set of parameters; the QM knows nothing about their internal operation except that they halt and can run at line rate.

Let $\mathcal{T}$ denote the time period of QM operation. The currently prevalent approach is to use a fixed policy configured by the network administrator for the entire $\mathcal{T}$. This static approach has limitations. First, the administrator must have a deep understanding of traffic patterns to make the right choice; even choosing the best policy by data collected about the incoming traffic is a very hard problem in general, let alone trying to predict future traffic patterns. Even more importantly, the network traffic patterns are likely to vary during $\mathcal{T}$, and as a result the performance of each individual policy will vary too. As an example, Figure 2.12 (which is a real evaluation result) shows a time interval $\mathcal{T}$ divided into T_1 and T_2, and each of the two policies *CoDel* [123] and *RED* [105] yields the best reward during its own interval. Thus, in order to achieve optimal performance we should

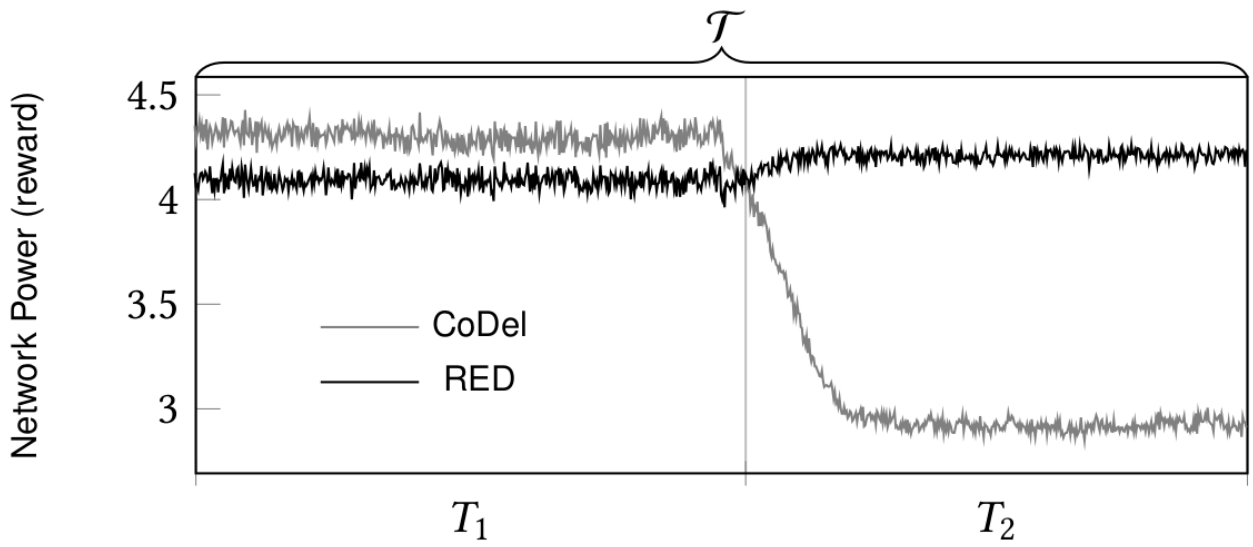


Figure 2.12: Sample behavior of two policies; shaded areas show the loss incurred by using a single policy over $\mathcal{T}$.

run different policies on different subintervals T_i; the shaded area on Figure 2.12 shows losses incurred by using a single policy throughout $\mathcal{T}$.

Our main goal in this section is to design the queue management infrastructure in such a way that would *automatically* adapt to traffic changes and identify the policy that is hypothetically best suited for current traffic patterns. The optimization objective is to choose such a sequence of policies over $\mathcal{T}$ that would maximize the overall performance $r(\mathcal{T})$. The QM infrastructure knows neither what kind of traffic is expected during $\mathcal{T}$ nor the performance of buffer management policies on that traffic.

We measure the performance of each policy over smaller *measurement intervals* and then choose the policies to use on future intervals. We denote the reward of a policy P active on a measurement interval t by $r_P(t)$ and divide $\mathcal{T}$ into *stationary* subintervals $T_1, \ldots, T_q$ on which the best policy does not change, i.e., for every j a single policy $P_j^* \in \mathcal{P}$ shows optimal expected performance over T_k; e.g., Figure 2.12 shows three subintervals with optimal policies P_A, P_C, and P_B respectively. Optimal $r(\mathcal{T})$ can be achieved by running P_j^* during each T_j.

Unfortunately, the QM cannot switch policies perfectly for two reasons: it know neither when each T_j begins nor what P_j^* is. The main goal of this section is to present approaches to approximate this desirable behavior by providing a solution to the *stationary optimization* problem: identifying and running the best policy P^* over a stationary interval T. We leave *non-stationarity detection*, i.e.,

identifying a switch between two stationary intervals, for further study.

The principal objective of stationary optimization is to run the best policy P^* for the largest possible fraction of T. In order to identify P^*, some part of T has to be spent on exploration through trial and error. There are two general approaches:

(a) *evaluate then exploit*: dedicate an *exploration interval* Δ in the beginning of T for the evaluation of different policies, and then use the resulting best policy for the rest of T;

(b) *continuous exploration*: combine evaluation with exploitation of the currently best policy; this approach keeps incurring exploration costs but can converge on the best policy with probability 1 and can handle dynamic policy rewards.

To apply known algorithms with proven results, we need to make assumptions regarding the properties of the process. We introduce several different cases that may lead to completely different problems.

The simplest case occurs if we are able to find the best policy after trying each policy once, i.e., if the best policy is *always* better than any other, on every time interval.

Assumption 2.8.1 (DISJ(Δ))**.** *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals and any $P \in \mathcal{P}$, $P \neq P^*$, we have $r_{P^*}^{\min} > r_P^{\max}$, where $r_P^{\min} = \min_i r_P(t_i)$, $r_P^{\max} = \max_i r_P(t_i)$, and P^* is the best policy as defined above.*

One way to relax DISJ(Δ) is to assume some (unknown) probability distribution over possible traffic behaviors. The stationarity assumption here would be that the expected reward for every policy remains constant, and rewards on different time intervals are independent random variables.

Assumption 2.8.2 (AVGEQ(Δ))**.** *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, any $P \in \mathcal{P}$, and any $1 \leq i < j \leq n$, $\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$, and $r(t_i)$ and $r(t_j)$ are independent.*

Still, in practice long-term dependencies often do exist, so one must look for an even more relaxed (but still hopefully tractable) setting. The next setting assumes only that the more often we try a policy, the closer the average reward gets to its true performance; this is a strict relaxation of AVGEQ(Δ).

Assumption 2.8.3 (LIMIT(Δ))**.** *For a set $\Delta = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, any $P \in \mathcal{P}$, and any sequence $1 \leq i_1 < \ldots < i_k \leq n$, there exists the limit $\lim_{n,k\to\infty} \frac{1}{k} \sum_j r_P(t_{i_j})$.*

Finally, we introduce the most general setting ADV(T), where the inputs can be completely arbitrary and even adversarial.

Assumption 2.8.4 (ADV(Δ))**.** *$r_P(t_i)$ are arbitrary.*

Note that in reality these assumptions pertain to properties of the traffic distribution.

Evaluate then exploit. In this setting, we assume that a special evaluation period Δ, a prefix of T, is reserved for pure exploration; this problem is known in machine learning as *best arm identification*. The main assumption here is that Δ is representative of the entire T, and the rewards of policies will not change over time, or at least the best policy will stay the same. The reward obtained during Δ is irrelevant for the learning algorithm's objective, but it does lead to a loss in the total regret, so the length of Δ represents a fundamental trade-off between exploration and exploitation, i.e., between the accuracy of identification and the fraction of time when suboptimal policies are active. We assume that Δ contains n reward measurement intervals $t_1, t_2, \ldots, t_n$.

Problem 2.8.1 (FindBest)**.** *For $\Delta = t_1 \sqcup \ldots \sqcup t_n$, find a policy $P \in \mathcal{P}$ that maximizes $\sum_{i=1}^{n} r_P(t_i)$.*

We now see a natural tradeoff between assumptions and performance: we would like to relax the restrictions as much as possible, but more general methods might perform worse.

Under DISJ(Δ) there is no problem: LocalGreedy is perfect. Under AVGEQ(Δ), there is a natural way to reduce the variance of estimates: average over several reward measurements. A straightforward Uniform strategy tries every policy $\lfloor n/|\mathcal{P}| \rfloor$ times and then chooses the policy with the highest average reward. It is known that under AVGEQ(Δ) the error probability of Uniform decreases exponentially with n [71]. However, intuitively, if a policy has already performed very poorly several times, the rest of Δ is probably better spent distinguishing between other policies. In [54], this intuition was formalized as the SuccessiveRejects algorithm that has a provably better performance than Uniform while still being parameter-free. Under LIMIT(Δ), one cannot guarantee that by the end of any given Δ the correct policy is chosen with high probability: it is always possible that n was not large enough. Still, the SuccessiveHalving algorithm [119] needs provably fewer measurements than the round-robin algorithm under LIMIT(Δ). Finally, under ADV(Δ) there is nothing we can do: the

Assumption	Constraints	Method	Accuracy
	FindBest		
DISJ(Δ)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	exact
AVGEQ(Δ)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	SR	$1 - e^{O(n)}$
LIMIT(Δ)	$\exists\ \lim_{n,k\to\infty} \frac{1}{k} \sum_{j=1}^{k} r_P(t_{i_j})$	SH	$>$ RR
	MaxReward		
DISJ(T)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	0
AVGEQ(T)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	UCB1	$O(\log n)$
ADV(T)	—	Exp3.1	$O(\sqrt{n})$

LG = LocalGreedy, SR = SuccessiveRejects,
SH = SuccessiveHalving, RR = RoundRobin

Table 2.1: FindBest and MaxReward summary.

exploration part of the interval has no bearing on what happens next. Table 2.1 contains a short summary of the results.

Continuous exploration. Making decisions based on a prefix Δ is a bad idea if Δ does not accurately represent the entire period T. At the end of the day, best arm identification is merely an improved version of Uniform, and it falls prey to the same issues: large error probability under AVGEQ(Δ) or insufficient rate of convergence under LIMIT(Δ). To address this, we allow to evaluate any policy at any moment, getting the classical *exploration vs. exploitation* tradeoff between gathering new information and using the currently best hypothesis.

Problem 2.8.2 (MaxReward)**.** *For a set $T = t_1 \sqcup \ldots \sqcup t_n$ of measurement intervals, assign a policy from $\mathcal{P}$ to each interval so that $\mathbb{E}\left[\sum_{i=1}^{n} r(t_i)\right]$ is maximized.*

In MaxReward, the exploration/exploitation trade-off is implemented by policy selection instead of choosing Δ. This is a more flexible approach: decisions can be made online, on the basis of already collected data. In what follows we extend the same assumptions from Δ to the entire interval T.

DISJ(T) is again trivial: LocalGreedy is still perfect. Under AVGEQ(T), average rewards of each policy are assumed to be equal among all measurement intervals $t_1, \ldots, t_n$. A policy's performance can still be evaluated by averaging over multiple reward measurements. This is the default setting for the multi-armed bandits problem, and classical results in this field apply [192]. For example, the ϵ-greedy algorithm (run the currently best policy with probability $(1-\epsilon)$ and a

random policy with probability ϵ) yields expected regret $O(\epsilon n)$, linear in $n = |T|$. Better algorithms are motivated by "optimism under uncertainty": assume the best if little is known. The UCB1 algorithm [56] has expected regret $O(\log n)$; on t_i, UCB1 chooses $P = \text{argmax}_P \left[\hat{\mu}_{P,i} + \sqrt{2 \log i / n_{P,i}}\right]$, where $\hat{\mu}_{P,i}$ is the current estimate of μ_P (empirical average reward), and $n_{P,i}$ is the number of times policy P has been chosen. Its variance-aware version UCB-V [55] uses priorities

$$\hat{\mu}_{P,i} + \sqrt{\frac{2V_{i,n_{P,i}} E_{n_{P,i},i}}{n_{P,i}}} + c\frac{3bE_{n_{P,i},i}}{n_{P,i}},$$

where $V_{i,n_{P,i}}$ is the empirical variance estimate for arm i, $E_{n_{P,i},i}$ is the exploration function, usually $E_{n_{P,i},i} = \zeta \log i$, and c, ζ are constants.

If there are no restrictions at all, ADV(T), an optimal solution to MaxReward may have to use several different policies during T, which was not the case for DISJ(T) or AVGEQ(T). There can be no general performance guarantees, but since we know that T is stationary we can restrict ourselves to comparing only against single-policy solutions, i.e., with $\max_P \sum_{i=1}^{n} r_P(t_i)$. Unlike FindBest, now it becomes possible to outplay even an adversary (at least on average) with a randomized algorithm. One possible such algorithm in this case is called Exp3.1 [57], shown to guarantee expected regret $O(\sqrt{n})$ [57], again sublinear, so per-interval loss still tends to zero, although slower than for UCB1 under AVGEQ(T). Note that learning algorithms should be fast since they should not slow down network elements; fortunately, all algorithms above have very low complexity: e.g., UCB1 or UCB-V need to keep a priority queue of policies and store a few numbers per policy; the number of different policies is low in practice and does not depend on the network load or traffic.

All of the above results have been supported by an extensive simulation study that validates the performance of proposed algorithms; this simulation study, as well as more details about the approach, are shown in the work [6].

Chapter 3 Packet classification algorithms

3.1 Introduction to packet classification and related work

Packet classification is a core functionality for popular commodity services such as Quality of Service (QoS) and access control; it has become even more prominent with the adoption of OpenFlow [169] that expresses features through hierarchical tuple matching with set actions; multi-field classification has become common. A classification rule that sets exact values for all fields can be represented by a concatenation of all fields; the classifier can then be implemented in content-addressable memory (CAM) or by a simple hash function in space linear in the number of fields. The problem becomes harder if a field is represented by a prefix (constraint on most significant bits) or a range (confining values inside an interval) since concatenation does not work.

Many sophisticated software-based approaches have been proposed (cf. the comprehensive survey [198]). Complexity bounds derived from computational geometry imply that a software-based packet classifier with N rules and $k \geq 3$ fields uses either $O(N^k)$ space and $O(\log N)$ time or $O(N)$ space and $O(\log^{k-1} N)$ time [110, 171]. Thus, software-based approaches are either too slow or too memory-intensive even with few prefix- or range-based fields: with just 100 rules and 4 fields, N^k space is about 100MB, and $\log^{k-1} N$ time is about 350 memory accesses [110].

Ternary content-addressable memory (TCAM) was introduced to overcome performance problems for prefix matching [166]. TCAM, the *de facto* standard for classifier implementation, is a fast class of memory for matching packet headers against a set of stored entries represented by tuples of Value and Mask words (hiding "don't care" bits). Unlike software-based solutions, TCAM can efficiently represent multi-field classification with prefixes, but suffers an exponential blowup from range expansion; each range-based field in a rule introduces an additional

multiplicative factor.

More sophisticated services have to be more expressive on existing fields or new ones, and many classifications are naturally represented by ranges. More efficient implementations for classifiers on ranges of IP or MAC addresses, dates, packet lengths, etc. are highly desired [85]. Modern packet classification mechanisms are limited and mostly consider five-tuples with at most two fields which include ranges, greatly restricting classification possibilities.

Let us formalize the problem setting. Packets are matched by their headers according to classification rules stored in a database. A packet header contains k fields; a field i, $1 \leq i \leq k$, is a string of W_i bits. A *classifier* $\mathcal{K}$ is an ordered set of N rules $R_1 \dots R_N$. A rule R_j is an ordered set of k fields and an associated action A_j; a field F_i is represented by a range of values on W_i bits, i.e., each rule contains k ranges $R = (I_1, \dots, I_k)$, $I_i = [l_i, u_i]$. We assume the last rule of each $\mathcal{K}$ is a "catch-all" rule $R_N = (*, \dots, *)$ with $A_N = \text{TRANSMIT}$ to transmit matched packets without changes. We denote by $\mathcal{K}^{-F}$ the classifier obtained from $\mathcal{K}$ by removing a set of fields F from each rule; by $\mathcal{K}^{+F}$, the classifier obtained from $\mathcal{K}$ by extending its rules with F (with values defined separately for every rule), and simplify notation to $\mathcal{K}^{-|F|}$ and $\mathcal{K}^{+|F|}$ when the fields are clear from context.

Rules are applied sequentially, i.e., the set of rules has non-cyclic ordering $\prec$, and if a header matches both R_x and R_y for $x \prec y$, rule R_x takes precedence. Classifiers are semantically equivalent if they match the same rule for every packet.

Research towards efficient implementations of packet classifiers falls into two main categories: algorithmic solutions (usually software-based) and TCAM-based solutions. Algorithmic solutions mainly rely on one of three techniques: decision trees, hashing, or coding-based compression. The works [113,182,197] suggest how to partition the multi-dimensional rule space. Possible matching rules are found by tracing a path in a decision tree. Techniques to balance the partition in each node exist, but rule replication often cannot be avoided; see a related approach in [211]. There is an inherent tradeoff between space and time complexities in these approaches. Song and Turner's ABC algorithm for filter distribution offers higher throughput with lower memory overhead and can tune the implementation for better time complexity or better space complexity [184]. The works [94, 122] discuss hash-based solutions to match a packet to its possible matching rules. Efficient coding-based representations are shown in [148, 174, 176, 180].

Other works discuss efficient TCAM implementations. TCAMs do not natively support ranges, and a range often cannot be represented by a single TCAM entry. Two popular range encoding schemes are the binary encoding [188] and the SRGE encoding [70]. In the binary encoding, a W-bit range is encoded as a union of disjoint subtrees in the binary tree of 2^W leaves. Each subtree is represented by a single prefix TCAM entry. With this scheme, the maximal number of entries to encode a range (worst-case expansion) is $2(W-1)$. For example, for $W = 5$ the range $[16, 23]$ is encoded by a single entry 10*** while the range $[1, 30]$ is described by $2W - 2 = 8$ entries 00001, 0001*, 001**, 01***, 10*** 110**, 1110*, 11110 that represent the eight subtrees of $[1], [2, 3], [4, 7], [8, 15], [16, 23][24, 27], [28, 29], [30]$, respectively. The SRGE encoding can improve the worst-case bound to $2(W-2)$ by representing values in Gray code. In this encoding, TCAM entries are not necessarily prefixes and do not necessarily represent disjoint subsets of the range. For example, for $W = 5$ the range $[1, 30]$ can be encoded by *0001, *001*, *01**, *1***. Both schemes encode a multi-field range with a number of entries that is exponential in the number of fields k with upper bounds of $(W-2)^k$ and $(W-4)^k$, respectively. When entries that negatively encode the complement of the ranges can be used ("deny" entries), the upper bound is improved to W entries [177,178]. The works [178,179] suggest schemes to encode a k-field range with complexity linear in k. However, they apply only to encoding a single rule; to encode a classifier with more rules, one has to change the conventional TCAM architecture.

Several works have tried to reduce the number of TCAM entries for a given classifier by relying on heuristics [75, 162]. These techniques include encoding the most common ranges with additional bits [76, 156], applying block permutations [201], designing tree-based architectures [196], and redundancy removal [155]. While many such heuristics could be improved with randomization [151], they usually limit the number of fields in a rule or perform badly as the number of fields increases. Schemes for representing rule updates by timestamp-based TCAM range rules have been shown in [193]. Efficient schemes for classification and update supporting that rely on rule disjointness have been proposed in [69, 185, 200].

Geometrically speaking, a rule with multiple prefixes represents a multidimensional hyper-rectangle in the search space, a classifier is a set of hyper-rectangles with assigned priorities, and the classification problem is to find the highest prior-

ity hyper-rectangle covering a given point. Point location problem involves finding the enclosing region of a point, given the set of non-overlapping hyper-rectangles. Complexity bounds for the point location problem imply that a software-based packet classifier with N rules and $k \geq 3$ fields uses either $O(N^k)$ space and $O(\log N)$ time or $O(N)$ space and $O(\log^{k-1} N)$ time [171]. Hyper-rectangles corresponding to classification rules can overlap, making packet classification at least as hard as a point location. In this chapter we exploit order-independence to reduce the number of fields participating in classification. For instance, reducing representations of classifiers down to two fields corresponds to the point location problem in two dimensions, leading to representations with logarithmic lookup time and (near-)linear space [84, 99, 143].

Exploiting order-independence to optimize power and lookup time had been previously discussed in [63, 64]. However, we know of no work prior to the results presented in this chapter that considers order-independence specifically to reduce classification width and support additional expressiveness. Whereas power optimization is out of scope of this chapter, in the TCAM case, assuming that power consumption linearly depends on the number of TCAM cells participating in the lookup, reducing classification width also improves power consumption.

3.2 Exploiting order independence for packet classification

In this section, we introduce an abstraction layer that hinges on the *order-independence* of classifiers with respect to the rules they encompass. We identify this property as essential for simple and efficient implementation of classifiers. We show that if a classifier is order-independent then new classification fields — even range-based — do not increase either time or space complexity. We use this to build efficient classification schemes. We also consider the general case with order-dependence, leading to a hybrid approach implementing order-independent rules in software with linear memory and logarithmic worst-case guaranteed lookup time, and the rest of the rules in TCAM. Specifically, our contributions are the following:

(1) we identify classifier properties that guarantee zero cost for additional fields represented by ranges or prefixes transparently to the implementation scheme;

(2) we propose a technique to reduce the number of classification fields represented by prefixes or ranges, yielding semantically equivalent classification results;

(3) we present a technique to reduce a multi-field classification problem to a classification problem on at most two fields with guaranteed worst-case performance and linear memory, improving upon previous reduction attempts which do not provide acceptable worst-case guarantees when classifiers start with more than two fields [185, 188].

We say that two rules R_1 and R_2 of a classifier *intersect*, or *conflict*, if there exists at least one header that matches both R_1 and R_2. For instance, for $w = 4$, and $R_1 = (1\ 0\ 0\ *)$, $R_2 = (0\ 1\ *\ *)$, and $R_3 = (1\ *\ *\ *)$, R_1 and R_3 intersect (the header (1 0 0 0) matches both rules). In contrast, R_1 and R_2 are *disjoint*. R_1 and R_2 are *order-independent* if the corresponding sets of matching headers are disjoint, and *order-dependent* otherwise. For such rules, every header matches at most one of them. Two rules R_1 and R_3 are *transitively* order-dependent on a subset of fields $\mathcal{F}$ if there exists a rule R_2 such that R_1 and R_2 are order-dependent and R_2 and R_3 are order-dependent.

A classifier $\mathcal{K}$ is called *order-independent* if any two of its rules are filter-order-independent; otherwise, $\mathcal{K}$ is order-dependent. Note that order-independence is equivalent to the fact that for any classifier $\mathcal{K}'$ with the same rules as $\mathcal{K}$ sorted in a different order (except the last "catch-all" rule) any packet header p is matched by the same rule in $\mathcal{K}$ and $\mathcal{K}'$. This condition is satisfied when rules do not intersect, i.e. for each pair of rules there is at least one field in which the corresponding ranges (or prefixes) are disjoint. For instance, a classifier $\mathcal{K}$ with two rules based on two fields $R_1 = ([1, 3], [4, 5])$ and $R_2 = ([5, 6], [4, 5])$ is order-independent, but $\mathcal{K}'$ with $R_3 = ([1, 3], [4, 5])$ and $R_4 = ([2, 4], [4, 5])$ is order-dependent: R_1, R_2 do not intersect since the ranges in their first field are disjoint while R_3 and R_4 do (e.g., $(3, 4)$ matches both R_3 and R_4). $\mathcal{K}(S)$ denotes a classifier that uses only a subset S of fields in classification.

Order-independence of classifiers, or parts thereof, is a prevalent and powerful property for efficient classifier representation and evaluation. Classifiers with longest-prefix matches on a single field are at least order-independent among all prefixes of the same length; forwarding entries in OpenFlow [169] likely preserve order-independence for simplicity of management; many classifiers representing services with "best-match" priority schemes are order-independent. Order-

dependent classifiers representing service-level-agreements (SLAs) are often unintended byproducts of uniting several services in one classifier. In our experience, many classifiers representing services such as QoS or ACLs are order-independent in practice. Increasing the number of rules of a classifier decreases the chance that it will stay order-independent *as a whole*, but the size of its *maximal order-independent part* increases. Adding fields to each rule increases the chances of order-independence. In this section, we give a first intuition of how order-independence can drastically reduce TCAM space in a classifier.

To study the impact of order-independence, we analyze classifiers from ClassBench generated with real parameters [83] and real-life classifiers provided by Cisco Systems. The data in Tables 3.1 and 3.2 indicate new methods for efficient representation of order-independent parts of classifiers can greatly reduce requirements for classification engines. If a classifier is not wholly order-dependent, we can lookup its order-independent and order-dependent parts separately and return the highest-priority match. An order-independent match preempts the need to match the order-dependent part; to this end, we can choose rules for the order-independent part which have higher priorities than any rule in the other part.

Example 3.2.1. Consider an order-independent classifier $\mathcal{K} = (R_1, R_2, R_3)$ with two fields of 5 bits each and rules

$$\begin{aligned} R_1 &= ([1,3],[4,31]),\\ R_2 &= ([4,4],[2,30]),\\ R_3 &= ([7,9],[5,21]). \end{aligned}$$

Let $\mathcal{K}^{+1}$ be a classifier that results from $\mathcal{K}$ by adding one new field of 5 bits, leading to the rules

$$\begin{aligned} R_1^{+1} &= ([1,3],[4,31],[1,28]),\\ R_2^{+1} &= ([4,4],[2,30],[4,27]),\\ R_3^{+1} &= ([7,9],[5,21],[3,18]). \end{aligned}$$

Since $\mathcal{K}$ is order-independent, the new fields can be skipped during (the main TCAM-based) classification and a single matched rule verified to avoid false positive matches (see Figure 3.2). The binary encoding [188] of $\mathcal{K}^{+1}$ requires 42+28+50 = 120 TCAM entries and the SRGE encoding [70] requires 24+8+32 = 64 entries. On the other hand, the binary and SRGE encodings of $\mathcal{K}$ require only $6+7+10 = 23$ and $6+4+8 = 18$ TCAM entries respectively.

	Rules		Original $\mathcal{K}$			By Theorem 3.2.2		
	Total	Order-ind., all fields	Width, bits	Space, Kb Binary	SRGE	OI wid, bits	Space, Kb Binary	SRGE
acl1	49870	49779	120	7922	7655	31	1517	1517
acl2	47276	44178	120	11289	11289	82	4189	4189
acl3	49859	47674	120	10771	10571	91	5018	5008
acl4	49556	46670	120	10079	9904	97	5379	5370
acl5	40362	38962	120	6121	6121	63	2950	2950
fw1	47778	43675	120	19454	19438	72	3926	3911
fw2	48885	48826	120	10866	10866	52	2498	2498
fw3	46038	41615	120	15090	15073	84	4161	4145
fw4	45340	42857	120	33500	33368	76	4025	4008
fw5	45723	39962	120	12478	12445	76	3759	3745
ipc1	49840	48294	120	8041	7924	50	2580	2579
ipc2	50000	50000	120	5859	5859	36	1757	1757
cisco1	584	538	120	78	78	52	34	34
cisco2	269	249	120	68	68	21	7	7
cisco3	95	92	120	11	11	30	3	3
cisco4	364	329	120	79	79	38	18	18
cisco5	148	120	120	19	19	17	5	5

Table 3.1: Space comparison. Left to right: # of rules (total and in a maximal order-independent set); TCAM space under two encodings (binary [188] and SRGE [70]) for ClassBench classifiers: standard representation and reduced by Theorem 3.2.2.

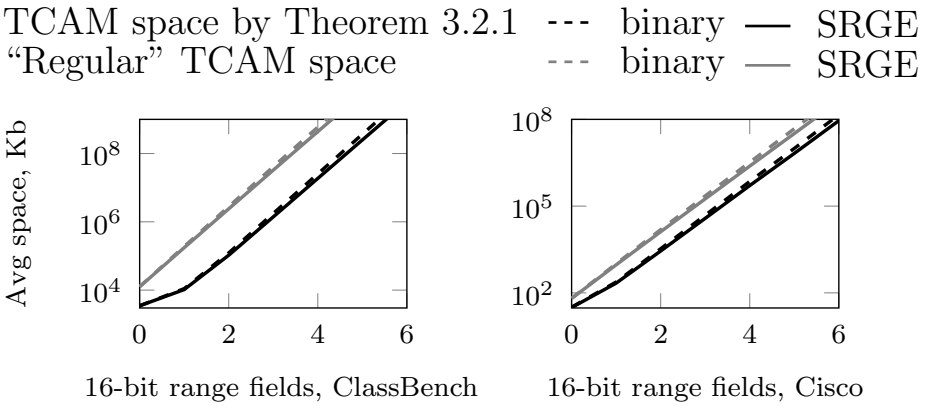


Figure 3.1: TCAM space for ClassBench and CISCO classifiers as a function of the number of additional 16-bit range fields.

Theorem 3.2.1 (Fields Expansion)**.** *Let $\mathcal{K}^{+m}$ be a classifier that results from an order-independent classifier $\mathcal{K}$ by adding m new fields of any width, i.e., a rule*

	Rules		Original $\mathcal{K}^{+2}$			By Theorem 3.2.1		
	Total	Order-ind., all fields	Width, bits	Space, Kb Binary	SRGE	OI wid, bits	Space, Kb Binary	SRGE
acl1	49870	49779	152	1752769	1462501	31	3857	3509
acl2	47276	44178	152	2499525	2159699	82	145925	126505
acl3	49859	47674	152	2391264	2027239	91	177771	152514
acl4	49556	46670	152	2234839	1895078	97	216994	186701
acl5	40362	38962	152	1359256	1172403	63	127818	110663
fw1	47778	43675	152	4303234	3720573	72	194313	164893
fw2	48885	48826	152	2399603	2071400	52	7382	6695
fw3	46038	41615	152	3337763	2873120	84	170688	144596
fw4	45340	42857	152	7438741	6390798	76	195130	165332
fw5	45723	39962	152	2745105	2366939	76	180420	152652
ipc1	49840	48294	152	1789100	1521153	50	52378	45391
ipc2	50000	50000	152	1301839	1123612	36	1757	1757
cisco1	584	538	120	78	78	52	34	34
cisco2	269	249	120	68	68	21	7	7
cisco3	95	92	120	11	11	30	3	3
cisco4	364	329	120	79	79	38	18	18
cisco5	148	120	120	19	19	17	5	5

Table 3.2: Space comparison. Left to right: # of rules (total and in a maximal order-independent set); TCAM space for the same classifiers as in Table 3.1 extended with 2 new random synthetic 16-bit range fields under two encodings (binary [188] and SRGE [70]): standard representation and reduced by Theorem 3.2.1.

$R = (I_1, \ldots, I_k)$ in $\mathcal{K}$ is replaced with $R^{+m} = (I_1, \ldots, I_k, \ldots, I_{k+m})$ in $\mathcal{K}^{+m}$. Then $\mathcal{K}$ with a false positive check of a single matched rule is a semantically equivalent representation of $\mathcal{K}^{+m}$.

By Theorem 3.2.1, introducing additional fields based on prefixes or ranges to an order-dependent classifier affects only the encoding size of its order-dependent part. The space and lookup time complexity of classification in the order-independent do not increase since new fields in the order-independent part can be ignored without affecting the classification outcome; an additional check of at most one matched rule is required to avoid a false positive match. Previously new fields based on prefixes or ranges significantly increased the complexity of software-based solutions (recall $O(\log^{k-1} N)$ lookup time in linear memory). Likewise, in TCAM-based solutions each range is converted to prefixes before configuration, and any new field based on ranges adds an additional multiplica-

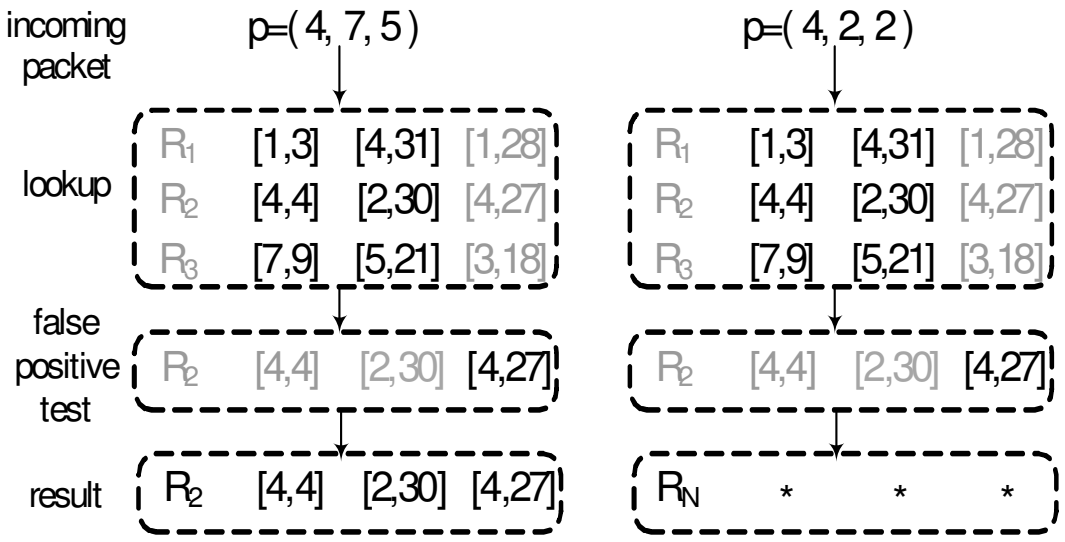


Figure 3.2: Visualizing the lookup procedure on rules with a subset of fields in Example 3.2.1. Fields checked in a certain step are in black; those irrelevant for the step, in grey. Packet $(4, 2, 2)$ matches R_2 but fails the false positive check on the added field of R_2^{+1}, so the classifier returns the catch-all rule.

tive factor for the required TCAM space. This contributes to the fact that 5-tuple classifiers with only source and destination port fields represented by ranges are the current industry standard. Support for additional fields amenable to range rules would greatly improve classification expressiveness (e.g., ranges on dates, packet length, etc.).

The effect of Theorem 3.2.1 is shown in Tables 3.1 and 3.2. In Table 3.2 we add two new ranges on 16 bits each to benchmark classifiers, garnering significant space savings. We compare memory requirements which result from Theorem 3.2.1 with those of two well-known range encoding schemes: binary [188] and SRGE [70]. In addition, Figure 3.1 demonstrates how average required space depends on the number of range-based fields; for our approach, the exponential growth is significantly deterred. Whereas Theorem 3.2.1 shows an equivalent representation of $\mathcal{K}^{+m}$ when new fields are added, Theorem 3.2.2 states we can ignore a subset of fields if the remaining fields maintain the order-independence of the classifier. Table 3.1 shows the effect of Theorem 3.2.2: for most benchmarks, significant space savings (by a factor of 2 to 5) caused by order-independence result immediately, on the classifiers themselves rather than their extended versions. The effect is independent of range encoding schemes.

Note that Tables 3.1 and 3.1 do not include memory requirements for the false positive check (on a single matched rule) since it does not require a conversion of

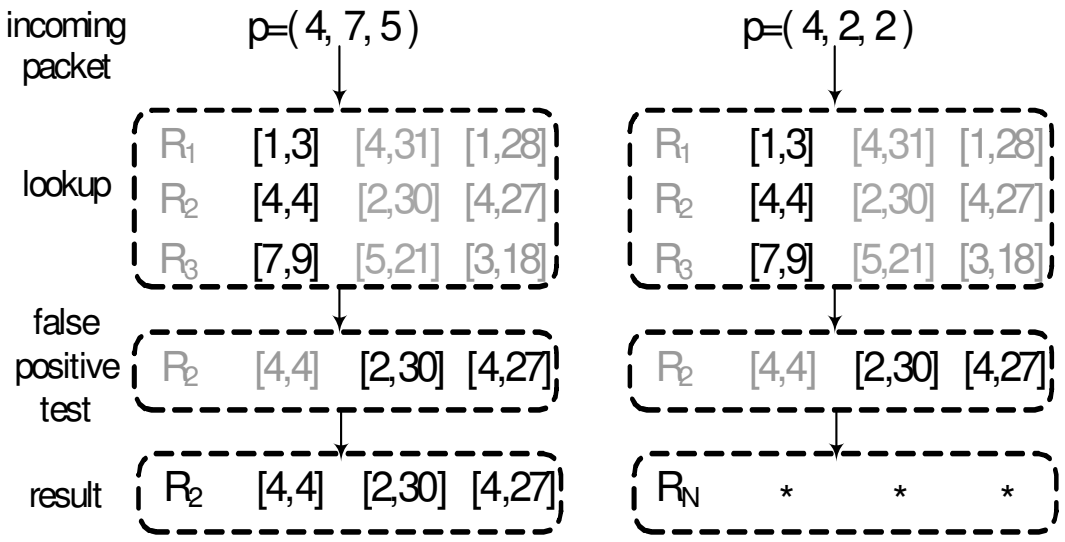


Figure 3.3: Visualizing the lookup procedure on rules with a subset of fields in Example 3.2.2. Fields to be checked in a step are shown in black; irrelevant for the step, in grey. Note that packet $(4, 2, 2)$ matches $R_2^{-\{2,3\}}$ but fails the false positive check, so the classifier returns the catch-all rule.

ranges to prefixes, and a different memory (e.g., SRAM, SDRAM) can be used for its implementation which is less of a bottleneck than TCAM. Moreover the required memory for false positive checks is only linear in the number of rules (before a conversion of ranges to prefixes) and classification width (in bits).

Example 3.2.2. Consider a classifier $\mathcal{K} = (R_1, R_2, R_3)$ with three fields of 5 bits each and rules

$$\begin{aligned} R_1 &= ([1,3],[4,31],[1,28]), \\ R_2 &= ([4,4],[2,30],[4,27]), \\ R_3 &= ([7,9],[5,21],[3,18]). \end{aligned}$$

In this case the first field is necessary but also sufficient to guarantee order independence, so we have that

$$\mathcal{K}^{-\{2,3\}} = \{R_1^{-\{2,3\}}, R_2^{-\{2,3\}}, R_3^{-\{2,3\}}\} = \{([1,3]),([4,4]),([7,9])\}$$

is order-independent (see Figure 3.3). That the binary encoding of $\mathcal{K}$ requires $42 + 28 + 50 = 120$ TCAM entries and the SRGE encoding of $\mathcal{K}$ requires $24 + 8 + 32 = 64$ TCAM entries. On the other hand, the binary encoding for $\mathcal{K}^{-\{2,3\}}$ requires $2+1+1 = 4$ TCAM entries and the SRGE encoding requires only $2 + 1 + 2 = 5$ TCAM entries.

Note that removing additional classification fields of $\mathcal{K}^{-\{2,3\}}$ based on ranges

can significantly reduce space complexity since the same $\mathcal{K}^{-\{2,3\}}$ is sufficient to represent $\mathcal{K}$ in Example 3.2.2. Only one false positive check of a matched rule $R_i^{-\{2,3\}}$ is required. To further improve performance, the removed fields of R_i can be checked in parallel. We believe that in many packet processing engines the false positive check can be done in parallel; for example, Cisco c12000 platform has a special logic mechanism that matches up to 32 ranges in parallel at line rate [79].

Theorem 3.2.2 (Fields Reduction)**.** *Let $\mathcal{K}^{-m}$ be a classifier that results from an order-independent classifier $\mathcal{K}$ by removing m fields, i.e., each $R = (I_1, \ldots, I_k)$ is replaced with $R^{-m} = (I_1, \ldots, I_{k-m})$. If $\mathcal{K}^{-m}$ is order-independent then $\mathcal{K}^{-m}$ with a false positive check of a single matched rule is a semantically equivalent representation of $\mathcal{K}$.*

By Theorem 3.2.2, detecting a false positive match of a rule $R_i^{-m} \in \mathcal{K}^{-m}$ requires only to check m additional fields of R_i. If a false positive is found the catch-all rule is used. At most one rule can match in $\mathcal{K}^{-m}$ classifiers, which hugely impacts the complexity and feasibility of implementation.

Since the tradeoff between lookup time and memory space for multi-field classification heavily depends on the number of fields k participating in classification, it is reasonable to explore ways to reduce k. For software-based solutions, any additional field that is represented by a prefix or range incurs additional complexity on the lookup time or memory space [171]. If the reduced classifier $\mathcal{K}^{-m}$ contains at most two fields, we can efficiently implement lookup in time logarithmic in N with (near-)linear memory [84, 99, 143, 188]. For TCAM-based solutions, if a classifier field is represented by a range, removing it reduces the required TCAM space proportionally to the number of prefixes used to express the range. In this section we present optimization problems as a step to achieve this goal. Moreover, removing fields can create a shorter classification format. Common TCAM formats include 72, 144, or 288 bits, so reducing the representation size across one of those barriers halves the required TCAM space. Over the course of this section, we present several ideas of how to reduce classifiers by exploiting order-independence properties, and introduce the corresponding combinatorial problems.

We begin by optimizing order-independent classifiers based on Theorem 3.2.2 (see Example 3.2.2).

Problem 3.2.1 (Fields Subset Minimization (FSM))**.** *Find a maximal subset of fields M of an order-independent classifier $\mathcal{K}$ such that $\mathcal{K}^{-M}$ is order-independent;*

if there are several such subsets, choose M with maximal total width (to minimize lookup word width).

FSM can only be applied to order-independent classifiers, and it does not always reduce the number of classification fields for order-independent classifiers. It may happen that, for instance, every field of just a single rule is required to keep order-independence of a classifier. One way to address these shortcomings is a *multi-group* representation. A subset of classification fields may suffice for order-independence of a subset of rules of a classifier. Assume that the rules of a classifier $\mathcal{K}$ can be assigned to β groups such that: (1) each rule belongs to a single group; (2) the rules of each group are order-independent on a subset of k fields of $\mathcal{K}$ (except the "catch-all" rule); (3) different groups can reuse the same fields to keep order-independence. By Theorem 3.2.2, a lookup to a group returns a single matched rule to be checked for false positives on all remaining fields; the matched non-false positive rule with the highest priority (from exactly β matched rules) is returned. We call this the *multi-group* representation of $\mathcal{K}$. Multi-group representations exist for any order-dependent classifier, but each group should be order-independent. The following theorem immediately follows by construction.

Theorem 3.2.3. *The multi-group representation of a classifier is semantically equivalent to the original classifier.*

Lookups to groups can be issued in parallel. If each group in a multi-group representation can be order-independent on at most two fields, we have a semantically equivalent SW-based representation of a classifier in linear memory and with worst-case guaranteed logarithmic lookup time. Example 3.2.3 shows how the rules of a classifier $\mathcal{K}$ can be split into groups.

Example 3.2.3. Consider $\mathcal{K} = \{R_1, R_2, R_3, R_4, R_5\}$ with three fields of four bits each and rules

$$\begin{aligned} R_1 &= ([5,10],[4,7],[4,5]),\\ R_2 &= ([1,4],[4,7],[4,5]),\\ R_3 &= ([1,9],[1,3],[4,6]),\\ R_4 &= ([1,9],[4,7],[1,3]),\\ R_5 &= ([1,9],[4,7],[5,6]). \end{aligned}$$

While the set of all rules $\{R_1, R_2, R_3, R_4, R_5\}$ is not order independent (for instance, $R_1 \cap R_5 \neq \emptyset$), the rules can be divided into two independent subsets, $\{R_1, R_2, R_3\}$ and $\{R_4, R_5\}$. The first two fields are necessary to keep the first

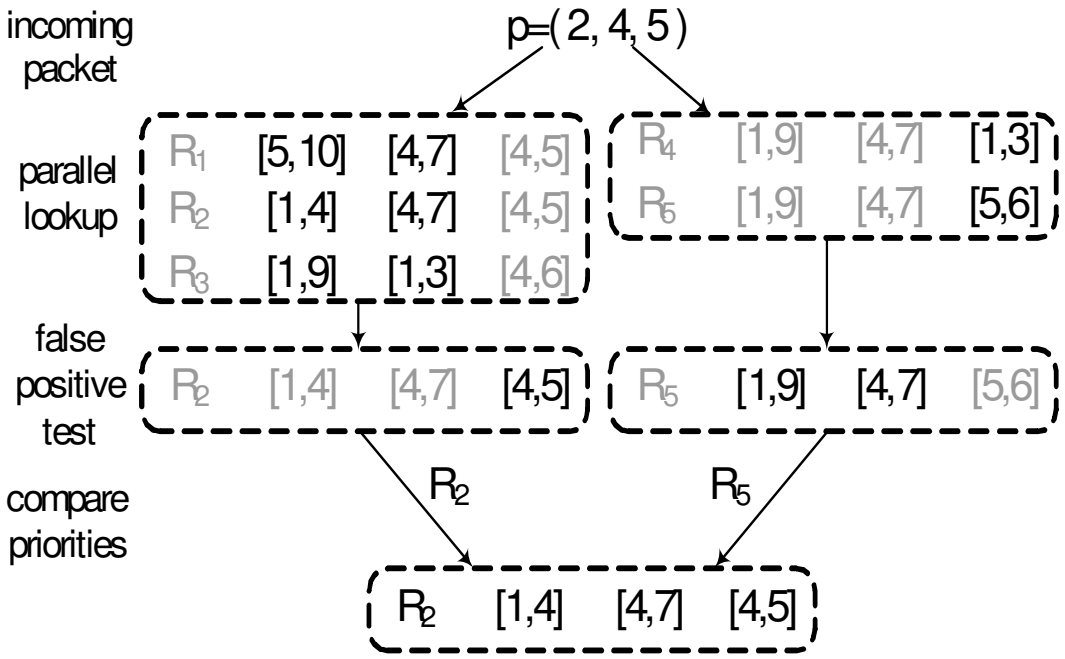


Figure 3.4: A visualization of the lookup procedure with a multi-group representation of the rules in Example 3.2.3. Fields to be checked on a certain step are in black; those that are irrelevant for this step, in grey.

group order-independent and the last field is enough to keep the second group order-independent.

Figure 3.4 shows a visualization of the lookup procedure after the grouping has been accomplished: for an incoming packet $p = (2, 4, 5)$, we issue parallel lookups for the two groups; these lookups return R_2 and R_5 and look only at fields $\{1, 2\}$ and $\{3\}$ respectively. Then we check if it was a false positive, matching p against the rest of the fields for the returned rules, and then select the match by rule priority. At most one rule will make it to the false positive test for each group. In case when no rule matches or the final test finds that the result was a false positive, the catch-all rule is sent to the priority comparison, so each group sends exactly one rule to the priority comparison for any packet.

In some order-independent classifiers removing any of the fields make it be order-dependent and accordingly the FSM cannot reduce the number of fields. This leads us to the following optimization problem that can be applied also to order-dependent classifiers.

Problem 3.2.2 (l-Groups of Rules (l-MGR))**.** *Given a classifier $\mathcal{K}$, find an assignment of $\mathcal{K}$'s rules to a minimal number of disjoint groups such that different groups of rules can be based on the same or different subset of at most $l \leq k$ fields,*

and each group is order-independent on these fields.

Problem 3.2.2 shows how to deal with order-dependent classifiers. However, the number of lookups that can be issued in parallel is a system parameter, and the minimal number of groups found by solving Problem 3.2.2 may be unacceptable. In this situation, we can decompose an order-dependent classifier $\mathcal{K}$ into order-independent part $\mathcal{I}$ and order-dependent part $\mathcal{D}$. We can perform parallel lookups on both parts and return the best matched rule. TCAM or another general implementation can be used to store $\mathcal{D}$. Since in practice a subset of order-dependent rules on k fields should be significantly smaller than N, significant TCAM space can be saved. The problem is finding which rules to put in I. One possibility is to find a maximal order-independent subset of rules on all k fields and run Problems 3.2.1 or 3.2.2 on $\mathcal{I}$. Finding maximal order-independent set on k fields is interesting when we extend already existing classifiers with new fields (Theorem 3.2.1). Another direction is finding an order-independent part that already has the desired properties.

Problem 3.2.3 (Maximum Rules Coverage (l-MRC)). *Given a classifier $\mathcal{K}$ and a positive number $l \leq k$, find a maximal subset $\mathcal{I} \subseteq \mathcal{K}$ which is order-independent on at most l fields.*

Example 3.2.4. Consider a classifier $\mathcal{K} = \{R_1, R_2, R_3\}$ with three fields and the first three rules from Example 3.2.3: $R_1 = ([5, 10], [4, 7], [4, 5])$, $R_2 = ([1, 4], [4, 7], [4, 5])$, $R_3 = ([1, 9], [1, 3], [4, 6])$. If $l = 2$, we can have a maximal subset $\mathcal{K}' = \mathcal{K}$ of three rules, as the first two fields guarantee that the set $\mathcal{K}$ of these three rules is order-independent. Likewise, if $l = 1$ and only one field can be selected to distinguish between the rules, we can have order-independent sets of two rules. By considering the first field, we can have the subset $\mathcal{K}' = (R_1, R_2)$. Using only the second field we can have other possible order-independent subsets such as $\mathcal{K}' = (R_1, R_3)$ or $\mathcal{K}' = (R_2, R_3)$. We can also see that the three rules together are not order-independent based on any single field and accordingly the three mentioned possible subsets are all maximal subsets.

The generalized version of l-MRC problem reuses the advantages of the decomposition into I and D and multi-groups representation.

Problem 3.2.4 ((β, l)-MRC). *Given a classifier $\mathcal{K}$ and two positive numbers $l \leq k$ and β, find a maximal subset of rules $\mathcal{I} \subseteq \mathcal{K}$ that can be assigned to at most β groups, where each group is order-independent on at most l fields.*

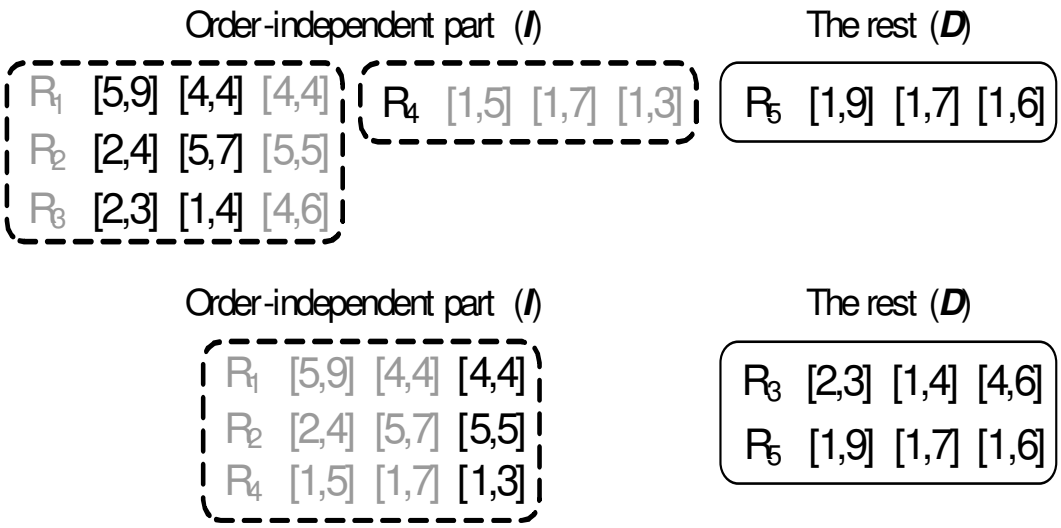


Figure 3.5: Illustration for Example 3.2.5. Top: a multi-group representation of $\mathcal{I} = \{R_1, R_2, R_3, R_4\}$ and $\mathcal{D} = \{R_5\}$. Bottom: a more compact representation of $\mathcal{I} = \{R_1, R_2, R_4\}$ and $\mathcal{D} = \{R_3, R_5\}$. Fields to be checked in a group lookup are in black; those that are irrelevant for this step, in grey.

In some cases, it may be feasible to send a few more rules to $\mathcal{D}$ and thus significantly reduce the number of groups; this usually happens in practice with more general rules at the bottom of a classifier.

Example 3.2.5. Consider a classifier $\mathcal{K}$ with three fields of five bits each and five rules

$$\begin{aligned} R_1 &= ([5,9],[4,4],[4,4]),\\ R_2 &= ([2,4],[5,7],[5,5]),\\ R_3 &= ([2,3],[1,4],[4,6]),\\ R_4 &= ([1,5],[1,7],[1,3]),\\ R_5 &= ([1,9],[1,7],[1,6]). \end{aligned}$$

The maximal order-independent subset on all three fields is $\mathcal{I} = \{R_1, R_2, R_3, R_4\}$, and one possible multi-group representation of $\mathcal{I}$ consists of two groups: $\{R_1, R_2, R_3\}$ (based on the first two fields) and $\{R_4\}$. But if we set $\mathcal{D} = \{R_3, R_5\}$, the rest will form a single group which is order-independent on a single field, the third one; see illustration on Fig. 3.5.

In this section, we discuss a special case when each field is represented by a prefix, so fields can be represented by a string of individual bits. This special case lets us represent classifiers as Boolean expressions and apply well known Boolean optimization techniques to reduce the width and number of rules. For this exposition, we concatenate all fields and treat a rule as a ternary bit-string

in the Value–Mask–Action format (since fields have a fixed width, we do not lose information with this transformation). In this part we consider order-independent classifiers; we decide whether an action should be applied to a packet, i.e., compute a Boolean function.

A rule $s = s_1 s_2 \dots s_k$, $s_j \in \{0, 1, *\}$, can be expressed as a conjunction $f_s(x_1, \dots, x_k) = \bigwedge_{s_i=1} x_i \wedge \bigwedge_{s_i=0} \bar{x}_i$. Further, an order-independent set of rules $\mathcal{R} = \{R_1, \dots, R_N\}$ with rules $s_1, \dots, s_k$ can be written as an unordered disjunction of individual rules, i.e., as a formula of depth 2 in disjunctive normal form (DNF, disjunction of conjunctions).

Example 3.2.6. The following set of rules with width 5

$$\begin{array}{ccccc} (0 & 1 & * & 0 & *) \to 1 \\ (* & 1 & 0 & 1 & *) \to 1 \\ (* & 1 & 1 & 1 & 0) \to 1 \\ (* & 1 & 1 & 1 & 1) \to 1 \end{array}$$

is order-independent and equivalent to

$$\begin{aligned} &f(x_1, x_2, x_3, x_4, x_5) = \\ &\quad = (\bar{x}_1 \wedge x_2 \wedge \bar{x}_4) \vee (x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_2 \wedge x_3 \wedge x_4 \wedge \bar{x}_5) \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_5). \end{aligned}$$

Thus, minimizing memory for representation of order-independent rules is equivalent to the following problem.

Problem 3.2.5 (MinDNF)**.** *For a given Boolean function f, find a minimal size DNF representation for f.*

DNF minimization may result in both reducing the number of rules (merging clauses in the DNF) and reducing the width of lookup necessary for classification (removing extra variables in the clauses).

Example 3.2.7. Let us minimize the set of rules from Example 3.2.6. We apply the well-known resolution heuristic [175] to clauses in the DNF of f:

$$\begin{aligned} (x_1 \wedge \bar{x}_2 \wedge \bar{x}_5) \vee (x_1 \wedge \bar{x}_2 \wedge x_5) &= (x_1 \wedge \bar{x}_2), \\ (x_1 \wedge \bar{x}_2) \vee (x_1 \wedge \bar{x}_2) &= x_2. \end{aligned}$$

At this point, we have $f(x_1, x_2, x_3, x_4, x_5) = (\bar{x}_1 \wedge x_2) \vee x_2$, and we can subsume

the first clause into the second: $f(x_1, x_2, x_3, x_4, x_5) = x_2$. After minimizing the corresponding DNF expression, we can express the result as a set of rules, a single rule in this case: $(*1***) \to 1$. Thus, in this example we have reduced the number of rules from four to one and have reduced the number of bits participating in the lookup from four (to apply the rules in Example 3.2.6 as stated, we would have to query all bits except bit 4) to one (bit 2).

The *MinDNF* problem has been extensively studied in complexity theory. Its decision version (given a formula φ in DNF and a number k, find whether there exists an equivalent DNF with size less than k) is unlikely to be in NP. It is obviously in $\mathrm{NP}^{\mathrm{NP}}$: to solve this problem, one needs to guess a small DNF and verify with the NP oracle that solves the satisfiability problem that these two formulas are indeed equivalent. In fact, *MinDNF* has been shown to be $\mathrm{NP}^{\mathrm{NP}}$-complete ($\Sigma_2^P$-complete), and some inapproximability results have also been proven [142, 195]. In practice, classical heuristics for *MinDNF* were based on Karnaugh maps and the Quine–McCluskey algorithm.

A different view of the problem appeared for the case of functions given by truth tables [50]; this means that the input size is exponential in the number of variables, and an algorithm is thus also allowed to use exponential memory. In this case, one can represent *MinDNF* as a special case of the set cover problem which leads to using the greedy set cover algorithm to find minimal DNFs. This algorithm has been shown to be $O(n)$-approximate, where n is the number of variables in the formula (for functions given by truth tables, $O(\log T)$-approximate, where T is the truth table size), and a matching lower bound has also been provided [50]. However, this approach is impractical for classifier optimization since practical classifiers look up several hundred bits, and it would be infeasible to construct the truth table explicitly.

At the first glance *MinDNF* is a more general problem than *FSM* since it can reduce both classification width and the number of rules. However, due to an additional check for a false positive match *FSM* can significantly reduce classification width beyond optimal results of *MinDNF*.

Example 3.2.8. Consider the following classifier:

$$\begin{aligned} R_1 &= (\ \texttt{100*},\ \ \texttt{001*}\ ), \\ R_2 &= (\ \texttt{1010},\ \ \texttt{0001}\ ), \\ R_3 &= (\ \texttt{000*},\ \ \texttt{****}\ ), \\ R_4 &= (\ \texttt{001*},\ \ \texttt{****}\ ). \end{aligned}$$

FSM can reduce the width of $\mathcal{K}$ from eight bits to four (if we treat fields as indivisible) or even two (if we are able to get down to the bit level). On the other hand, the only MinDNF heuristic applicable here is the resolution rule that can be applied to R_3 and R_4, getting

$$\begin{aligned} R_1 &= (\ \texttt{100*},\ \ \texttt{001*}\ ), \\ R_2 &= (\ \texttt{1010},\ \ \texttt{0001}\ ), \\ R'_3 &= (\ \texttt{00**},\ \ \texttt{****}\ ). \end{aligned}$$

This classifier has width 8; if we discard bits with identical values (second bit in the first field), we get width 7, but the savings are still small compared to FSM.

Table 3.3 shows how MinDNF heuristics apply to our experimental set of classifiers; we have applied MinDNF to the order-independent set from Tables 3.1 and 3.2. The number of binary rules is sometimes significantly larger than the number of original rules, and MinDNF does not have a significant effect on width, i.e., few bits become purely "don't care" bits even after MinDNF-style reductions. In Table 3.3, we show both "pure" width (cutting out bits that are purely "don't care") and further reduced width where we have cut out bits that have the same value, not necessarily "don't care". This is a Boolean counterpart of our previous reduction since we can use a single look up to check bits that always have the same value, and they do not change which rule matches an incoming packet. MinDNF does not significantly decrease width even with this reduction.

The example of FSM shows that algorithms that employ an additional constant number of false positive checks can significantly reduce the space required for optimal representations of Boolean expressions.

Note that TCAM minimization in case of order-dependent classifiers can be done via the following heuristic approach: reduce each set of rules that produce the same action separately; in what follows, we denote this port by 1 and all other

	Original classifier					MinDNF reduced						OI
	Rules, orig.	Rules, OI	Rules, binary	Rules, SRGE	Wid, bits	Rules, binary	Wid, bits	Red., bits	Rules, SRGE	Wid, bits	Red., bits	wid, bits
acl1	49870	49779	67511	65240	120	67505	90	90	65234	90	90	31
acl2	47276	44178	90772	90772	120	90233	104	104	90230	104	104	82
acl3	49859	47674	85252	83630	120	85226	106	106	83605	106	106	91
acl4	49556	46670	77837	76424	120	77755	106	106	76338	106	106	97
acl5	40362	38962	47514	47514	120	46261	96	94	46249	96	94	63
fw1	47778	43675	159525	159519	120	159461	112	112	159458	112	112	72
fw2	48885	48826	92646	92646	120	92316	88	88	92316	88	88	52
fw3	46038	41615	122495	122477	120	122259	112	112	122244	112	112	84
fw4	45340	42857	278887	277921	120	277799	104	104	276807	104	104	76
fw5	45723	39962	99574	99412	120	99421	112	112	99273	112	112	76
ipc1	49840	48294	66718	65734	120	66715	112	112	65731	112	112	50
ipc2	50000	50000	50000	50000	120	50000	112	112	50000	112	112	36
cisco1	584	538	603	603	120	475	104	86	474	104	86	52
cisco2	269	249	565	565	120	564	104	84	564	104	84	21
cisco3	95	92	92	92	120	92	88	68	92	88	68	30
cisco4	364	329	629	629	120	629	104	84	629	104	84	38
cisco5	148	120	139	139	120	139	104	84	139	104	84	17

Table 3.3: Experimental results of MinDNF reduction in the order-independent subsets of classifiers.

ports by 0. The difference from the previous case is that we need to encode the fact that a rule applies and none of the previous rules that produce some other action apply. Since we reuse the same zero-producing rules many times, the most natural representation is a Boolean circuit.

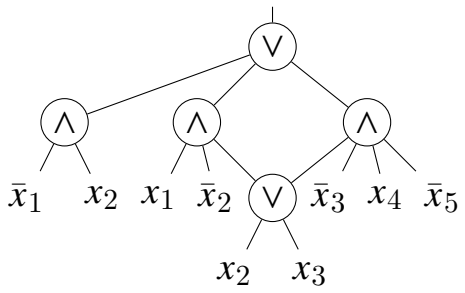
Example 3.2.9. The following ordered set of rules:

$$\begin{aligned}
(0\ \ 1\ \ *\ \ *\ \ *) &\rightarrow 1\\
(*\ \ 0\ \ 0\ \ *\ \ *) &\rightarrow 0\\
(*\ \ *\ \ 0\ \ 1\ \ 0) &\rightarrow 1\\
(1\ \ 0\ \ *\ \ *\ \ *) &\rightarrow 1\\
(*\ \ *\ \ *\ \ *\ \ *) &\rightarrow 0
\end{aligned}$$

can be represented as the following Boolean formula:

$$f(X) = (\bar{x}_1 \wedge x_2) \vee ((x_2 \vee x_3) \wedge \bar{x}_3 \wedge x_4 \wedge \bar{x}_5) \vee ((x_2 \vee x_3) \wedge x_1 \wedge \bar{x}_2)\,,$$

which is equivalent to the following circuit (we copy input variables for clarity):



Every ordered set of rules corresponds to a circuit of depth 3 with OR–AND–OR alternation, i.e., an AC^0_3 circuit. Moreover, TCAM rules reduce to a special case of AC^0_3: all $\wedge$-gates corresponding to rules below a certain rule with action 0 have to be connected to the corresponding level-1 $\vee$-gate. Thus, we arrive at the *Ordered Min-*AC^0_3 problem: given a Boolean function f, find its minimal size *ordered* AC^0_3 circuit, i.e., a circuit such that there exists a linear ordering $\prec$ on level-1 $\vee$-gates such that if a level-2 $\wedge$-gate is connected to a $\vee$-gate g, it is also connected to all $\vee$-gates g' such that $g' \prec g$.

From the worst-case complexity point of view, the Min-AC^0_d problem has been studied in [50]. It is obviously even harder than *MinDNF*, although to the best of our knowledge there are no known inapproximability results for *Min-*AC^0_d for small values of d such as $d = 3$ (there exist inapproximability results for larger d). We are not aware of any results about Ordered Min-AC^0_3 specifically, and for practical algorithms we recommend a generalization of heuristic rules for *MinDNF* discussed in the previous section. In the order-dependent case there is no easy reduction from multiple actions to two actions: rules with different actions may interleave. We can only reduce the general TCAM minimization problem to optimizing an ordered AC^0_3 circuit with multiple actions.

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. The viability of our approach was demonstrated through simulations on 12 classifiers from Classbench generated with real parameters [83], each with $\approx$ 50K rules on 6 fields, and on 5 real-life classifiers from Cisco Systems; we show that up to 90-95% of the rules can be implemented in linear space and with worst-case guaranteed logarithmic time, leaving only 5-10% to be implemented in TCAM. Detailed results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [14, 26, 37, 42].

3.3 Multigroup representations of packet classifiers

In this section, our results are primarily motivated by the (highly relevant for networking) problem of representing IPv6 routing tables on existing infrastructure designed for IPv4. While on the surface implementing IPv6 forwarding table lookups should be similar to IPv4 (longest prefix matching semantics remains the same), there are a number of details very important for building an efficient IPv6 forwarding engine. Below, we discuss the main design guidelines and sketch possible implementation strategies.

One recurring theme across IPv6 software FIB implementations is various forms of decision tree representations, e.g., *prefix trees* [62, 92, 98, 168, 183, 186]. Usually, tree nodes correspond to prefixes, node labels define an "action" (i.e., forward to given next-hop), and an LPM search involves traversing the tree along the path traced by the consecutive bits of the search key (i.e., the destination address). However, prefix trees are inherently *sequential*, meaning that an longest prefix match (LPM) search involves multiple consecutive lookups, each using the result of the previous one, and the total number of steps is bounded only by the width of the address space (128 for IPv6) because a prefix tree can contain multiple matching entries for a given IPv6 address, and in LPM we need to find the most specific one. Note that each step needs a separate random access to memory. There is an upper limit on the number of consecutive memory accesses performed per packet at line-rate. This imposes a strict budget on the number of levels for tree traversal, and hence most IPv4 FIB prefix tree representations cannot be naively extended to IPv6 within desired lookup time and memory requirements. Since routed address space is sparse, traditional forms of reducing the number of tree traversal steps (e.g., level-compression as in IPv4 [168,186]) cannot be applied for IPv6. Therefore, most proposals go to great depths to efficiently encode sparse multi-level subtrees inside tree nodes to save time on RAM accesses [62,92,98,183].

To address this limitation, IPv6 FIB representations can exploit the level of *parallelism* (how many parallel lookups can be done per packet at line-rate) that is an internal characteristic of each network element. A straightforward choice for parallelization of IPv6 FIB lookup would be classifying prefixes to groups, where each group would contain all prefixes of the same length [152, 199, 213] and perform lookups on each of the prefix groups simultaneously. Eventually, the

match within the group with the largest prefix length is output as the result. This scheme is extremely appealing for implementations because lookup inside groups boils down to a simplistic exact match, where the key is taken as the initial bits of the address as specified by the group's prefix length; this match can be easily done with a CAM or a hash. Unfortunately, the number of different prefix lengths, and hence the number of lookup threads needed to be executed in parallel, is too high in the IPv6 case, as most hardware (except GPUs [115]) lacks the necessary number of independent cores.

This discussion brings us to the main question we address in this section: how can we find alternative decompositions for an IPv6 FIB, more efficient and more amenable to parallelization than by prefix length? The decomposition must be such that:

(1) lookups inside each group are as simple as possible, preferably exact match,

(2) the number of groups is reasonable, e.g., in the order of a couple of dozen, and

(3) the number of bits participating in classification in every group is significantly smaller than the original classification width (e.g., it would be great to reduce from 128 to 32 or even 13 bits).

The latter requirement would allow to represent IPv6 FIBs on existing IP and MPLS infrastructure.

In this section, we give a positive answer to this question, introducing several heuristics that serve as a new abstraction layer, reducing lookup in IPv6 FIBs to several parallel lookups in longest prefix match classifiers with much smaller width, down to 32 and in some cases even 13 bits. These heuristics are based on the notion of *order independence* for sets of classification rules.

Similar to previous section, we introduce two operations on packet classifiers, namely (width) reduction and partitioning. Let B be an ordered set of bit indices, $B \subseteq \{0, \ldots, w-1\}$, referring to a subset of the bits in packet headers. For a header H, the (sub)header of H obtained by taking only bits of H with indices in B is called a *reduction of H to B*, denoted by H^B. Likewise, for a rule R (or a filter F) the *reduction of rule R* (*reduction of filter F*) to B, denoted by R^B (F^B, respectively), is the (sub)rule (subfilter) defined on the bit positions in B. Finally, for a classifier $\mathcal{K} = (R_1, \ldots, R_N)$, let $\mathcal{K}^B$ be the (sub)classifier obtained

from $\mathcal{K}$ by replacing each rule R in the classifier by R^B. The notions of matching, intersection, and order-independence carry over to such subsequences.

Note that any reduction of a prefix classifier is again a prefix classifier. Further notice that a reduced rule matches a larger set of packets than an underuced one (as certain, potentially valuable, bits in the filters are ignored), therefore a match in the reduced classifier does not immediately signal a match in the original classifier. Thus a match in a reduced classifier must always be validated using a false positive check, except for the case when only "don't care" bits are eliminated (like for prefix-length-based decompositions).

We also introduce the notion of classifier partitions, as a division of the rules in a classifier into distinct groups of rules. Consider a classifier $\mathcal{K} = (R_1, \ldots, R_N)$ and a subset of the index set $C \subseteq \{1, \ldots, N\}$. Now, the classifier $\mathcal{K}_C = (R_i : i \in C)$ is called the (*classifier*) *group* of $\mathcal{K}$ defined by C. Given any partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ of $\{1, \ldots, N\}$, where $\bigcup C_i = \{1, \ldots, N\}$ and $C_i \cap C_j = \emptyset$ whenever $i \neq j$, the *classifier partition* defined by $\mathcal{C}$ is given by exactly k classifier groups on $\mathcal{C}$: $\mathcal{K}_\mathcal{C} = \{\mathcal{K}_{C_1}, \ldots, \mathcal{K}_{C_k}\}$. Note that any partition of a prefix classifier is semantically equivalent with the original classifier, using the convention that from the matching rules of each group the output is chosen by longest prefix length.

Intuitively, prefix-length-based decompositions enjoy an appealing structural property: *for every input each subclassfier contains at most one matching rule.* This property makes representation of every group simple since all prefixes in a group can be represented as exact values; this is *strong* order independence. Unfortunately, prefix-length-based decompositions can contain too many groups in the worst case for IPv6. One interesting question we tackle in this section is the existence of structural properties in classifiers that would allow to consider prefixes with heterogeneous length to be mixed in the same group while still keeping the simplicity of representation. We argue that order-independent decompositions are a natural generalization of prefix-length-based decompositions in this context: given any classifier $\mathcal{K}$, it holds that $\mathcal{K}$ contains at most one matching rule for any header if and only if $\mathcal{K}$ is order-independent. This is a special type of order-independence that we call *weak*. This discussion motivates to limit our attention to the classifier decompositions that guarantee order-independence (weak or strong) inside each group on a subset of bit identities that allows to mitigate time-space tradeoffs in representations of IPv6 FIBs.

Problem 3.3.1. Minimal weak (strong) order-independent reduction. *Given a classifier $\mathcal{K}$ (not necessary order-independent) and a constant l, assign $\mathcal{K}$'s rules to a minimal number of disjoint groups such that different groups of prefixes can be based on subsets of at most l bit identities, and each group is weak (strong) order-independent on these bits.*

Finding a subset of bit identities that keeps order-independence on these bits of an order-independent (both weak and strong) classifier is already computationally hard for a single group; we show it by reducing the minimum test set problem [108]. For a set X of objects, a set $\tau \subseteq 2^X$ is called a *test set* if for every pair of elements $x, y \in X$ there is a test $t \in \tau$ such that $x \in t$ but $y \notin t$ (or $y \in t$ and $x \notin t$).

Problem 3.3.2 (Minimum test set)**.** *Given a test set τ on X, find a test set $Y \subseteq X$ with minimal $|Y|$.*

Theorem 3.3.1. *The special case of the minimum (weak or strong) order-independent reduction problem where all rules in $\mathcal{K}$ consist of 0 and 1 bits only is equivalent to the minimum test set problem.*

Theorem 3.3.1 shows that both weak and strong forms of minimum order-independent reduction are at least as hard as minimum test set, so intractability and inapproximability results for the latter carry over to the former. Thus, we obtain that for a prefix classifier with N rules, both order-independent reduction problems are not approximable within $(1 - \epsilon) \ln N$ for any $\epsilon > 0$ unless $NP \subseteq DTIME(N^{\log \log N})$ [66].

Although finding exact minimal reductions is computationally hard, we propose several heuristics that can be used in practice. All proposed algorithms follow the same general scheme shown as Algorithm 3: one by one, they construct a group with some OneGroup procedure, remove it from the current set of prefixes, and then call OneGroup again until the stopping condition is satisfied, i.e., either all prefixes of a given classifier are already covered or the number of created groups has reached a given level of parallelism β. Another parameter is the bit *exact* that shows whether we are trying to construct groups with only exact values or groups where $*$ bits are allowed. The building block for our heuristics is the MaxOI procedure that finds, for a set of ranges $[s_i, f_i]$, $i = 1..n$, the maximal set of non-intersecting ranges; the greedy Algorithm 4 (Earliest Deadline First, EDF) for this problem is optimal with $O(N \log N)$ time complexity [144].

We begin with greedy heuristics without maximal length. Algorithm 1 (that

Algorithm 1: OneGroup, greedy, no maximal width.

```
Data: Classifier 𝒦 of width k
1 begin
2     C ← MaxOI(𝒦)
3     B ← {1, ..., k}
4     for i = k, ..., 1 do
5         if OI(𝒦^C_{B\{i}}) then
6             B ← B \ {i}
7     return B ,C
```

Algorithm 2: The Regroup procedure.

```
Data: Classifier 𝒦, groups {(B_1, C_1), ..., (B_g, C_g)}
1 begin
2     for i = 1, ..., g − 1 do
3         for F ∈ C_i do
4             for j = g, g − 1, ..., i + 1 do
5                 if OI(𝒦^{C_j ∪ {F}}_{B_j}) then
6                     C_j ← C_j ∪ {F}
7                     C_i ← C_i \ {F}
8     for j = 1, ..., g − 1 do
9         for i ∈ B_i do
10            if OI(𝒦^{C_i}_{B_i \ {j}}) then
11                B_i ← B_i \ {j}
12    return {(B_1, C_1), ..., (B_g, C_g)}
```

is, Algorithm 3 with OneGroup procedure in Algorithm 1) simply selects the next group greedily with MaxOI and then tries to cut down its width (starting from the back because there are more $*$ bits there that are more desirable to cut). The OI procedure checks whether its input is order-independent (i.e., whether all ranges in the input are disjoint).

The main problem of Algorithm 1 is that the first group is usually very large and only order-independent on many bits, so the space savings are small. To alleviate this issue, we designed a postprocessing Regroup procedure (Algorithm 2). It tries to move rules from earlier to later groups without changing the bits of each group, and then reduce the widths once again.

Algorithm 3: MultiGroup

Data: Classifier $\mathcal{K}$, max. width l, parallelism β, *exact*

```
1 begin
2     i ← 0, T ← K, G ← ∅
3     while T ≠ ∅ and i < β do
4         G_i = K_{C_i}^{B_i} ← OneGroup(T, l, exact)
5         G ← G ∪ {G_i}
6         T ← T \ G_i
7         i ← i + 1
8     return G
```

Algorithm 4: MaxOI

Data: Set of prefixes $\mathcal{K} = \{F_1, \ldots, F_n\}$

```
1 begin
2     represent K as set of ranges F_i = [a_i, b_i]
3     sort K by upper bounds b_i, set k ← 1, S ← ∅
4     for i = 2, ..., n do
5         if a[i] > b[k] then
6             S ← S ∪ {F_i}, k ← i
7     return S
```

Next set of heuristics assign prefixes of a given classifier to groups with fixed maximal width l. The first approach, which we call MinSimilar, is to remove bit identities that have the most similar values in all prefixes. Let $\mathbf{0}_i$ ($\mathbf{1}_i$) be the number of prefixes whose ith bit is either 0 (1) or $*$, and let $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$. Algorithm 5 removes bit identities with minimal S_i: run MaxOI on the l bits that minimize S_i. If we need exact values, we choose them from the reduced classifier.

The alternative approach is to choose bit identities that distinguish as many pairs of prefixes as possible. Denoting by D_i the number of prefix pairs that differ in bit i, we get the MaxPair heuristic shown in Algorithm 6.

In Algorithms 5 and 6, we explored two ways to achieve strong order-independence (i.e., no $*$ bits in the filters): either removing all prefixes with $*$ bits prior to MaxOI or running the algorithm as usual and then expanding $*$ bits with an exponential blowup. These are the two extremes with respect to our two conflicting objectives: reduction of width vs. the number of filters. In the next heuristic, we consider a middle ground between them. Algorithm 7 shows the MinSimilar$_\delta$ heuristic where we remove filters with at most δ $*$ bits before

Algorithm 5: OneGroup, the MinSimilar heuristic.

Data: $\mathcal{K}$, max. width l, *exact*

begin
  for every bit $i = 1, \ldots, k$ **do**
    Compute $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
  Choose bit identities $B = \{b_1, \ldots, b_l\}$ with min S_i
  $C \leftarrow \mathcal{K}$
  if *exact* **then**
    $C \leftarrow \{F \in C \mid * \notin F^B\}$
  return MaxOI($\mathcal{K}_C^B$)

Algorithm 6: OneGroup, the MaxPair heuristic.

Data: $\mathcal{K}$, max. width l, *exact*

begin
  for every bit $i = 1, \ldots, k$ **do**
    compute $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
  Choose bit identities $B = \{b_1, \ldots, b_l\}$ with max D_i
  $C \leftarrow \mathcal{K}$
  if *exact* **then**
    $C \leftarrow \{F \in C \mid * \notin F^B\}$
  return MaxOI($\mathcal{K}_C^B$)

running MaxOI; the rest is exactly like MinSimilar. Algorithm 8 does the same for MaxPair.

We have extensively compared these algorithms with prior state of the art and between each other with a comprehensive experimental evaluation study. Detailed results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [20, 34, 35].

3.4 Packet classifier reduction on distributed platforms

As we have already discussed, the Internet routing infrastructure is widely acknowledged to be facing severe challenges in the long haul [149, 163]. While these challenges may not be critical for the control plane, the data plane is bogged down by the rapid expansion of forwarding tables constituting forwarding information bases (FIBs), a largely unsolved problem to date [100, 101]. Efficient FIB representations in the data plane become even more important with IPv6 since

Algorithm 7: OneGroup, the MinSimilar$_\delta$ heuristic.

Data: $\mathcal{K}$, l, δ, *exact*

begin
  for every bit $i = 1, \ldots, k$ **do**
    Compute $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
  Choose bit identities $B = \{b_1, \ldots, b_l\}$ with min S_i
  $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leq \delta\}$
  /* $*_{F^B}$ denotes the number of $*$ bits in F^B */
  return MaxOI($\mathcal{K}_C^B$)

Algorithm 8: OneGroup, the MaxPair$_\delta$ heuristic.

Data: $\mathcal{K}$, max. width l, *exact*

begin
  for every bit $i = 1, \ldots, k$ **do**
    $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
  Choose $B = \{b_1, \ldots, b_l\}$ with max D_i
  $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leq \delta\}$
  /* $*_F$ denotes the number of $*$ bits in F */
  return MaxOI($\mathcal{K}_C^B$)

most methods that efficiently represent IP-based FIBs do not scale well to IPv6 due to its significantly larger 128-bit address width [98, 174, 187, 207]. Distributed systems and network-wide approaches introduce additional complexities in FIB representations.

In this section, we consider the problem of representing FIBs on distributed platforms [79,81], where several line-cards (LCs) are interconnected by a switching fabric (see Fig. 3.6b). Each ingress (RX) LC must separately maintain a FIB table (or parts of one) to forward traffic to the correct egress (TX) LC which transmits it further over an output port. The fundamental question here is how to efficiently implement a FIB table (Fig. 3.6a) across such a distributed switching platform.

There exist two major types of FIB implementations for such distributed platforms: *one-stage* and *two-stage* forwarding. In the former, pointers to output ports are already found on ingress LCs (Fig. 3.7a). Later, an egress LC encapsulates the corresponding traffic with a pointer coming from the ingress LC [79], so every ingress LC should contain information about "routable" prefixes of all egress LCs, whilst egress LCs encapsulate packets based on the pointer to the output port from the ingress LC. A major drawback of this representation is that main-

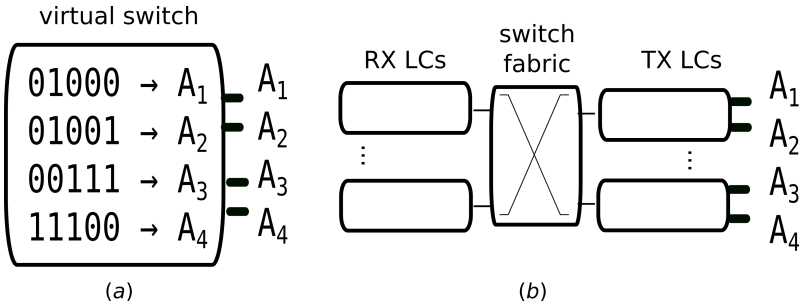


Figure 3.6: How to represent a FIB table (a) on a distributed switching platform (b).

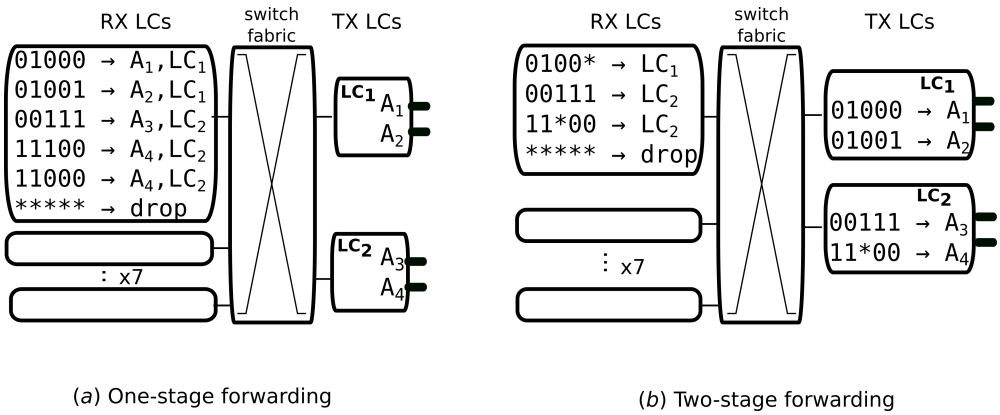


Figure 3.7: One- and two-stage forwarding implemented on existing distributed platforms [80, 81]; lookups are done on all address bits.

taining prefixes mapped to output ports of all egress LCs on every ingress LC is not scalable. While such one-stage forwarding representations do not scale well, they are simple and require only dummy processing on egress LCs (processing pipelines are usually doubled to isolate ingress and egress traffic); e.g., different generations of Cisco C12000 [79] LCs implement one-stage forwarding.

In contrast to one-stage forwarding, where output ports are already computed at ingress LCs, ingress LCs in two-stage forwarding only have to choose which egress LC to send a packet to (see Fig. 3.7b). Now egress LCs have to choose a specific output port by running an additional FIB lookup on the whole destination address for every packet. This allows to mitigate memory constraints on ingress LCs, and each egress LC now contains only prefixes corresponding to its output ports. For a specific router example, different generations of Cisco CRS-1 [79] LCs implement two-stage forwarding.

FIB tables can be represented in different ways. Informally, we say that two representations $\mathcal{K}$ and $\mathcal{K}'$ are *equivalent* if for any input header both $\mathcal{K}$ and $\mathcal{K}'$ yield the same action. Different equivalent FIB representations can optimize

distinct objectives, e.g., minimize required memory or number of entries [148,174].

The one- and two-stage forwarding representations are equivalent to FIB representation on a virtual switch (see Fig. 3.6a) and are currently used in real-life distributed platforms [79, 81]. Note that all bits of addresses are used in both one- and two-stage forwarding which extremely limits efficient implementations, especially on ingress LCs.

The primary contribution of this section is to exploit structural FIB properties to balance bits that are looked up between ingress and egress LCs, minimizing required memory and lookup time. We propose an *abstraction layer* that defines a subset of bit indices that should be involved in the lookup process on ingress and egress LCs respectively. As a result, a classifier based on these bit indices can be transparently represented by other schemes (e.g., [174, 184]). Our abstraction, based on the proposed structural properties, generalizes beyond IPv4, IPv6, or flow-based FIB representations. It does not require changes to network-wide control plane signaling and underlying network infrastructure: FIBs are given and computed by any existing routing protocol. In contrast to compact routing [149], our approach does not affect routing decisions based on requested address space, yet our methods can achieve significant memory savings without increasing lookup time (actually improving it) and are applicable even if the input FIBs have already been optimized. As a byproduct, we show a counter-intuitive result: IPv6 FIBs can be implemented on existing IPv4 or MPLS hardware- or software-based FIB implementations without increasing lookup time or memory requirements or requiring changes in hardware. We do not limit ourselves to prefix-based classifiers with longest-prefix-match (LPM) priorities either and consider the general case of classifiers applicable to clean slate architectures. We also demonstrate how to achieve additional savings when network-wide representations are considered.

We begin with the model description. Again, a packet *header* $H = (h_1, \cdots, h_w)$ is a sequence of w bits from a packet and internal databases of network elements, where each bit h_i of H has a value of zero or one, $1 \leq i \leq w$. We denote by W the ordered set of w *indices* of the bits in headers (i.e., $(1, ..., w)$). A *classifier* $\mathcal{K} = (R_1, ..., R_N)$ is an ordered set of rules, where each rule R_i consists of a *filter* F_i and a pointer to the corresponding *action* A_i. A filter F is an ordered set of w values corresponding to the bits in headers. Possible values are 0, 1, and $*$ ("don't care"). A header H *matches* a filter F if for any bit of H, the corresponding bit of

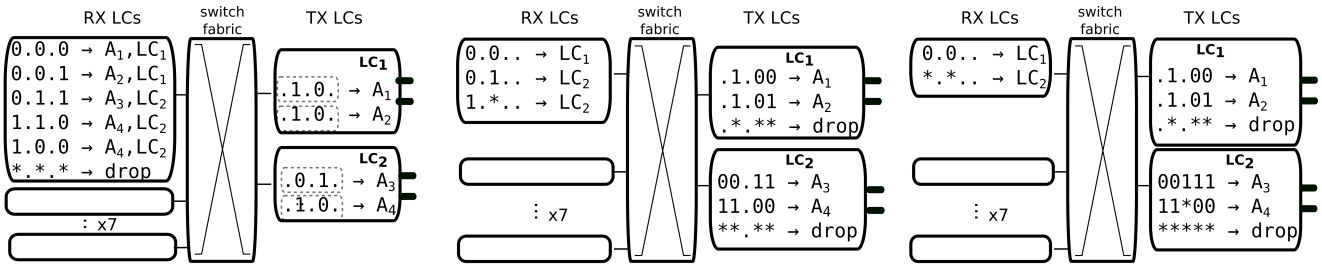
F has the same value or $*$. A header H matches a rule R_i if R_i's filter is matched. The set of rules has non-cyclic ordering $\prec$; if a header matches both R_i and R_j for $R_i \prec R_j$, the action of rule R_i is applied.

We say that two filters F and F' of a classifier *intersect* if there is at least one header that matches both F and F'; otherwise, F and F' are *disjoint*. For instance, for $w = 4$, $F_1 = (1\ 0\ 0\ *)$, $F_2 = (0\ 1\ *\ *)$, and $F_3 = (1\ *\ *\ *)$, F_1 and F_3 intersect (for instance, the header (1 0 0 0) matches both filters), while F_1 and F_2 are disjoint. A classifier is called *complete* iff it matches any possible header. Any incomplete classifier $\mathcal{K}$ can be transformed to a complete one by appending $\mathcal{K}$ with a *default* rule that matches all unmatched headers.

Let B be a set of bit indices, $B \subseteq W$, referring to a subset of the bits in packet headers. For a header H, we denote by H^B the (sub)header of $|B|$ bits obtained by taking only bits of H (in their original order in W) with indices in B. For example for $B = \{1, 3, 5, ..., 2\lfloor\frac{w+1}{2}\rfloor - 1\}$, $H^B = (h_1, h_3, h_5, ..., h_{2\lfloor\frac{w+1}{2}\rfloor-1})$. Likewise, for a rule R or a filter F we denote by R^B and F^B, respectively, the (sub)rule and the (sub)filter defined on $|B|$ bits by removing the bits from $W \setminus B$. Finally, for a classifier $\mathcal{K} = (R_1, ..., R_N)$, let $\mathcal{K}^B$ be the (sub)classifier obtained from $\mathcal{K}$ by replacing each rule R in the classifier by R^B. The notions of matching, intersection, and disjointness carry over to such subsequences. Similarly, for a header H we denote by H^{-B} the header of $w - |B|$ bits obtained from H by considering only bits with indices in $W \setminus B$. Likewise, let R^{-B} and F^{-B} be the rule and the filter, respectively, defined on $w - |B|$ bits by eliminating the requirement on the bits in B for a rule R or its filter F. Let $\mathcal{K}^{-B}$ be the classifier obtained by replacing each rule R in a classifier $\mathcal{K}$ by R^{-B}. We denote the set of all filters in $\mathcal{K}$ based on the ordered set of bit indices $B \subseteq W$ by $\mathcal{F}^B$; $\mathcal{F} = \mathcal{F}^W$. We denote the set of all actions in $\mathcal{K}$ by $\mathcal{A}$. Likewise, we denote by $\mathcal{F}^i$ the set of filters based on bit identities of B whose rules have the same action A_i.

Consider a classifier $\mathcal{K}$ with a classification function $f : \{0,1\}^w \rightarrow \mathcal{A}$. For a set of bit indices $B \subseteq W$, we say that a classifier $\mathcal{K}^{-B}$ with classification function $g : \{0,1\}^{w-|B|} \rightarrow \mathcal{A}$ is an *equivalent* representation of $\mathcal{K}$ if $f(H) = g(H^{-B})$ for every header $H \in \{0,1\}^w$.

Before we introduce representations based on the proposed structural properties formally, we present an illustrative example (Fig. 3.6a) to motivate our contributions. Consider a virtual switch (Fig. 3.6b) representing a distributed



(a) One-stage with RX filter-order independence and false-positive check on TX.

(b) Two-stage with RX action-order independence and TX lookup to find an action.

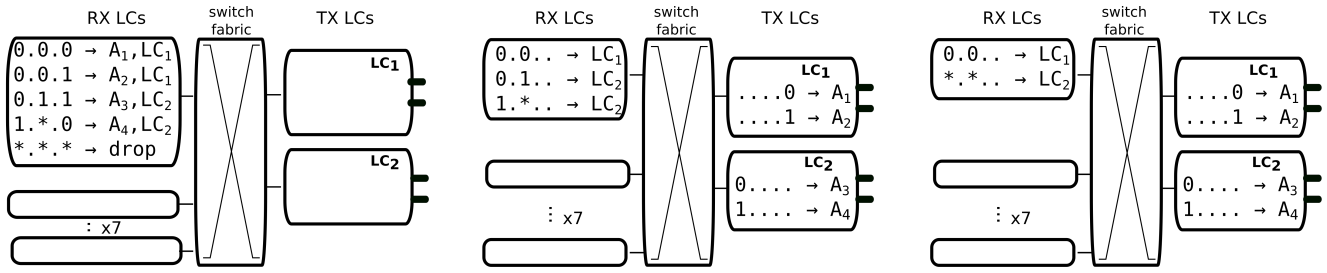
(c) Two-stage with RX non-conflicting rules and TX lookup to find an action.

Figure 3.8: Equivalent FIB representations with reduced address bits.

system. The major drawback of one- and two-stage forwarding approaches is the fact that all bits of a lookup address are represented in FIB tables. In particular, in the case of one-stage forwarding (Fig. 3.7a), every ingress LC requires five 5-bit entries (not including a default entry) without additional FIB table on egress cards (Fig. 3.7a), so the total memory needed for eight slots is $6 \times 5 \times 8 = 240$ bits, and there is a single FIB lookup per packet that is done on ingress LCs.

In the two-stage forwarding case (Fig. 3.7b in our recurring example), actions A_1 and A_2 for output ports are unified to a single "action" $\boldsymbol{LC}_1$, and actions for output ports A_3 and A_4 to $\boldsymbol{LC}_2$ on ingress LCs. This allows to further compress FIB entries on ingress LCs, uniting for instance prefixes 01000 and 01001 with the same action $\boldsymbol{LC}_1$ into a filter 0100∗. Now every ingress LC requires three 5-bit entries (not including a default entry). In addition each egress $\boldsymbol{LC}_1$ and $\boldsymbol{LC}_2$ requires two 5-bit entries. The total memory needed on all ingress and egress LCs in our example is $4 \times 5 \times 8 + 2 \times 5 \times 2 = 180$ bits.

Note that interconnecting switch fabrics should support full line-rate for any combination of ingress and egress LCs, and therefore should not become a bottleneck. Also FIB lookups on egress LCs in the case of two-stage forwarding (or dereferencing of a pointer found on ingress in the case of one-stage forwarding) is usually the first thing to be completed on egress LCs to determine further processing of packets (e.g., the pipeline architecture of packet processing engine in Cisco C12000 router [172]). As long as these invariants are maintained we can move away from equivalent representations on the LC level to equivalent representations in the boundaries of the same distributed system by balancing between evaluating bits of a represented FIB table on ingress LCs versus egress LCs. In the rest of this section, we propose several structural properties of classifiers and show how to exploit them to find more efficient representations of FIBs on distributed



(a) One-stage with RX action-order independence and no false-positive check on TX.

(b) Two-stage with RX action-order independence and TX lookup to distinguish between actions.

(c) Two-stage with RX non-conflicting rules and TX lookup to distinguish between actions.

Figure 3.9: Non-equivalent FIB representations with reduced address bits.

switching platforms.

When we reduce a classifier's width, we often get a classifier that does not preserve the semantics (we do not have *strict* equivalence). However, removing a given bit index from a classifier is in essence equivalent to changing the bit's value to $*$ in all filters, so reducing width cannot decrease the set of headers covered by filters. Hence, for this relaxation we just need to guarantee a correct match of headers that are matched by the original classifier.

We introduce a novel property, *match equivalence*, to improve the efficiency of incomplete classifiers. This property relaxes the classification function by keeping only bit indices that preserve the original classifier $\mathcal{K}$'s action for any header *matched* by $\mathcal{K}$. Formally, an incomplete classifier $\mathcal{K}^B$ is *match equivalent* (ME) to a classifier $\mathcal{K}^W$, $B \subseteq W$, if the action for any matched header H^W in $\mathcal{K}^W$ coincides with the action for the corresponding header H^B in $\mathcal{K}^B$. If H^W has no match in $\mathcal{K}^W$ then H^B can either be matched or remain unmatched in $\mathcal{K}^B$[1].

Problem 3.4.1 (Minimal match equivalence, MINME)**.** *For a given classifier $\mathcal{K}$, find a match equivalent classifier to $\mathcal{K}$ whose filters require minimal memory (in bits).*

This definition of match equivalence does not directly yield algorithms for the MINME problem. Next we propose different flavors of match equivalence based on three structural properties.

First we consider a family of match equivalent classifiers that exploit *filter-order independence*. Classifiers whose filters are all pairwise disjoint (that do not match the same headers) are *filter-order-independent*. Intuitively, representations based on filter-order independence (alone) do not take into account the number

[1]match equivalence can also be applied to complete classifiers when the action of the default entry drops all unmatched traffic and does not implement routing decisions. In this case we can compute a match equivalent classifier on $\mathcal{K}$ without the default entry and later append the default entry back to the result.

of different actions.

In Sections 3.2 and 3.3, we demonstrated the effect of this characteristic on classifiers whose fields are represented by ranges; the impact of filter-order independence on FIB representations was left unaddressed. Filter-order independence guarantees that only a single filter is matched, so to keep this property, match equivalent classifiers on ingress linecards should preserve filter-order independence. In addition, to build an *equivalent* solution we need to check on egress LCs if the remaining bits (that did not participate in the classification process on ingress) of the packet still match an admissible packet (see Fig. 3.8a). If yes, the forwarded packet is sent out through the corresponding output port, otherwise it is dropped, similarly as it is done by representations in Fig 3.7. Observe that in this case there is no lookup table on egress LCs and the remaining bits are fetched from the pointer found on ingress, so filter-order independence has a good fit for one-stage forwarding. Every packet matching the ingress FIB (except the default entry) in Fig. 3.7a (that implements one-stage forwarding) applies the same action as in the compressed ingress FIB in Fig. 3.8a. But now packets previously matched by the default entry in the ingress FIB in Fig. 3.7a may be matched by the compressed ingress FIB in Fig. 3.8a and are dropped only on egress LCs. Still this representation is equivalent in the system boundaries and allows to balance lookup time and memory requirements between ingress and egress LCs. In the running example the prefixes of a given FIB are filter-order independent and the FIB on every ingress LC can be reduced to only five 3-bit entries (see Fig. 3.8a) (not including a default entry), with a total memory requirement for this representation being $6 \times 3 \times 8 + 2 \times 2 \times 2 = 152$ bits. As a significant side effect, this representation allows to represent IPv6 on already existing IP or even MPLS ingress LCs. We demonstrate the feasibility of this representation in the evaluation study for real IPv6 FIBs (see the works [10, 24]).

Note that filter-order independence can be too restrictive; it is unnecessary to require this property from rules with the same action.

One major drawback of filter-order independence is that it can reduce only a number of bit identities that are involved in FIB representations on ingress LCs but cannot reduce the number of entries in these tables on ingress. To address this limitation, we introduce a family of match equivalent classifiers that exploit *action-order independence*. This characteristic is based on the observation that a

classification result really is the *action* of the rule with the highest priority, rather than the filter associated with this rule. Two rules with filters F_1^B and F_2^B are *action-order-independent* if either they have the same action or F_1^B and F_2^B are disjoint; otherwise, we say that two rules are *action-order-dependent*. Similarly, a classifier $\mathcal{K}$ as a whole is called action-order-independent if every pair of its rules are action-order-independent; otherwise $\mathcal{K}$ is action-order-dependent. Conversely, a classifier $\mathcal{K}'$ that implements action-order independence on a subset $B \subset W$ of bit indices of an originally action-order-independent classifier $\mathcal{K}$ matches the same headers as $\mathcal{K}$, and as a result $\mathcal{K}'$ is match equivalent to $\mathcal{K}$ with respect to B.

Action-order independence allows to further reduce the number of entries for filters with the same action. In this case we can reduce both the number of entries and the number of bit identities participating in the FIB lookup on ingress. The price for this is a slightly more complex "false positive" check and thus more complex processing on egress LCs, that in contrast to filter-order independence requires a FIB lookup table based on a subset of bit identities (see Figure 3.8b). Note that the two-stage forwarding approach with action-order independence leads to an additional reduction in the number of entries on ingress LCs since the number of actions is reduced to the number of different LCs. In our example, the memory requirements for this representation type is $3{\times}2{\times}8{+}3{\times}3{+}3{\times}4 = 69$ bits (note that in this specific example no default entry is required on ingress LCs since all combinations of values are already covered). Since this representation is more general than the previous one it supports an even better balance of memory and lookup times between ingress and egress LCs. In practice (see the evaluation in [10, 24]), it suffices to achieve action-order independence on a subset of bit indices to find much more space-efficient representations of classifiers than representations that exploit filter-order independence.

While exploiting action-order independence to implement match equivalence has a huge potential for memory savings, it becomes apparent that we need to better understand the applicability and expression of match equivalence. To that end we suggest to express the third type of match equivalence through the notion of *conflicting* rules. Two rules R_1^X and R_2^X with different actions, $R_1^X \prec R_2^X$, are *conflicting* with respect to bit indices $B \subset X \subseteq W$ if there is a header H^X matching R_2^X that is not matched by R_1^X and R_1^B matches H^B.

Example 3.4.1. In the following classifier with $|X| = |W| = 3$

$$\begin{array}{cccccc} & \#1 & \#2 & \#3 & & \\ R_1^X & (* & 1 & 0) & \rightarrow & A_1 \\ R_2^X & (1 & 1 & *) & \rightarrow & A_2 \end{array}$$

we have that R_1^X and R_2^X are conflicting with respect to $B = \{1, 2\}$ since R_1^B matches $H^B = (1\ 1)$ but R_1^X does not match $H^X = (1\ 1\ 1)$ when R_1^B matches H^B and yet it matches R_2^X. On the other hand, for $B = \{2, 3\}$, the rules R_1^X and R_2^X are not conflicting with respect to B.

A classifier $\mathcal{K}^B$ is match equivalent to a classifier $\mathcal{K}^X$ with respect to B, $B \subset X$, if no two rules $R_1^X \prec R_2^X$ in $\mathcal{K}^X$ with different actions are conflicting with respect to B.

For our running example the FIB representation based on non-conflicting rules on each ingress LC requires two 2-bit entries (see Figure 3.8c) with the total memory requirement of $2 \times 2 \times 8 + 3 \times 3 + 3 \times 5 = 56$ bits.

Representations based on three types of match equivalence are applicable to FIBs with multi-field rules with general priorities such as in OpenFlow. Complexity bounds derived from computational geometry imply that a software-based packet classifier with N rules and $k \geq 3$ fields uses either $O(N^k)$ space and $O(\log N)$ time or $O(N)$ space and $O(\log^{k-1} N)$ time [111, 171]. So in the general case, new fields that do not have exact values in all rules can lead to significant overhead in either memory requirements or lookup time.

Consequences of all three types of match equivalence that we consider are counter-intuitive. For equivalent representations on the LC level, more bit identities in given classifiers with a fixed number of filters implies increased memory footprint (e.g., IPv6 or OpenFlow versus IP). This is not the case for match equivalence. Moreover, match equivalent FIB tables with A actions based on action-order independence and non-conflicting rules has $A \log A$ minimal memory requirements, since we need at least one filter for every action, and $\log A$ bits are required to distinguish among A actions. With match equivalence, IPv6 and OpenFlow have better chances than IP to find bit indices to distinguish between actions. This not only significantly reduces memory footprint versus optimal equivalent representations on the LC level, but also allows to implement ingress IPv6-based FIBs or other clean-slate solutions (e.g., OpenFlow) on existing IPv4 or even MPLS in-

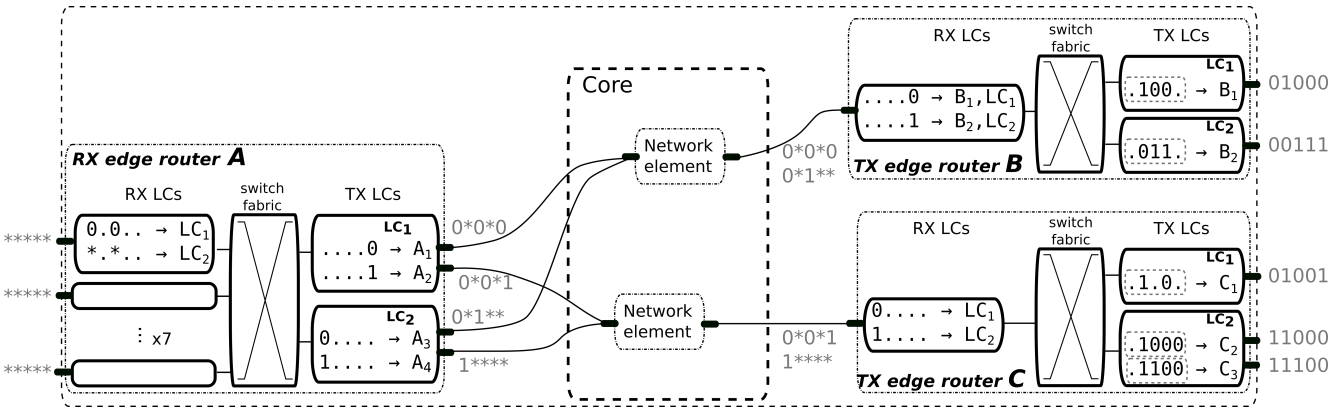


Figure 3.10: A sample network-wide FIB representation.

frastructures with the same memory requirements and lookup time, transparently to the hardware. Fig. 3.8 summarizes all proposed representations which are at the system level equivalent to the one- and two-stage forwarding representations in Fig. 3.7 and introduce substantial savings on ingress and egress LCs.

So far all proposed representations operate on a given (possibly already optimized) FIB and implement equivalence within the confines of a single network element. As a result, as opposed to compact routing, these schemes do not require changes in control plane signaling and do not affect the "quality" of implemented routing decisions. In this part we generalize previously proposed FIB representations and discuss equivalence with network-wide resolution; this can yield further savings in memory footprint but will require changes in control plane signaling and possibly of the network-wide view. For instance, this approach can fit well for software-defined networking with centralized management.

We return to our original question of how to represent the FIB table of a virtual switch from Fig. 3.6a on a distributed system in Fig. 3.6b, but now the distributed system is also virtual and can represent the whole network. Various prior works represent the whole network as a single virtual switch (e.g., [49, 121]). Clearly, equivalent representations at the LCs or system levels are applicable during mapping of this virtual switch to physical network elements.

But we can go further and decide to skip false positive tests on egress LCs of the edge router *A*, ideally on egress LCs of relevant edge routers *B* and *C* facing core interfaces (see Fig 3.10; the core network in Fig 3.10 can run IP/IPv6 or MPLS). Figure 3.9 summarizes non-equivalent solutions that allow to achieve further memory savings. Amazingly in the case of IPv6, we can exploit these non-equivalent representations to decrease the size of FIB tables on ingress and egress

LCs of router A, and for instance, representations based on one-stage forwarding with filter-order independence on ingress of the routers B and C. Note that in this case egress LCs of these routers do not need to run FIB lookups just a false positive check based on the "remaining bits" of IPv6 address based on the found on ingress pointer.

A combination of these two representations allows to avoid complex 128-bit representations on routers A, B, and C. For instance, the engine 3 LC family of Cisco C12000 [79] implement IP FIB as a three level 16-8-8 bit-trie, where every level requires exponential memory. Usually these schemes can clearly not be extended for IPv6. As a result engine 3 LCs used TCAM for IPv6 that was sharable with other services such as QoS, ACL, etc. So IPv6 scalability was very limited for this LC family. The following engine 5 LC family already had four TCAMs with 10^6 of 72-bit entries on every ingress and egress LC to support the desired level of IPv6 scalability, significantly increasing cost and power requirements. Network-wide FIB representations can significantly reduce these resource requirements of representing edge routers, and allow to deploy IPv6 on existing IP and MPLS infrastructure. These network-wide representations can be considered as equivalent in the boundaries of the whole network since in the worst case some packets dropped in router A with equivalent solutions from Fig. 3.7 and Fig. 3.8 will now be dropped on egress LCs of routers B and C. In contrast to equivalent representations on the system level, where switching fabric is not a bottleneck, here we have a fundamental tradeoff between memory requirements on edge routers and potentially "wasted" core bandwidth for traffic that is now dropped only at the egress of edge routers B and C, but these alternatives are still worth considering. All proposed methods that exploit proposed structural properties are valid also for general classifiers as in OpenFlow.

Further in this section we propose algorithms that compute optimized match equivalent classifiers applicable to all three considered structural properties: filter-order independence, action-order independence, and non-conflicting rules.

Removing redundant entries and bit indices further reduces the memory footprint bottleneck on ingress and egress linecards. Instead of proprietary algorithms, we suggest to use heuristics based on Boolean minimization techniques for ordered AC^0_3 circuits [50] in all considered FIB representations (incl. traditional one- and two-stage forwarding). We denote this building block by SE (*strict* equivalence);

SE computes an equivalent representation in minimal memory. Unfortunately, some of the more sophisticated SE algorithms have complexity $O(N^3 \cdot w)$, where N is the number of rules and w is their width. We proposed a possible SE implementation (OptimizeClassifier) in Section 3.2. Hence, in our evaluations we used a simplified version of SE that exploits only a part of the SE heuristics and runs in $O(N^2 \cdot w)$ time; in preliminary evaluation on hundreds of FIBs we did not find any significant difference in memory reduction between simplified and full versions of SE for all considered representations. Here we use SE in two major cases:

(1) to remove redundant filters and bit indices;

(2) to check potential memory savings while removing a specific bit index.

We begin with Algorithm 10 (which uses Algorithm 9) that tests whether $\mathcal{F}^B$ is match equivalent (IsME) to a given $\mathcal{F}^X$, where B a subset of bit indices X. We say that two rules *break a structural property* if they are filter-order-dependent, action-order-dependent, or conflicting. In this way all proposed algorithms can be used for all structural properties. Note that actually, Algorithm 10 tests if all different pairs of rules do not break a structural property on a specific subset of bit indices. Algorithm 10 is implemented through Algorithm 9 to increase reuse with other algorithms in this section. In Algorithm 10, we have denoted

$$\textsc{Succ}(\mathcal{F}, F) = \{F' \in \mathcal{F} \mid F \prec F'\},$$

i.e., the set of all rules in $\mathcal{F}$ after F. All algorithms proposed below will check if removing a certain bit still retains match equivalence, and will differ in how they choose which bit to remove.

Algorithm 11 considers bit indices that can be removed while preserving match equivalence. At each step, it removes one bit index from all filters; the bit index is selected to maximize the memory savings obtained by the reduction (Lines 6-7). The algorithm continues as long as there is a bit index that can be removed while preserving match equivalence. In Line 9, SE removes rules that collapse after reducing a subset of bit indices.

Algorithm 9: FINDBREAKERS(F,$\mathcal{F}$,B)

```
Data: F,F,B
1 begin
2     A ← ∅
3     for every F′ ∈ F starting from F do
          /* two filters that break a struct. property can be
             filter- or action-order-dependent, or conflicting
             rules                                              */
4         if F and F′ break a structural property with respect to B then
5             A ← A ∪ {F′}
6 return A
```

Algorithm 10: ISME($\mathcal{F}^X$,B)

```
Data: F^X,B
1 begin
2     for every F_1^X ∈ F^X do
3         if FINDBREAKERS(F_1^X, SUCC(F^X, F_1^X), B) ≠ ∅ then
4             return False
5     return True
```

Example 3.4.2. Consider the following classifier:

#1	#2	#3	#4	
(0	1	0	0)	$\rightarrow A_1$
(0	1	1	1)	$\rightarrow A_2$
(1	1	1	0)	$\rightarrow A_2$

The proposed heuristic MINME will consider three bits that can be potentially removed in the first step: #1, #2, and #4. Among these bits, removing #2 does not lead to further memory reductions, while removing, say, #1 leads to $F_2^{-\{1\}} = (1\ 1\ 1)$ and $F_3^{-\{1\}} = (1\ 1\ 0)$, which can now be united by resolution to $F' = (1\ 1\ *)$; similarly, removing #4 leads to $F' = (*\ 1\ 1)$ on bits 1-3. Therefore, the heuristic chooses either #1 or #4, say #1, leading to the following match

Algorithm 11: MINME ($\mathcal{K}^X$)

```
   Data: K^X
 1 begin
 2     B ← X, I^B ← K^X
 3     repeat
 4         B' ← B
 5         for each j ∈ B do
 6             if IsME(I^{B\{j}}, B \ {j}) then
                   /* remove j-th bit index whose removal lead to
                      the minimal size reduction                    */
 7                 j* ← arg min_j SE(I^{B\{j}})
 8                 B ← B \ {j*}
 9     until B = B'
       /* reduce collapsing rules after removed bit indices        */
10     I^B ← SE(I^B); return B, I^B;
```

equivalent classifier:

#2	#3	#4	
(1	0	0)	$\rightarrow A_1$
(1	1	$*$)	$\rightarrow A_2$

It is impossible to further reduce the set of rules (all rules have different actions now), so the proposed heuristic will not have any preference in subsequently removing bits #2 and #4; after removing them, we get the final representation that is based only on bit #3.

Theorem 3.4.1. *For a classifier with N rules and headers of w bits, the time complexity of Algorithm 11 is $O(N^2 \cdot w^3) + O(w^2) \cdot T(\mathrm{SE})$, where $T(\mathrm{SE})$ is the time complexity of the* SE *procedure.*

We can further improve efficiency of representations and also deal with match equivalent classifiers whose classification width is irreducible while preserving one of the structural properties. In particular, we propose to find a match equivalent classifier on a *subset* of rules $\mathcal{I}$ of a given classifier $\mathcal{K}$ and moving "conflicting" rules of $\mathcal{I}$ to $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$, so that $\mathcal{I}^B$ is match equivalent to $\mathcal{I}$ for some subset of bit indices $B \subseteq W$, and $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$ contains all other rules of the original classifier. The header of an incoming packet is matched in both groups of rules and the rule with the highest priority is chosen.

Example 3.4.3. In the following classifier

#1	#2	#3	#4	#5	#6	
(0	1	1	0	0	0)	$\rightarrow A_1$
(0	0	0	1	0	0)	$\rightarrow A_2$
(1	0	0	0	1	0)	$\rightarrow A_3$
(1	1	0	0	0	1)	$\rightarrow A_4$
(∗	∗	0	0	0	0)	$\rightarrow A_5$

R_5 does not let us remove bits 3-6 because it conflicts with rules R_1-R_4 with respect to these bits. However, once we set $\mathcal{D} = \{R_5\}$, the remaining $\mathcal{I}^{\{1,2\}}$ becomes match equivalent to the following classifier on fields $\{1, 2\}$: (0 1) $\rightarrow A_1$, (0 0) $\rightarrow A_2$, (1 0) $\rightarrow A_3$, (1 1) $\rightarrow A_4$.

Note that $\mathcal{D}$ is maintained in a traditional way, which can be expensive. Varying B we can find a feasible representation of $\mathcal{I}^B$ and reduce $\mathcal{D}$; applying an additional constraint on the maximal value of $|B|$ can reduce the size of $\mathcal{I}^B$. This additional constraint can allow to represent IPv6 on existing IP infrastructure. Splitting of an original classifier into $\mathcal{I}$ and $\mathcal{D}$ proves to be very fruitful for size reduction.

Problem 3.4.2 (MINME)**.** *Given a classifier $\mathcal{K}$ on a set of bits W and a positive number $l \leq w$, find a subset of bits B, $B \subseteq W$, $|B| \leq l$, and a subset of rules $\mathcal{I} \subseteq \mathcal{K}$ such that $\mathcal{I}^B \cup \mathcal{D}$ for $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$ is match equivalent to $\mathcal{K}$ and minimizes the value of $|B| \cdot |\mathcal{I}| + w \cdot |\mathcal{D}|$.*

Algorithm 12 removes at least $w - l$ bits by greedily minimizing $|\mathcal{D}|$, which is equivalent to minimizing the value of $|B| \cdot |\mathcal{I}^B| + w \cdot |\mathcal{D}|$.

Theorem 3.4.2. *For a classifier with N rules and headers of w bits, the time complexity of Algorithm 12 is $O(N^2 \cdot w^3) + O(w^2) \cdot T(\text{SE})$, where $T(\text{SE})$ is the time complexity of the* SE *procedure.*

We can further generalize the idea of implementing match equivalence on a subset of rules to improve efficiency of FIB representations. Actually we can assign all rules into multiple disjoint groups, where every group obeys a structural property on a subset of bit indices. The number of such groups can be defined by the number β of parallel and serial lookups that can be performed at linerate, which is a property of a network element. Hence, we can further generalize Problem 3.4.2.

Algorithm 12: MINME $(\mathcal{K}^X, l)$

Data: $\mathcal{K}^X, l$

begin

  $\mathcal{D}_0 \leftarrow \emptyset$, $B \leftarrow X$, $\mathcal{K} \leftarrow \mathcal{K}^X$, $\mathcal{I} \leftarrow \mathcal{K}$, $w \leftarrow |B|$

  for $i = 1..w$ **do**

    for *each* $j \in B$ **do**

      /* find the set $D_{i,j}$ of rules conflicting w.r.t. all bits except j */

      $D_{i,j} \leftarrow \emptyset$

      for *each* $R \in \mathcal{I}$ **do**

        $D_{i,j} \leftarrow D_{i,j} \cup \text{FINDBREAKERS}(R, \text{SUCC}(\mathcal{K}, R), B \setminus \{j\})$

    /* choose the best j^*, with the smallest number of conflicting rules */

    $j^* \leftarrow \arg\min_j |D_{i,j}|$

    /* add D_{ij} to $\mathcal{D}$, update $\mathcal{I}_i$, B, $\mathcal{I}$, and $\mathcal{K}$ */

    /* optimize the current $\mathcal{D}_i$ */

    $\mathcal{D}_i \leftarrow SE(\mathcal{D}_{i-1} \cup D_{i,j^*})$

    /* optimize the current $\mathcal{I}_i$ */

    $\mathcal{I}_i \leftarrow SE(\mathcal{K} \setminus \mathcal{D}_i)$

    $B \leftarrow B \setminus \{j\}$, $\mathcal{I} \leftarrow \mathcal{I}^B$, $\mathcal{K} \leftarrow \mathcal{K}^B$

  $i^* \leftarrow \arg\min_{i \geq w-l} (w|\mathcal{D}_i| + (w-i)|\mathcal{I}_i|)$

  return i^*, $\mathcal{D}_{i^*}$, $\mathcal{I}_{i^*}$

Algorithm 13: PARMINME $(\mathcal{K}, l, \beta)$

Data: $\mathcal{K}, l, \beta$

begin

  $\mathcal{D} = \mathcal{K}$, $\mathcal{I}_0 = \emptyset$, $k = 0$

  while $\cup_{s=0}^{k} \mathcal{I}_s \neq \mathcal{K}$ *and* $k \leq \beta$ **do**

    $k = k + 1$

    $i_k, \mathcal{D}, \mathcal{I}_k = \text{MINME}(\mathcal{D}, l)$

  return k, $\{(i_s, \mathcal{I}_s)\}_{s=1}^{k}$, $\mathcal{D}$

Problem 3.4.3 (PARMINME)**.** *Given a classifier $\mathcal{K}$ on bit indices W and two positive numbers $l \leq w$ and β, assign the rules of $\mathcal{K}$ to β disjoint groups, where a group $\mathcal{I}_i \subseteq \mathcal{K}$ preserves a structural property on $B_i \subseteq W$ bit indices, $|B_i| \leq l$, and all remaining rules are assigned to $\mathcal{D} = \mathcal{K} \setminus \cup_{i=1}^{\beta} \mathcal{I}_i$, such that $\sum_{i=1}^{\beta} |B_i| \cdot |\mathcal{I}_i| + w \cdot |\mathcal{D}|$ is minimized.*

Algorithm 13 shows the basic idea of greedy multi-group optimization with

match equivalence; in a nutshell it iteratively calls to Algorithm 12 to build a group without backtracking (while there are unused groups) on the currently still unassigned rules.

Theorem 3.4.3. *For a classifier with N rules and w-bit headers, the time complexity of* PARMINME *is $O\left(\beta N^2 \cdot w^3\right) + O(\beta w^2) \cdot T(\text{SE})$, where $T(\text{SE})$ is the time complexity of the* SE *procedure.*

Note that in practical implementations we can mitigate time complexity of the proposed algorithm in two different ways. First, we can operate on blocks of bits. Second, if the classification width is relatively large (e.g., IPv6), we can just drop bits that still preserve structural properties, not necessarily looking for maximal memory savings every time. We applied these shortcuts in our evaluation study, and did not find significant differences between these simplified versions and the full algorithms proposed in this section.

We have extensively compared these algorithms with prior state of the art and between each other with a comprehensive experimental evaluation study. Detailed results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [10, 24].

3.5 Classifiers for policies reusing the same classes

In this section, we exploit other alternatives to achieve additional efficiency of policy state represented on the data plane. Our ideas hinge on the fact that similar "classification patterns" (*classes*) are reused in different policies, where each class consists of ternary-bit filters determining a set of matched packet headers. Various vendors already support the notion of classes in policy declarations [82,173] allowing to abstract and manage classification patterns more efficiently. For instance, Cisco IOS supports up to 256 different QoS policies and up to 4096 classes per box [86]. In real deployments, the number of classes per policy ranges from tens to hundreds depending on the application model [86]. The size of a class depends on the complexity of represented patterns.

Traditionally, a separate class instance is allocated for each policy instance that contains it (see policies P_1, P_2, and P_3 in Fig. 3.11a). Each allocated class instance has an attached action specified by the corresponding policy (in Fig. 3.11a, an action $A_{i,j}$ is attached to the instance of c_j in the policy P_i). Since classes are

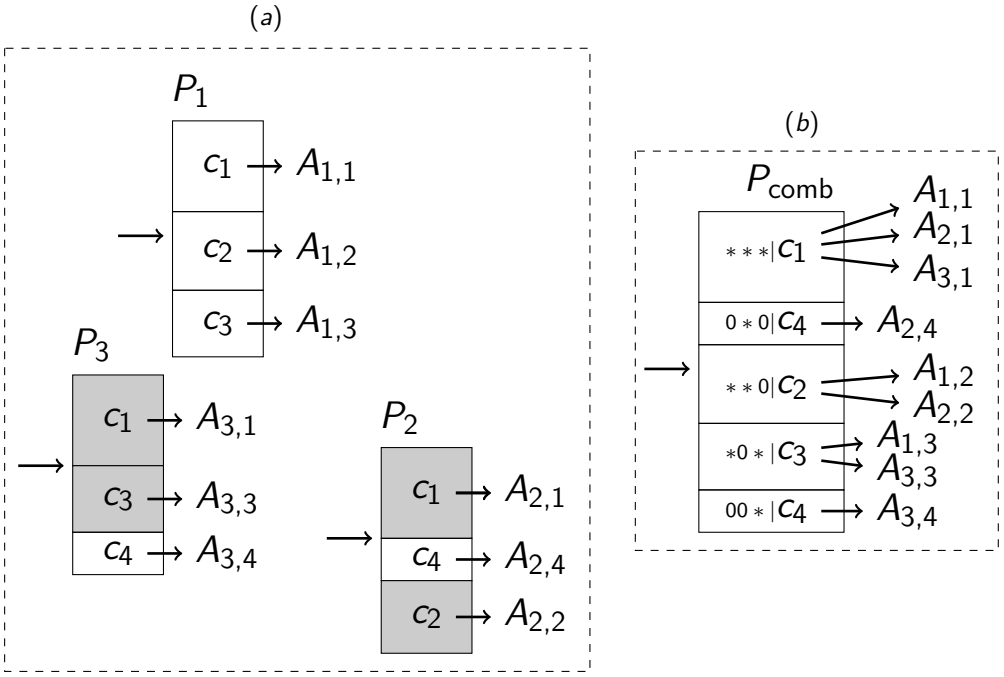


Figure 3.11: Sample representations: (a) separate policies P_1, P_2, P_3; (b) a representation P_{comb} that emulates the policies; class instances that have been cut in P_{comb} are shown in gray.

used in different policies, it allows us to look at combined service policy representations, where ideally each class appears only once, providing substantial savings in representations of underlying classifiers in expensive memory such as TCAM. Usually, the complexity of structural properties of classifiers can be alleviated with additional classification lookups, but this is a shareable resource for the overall processing. The number of classification lookups per packet is one of the major constraints limiting line-rate characteristics. For instance, Cisco C12000 [153] supports at most six TCAM lookups per packet at line-rate for all services. As a result, in this section we consider combined policy representations that do not increase the number of classification lookups. Informally, proposed combined policy representations "emulate" the behaviours of represented policies.

Classes represent an intermediate level of abstraction: a *class* c is a set of filters. We denote by $w(c)$ the number of filters in c. A header H *matches* a class c if H matches at least one filter in c. Two classes (sets of filters) c and c' are *disjoint*, denoted by $c \perp c'$, if there are no headers matching both c and c' (all filters of c and c' are pairwise disjoint), otherwise, they *intersect*. To define a *policy* P over a given set of classes C, one needs to select a set of classes $C_P \subseteq C$ belonging to P, specify a sequence $\mathbb{S}(P)$ containing each class from C_P only once,

and associate an action with each class in C_P. For an incoming header, the action of the first matched class in $\mathbb{S}(P)$ is returned. If an incoming header is not matched by any class in $\mathbb{S}(P)$ then the policy *default* action is returned. Since classes can intersect, a policy is defined by a sequence rather than a set. Originally, classes were introduced to define common classification patterns [82, 173] that can significantly simplify policy management. In this way a single classification pattern should not be redefined during the declaration of another policy.

Two policies P_1 and P_2 are *equivalent* if for every given header both yield the same action. Note that different sequences on the same set of classes C_P can lead to several equivalent policies due to possible pairwise disjointness of classes in C_P. For instance, Figure 3.12a shows a policy P defined by the sequence $\mathbb{S}(P) = c_3, c_2, c_4, c_1$, which is equivalent to P' defined by $\mathbb{S}(P') = c_3, c_2, c_1, c_4$ since c_1 and c_4 are disjoint.

To define policies that are equivalent to P on C_P, we introduce the following partial order $\prec_P$ of classes in C_P. We say that $c_i \prec_P c_j$ if at least one of the following conditions is satisfied:

- c_i intersects with c_j, c_i appears before c_j in $\mathbb{S}(P)$, and the actions assigned to c_i and c_j in P are different (disjointness constraint);
- there is a class $c_k \in P$ such that $c_i \prec_P c_k$ and $c_k \prec_P c_j$ (transitivity of the partial order).

For instance, Figure 3.12a defines a policy P whose corresponding partial order $\prec_P$ is illustrated on Fig. 3.12b. In all illustrations of partial orders we omit arrows showing that $c_i \prec_P c_j$ if c_i precedes c_j in $\prec_P$ that follow from transitivity.

Note that a policy P' is equivalent to a policy P if $\prec_{P'}$ coincide with $\prec_P$ and each class in P' has the same action as in P (the default actions in P and P' should coincide).

Informally, if we represent a policy P as a graph G with vertices corresponding to classes of P and edges corresponding to partial order constraints of $\prec_P$, then any topological order on the vertices of G forms a sequence of classes in the policy that is equivalent to P.

We denote by $\mathcal{P} = \{P_1, P_2, \ldots P_{|\mathcal{P}|}\}$ a set of $|\mathcal{P}|$ policies over the same set of classes C; by $|C|$, the number of classes in C. Note that for an incoming header H, the corresponding policy P is retrieved from internal switch data structures,

a

Class	Filter	#1	#2	#3	#4	Action
c_3	F_1	0	1	*	*	A_1
	F_2	*	1	*	0	A_1
c_2	F_3	*	*	0	0	A_2
c_1	F_4	1	0	*	*	A_3
c_4	F_5	0	0	*	*	A_4

b

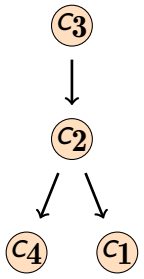


Figure 3.12: (a) definition of the policy $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) partial order $\prec_P$: $c_3 \perp c_1, c_3 \perp c_4$, and $c_1 \perp c_4$.

where the classification of H should be performed; in this case, we say that H is coming in the *context* of the policy P. A *combined policy* P_{comb} representing a group of policies $\mathcal{P}' \subseteq \mathcal{P}$ *emulates* $\mathcal{P}'$ if for any header H and any policy $P_i \in \mathcal{P}'$, the lookup of H in P_i and the lookup of H in P_{comb} in the context of P_i yield the same action. Informally, this means that P_{comb} mimics the behaviour of any policy in $\mathcal{P}'$.

In general, a *representation* $\mathcal{P}_{comb}$ implementing the classification in $\mathcal{P}$ can consist of more than one combined policy. Due to the single lookup constraint, a policy in $\mathcal{P}$ should be represented only by a single combined policy in $\mathcal{P}_{comb}$. Therefore, in order to construct $\mathcal{P}_{comb}$, the policies in $\mathcal{P}$ should be assigned into multiple disjoint groups $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$, where each group $\mathcal{P}^i$ is represented by a separate combined policy P^i_{comb}. By m we denote the number of groups. For every header incoming in the context of a policy $P \in \mathcal{P}$, the lookup is done in P^i_{comb} corresponding to the group $\mathcal{P}^i$ containing P. We say that $\mathcal{P}_{comb}$ *emulates* $\mathcal{P}$ if each $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ emulates the corresponding group of policies $\mathcal{P}^i$.

We introduce the notion of a *joint graph* G^{jnt} for a set of policies $\mathcal{P}$ over classes C; this is a directed graph $G^{\text{jnt}}(\mathcal{P}) = (C, E^{\text{jnt}})$, where E^{jnt} contains an edge from c_i to c_j for $c_i, c_j \in C$ if and only if $c_i \prec c_j$ in at least one policy in $\mathcal{P}$ (see Fig. 3.13b for an example).

For a given set of policies $\mathcal{P}$, there exists an ideal P_{comb} if the corresponding G^{jnt} is acyclic (this is proven in [3, 19] but is not part of the present dissertation). This implies an algorithm that constructs an ideal representation if it exists. The time complexity of this algorithm equals to the time $T_{G^{\text{jnt}}}(\mathcal{P}) = O(\sum_{P \in \mathcal{P}} (|P|^2 + |P| \cdot D(P)))$ needed to construct G^{jnt}, where $D(P)$ is the number of intersecting class pairs from P that have different attached ac-

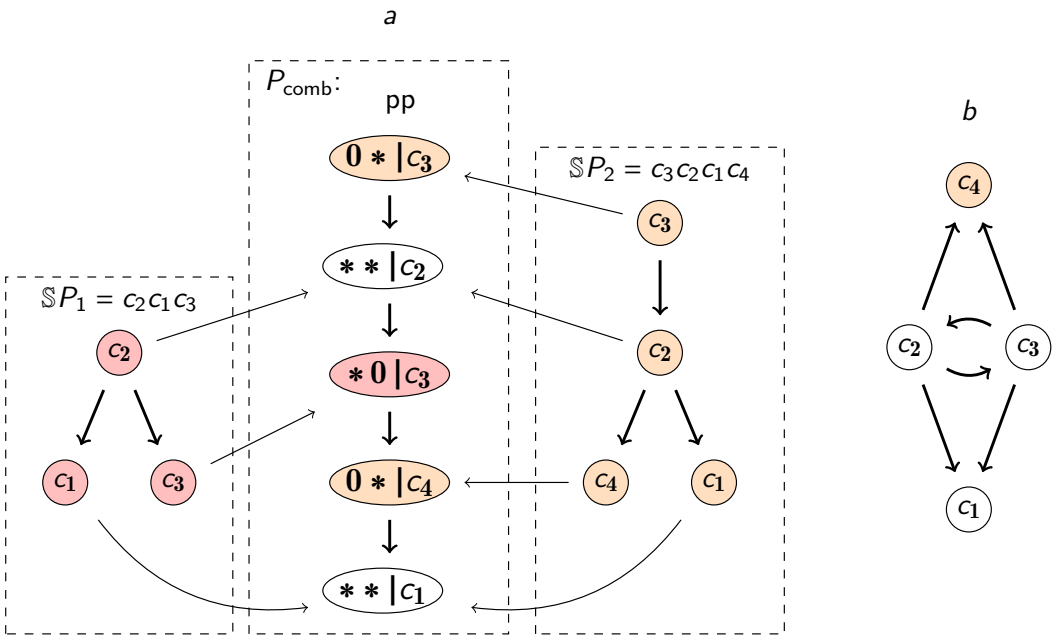


Figure 3.13: Sample representations: (a) $\mathcal{P} = \{P_1, P_2\}$ and P_{comb} with duplicated c_3; (b) $G^{\text{jnt}}(\mathcal{P})$.

tions.

In case when an ideal representation cannot be constructed, to deal with incompatible partial orders in policies we duplicate some instances of classes. Formally, a sequence $\mathbb{S}$ defining the order of class instances in P_{comb} is *compatible* with a policy P_i if there exists a subsequence $\mathbb{S}'$ of $\mathbb{S}$ that consists of a single instance of every class in P_i and for any two classes c_j, c_t in P_i, if $c_j \prec_{P_i} c_t$ then c_j appears before c_t in $\mathbb{S}'$. Only instances of classes from this subsequence participate in the classification by policy P_i, i.e., in the corresponding P_{comb} only for them the ith bit of the policy prefix is set to $*$, while for all other instances the ith bit of the policy prefix is set to zero. Header prefixes are exactly the same as in the case of ideal policies.

Figure 3.13a illustrates two policies P_1 and P_2. Since $c_2 \prec_{P_1} c_3$ and $c_3 \prec_{P_2} c_2$, there is no ideal P_{comb} satisfying partial orders of classes in both P_1 and P_2. Figure 3.13b shows the corresponding joint graph, and it indeed contains a cycle. To satisfy the partial orders of P_1 and P_2 at the same time, we can add an additional instance of c_3 to P_{comb} with the corresponding bits of the policy prefix. In particular, the sequence of classes in P_{comb} shown on Fig. 3.13a is $\mathbb{S} = c_3c_2c_3c_4c_1$; its subsequence $c_2c_3c_1$ is compatible with partial order $\prec_{P_1}$, and another subsequence $c_3c_2c_4c_1$ is compatible with partial order $\prec_{P_2}$. Now P_{comb} contains two instances of c_3: the first is used during classification in P_2, and its policy prefix is $0*$; the

second instance is used during classification in P_1, and its policy prefix is $*0$. In this case P_{comb} is non-ideal but still emulates P_1 and P_2.

Clearly, the number of filters in classes should be taken into account during class duplications. We denote by $\mathrm{W}^{+}(\mathbb{S})$ the total overhead in filters from duplicated class instances in the resulting sequence of classes $\mathbb{S}$, i.e., the difference between the total number of filters in all class instances from $\mathbb{S}$ and the total number of filters in original classes without duplications.

Problem 3.5.1 (Policy Sequence Packing, PSP)**.** *Given a set of policies $\mathcal{P}$, find a sequence of classes $\mathbb{S}$ compatible with all policies in $\mathcal{P}$ that minimizes* $\mathrm{W}^{+}(\mathbb{S})$.

Theorem 3.5.1. *PSP is NP-hard even for two policies, $|\mathcal{P}| = 2$.*

We denote by $|\mathbb{S}|$ a number of class instances in $\mathbb{S}$. If the partial orders of all policies in $\mathcal{P}$ are linear, the PSP problem is a weighted version of the classical Shortest Common Supersequence (SCS) problem [106]. For a set of strings, SCS finds a string with a minimal total length containing all these strings as subsequences. The algorithm from [106] finds an optimal solution for the weighted version of SCS in time $O(|C|^{|\mathcal{P}|})$; therefore, this algorithm can find an optimal solution of the PSP problem but only if the number of policies is relatively small and the partial orders of all policies in $\mathcal{P}$ are linear.

Our algorithms for PSP will use algorithms for the *Weighted Feedback Vertex Set* (WFVS) problem [108], which is NP-complete. The *feedback vertex set* is a set of vertices in a directed graph $G = (V, E)$ with weighted vertices such that removing them forms an acyclic graph, and the WFVS problem is to find a feedback vertex set of minimal total weight. For instance, the work [104] proposes an algorithm for WFVS with approximation factor $O(\log |V| \log \log |V|)$, but there are other alternatives [93]. In what follows we denote by $\alpha(G)$ the approximation factor of an algorithm for the *WFVS* problem on a graph G. Note that WFVS is not harder than PSP (see the proof of Theorem 3.5.1). This is why the algorithms proposed below exploit WFVS as a building block.

The main reason for class duplications are cycles in the joint graph. The algorithm AllOrOne (AO) constructs G^{jnt} and transforms it into an acyclic graph G^{*} whose topological order produces a valid sequence of classes $\mathbb{S}$ for P_{comb}.

AO finds a feedback vertex set V^{wfvs} in G^{jnt} with minimal total weight, where vertex weight equals the number of filters in the corresponding class. By $W(V)$ we denote the total weight of vertices in V. An induced subgraph on vertices that

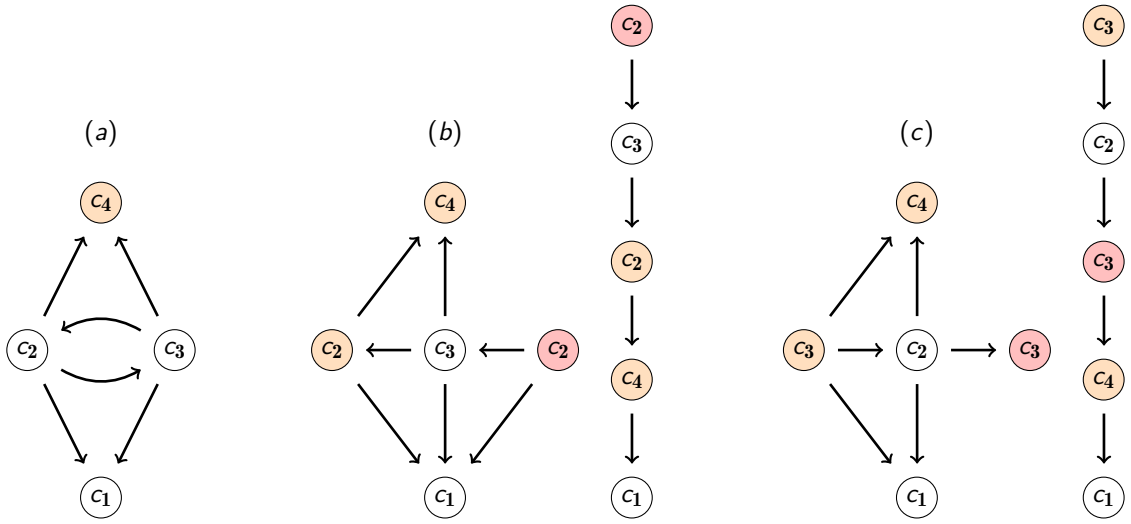


Figure 3.14: (a) G^{jnt}; (b)-(c) two solutions depending on the values of $w(c_3)$ and $w(c_2)$.

Algorithm 14: AO($\mathcal{P}$)

```
1  begin
2      construct a graph G^jnt(P_1, ..., P_l);
3      V^wfvs ← WFVS(G^jnt), with vertex weights w(c) = |c|;
4      initialize G* as a subgraph of G^jnt induced by V \ V^wfvs;
5      for each c ∈ V^wfvs do
6          for each P_i containing c do
7              add to G* an instance c̃_i of c;
8      for each P_i do
9          for each c ≺_{P_i} c' s.t. c or c' are in V^wfvs do
10             add edge (c̃_i, c̃'_i) to G*;
               /* here c̃_i = c if c ∉ V^wfvs                    */
11     let 𝕊 be a topological ordering of the vertices of G*;
12     return 𝕊
```

are not in V^{wfvs} is acyclic, therefore, the corresponding classes appear only once in $\mathbb{S}$. For a class $c \in V^{\mathsf{wfvs}}$, the sequence $\mathbb{S}$ contains a separate instance of c for each policy containing c.

To transform G^{jnt} into an acyclic graph G^*, the algorithm AO first removes all classes that are in V^{wfvs} (line 3 in Algorithm 14). Then for every class $c \in V^{\mathsf{wfvs}}$ and every policy P_i containing c, a vertex $\tilde{c}_i$ is added into G^* (lines 5-6); other vertices in G^* will be connected with $\tilde{c}_i$ by edges induced by the partial order on

$\prec_{P_i}$ (lines 8-9). A topological order of the vertices of G^* (line 10) forms a correct solution for the PSP problem (see Theorem 3.5.2). Since G^* can be constructed in at most $T_{G^{\mathsf{jnt}}}(\mathcal{P})$ time, the running time of AO is $T_{\mathrm{FVS}}(G^{\mathsf{jnt}}) + T_{G^{\mathsf{jnt}}}(\mathcal{P})$, where $T_{\mathrm{FVS}}(G)$ is the running time of the algorithm for the WFVS problem.

Theorem 3.5.2. AO *correctly solves the PSP problem.*

A much more detailed treatment of classifiers reusing the same policies, including a comprehensive simulation study, and many more results on this topic (which are not part of the present dissertation) can be found in the works [7, 17].

3.6 Approximate packet classification

Exact computations may require excessive resources. *Approximate computing* deals with potentially inaccurate computations helping alleviate resource constraints [165]. In this section, we generalize a classical *packet classification problem* (the exact case), where each rule is composed from a ternary bit string and an action; since rules can overlap they have priorities and the first *matched* rule is returned for an incoming packet. There is a long line of research exploring various optimization methods to find *semantically equivalent* packet classifiers, where each header matches the same action in an originally given and optimized classifiers [53,96,154]; our own research in this direction has been the subject of previous sections in this chapter. In this section we consider the case when semantically equivalent classifiers fail to achieve desired optimization results and introduce approximate representations of packet classifiers allowing to "multiplex" multiple actions. This additional level of flexibility allows to improve resource requirements by still keeping desired level of accuracy.

The efficiency of approximate representations depends on two major factors: a clear understanding of the allowed classification results for each rule and structural properties of classifiers. The first factor provides additional flexibility that can improve the efficiency of classifiers even if they are irreducible by conventional methods; the second one is mostly constraining. The efficiency of approximate representations is defined by a balance between these two factors.

The notion of *exact* and *approximate* classifiers is central for this section. We say that a classifier $\mathcal{K}$ is *exact* if for each rule $R_i \in \mathcal{K}$, the corresponding $\mathcal{A}_i$ consists of a single element; otherwise, $\mathcal{K}$ is *approximate*. Exact classifiers

are a traditional way to represent packet classifiers. Approximate classifiers are a generalization where $\mathcal{A}_i$ represents multiple options that can be taken for the classifying rule.

Example 3.6.1. In the following classifier with $w = 4$, $H = (0\ 1\ 1\ 0)$ matches R_1, R_4, $R_\perp$, so H is classified by R_1.

$\mathcal{K}$	#1	#2	#3	#4	Exact	Approx
R_1	0	1	1	0	A_1	A_1, A_4
R_2	1	$*$	0	$*$	A_2	A_2
R_3	1	0	1	$*$	A_3	A_3
R_4	0	$*$	1	$*$	A_4	A_4
R_5	$*$	0	0	$*$	A_5	A_5
$R_\perp$	$*$	$*$	$*$	$*$	D	D

In Example 3.6.1, filters of the classifier combined with the action sets in Exact and Approx columns form the exact and approximate classifiers respectively. In particular, if classifiers represent forwarding tables then an action set $\mathcal{A}_1 = \{A_1, A_4\}$ means that a packet can be transmitted through either A_1 or A_4. Note that in this case $\mathcal{A}_1 = \{A_1, A_4\}$ does not represent possibilities for load balancing but adds options for classifier optimizations. We say that a classifier $\mathcal{K}'$ is *approximately equivalent* to a classifier $\mathcal{K}$ if for every header H, the classifying action set $\mathcal{A}' \in \mathcal{K}'$ is a subset of the classifying action set $\mathcal{A} \in \mathcal{K}$. In Example 3.6.1, the classifier $\mathcal{K}_e$ is approximately equivalent to $\mathcal{K}_a$.

An *optimization process* consists of a set of *basic operators* (that we have introduced above) and a *heuristic* choosing a sequence of applied operations (*operation sequence*). In this section we study properties of operation sequences allowing us to construct efficient heuristics.

Let $\mathcal{S}(\mathcal{K})$ be a classifier obtained after applying $\mathcal{S}$ on a classifier $\mathcal{K}$. By definition, each applied operation removes exactly one rule, so $\mathcal{S}(\mathcal{K})$ has $|\mathcal{K}| - |\mathcal{S}|$ rules, where $|\mathcal{S}|$ is the number of operations in $\mathcal{S}$ (*length* of $\mathcal{S}$). Thus, considered heuristics can shift from minimizing number of rules in the optimized classifier to constructing a longest operation sequence. An operation sequence $\mathcal{S}$ is *extensible*, if $\mathcal{S}$ can be appended with additional operations; otherwise, $\mathcal{S}$ is *inextensible*. A longest operation sequence is always inextensible, but inextensible sequences are not necessary longest.

Example 3.6.2. This example shows that different operation sequences even in the exact case can lead to different results.

$\mathcal{K}$	#1	#2	#3	#4	#5	Action
R_1	1	1	1	$*$	$*$	A_1
R_2	1	0	$*$	$*$	0	A_1
R_3	1	1	0	$*$	0	A_1
R_4	1	1	1	$*$	0	A_1
$R_\perp$	$*$	$*$	$*$	$*$	$*$	D

Here, we can apply either the operation sequence $\mathcal{S}_1 = \mathcal{F}(R_4)$ or $\mathcal{S}_2 = \mathcal{R}(R_3, R_4)\mathcal{R}(R_2, R)$, where R is rule constructed by $\mathcal{R}(R_3, R_4)$. Both $\mathcal{S}_1$ and $\mathcal{S}_2$ are inextensible, but only one of them is longest.

Example 3.6.2 shows that a combinatorial search for longest operation sequences is a hard problem even in the exact case. At the beginning, to reduce the complexity of an optimization process, we consider only operations that are applicable on the original classifier $\mathcal{K}$. We say that an operation sequence $\mathcal{S}$ is *initial* if $\mathcal{S}$ contains only operations applicable on $\mathcal{K}$. For instance, in Example 3.6.2 $\mathcal{S}_2 = \mathcal{R}(R_3, R_4)\mathcal{R}(R_2, R)$ is not an initial sequence since the second resolution in $\mathcal{S}_2$ merges a rule constructed by the first resolution in $\mathcal{S}_2$. Note that all operations in the initial sequence $\mathcal{S}$ do not take the same rule as an argument; we define such operations as *non-conflicting* (for instance, $\mathcal{R}(R_3, R_5)$ can not be in the same initial sequence with any subsumption applied on R_3 or R_5). The longest initial sequence provides a lower bound on the length of a longest operation sequence. Below, we consider problems of finding a longest initial sequence in the exact and in the approximate case, respectively; this study shows the combinatorial complexity of the approximate classifier minimization, and establishes building blocks for heuristics finding longest operation sequences.

We say that an operation sequence $\mathcal{S}$ on $\mathcal{K}$ is ordered according to the *rule priority* if all operations in $\mathcal{S}$ are ordered according to priorities of their basic rules. We denote by $\mathbf{O}$ the set of operations applicable on the input classifier $\mathcal{K}$. Even non-conflicting operations in $\mathbf{O}$ implicitly depend on each other; for example, a resolution can create a new possible rule that will intersect with another rule in a backward subsumption. We make one additional assumption: if for a rule R_i both $\mathcal{F}(R_i)$ and $\mathcal{B}(R_i)$ are applicable on the original classifier, then $\mathbf{O}$ contains only

$\mathcal{F}(R_i)$ since it makes no sense to apply backward instead of forward subsumption.

The works [1, 16] contain a detailed treatment of operation sequences which is not part of the present dissertation. It shows that while finding the best possible operation sequence is always hard, in the exact case it is relatively easy to construct a longest initial sequence. Next we turn to the approximate case.

In the approximate case, $\mathbf{O}$ can increase compared to the exact case since generalized operations are applicable more often. But unlike the exact case, in the approximate case not every subset $\mathbf{O}'$ of $\mathbf{O}$ without conflicting operations has a corresponding initial sequence consisting of all operations from $\mathbf{O}'$ (this is a key result shown in [1, 19]), and the length of a longest initial sequence can be smaller than the size of a biggest subset of $\mathbf{O}$ without conflicting operations.

Example 3.6.3. Non-conflicting operations from $\mathbf{O}$ are not always applicable together.

$\mathcal{K}$	#1	#2	#3	#4	$\mathcal{A}$
R_1	0	1	1	$*$	A_1
R_2	1	1	$*$	1	A_1, A_2
R_3	$*$	1	1	$*$	A_1, A_2
R_4	0	1	$*$	1	A_2
$R_\perp$	$*$	$*$	$*$	$*$	D

For instance, here we can apply resolution $\mathcal{R}(R_2, R_4)$ or backward subsumption $\mathcal{B}(R_1)$, but we cannot apply both since otherwise $\mathcal{A}_2$ would become empty.

The complexity of constructing the longest initial sequence arises from backward subsumptions. For any subset $\mathbf{O}'$ of $\mathbf{O}$ without conflicting operations and backward subsumptions, there is an initial sequence consisting of all operations from $\mathbf{O}'$ (this is a key result shown in [1, 19]). But if $\mathbf{O}$ has backward subsumptions then finding a longest initial sequence turns into an intractable problem (its NP-hardness is shown in [1, 19]).

So far we have considered a general case where each filter is a ternary bit string. Since minimizing a given classifier is an intractable problem, we have defined our optimization process as constructing a maximal sequence over a predefined set of operators. There is, however, an important special case of LPM (longest prefix match) classifiers, where each filter contains $*$ only after all 0 and 1. Each such filter can be represented as a prefix over binary alphabet, where the position of a first $*$ defines the prefix length. Rules in an LPM classifier are ordered according

to their prefix length, where the highest priority rule has a longest prefix. We assume that an LPM classifier does not contain two rules with the same prefix. In the following, we denote by $\mathbf{A}$ the set of all actions in a given classifier $\mathcal{K}$.

Since filters in LPM classifiers are prefixes, the construction of a minimal LPM classifier can be done in polynomial time. In the exact case, the works [97, 191] introduce the algorithm eLP that computes a minimal LPM classifier in time $O(|\mathcal{K}| \cdot w \cdot |\mathbf{A}|)$, where w is the classification width (in bits). We have been able to generalize eLP to the approximate case while preserving optimality; we call the generalized version aLP. As a byproduct, we improve the time complexity of eLP to $O(N \cdot w)$. Both the eLP algorithm for the exact case and the aLP algorithm that extends it to the approximate case are shown and discussed in detail in the works [1, 19]; these works also study the complexity of these algorithms.

Theorem 3.6.1. *The time complexity of aLP is $O(|\mathcal{K}| \cdot w \cdot l_{\max})$, where $l_{\max}$ is the size of the biggest action set in the originally given classifier $\mathcal{K}$.*

Theorem 3.6.2. *Subsumptions can not be applied on a maLPM classifier $\mathcal{K}_a$ or on the classifer $\mathcal{S}(\mathcal{K}_a)$, where $\mathcal{S}$ is an operation sequence consisting of resolutions.*

A much more detailed treatment of approximate classifiers, including a comprehensive simulation study and many more results on this topic (which are not part of the present dissertation) can be found in the works [1, 16].

Chapter 4 Control plane algorithms: network simplification, cloud computing, and smart grids

4.1 Network simplification algorithms

The growing complexity of network management has been the subject of many debates. In general, there are two major factors that impact the complexity of network operations: the size and structure of a manageable state and the frequency of its changes/updates. To address this problem, the works [95, 146] proposed a framework with control plane composition that can use controllers from different vendors. The framework applies services implemented on heterogeneous controllers to the same network traffic by requiring only a standardized south-bound API. This prevents vendor lock-in at the control plane by allowing network operators to deploy services implemented on heterogeneous controllers. The framework is designed to operate in a way that is transparent to the controllers and does not require additional standardization as north-bound API. Thus, an important direction of study intended to simplify network management is to develop representations of the original network by a simpler virtual network while still implementing desired properties of the original.

There have been attempts to virtualize specific network topologies, optimizing various objectives [49, 121], but at present there is no way to systematically quantify the "efficiency" of implemented network representations versus the complexity of operations. Preserving "bandwidth" and "routing" network capabilities in simpler representations allows to operate services transparently and independently from the physical network infrastructure. In addition, understanding network constraints allows for better resource planning in case additional investments are needed. In this section, we present a first step in this direction.

The first problem we explore in this section is *capacity planning*, by which we

mean minimizing the aggregate capacity of the links in a network while maintaining its original topology. Since different coexisting applications can use bandwidth allocation methods that optimize different objectives (e.g., aiming for shortest, cheapest, or load-balanced traffic over several paths), we do not assume any knowledge about bandwidth allocation methods and routing for interconnecting sources and destinations. In fact, the resulting set of link capacities must allow for any bandwidth allocation and routing the original did, a property we call *bandwidth equivalence*. For a simple example, the three networks in Figure 4.1 are all *bandwidth equivalent*: any set of routes from s_1 and s_2 to d that satisfies capacity constraints shown on the edges of one of the graphs will also satisfy them in the two others. However, the graphs have very different total capacities, and the result of capacity planning for all three networks in Fig. 4.1 should be the network on the right, which has minimal aggregate capacity and cannot be reduced further. Useful applications of capacity planning include various WAN optimizations for interconnected geo-distributed datacenters, where WAN links are extremely expensive [116, 118].

The second problem goes even further: starting from a given network topology, we allow shrinking edges and merging nodes of the original network, simplifying it while preserving its routing capabilities. We first define the notion of *routing equivalence* between two networks, a property that allows to map coherent sets of routes (called *bandwidth allocations* in what follows) between the networks while preserving capacities. The simplified network can be used to find bandwidth allocations, and they can be mapped back to the original topology. Hence, we introduce network transformations that simplify the network while preserving routing equivalence. For instance, Fig. 4.2a shows a simplified routing equivalent graph G' for a more complex graph G; Fig. 4.2b shows a bandwidth allocation from s to d on G' and how it maps to G; and vice versa. One can construct simple routes on G' and then map them back to G.

Unfortunately, routing equivalence is a strong property. We explore additional transformations that relax it, allowing further simplifications: even without a 1-to-1 mapping between bandwidth allocations, one can guarantee that any bandwidth routed from sources to destinations in one network is also feasible in the other. Moreover, we show an instance of the well-known Braess' Paradox in our context: removing a link from a network may not affect its capabilities negatively

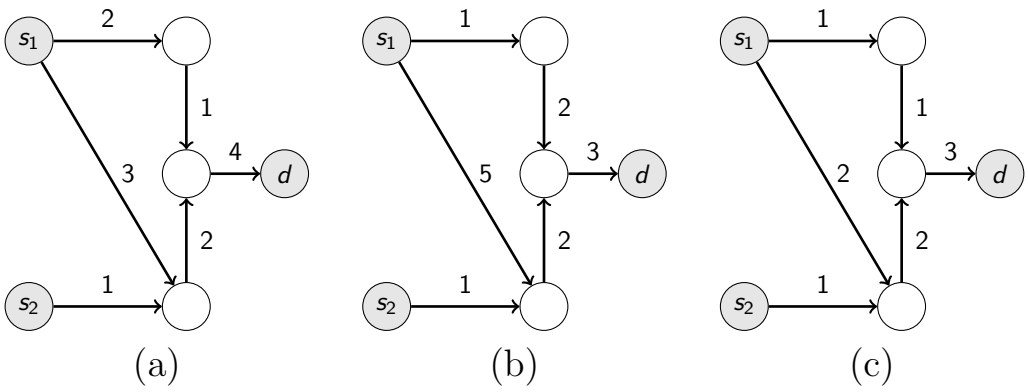


Figure 4.1: Three bandwidth equivalent networks.

but rather allow to drastically reduce its size. We find a fundamental tradeoff between additional network topology simplification and extra capacity that would be required to achieve it.

As a network interconnect, we consider a weighted directed graph $G = (V, E, w, S, D)$ without self-loops, where V is a set of vertices, E is a set of edges among them, and $w : E \to \mathbb{R}^+$ is a weight function that represents an available capacities on edges. In addition some vertices in V are marked as *sources* $S \subset V$ and *destinations* $D \subset V$. We assume that source vertices do not have incoming edges and destination vertices do not have outgoing edges (if they do, we can simply replace an internal source vertex s by an internal node v_s and a new source s' linked to that internal node v_s with an infinite capacity edge, and similarly for an internal destination vertex).

We begin with the notion of bandwidth allocation; the intuition is to define a set of routing paths together with specific bandwidths allocated along these paths.

Definition 4.1.1. *A* bandwidth allocation $A = (P, f)$ *on a network* $G = (V, E, w, S, D)$ *is a set of edge-simple paths* $P = \{p_1, \ldots, p_k\}$ *and a function* $f : P \to \mathbb{R}^+$ *such that:*

(1) $\forall i \in [1, k]$, *path* $p_i = (v_1, \ldots, v_{l_i})$ *starts at a source vertex* $v_1 \in S$ *and ends at a destination vertex* $v_{l_i} \in D$*;*

(2) $f(p_i) \geq 0$ *for every* i *(nonnegativity);*

(3) for every $e \in E$, $\sum_{p \in P : e \in p} f(p) \leq w(e)$ *(capacity constraints).*

Note that paths in a bandwidth allocation can contain loops (but cannot cross the same edge twice). The notion of bandwidth allocation is similar in spirit to flow

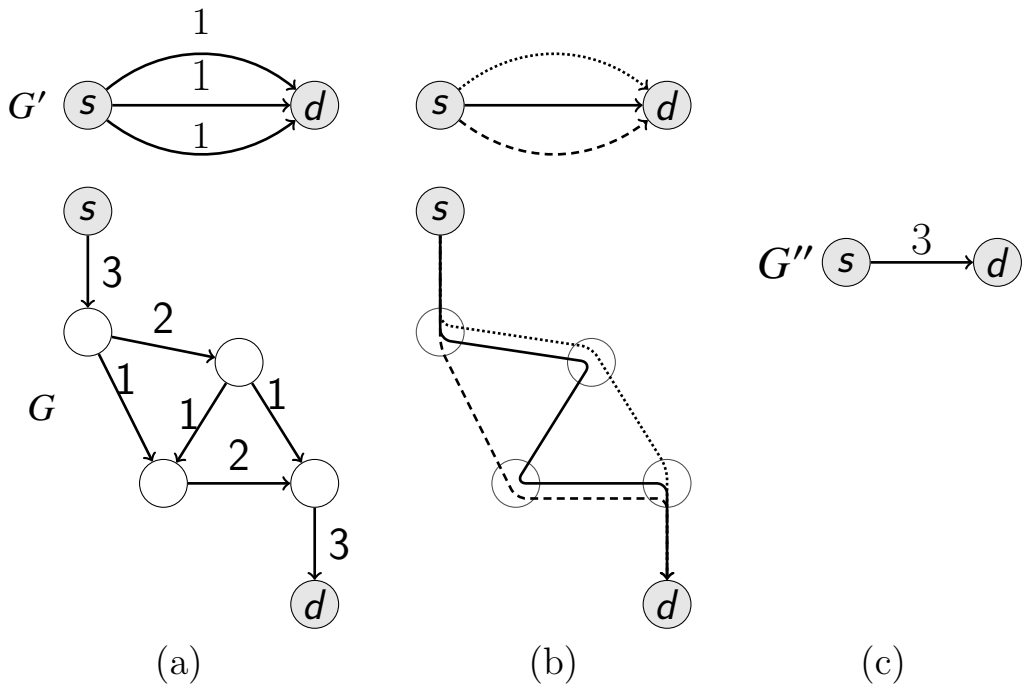


Figure 4.2: Simplifications of network topologies: (a) two routing equivalent graphs, (b) a bandwidth allocation in G' with three routes of bandwidth 1 maps to G, (c) any bandwidth routed from s to d in G'' is also feasible in G' and G.

decomposition: we partition a flow into a set of weighted paths; the difference is that in flow decomposition no cycles are allowed, and all path weights are positive.

Definition 4.1.2. *Two weighted networks* $G = (V, E, w, S, D)$ *and* $G' = (V, E, w', S, D)$ *with the same graph structure but possibly different weight functions* w *and* w' *are called* bandwidth equivalent*, denoted* $G \simeq G'$*, if every bandwidth allocation* A *on network* G *is also a bandwidth allocation on network* G'*, and vice versa.*

Definition 4.1.3. *We call a network* $G^* = (V, E, w^*, S, D)$ *a* minimal bandwidth network *if, for every network* $G = (V, E, w, S, D)$ *such that* $G \simeq G^*$*, it holds that* $w^*(e) \leq w(e)$ *for every* $e \in E$*.*

Note that this definition uses a specific objective: to minimize the total capacity of all edges $\sum_{e \in E} w(e)$.

The first problem we consider is to find the minimal bandwidth network G^* in the bandwidth equivalence class of a given network G. Note that if a minimal bandwidth network equivalent to a given G exists and is unique, it obviously also minimizes the total capacity of all edges $\sum_{e \in E} w^*(e)$. However, it is not obvious that it exists at all.

Fortunately, we can provide a constructive definition: an explicit algorithm

to construct minimal bandwidth networks. Given a network G, the *brute force* algorithm to find G^* works as follows:

- for every subgraph $G' \subseteq G$ with induced edge capacities, compute max-flow $f_{G'}$ from sources S to destinations D;
- for every edge $e \in E$, set its capacity in G^* to the maximum of all flow values, i.e., $w^*(e) = \max_{G' \subseteq G} f_{G'}(e)$.

Theorem 4.1.1. *For every input network G, the brute force algorithm computes a minimal bandwidth network G^* that is unique.*

The brute force algorithm is obviously exponential since it requires to enumerate all subgraphs. Next, we show an algorithm that finds the minimal bandwidth network in polynomial time if the graph has no cycles; ince we restrict the algorithm to directed acyclic graphs (DAGs), we call it *DAG-OPT*. To find a minimal bandwidth network equivalent to a DAG $G = (V, E, w, S, D)$, DAG-OPT proceeds as follows:

(1) sort vertices of G in topological order: $u \leq v$ if G has a path from u to v (here we use the fact that G is a DAG);

(2) for every edge $e = (u, v)$:

   (i) consider the subgraph G_e of G induced by the vertex set V_e which contains vertices comparable to u in the topological order:

$$V_e = \{v' \in V \mid v' \leq u\} \cup \{v' \in V \mid v' \geq v\};$$

   (ii) find a maximal flow $f^*_{G_e}$ on G_e from sources to destinations;

   (iii) set the final capacity $w^*(e)$ to the flow through e in this maximal flow, $w^*(e) = f^*_{G_e}(e)$.

The DAG-OPT algorithm is illustrated in Fig. 4.3. Essentially, to find the optimal capacity for an edge e we remove all vertices and edges that are not connected to paths through e and solve the maximal flow problem on the resulting subgraph, which can be done in time $O(|V||E|)$ [170].

Theorem 4.1.2. *The algorithm DAG-OPT outputs the minimal bandwidth network G^* equivalent to a given network (DAG) G in time $O(|V||E|^2)$.*

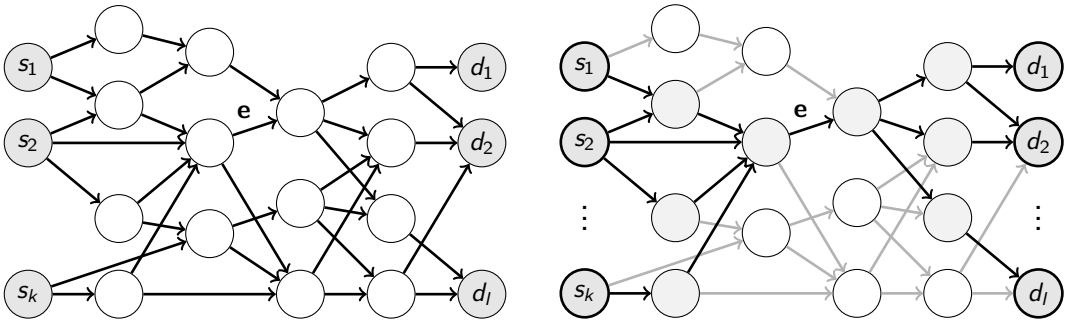


Figure 4.3: The subgraph G_e used in the DAG-OPT algorithm.

We next present a local heuristic that, although it does not find the minimal bandwidth graph in general, produces good results in practice. We also show that this heuristic is optimal on graphs whose undirected version is a forest, and, unlike DAG-OPT, it can be applied to graphs with cycles. Moreover, this algorithm, called *WPP*, has computational complexity $O(|E| + |V| \log |V|)$, as opposed to $O(|V||E|^2)$ of DAG-OPT.

Consider a graph $G = (V, E, w, S, D)$; the goal is to reduce its total capacity $\sum_{e \in E} w(e)$. Let us denote the total capacity of the incoming and outgoing links of a vertex v by I_v and O_v, respectively; we also denote $I_e = I_u$ and $O_e = O_v$ for an edge $e = (u, v) \in E$.

The algorithm is based on a local capacity adjustment that we call *weight propagation* (WP). WP uses a simple observation that an edge $e \in E$ cannot transmit more bandwidth than I_e or O_e. In this section, we assume that every source s has $I_s = \infty$ and every destination d has $O_d = \infty$; recall that sources have no incoming links and destinations have no outgoing links.

To apply WP to an edge $e \in E$ is to set $w(e) \leftarrow \min\{w(e), I_e, O_e\}$; this yields a new graph which is bandwidth equivalent to G. If $w(e) > \min\{I_e, O_e\}$, by applying WP to e we reduce its capacity and hence the total capacity. WPP applies WP to the edges until no edge e satisfies $w(e) > \min\{I_e, O_e\}$.

To bound the complexity, we need to bound the number of WP applications, which is done by carefully choosing the edges on which to apply WP next. WPP is presented below; it processes the nodes of G, chooses on each iteration an unprocessed node v with minimal $\min\{I_v, O_v\}$, and then applies WP to all edges incident to v.

Theorem 4.1.3. *The algorithm WPP, after processing a graph*

Algorithm 15: Algorithm WPP.

```
Data: network G = (V, E, w, S, D)
1 begin
2     N ← V
3     while N ≠ ∅ do
4         Choose v ∈ N : min{I_v, O_v} = min_{u∈N}{I_u, O_u}
5         Apply WP to all edges e incident to v
6         N ← N \ {v}
```

$G = (V, E, w, S, D)$, outputs a graph $G' = (V, E, w', S, D)$ such that $G' \simeq G$ and for all $e \in E$ it holds that $w'(e) \leq w(e)$ and $w'(e) \leq \min\{I'_e, O'_e\}$, in time $O(|E| + |V| \log |V|)$.

WPP is not guaranteed to find the minimal bandwidth graph in general. However, WPP achieves excellent results in practice, and moreover, in the common special case of a forest (when the graph topology is a set of trees), WPP is optimal.

Theorem 4.1.4. *Let $G = (V, E, w, S, D)$ be a graph such that its undirected version is a forest. Then WPP applied to G outputs the minimal bandwidth network G^* equivalent to G.*

We have shown how to reduce the total capacity of a network while preserving its original topology. This is required in some problems (e.g., capacity planning) but in case of network simplification/virtualization the objective is different: simpler network management. The idea of network topology transformations that we introduce in this section is to find, for a network G, a simpler network G' that can satisfy exactly the same bandwidth requests as G. Since G' is simpler, the feasibility of bandwidth requests can be checked more efficiently. Such transformations can be used to simplify management, representing a complex network with a much simpler structure which is easier to manage/configure.

Formally, two networks $G = (V, E, w, S, D)$ and $G' = (V', E', w', S, D)$ with the same set of sources and destinations are *routing equivalent* if there is a one-to-one mapping $g : \text{Path}_G(S, D) \to \text{Path}_{G'}(S, D)$ (where $\text{Path}_G(A, B)$ is the set of paths in network $G = (V, E)$ that begin in $A \subseteq V$ and end in $B \subseteq V$) between paths from sources to destinations on G and G' that preserves bandwidth allocations. That is,

(1) for every bandwidth allocation $A = (\{p_1, \ldots, p_k\}, f)$ on network G,

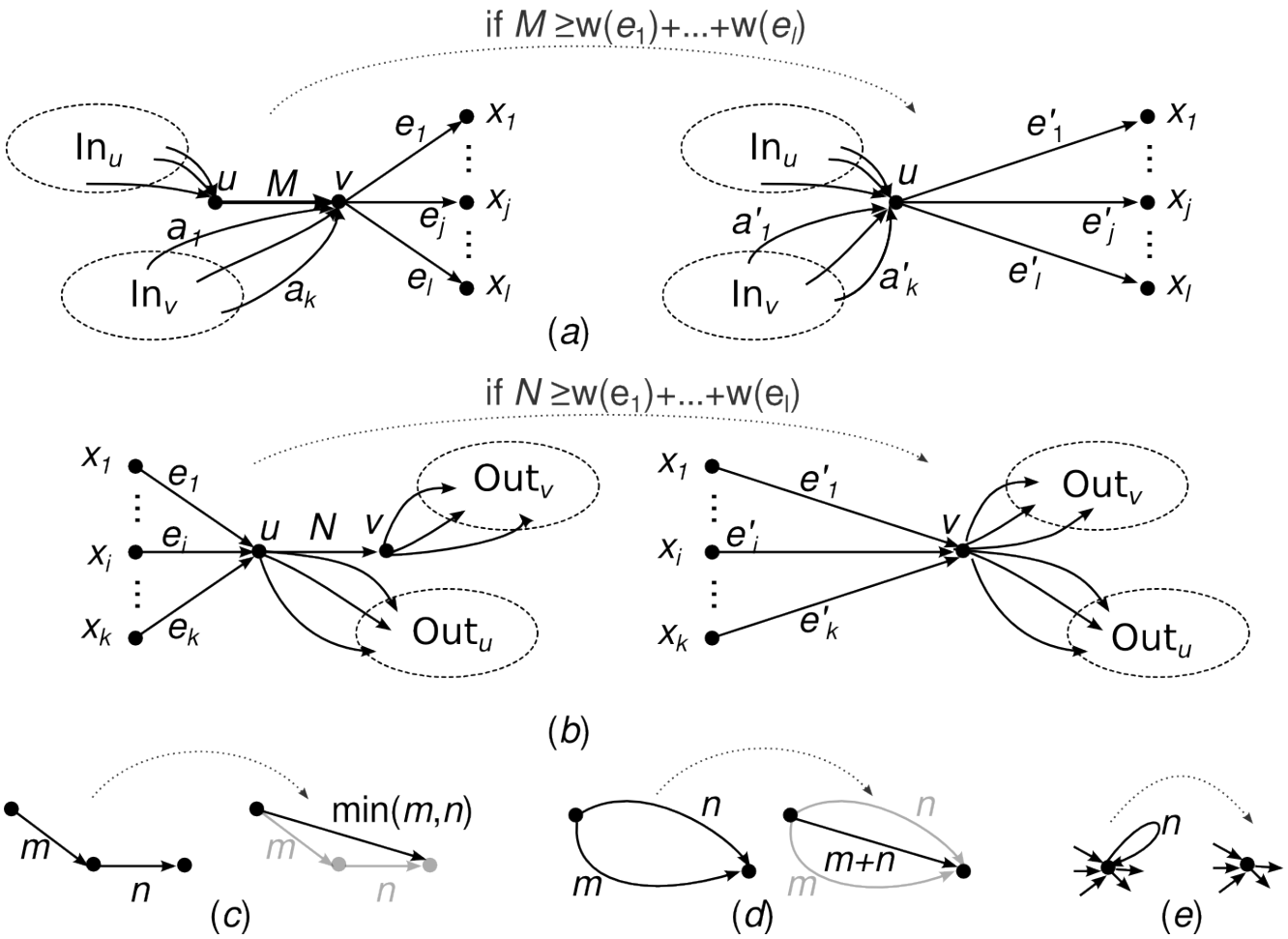


Figure 4.4: Topology transformations: (a) shrinking non-bottleneck incoming edge; (b) shrinking non-bottleneck outgoing edge; (c) merging parallel edges; (d) removing self-loops; (e) path of length 2.

$A' = (\{g(p_1), \ldots, g(p_k)\}, f')$, where $f'(g(p)) = f(p)$, is a bandwidth allocation on G';

(2) and vice versa, for every bandwidth allocation $A' = (\{p'_1, \ldots, p'_k\}, f')$ on network G', $A = (\{g^{-1}(p'_1), \ldots, g^{-1}(p'_k)\}, f)$, where $f(p) = f'(g(p))$, is a bandwidth allocation on G.

Our objective is to propose ways to transform a network G into a simpler routing equivalent network G'. The transformation process must also provide the function g^{-1} that maps any bandwidth allocation on the simple network G' back to G. The proposed transformations are illustrated in Figure 4.4. They can be summarized in the following two heuristics.

1. *Shrinking a non-bottleneck incoming edge.* The most general form of this heuristic is shown in Fig. 4.4a. Consider an edge $e = (u, v)$ such that $u \notin S$, $v \notin D$, and $u \neq v$. If e has capacity $w(e) = M$ and its outgoing edges have

aggregate capacity $O_v = \sum_{e_j \in Out_v} w(e_j) \leq M$, then e is not a bottleneck in any bandwidth allocation and can be shrunk into a single node u. This means that node v and edge e disappear, each edge $e_j = (v, x_j) \in Out_v$ is replaced by a new edge $e'_j = (u, x_j)$ with capacity $w(e'_j) = w(e_j) = n_j$, and each edge $a_i = (y, v) \in In_v$ is replaced by a new edge $a'_i = (y, u)$ with capacity $w(a'_i) = w(a_i)$. The function g maps a path $p = (\ldots, e', e, e_j, \ldots) \in \text{Path}_G(S, D)$ into $g(p) = (\ldots, e', e'_j, \ldots)$. Note that g is 1-to-1 since a path $p' = (\ldots, e', e'_j, \ldots) \in \text{Path}_{G'}(S, D)$ is mapped to either $g^{-1}(p') = (\ldots, e', e, e_j, \ldots)$ (if $e' \in In(u)$) or to $g^{-1}(p') = (\ldots, a_i, e_j, \ldots)$ (if $e' = a'_i$ is one of the new edges that replaced In_v). Fig. 4.4a illustrates this case, and Fig. 4.4c shows an important special case when $l = 1$.

2. *Shrinking a non-bottleneck outgoing edge.* This heuristic is symmetric to the previous one, as shown in Fig. 4.4b, but with respect to the aggregate capacity of the edges incoming to u. If $I_u = \sum_{j \in In_u} w(e_j) \leq N$ the link $e = (u, v)$ is shrunk into a single node v. The mapping function g is similar to the previous case. As before, Fig. 4.4c is a special case of this heuristic when $k = 1$.

Note that transformations 1 and 2 reduce simultaneously the number of edges in the network, the number of vertices in the network, and the total capacity of all edges.

Theorem 4.1.5. *The network $G' = (V', E', w', S, D)$ obtained by applying Transformations 1 and 2 to a network $G = (V, E, w, S, D)$ is routing equivalent to G.*

Transformations 1 and 2 proposed above produce routing equivalent simpler graphs. Next, we introduce a relaxation of the routing equivalence property, called bandwidth-preserving routing transformation, allowing for even simpler graphs. Intuitively, the property that is guaranteed by a bandwidth preserving routing transformation is that any feasible allocation in the derived graph G' can be realized in graph G. The bad news is that the routes are not directly given by mapping g, since it is not 1-1 anymore. The good news is that this relaxed property allows to apply new graph transformations that in most cases allow to end up with a much simpler graph G'.

For two networks $G = (V, E, w, S, D)$ and $G' = (V', E', w', S, D)$ with the same set of sources and destinations, a *bandwidth-preserving routing transformation* is a mapping $g : \text{Path}_G(S, D) \rightarrow \text{Path}_{G'}(S, D)$ between paths from sources to destinations on G and G' that satisfies the following:

(1) for every bandwidth allocation $A = (\{p_1, \ldots, p_k\}, f)$ on network G,

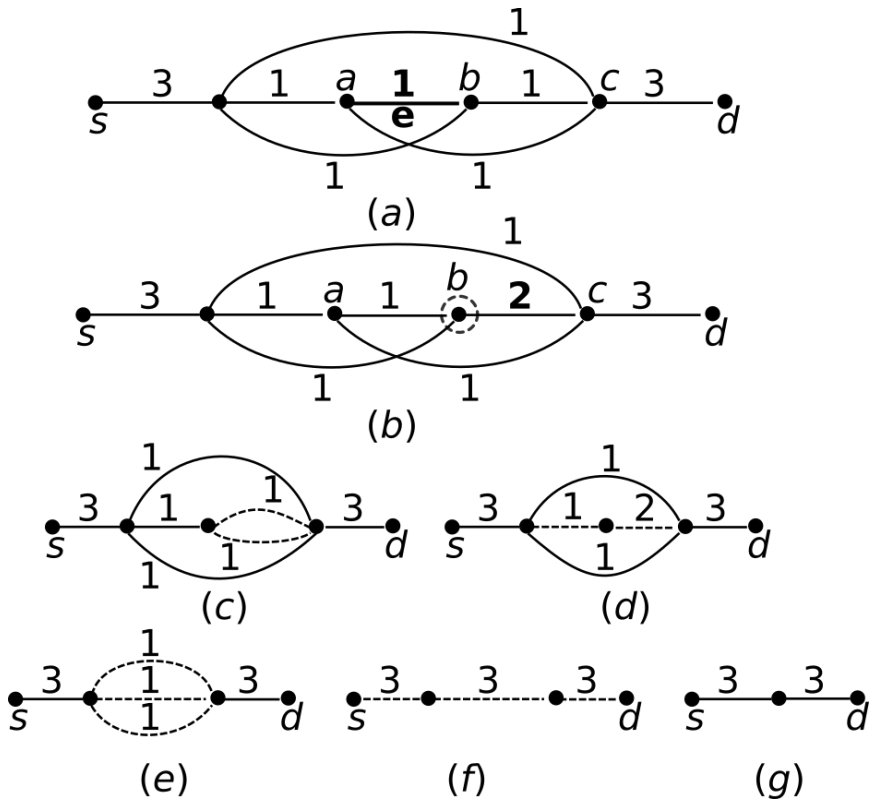


Figure 4.5: Example of Braess' Paradox and the tradeoff between total capacity and topology optimization: (a) original graph; (b) graph with one edge capacity increased; Transformation 2 is applied to link (b, c); (c-g) further simplifications (dashed lines) lead to total collapse of the graph.

$A' = (\{g(p_1), \ldots, g(p_k)\}, f')$, where $f'(g(p)) = \sum_{q \in g^{-1}(g(p))} f(q)$, is a bandwidth allocation on G';

(2) for every bandwidth allocation $A' = (\{p'_1, \ldots, p'_k\}, f')$ on network G', there exists a function $f : \cup_{i=1}^{k} g^{-1}(p'_i) \to \mathbb{R}$ such that $f'(p') = \sum_{p \in g^{-1}(p')} f(p)$ and $A = (\cup_{i=1}^{k} g^{-1}(p'_i), f)$ is a bandwidth allocation on G.

We add the following two transformations.

3. *Merge parallel edges.* If there are two parallel edges with capacities m and n from vertex u to vertex v, we can replace (merge) them by one new edge from u to v with total capacity $m + n$. This transformation is illustrated in Fig. 4.4d.

4. *Remove self-loops.* After some transformations, self-loops may arise in the graph; they can be simply removed (Fig. 4.4e).

Figure 4.5b-g shows how repeated application of the above transformations can drastically simplify a graph.

Theorem 4.1.6. *Transformations 3 and 4 are bandwidth preserving routing transformations.*

For the special case in which the original graph G is a forest, it is easy to see that the graph G' obtained with any bandwidth-preserving routing transformation g is in fact routing equivalent.

Theorem 4.1.7. *A bandwidth preserving routing transformation applied to a forest $G = (V, E, w, S, D)$ generates a graph $G' = (V', E', w', S, D)$ that is routing equivalent to G.*

Figure 4.5a shows a paradoxical network: this network allows to route bandwidth of 3 from s to d, and none of the transformations proposed above can be used to simplify the network. Surprisingly, after removing one edge, the central edge $e = (a, b)$ of capacity 1, we can start a chain of transformations that lead to the trivial graph shown in Fig. 4.5g. Note that the bandwidth that can be sent from s to d in the network is not affected by the removal of edge e from the graph in Fig. 4.5a, so this is a bandwidth-preserving routing transformation. This phenomenon can be regarded as an instance of the well-known Braess' Paradox [190], and it is worth to be studied further; e.g., is there an efficient algorithm for identifying edges that hinder simplification without any side benefits?

Starting again from a network that cannot be simplified with transformations shown above, we consider a fundamental tradeoff between network topology simplification and extra capacity required to achieve it. As a motivating example, see Fig. 4.5 again: the original graph in Fig. 4.5a is irreducible, but if we increase the capacity of a single edge $e' = (b, c)$ by 1 (Fig. 4.5b), the graph again collapses to Fig. 4.5g.

One can define optimization problems related to this tradeoff in a natural way. The general problem is to find the best possible allocation of extra capacity to simplify the resulting graph. The simplest possible graph is the *star graph*, that has one inner node connected to all sources via incoming edges and to all destinations via outgoing edges (Fig. 4.5g is a star graph). The problem can be posed in several versions:

(1) given a capacity budget C, find the allocation of this capacity budget to edges of the input graph such that the resulting graph can be simplified as much as possible (in terms of the number of edges, number of vertices, or both) with bandwidth-preserving routing transformations;

(2) find the minimal total capacity that has to be added to the graph so that

there exists a bandwidth-preserving routing transformation to a star graph.

Both (1) and (2) are hard optimization problems. For problem (1) we propose a local heuristic to find a reasonable tradeoff value. After repeatedly applying the transformations introduced above, we are left with a graph where every edge is a bottleneck (and hence transformations 1 and 2 cannot be applied). We choose the best edge e to augment its capacity with the following greedy algorithm:

(1) for every edge $e \in E$ of graph G,

   (i) find the minimal capacity w_e such that, by increasing its capacity to $w(e) + w_e$, e stops being a bottleneck;

   (ii) repeatedly apply transformations 1–4 to the graph obtained from G by setting $w(e) \leftarrow w(e) + w_e$, getting reduced bandwidth equivalent graph G_e;

   (iii) measure the resulting graph complexity $C(G_e)$;

(2) choose e such that $\frac{C(G)-C(G_e)}{w_e}$ is maximal.

Here one can use different graph complexity measures $C(G)$ depending on the objectives, e.g., number of edges, number of vertices, their sum, and so on. To solve problem (2) above this process has to be repeated until a star graph is obtained.

In general, in this section we have shown how a network can be reduced to a much simpler network where sensible management and routing decisions can be made. This is, however, only the first step towards a much more ambitious goal: represent the original network as a small virtual switch, where a richer set of decisions (e.g., scheduling) can be made and mapped to operations in the physical network.

Analytic results of this section are supported by a comprehensive simulation study on real network topologies. Our experiments confirm that proposed transformations are applicable in practice. In particular, adding 10-30% extra capacity to real world networks allows to simplify them to a star topology (connecting sources and destinations through a single switch), where all routing and bandwidth allocation decisions can be mapped back to the original network. This represents

an important step towards improving the management of virtualized network infrastructure. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the work [18].

4.2 Formalization and taxonomy of compute-aggregate problems for cloud computing applications

Datacenters store data at different interconnected locations. Modern big data applications are highly distributed, and requests need to satisfy different, often conflicting objectives: latency, cost efficiency, and others [47, 74, 150]. *Compute-aggregate* problems, where several data chunks must be aggregated in a network sink, encompass an important class of big data applications implemented in modern datacenters. Traditionally, applications have little control over how network transport handles the data. Latency optimization should account for the properties of underlying transports in order to avoid, e.g., the *incast problem* [78, 210], and optimizing latency for several compute-aggregate tasks can overload "fastest" (and more expensive) links. We believe that more fine-grained control is required to implement desired objectives transparently for applications.

In this section, we consider the problem setting under the assumption that each compute-aggregate task should conform to a budget constraint since different cloud tenants are able to invest different economic resources to compute their aggregations. To avoid oversubscription of "fastest" links, they can also have different costs of sending data over a link. The problem now divides into two completely decoupled phases:

(1) find a "cheapest" plan given a distribution of data over the network, an aggregation function (that computes the size of the result of aggregating two different chunks of data), and the cost of sending a unit of data over a link, and

(2) actually redistribute aggregations computed on the first phase while optimizing desired objectives.

In this setting, we can solve the first phase in a serial way, independently of the properties of underlying transport protocols, while the second phase can address

such problems as incast. This is a natural generalization of traditional transports designed to implement efficient aggregations.

The first phase is of separate interest since it can represent various economic settings (e.g., energy efficiency) during aggregation; this phase can also lead to better utilization of network infrastructure since the cost to send a unit of data through the links can differ for different compute-aggregate instances. Hence, our primary goal is to identify universal properties of compute-aggregate tasks that allow for unified design principles of "perfect" aggregations on the first phase. Incorporating properties of aggregation functions into final decisions requires new insights on the model level and may lead to more efficient aggregation. There is definitely room for it: the average final output size of a job has been estimated at 40.3% of the initial data sizes in Google [91], 8.2% in Yahoo, and 5.4% in Facebook [77].

Our main objective is to use a network in the best possible way for a given compute-aggregate task. This is a problem with many variables. In this section, we leave most of them to the network transport layer (e.g., it chooses how transmissions should be spread in time), concentrating on the *aggregation plan* that defines the order of aggregation. The aggregation plan is fully decoupled from transport implementation and will be formalized below.

We model a network as an undirected connected graph $G = (V, E)$, where V is a set of computing nodes connected by links (edges) E. Note that since we operate on an application level, we are free to use any overlay topology in place of G that captures only information relevant to a specific compute-aggregate task. The task is represented as a set of initial data chunks $C = \{\boxed{x_0}, \boxed{x_1}, \ldots, \boxed{x_k}\}$, with each chunk $\boxed{x_i}$ characterized by its location $\mathrm{v}(\boxed{x_i})$ and size $\mathrm{size}(\boxed{x_i})$. Since many compute-aggregate tasks require the result to be fully available on a specific node (e.g., to allow for low-latency responses), we assume a special root vertex $t \in V$ where all data chunks should be finally aggregated.

The hardest part to define is "the best possible way": objectives are application-specific and may include latency, throughput, or a more subtle objective such as congestion avoidance. We model them with a single per-link parameter, the *cost*, a flexible way to both freely combine objectives and keep the optimization problem clear. Formally, the cost function $c : E \to \mathbb{R}_+$ on the topology graph G maps each link e to its transmission cost per data unit $c(e)$; to

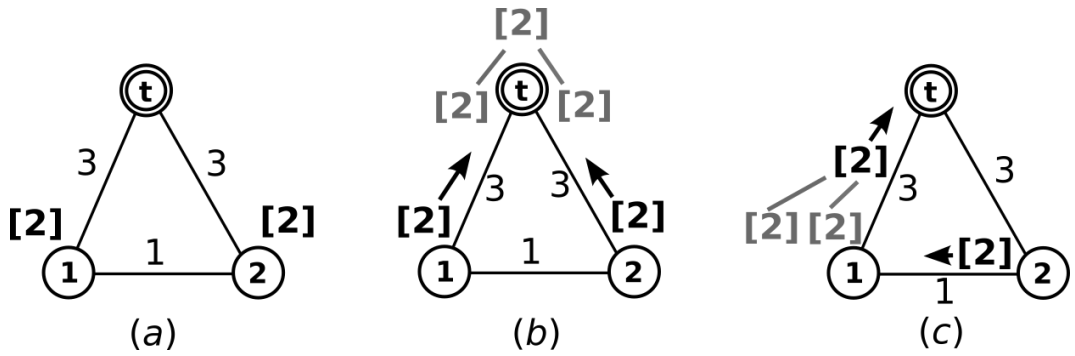


Figure 4.6: A sample compute-aggregate task with three vertices t (target), u, and v: (a) the problem; (b) "move to root" plan with cost 12; (c) optimal aggregation plan with cost 8. Transmission cost of every edge is specified near the middle of the edge (e.g., $c(u, v) = 1$).

transmit $\boxed{x}$ through e one must pay $c(e) \cdot \text{size}(\boxed{x})$. If load balancing is needed, it can be achieved by using low values of c for different subsets of links for different tasks.

A simple example of a compute-aggregate task is shown in Fig. 4.6a. Costs are shown on the edges, square brackets denote chunks, and the root vertex is marked by t.

To define an aggregation plan in a clean but practical way it is important to understand the impact it may have on the rest of the system (e.g., on a transport layer or computing infrastructure). We begin with the simplest form of an aggregation plan that we call "move to root": bring everything to the root node t (we say that an aggregation plan *moves* or *aggregates* for simplicity; in practice data transmission and aggregation are handled by the transport and application layers respectively).

"Move to root" can be suboptimal with regard to transmission costs. Suppose that in the example in Fig. 4.6 the aggregation function chooses the best chunk, so the aggregated size does not exceed the maximal size of initial chunks. Now "move to root" has total cost 12 (Fig. 4.6b: two chunks of size 2 each moving along edges of cost 3), while in Fig. 4.6c one chunk moves to vertex 1 paying 2, then chunks merge, and chunk of size 2 moves to t with total cost 8.

But concerns arise even apart from transmission costs. A naive implementation of the "move to root" plan that moves all data chunks to the root and then aggregates leads to the transport layer directing a lot of traffic towards t, possibly overflowing ingress buffers there and increasing latency due to the notorious TCP-incast problem. Moreover, in a low-latency application that aggregates in

RAM [72, 209] the storage capacity can be exhausted when all data chunks are stored at t.

This problem can be alleviated with intermediate aggregations. Data chunks can be sent to t sequentially in some order, and in the process some arriving chunks are immediately aggregated. Recent studies [77, 91] show that the final result of a compute-aggregate task is often only a small fraction (usually less than half) of the total size of initial data chunks; e.g., in counting problems the aggregation result is just a few numbers. Thus, keeping in memory a single intermediate chunk instead of several initial data chunks can significantly reduce storage requirements.

In general, not every order can be used for intermediate aggregations because the final aggregation result might depend on this order (e.g., string field concatenation), and it is undesirable for an aggregation plan to affect the result [205]. Fortunately, most aggregation functions do not depend on the aggregation order, that is, they are *associative*: $\mathbf{aggr}(\boxed{x}, \mathbf{aggr}(\boxed{y}, \boxed{z})) = \mathbf{aggr}(\mathbf{aggr}(\boxed{x}, \boxed{y}), \boxed{z})$, and *commutative*: $\mathbf{aggr}(\boxed{x}, \boxed{y}) = \mathbf{aggr}(\boxed{y}, \boxed{x})$. Below we assume that aggregations are both associative and commutative; such systems as *MapReduce* already assume this for most *reduce* functions and allow aggregations of intermediate data chunks with combiner functions [91].

The basic principle of *data locality optimization*, which lies at the heart of the *Hadoop* framework [202], is to *move computation to data* and as a result save on data transmission. We extend this strategy and try to move aggregation to data by allowing an aggregation plan to exploit intermediate nodes. Formally, an aggregation plan is a sequence P of operations $(o_0, o_1, \ldots, o_m)$, where each o_i is one of the following:

- either $\mathbf{move}(\boxed{x}, v)$, which moves a chunk x to a vertex v, or
- $\mathbf{aggr}(\boxed{x}, \boxed{y})$, which merges chunks $\boxed{x}$ and $\boxed{y}$ located at the same vertex; the result is a new chunk $\boxed{xy}$ at that vertex.

After all operations have been applied, the result must be a single data chunk $\boxed{z}$ at the root: $\mathrm{v}(\boxed{z}) = t$. For example, Figures 4.6b and 4.6c show aggregation plans for the problem shown in Fig. 4.6a. Aggregation plans are fully decoupled from the transport layer, producing instructions and constraints that the transport layer must satisfy.

With this definition of an aggregation plan, it is easy to assign it with a transmission cost and pose the minimization problem. First, every operation o_i has an associated cost that we denote $\text{cost}(o_i)$, which is defined as follows:

- $\text{cost}(\textbf{aggr}(\boxed{x}, \boxed{y})) = 0$ (there is no data transmission), and
- $\text{cost}(\textbf{move}(\boxed{x}, v)) = \text{size}(\boxed{x}) \cdot d(\text{v}(\boxed{x}), v)$, where $d(u, v)$ is the total cost of the cheapest path from u to v (if there are several such paths, any one can be used).

The total cost of an aggregation plan P, $\text{cost}(P)$, is the sum of costs of all operations in P.

In order to formally define the optimization problem for aggregation, we have to know the following: given $\boxed{x}$ and $\boxed{y}$, what is the size of their aggregation result $\boxed{xy}$? This directly affects the cost of an aggregation plan, and different aggregation result sizes can lead to very different solutions. For example, if in the example in Fig. 4.6 we assumed that the task is, e.g., sorting, where the size of an aggregated chunk is the sum of input sizes, the cost of the first plan would still equal 12, but the plan shown in Fig. 4.6c would now cost 14 and become suboptimal.

Unfortunately, the size of an aggregation result is application-specific, and in most cases the exact value depends on the actual content of $\boxed{x}$ and $\boxed{y}$; moreover, to determine this value we may need to actually perform aggregation (e.g., the number of key-value pairs in the counting problem cannot be predicted exactly unless we actually count). This is clearly infeasible since an aggregation plan must be constructed (and its cost evaluated) before the application performs any aggregations and the transport layer transmits any data. Therefore, we require each application to supply the *aggregation size function* $\mu : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$ that would estimate this size using only sizes of the inputs, so that for the purposes of optimization $\text{size}(\boxed{xy}) = \mu(\text{size}(\boxed{x}), \text{size}(\boxed{y}))$. We do not expect these functions to be exactly correct, but they should provide the correct order of magnitude in order for the optimal solution to be actually good in practice. Since **aggr** is assumed to be associative and commutative, every aggregation size function μ should also have these properties. Some examples of μ for practical problems include:

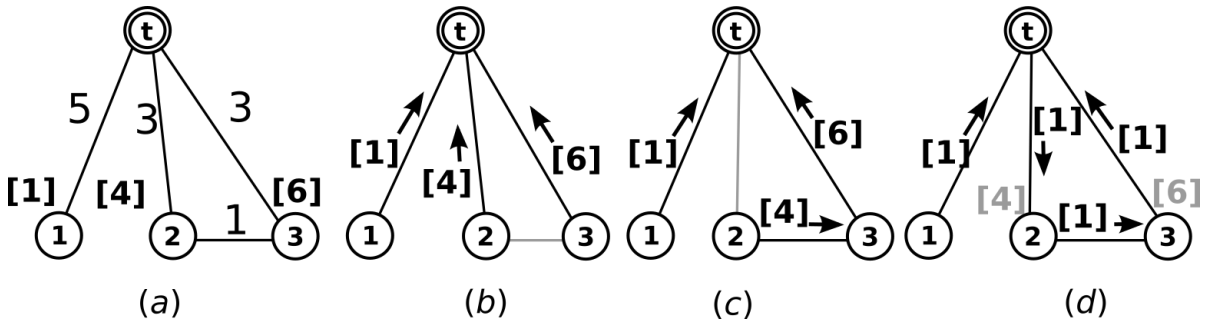


Figure 4.7: Different μ lead to different plans: (a) a sample task; (b) optimal plan for $\mu(a,b) = a + b$; (c) optimal plan for $\mu(a,b) = \max(a,b)$; (d) optimal plan for $\mu(a,b) = \min(a,b)$.

- $\mu(a,b) =$ const for finding the top k elements in data with respect to some criterion;
- $\mu(a,b) = \min(a,b)$ or $\mu(a,b) = \max(a,b)$ for choosing the best data chunk;
- $\mu(a,b) = a + b$ for concatenation or sorting;
- $\max(a,b) \leq \mu(a,b) \leq a + b$ for set union (e.g., word count).

Figure 4.7 shows how the choice of μ can affect the optimal aggregation plan. Fig. 4.7a shows chunks of size 1 at vertex 1, of size 4 at vertex 2, and of size 6 at vertex 3, and the goal is to aggregate them at vertex 0. For $\mu(a,b) = a + b$, the optimal plan is to move each chunk to the root separately (Fig. 4.7b). For $\mu(a,b) = \max(a,b)$, it is cheaper to first move the chunk of size 4 along edge $2 \to 3$ and merge it, then move the resulting chunk of size 6 to the root (Fig. 4.7c). Finally, for $\mu(a,b) = \min(a,b)$ the optimal plan is to traverse the whole graph with the smallest chunk, merging larger ones along the way (Fig. 4.7d). Thus, even in a simple example the aggregation plan can change drastically depending on μ. We get the following optimization problem.

Problem 4.2.1 (CAM — compute-aggregate minimization)**.** *Given an undirected connected graph $G = (V, E)$, cost function c, a target vertex t, a set of initial data chunks C, and an aggregation size function μ, the* CAM[μ] *problem is to find an aggregation plan P such that* cost(P) *is minimized.*

Interestingly, unless the aggregation size function is well-behaved and at the same time there are constraints on the graph structure, there is not much we can do in the worst case.

Theorem 4.2.1. *Unless $P = NP$, there is no polynomial time constant approximation algorithm for* CAM *without associativity constraint on μ even if G is restricted to two vertices.*

There exist different types of big data applications, a large variation in datacenter network topologies, and countless data distributions, all of which collectively define constraints for a compute-aggregate task. Handling each and every variation of these constraints separately does not scale, so a generalized decision procedure should be used to construct an aggregation plan. In this section, we present such a procedure and show worst-case guarantees for every choice.

Intuitively, stricter constraints may lead to better decisions, both in terms of the *cost of an aggregation plan*, which is our primary objective, and in terms of *performance* (running time). For instance, if the network graph is a tree, it might be possible to construct an aggregation plan in linear time. Similarly, a better algorithm (in terms of the resulting cost) can be chosen under certain constraints on the aggregation size function. Thus, a possible solution may have to account for *network topology*, *aggregation size function*, and *initial chunk distribution*. Chunk distribution varies from one problem instance to another, and is unlikely to provide useful information since it is expected to be roughly uniform (big data storage systems try to achieve even load distribution). Although there exist common network topologies, such as *hypercube*, *fat-tree*, or *jellyfish*, there are plenty of variations and exceptions.

We mostly consider the aggregation size function, with only two special cases. First, a tree is a topology that is both widespread and has a high potential for better algorithmic solutions; we call the CAM problem where G is a tree TCAM (tree CAM). Second, sometimes it is reasonable to limit aggregation to only those nodes that contain data chunks initially, either for security reasons or due to the need for additional resource provisioning on intermediate nodes that may significantly increase latency, while nodes with initial chunks usually already have computing resources for a preprocessing stage. If either security or provisioning impose the aforementioned restriction then a network graph G can be reduced to a complete graph over the nodes that contain chunks, and we call this special case CCAM (complete CAM).

We have summarized the theoretical results presented in the works [4, 21, 30] in Table 4.1. Cells in the table show approximation ratios for various cases of the

	CAM	TCAM	CCAM
Arbitrary μ (general case)	$\alpha\frac{W_C[\mu]}{w_C[\mu]}$ *[4, Thm. 3]*	$\frac{W_C[\mu]}{w_C[\mu]}$ *Thm. 4.2.4*	$\frac{W_C[\mu]}{w_C[\mu]}$ *Thm. 4.2.4*
$\mu(a,b) \leq \min(a,b)$	∞ *Thm. 4.2.5*	∞ *Thm. 4.2.5*	∞ *Thm. 4.2.5*
$\mu(a,b) = \min(a,b)$	2α *[4, Thm. 13]*	1 *Thm. 4.2.6*	2 *[4, Cor. 14]*
$\min(a,b) \leq \mu(a,b) \leq \max(a,b)$	$\alpha\frac{W_C}{w_C}$ *[4, Thm. 9]*	$\frac{W_C}{w_C}$ *[4, Thm. 10]*	$\frac{W_C}{w_C}$ *[4, Thm. 10]*
$\mu(a,b) = \max(a,b)$	4α *Thm. 4.2.7*	1 *[4, Thm. 12]*	4α *Thm. 4.2.7*
$\max(a,b) \leq \mu(a,b) \leq a+b$	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$ *[4, Thm. 11]*	1 *[4, Thm. 12]*	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$ *[4, Thm. 11]*
$\mu(a,b) \geq a+b$	1 *[4, Thm. 8]*	1 *[4, Thm. 8]*	1 *[4, Thm. 8]*

Table 4.1: A taxonomy of our theoretical results in relation to the aggregation size function μ and specific problem (CAM, TCAM, CCAM). Numbers show approximation factors of polynomial algorithms presented in the corresponding theorems, and α denotes the approximation factor for a polynomial algorithm solving the MStT problem.

aggregation size function and μ and various problem settings (CAM, TCAM, and CCAM) together with references to specific theorems and corollaries where these results are proven below. Rows of Table 4.1 correspond to various special cases of μ, ranging from the slowest to the fastest growing functions, and columns correspond to the three problem variations. In the table, α denotes the approximation factor for the polynomial algorithm for the minimum Steiner tree problem (MStT) used as a subroutine in our algorithms. An approximation ratio of 1 means that there is a polynomial-time optimal algorithm for this case, and ∞ means that there can be no constant approximation factor achieved by a polynomial algorithm unless P=NP. As a visualization, in order to simplify the two-dimensional structure of our result, we have also presented them in three figures corresponding to the three cases of topologies we consider: Fig. 4.8 shows the results for

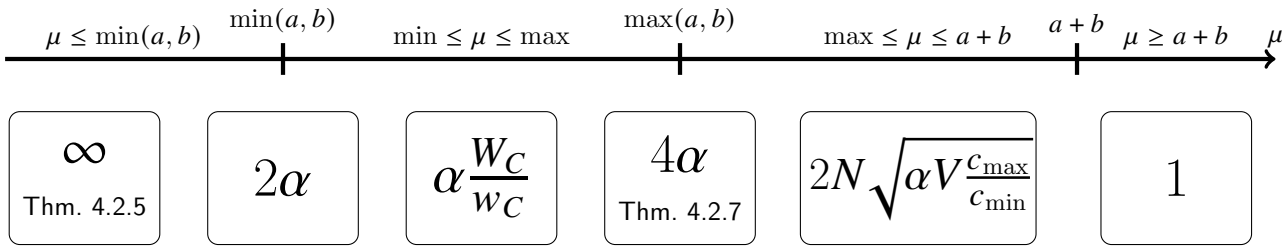


Figure 4.8: A visual summary of our theoretical results for the CAM problem.

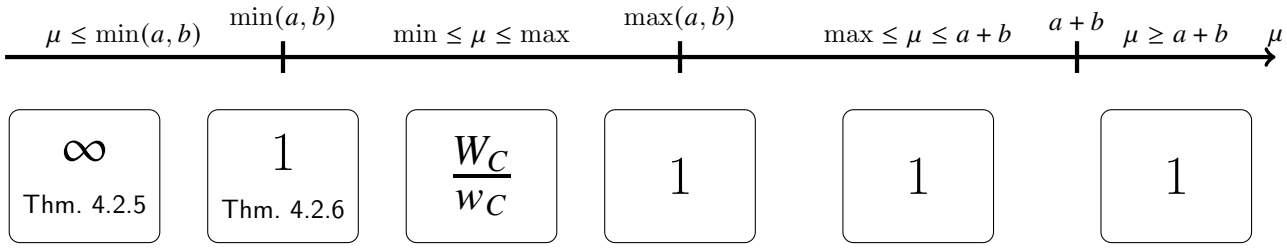


Figure 4.9: A visual summary of our theoretical results for the TCAM problem.

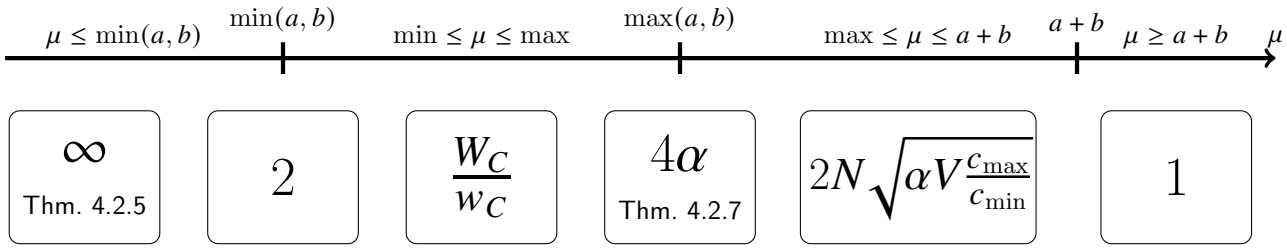


Figure 4.10: A visual summary of our theoretical results for the CCAM problem.

CAM, Fig. 4.9 shows approximation factors for TCAM, and Fig. 4.10 for CCAM. On each figure, the horizontal axis corresponds to how fast μ grows, and each rectangular block refers to a result in the form of an approximation ratio for the corresponding problem and a reference to a specific theorem proving it for the corresponding algorithm.

First, we assume no constraint on the behavior of the aggregation size function μ. To devise a meaningful strategy in this general case, we begin by considering a simpler setting where all chunks have size x, and a merge result of two chunks has the same size as one of the chunks, i.e., $\mu(x,x) = x$. In this case, when paths of two chunks intersect it is always better to merge at the intersection. Thus an optimal aggregation plan always proceeds along a tree subgraph of G, and the weight of that tree multiplied by x equals the aggregation plan cost since μ does not change weights. Thus, the problem reduces to finding a minimum weight tree that connects a given set of vertices, which is a well-studied minimum Steiner tree problem [120], MStT, that has many constant approximation algorithms. This

reduction and the associated results are not all part of the dissertation and can be found in [4, 21, 30].

We note two special cases.

Theorem 4.2.2. *If every vertex in G contains a data chunk, then* MStT *can be solved exactly in polynomial time.*

Theorem 4.2.3. *If G is a tree, then* MStT *can be solved exactly in polynomial time.*

Theorem 4.2.2 and Theorem 4.2.3 essentially say that in these special cases we have 1-approximation algorithms for MStT.

Theorem 4.2.4. *There exist polynomial algorithms that solve* CCAM$[\mu]$ *and* TCAM$[\mu]$ *on a set of chunks C with approximation factor $\frac{W_C[\mu]}{w_C[\mu]}$.*

However, for many μ, including important ones (e.g., set union), Theorem 4.2.4 provides rather weak approximations; in particular, we would like to have approximation ratios independent of chunk sizes and specific values of μ since in practice $\frac{W_C}{w_C}$ may be very high. Unfortunately, it is impossible even for a restricted class of functions μ that reduce the weights, that is, for functions smaller than min.

Theorem 4.2.5. *There exists an aggregation size function μ such that $\forall a, b$ $\mu(a,b) \leq \min(a,b)$, and no polynomial time constant approximation algorithm for* CCAM$[\mu]$ *or* TCAM$[\mu]$ *exists unless $P = NP$.*

Depending on the application, the value of μ may be known to lie in a certain range. For example, if **aggr** represents *set union* then $\mu(x,y) \in [\max\{x,y\}, x+y]$, and if **aggr** represents *outer join* then $\mu(x,y)$ is likely to be always larger than $x+y$. We show a taxonomy of algorithms for different μ; the goal of this subsection is to find a decomposition of a general case into subranges that allow for better worst-case guarantees. These results, summarized in Table 4.1, are not part of this dissertation and are shown in the works [4, 21, 30].

We present several results that deal with specific important aggregation functions.

Theorem 4.2.6. *There exists an optimal algorithm for* TCAM[min] *running in polynomial time.*

Theorem 4.2.7. *If there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial 4α-approximate algorithm for* CAM[max]*, which we call* `RECH_MStTMax`*.*

Theorem 4.2.8. *If there exists $a > 0$ such that $\mu(a, a) = a$ then* CAM[μ] *is NP-hard and does not have less than $\frac{19}{18}$-approximate polynomial algorithms even if all edge weights are equal, unless P=NP.*

All of the above results have been supported by extensive simulation studies that validate the performance of proposed algorithms. Results of these simulation studies, as well as proofs of all the above theorems, are shown in the works [4, 21, 23, 30].

4.3 Efficient demand assignment in multi-connected microgrids

In this section, we consider a very different application of competitive analysis that, however, employs methods similar to Chapter 2. The focus of this section is on efficient demand satisfaction in the context of multi-connected microgrids, where a demand can be met by different generation resources: local, nearby, or on a regional grid that defines a *load balancing vector* for each demand that determines which generators are connected/available to satisfy this demand. In the simplest case, the load balancing vector of each demand consists of a single generator, so each input generator can be scheduled independently. In this setting the problem reduces to the well-known two-dimensional strip packing problem that has been extensively studied in literature [158]. In our model, each demand may have a load balancing vector with several inputs, which makes the problem more complex.

Demands can be *elastic* and *non-elastic* [125]; a demand is non-elastic if its scheduling should start immediately upon arrival. A demand is *non-preemptive* if it cannot be preempted until completion once it has begun servicing, and *preemptive* otherwise. When generation capacities are insufficient to satisfy all demands some non-preemptive demands should be delayed, which leads us to the need for switching of electricity. We concentrate on scheduling elastic non-preemptive demands since

(1) they encompass an important class,

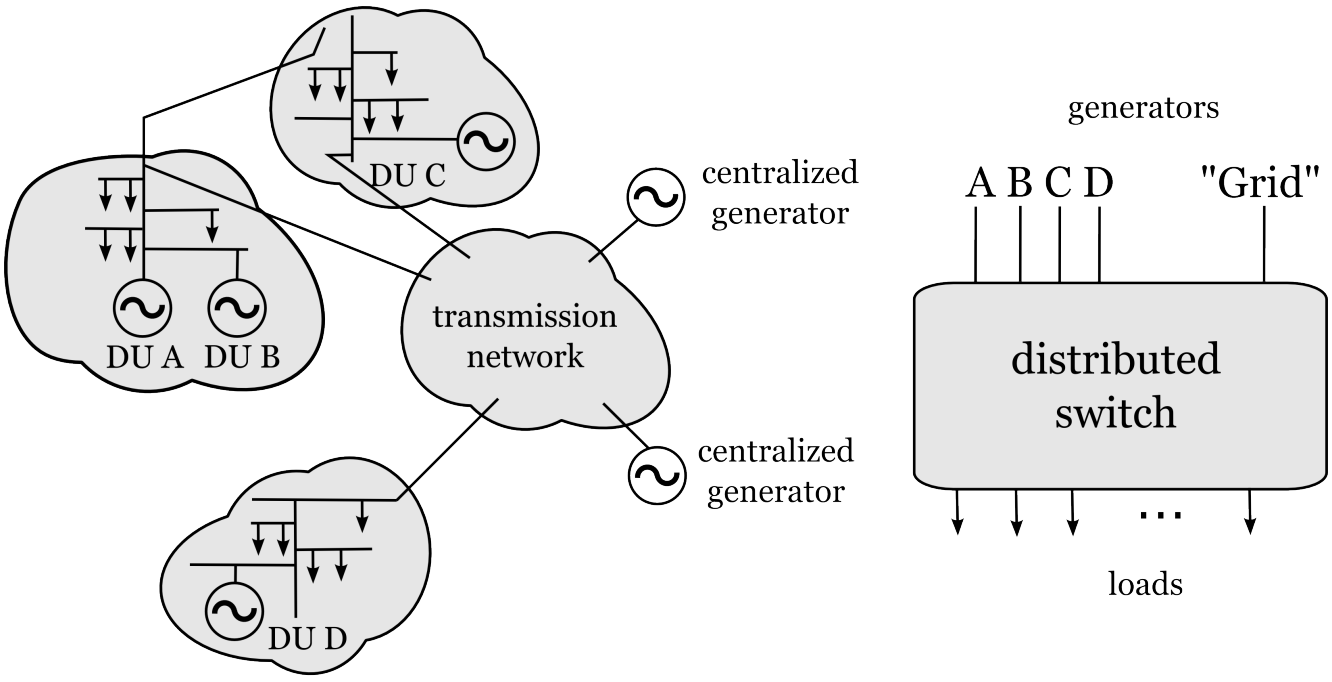
(2) with higher resolution of scheduling non-elastic demands may become elastic, and

(3) preemptive demands can be used to complement a schedule to reuse resources not utilized by non-preemptive demands.

The concept of "packetized" electricity is not new [43, 181]. A natural representation of a demand by power level and duration is isomorphic to a data packet represented by length (duration) and CPU processing requirement (power level). Under some simplifying assumptions, the abstract problem of meeting time-limited loads (that require a certain power for a certain time) from a set of generation resources with distributed switches is similar to assigning packets of a certain length arriving to a set of output ports through a rearrangeable optical switch [129,204]. Each packet corresponds to a demand, each input port to a generation resource, and each output port to a load; in fact, electricity switches with packets of electricity have already been designed [204].

When a central grid has limited capacity, optimal reuse of the underlying infrastructure becomes more important than cost savings for individual consumers. Pricing methods that try to move generation peak are only a tool to achieve better utilization of the underlying central grid infrastructure [73, 164]. For scheduling with "cost constraints", input ports with higher costs can be excluded from the load balancing vector, and scheduling with cost constraints for this microgrid architecture can be reduced to optimizing the makespan or some other objective on a given set of demands with modified load balancing vectors. Moreover, cost minimization implies waiting for the cheapest source, which may be unacceptable even for non-preemptive elastic demands. Therefore, we are especially concerned with minimizing the delay in satisfying a set of resource demands. We assume that there is a non-negligible penalty, called *configuration overhead*, of V time slots to change the configuration in a switching system; note that V may impact a scheduling decision. We also consider an additional objective: minimizing the total number of configurations. Among other things, this reduces the wear and tear of switching equipment that directly depends on the number of necessary switching operations. We extend past work in demand assignments in rearrangable optical switches [129] to compute lower and upper bounds on the minimum number of rearrangements needed to meet a set of demands and show a relation between objective functions.

We model a set of multi-connected microgrids with a switching system $(I, \mathcal{D})$ that consists of a set of inputs I (generators) with port capacities c_i (their nominal



(a) physical connectivity (b) logical model

Figure 4.11: The physical infrastructure in (a) is represented logically in (b). Note that "Grid" represents all centralized generators available through the transmission network. The "distributed switch" refers to the set of switches in microgrids that determines how loads are matched to generators.

power) and a set of demands $\mathcal{D}$ that are to be scheduled; a demand d is characterized by its length $l(d)$ (how long it lasts), width $w(d)$ (power level), and a *load balancing vector* $v(d)$ that contains the set of input ports available to process d (set of generators that can meet this demand). We assume that $(I, \mathcal{D})$ is a directed acyclic bipartite graph. Scheduling proceeds in batches. While the previous batch is served, we complete a preparation phase for the next batch. We assume that demands can be delayed at least until the end of the preparation phase. A preparation phase is divided into *demand prediction*, when sub-demands are predicted for the next batch, *auction*, when each demand receives a load balancing vector, and *scheduling*, when the desired objective function is optimized. When a batch has finished scheduling, the resulting schedule is served.

Time is discrete; L and l are respectively the longest and shortest length in time slots among all demands. If a demand d is assigned to input i at time t, d uses $w(d)$ bandwidth of port i during the time interval $[t, t + l(d) - 1]$. A *schedule* P is a sequence of *configurations*, partial mappings of demands to inputs that have to satisfy constraints imposed by port capacities and load balancing vectors. The *length* of a configuration C is defined by the longest demand scheduled

during C; a configuration can be represented by a vector where each element is a set of demands assigned to the corresponding input port. We use $\in$ liberally, e.g., a demand can be said to belong to a configuration (which is formally a set of demand-input pairs). We denote by $\mathcal{D}_i^t$ the set of demands that have the first (universally available) and the ith ports enabled in the load balancing vector at time t; by n_i^t, the number of demands in $\mathcal{D}_i^t$ at time t. We also denote by $k(\mathcal{D}_i^t) = \left(\sum_{j=1}^{n_i^t} w_2(j)\right)/c_i$ the total "normalized load" for each port, that is, sum of all widths $w_i(j)$ of demands from $\mathcal{D}_i^t$ divided by the ith port capacity c_i. We omit the superscript t when time is clear from context. W.l.o.g. we assume that in the beginning $k(\mathcal{D}_2) \geq k(\mathcal{D}_3) \geq \ldots \geq k(\mathcal{D}_I)$.

To compare the performance of different policies, we use worst-case competitive analysis. An algorithm A has *approximation ratio* α (is α-*approximate*) with respect to some (minimized) objective function if for every input $(\mathcal{D}, I)$, A produces a schedule with objective function value at most α times greater than the optimal one.

We concentrate on the practically interesting case when each demand can be met by two generators, and one of the generators, corresponding to the central grid, is shared among all demands. This situation arises naturally if local distribution networks, each covering its own region, are supplemented by a central grid. In this case, the problem reduces to reusing the central grid (of limited capacity) in the most efficient manner.

We consider simple policies amenable to efficient implementation; such policies can also scale well. A general scheme for such policies is presented in Algorithm 16. Given a set of demands $\mathcal{D}$ and a set of input ports I with capacities c_i, $i \in I$, a greedy scheduling policy populates each configuration by greedily choosing the next demand to process. Once there are no more ports (generators) available for existing demands, the current configuration is finalized and a new one begins.

The ChoosePortDemand procedure is the heart of Algorithm 16; it takes the current state (remaining demands and leftover capacities) as input and outputs the input-demand pair (i, d) for the next assignment. Various algorithms considered in this section differ in their ChoosePortDemand procedures. The obvious general algorithm for this case is SG, which stands for "Shared Greedy" (Algorithm 17): fill the capacities of every port except the first, then choose demands for the first port. Different algorithms may differ in choosing a demand

Algorithm 16: GreedySchedulingPolicy($\mathcal{D}$, I)

```
1 begin
2     D := 𝒟, C := ∅
3     while D ≠ ∅ do
4         Start new configuration C := ∅, I' := I
5         while there are available ports and demands do
6             (i, d) := ChoosePortDemand(D, I')
7             C := C ∪ {(i, d)}, c'_i := c'_i − w(d), D := D \ {d}
8         C := C ∪ {C}, D := D \ {d | d ∈ C}
9     return C
```

Algorithm 17: SG

```
1 Function ChoosePortDemand{𝒟_i}_i, I:
2     for i := 2 to I do
3         if c_i > w(d) for some d ∈ 𝒟_i then
4             return (i, ChooseDemand(𝒟_i, c_i));
5     return (1, ChooseFirst({𝒟_i}_i, I)).
```

for a single port (ChooseDemand procedure) and in choosing which demand to send to the first port for extra processing (ChooseFirst procedure). The basic tradeoff here is the balance between minimizing the number of configurations and minimizing their total length (duration). In this regard, we define two algorithms from the SG family: SLD ("Shared Longest Demand", Algorithm 18) and SLP ("Shared Longest Port", Algorithm 19). SLD chooses the longest available demand for the current configuration; it does not matter which one for the ChooseDemand procedure, and the ChooseFirst procedure SLD splits ties with the largest port heuristic (maximal $k(\mathcal{D}_i)$). SLP, on the other hand, chooses for the ChooseFirst procedure a demand from the port with maximal normalized load $k(\mathcal{D}_i)$; for splitting ties and ChooseDemand, it uses the longest demand. Note that in all cases, maximization and minimization is done over demands that can fit into available inputs, i.e., the constraint $w(d) < c_i$ is always present but omitted for clarity.

Four important parameters define the behaviour of a scheduling policy:

(1) input port capacities,

Algorithm 18: SLD

Function ChooseDemand($\mathcal{D}_i, c_i$):
    return $\arg\max\{l(d) \mid d \in \mathcal{D}_i\}$.
Function ChooseFirst($\mathcal{D} = \{\mathcal{D}_i\}_i, I$):
    return $\arg\max_{d \in D} l(d)$;

Algorithm 19: SLP

Function ChooseDemand($\mathcal{D}_i, c_i$):
    return $\arg\max\{l(d) \mid d \in \mathcal{D}_i\}$
Function ChooseFirst($\mathcal{D} = \{\mathcal{D}_i\}_i, I$):
    return $d \in \arg\max_{\mathcal{D}_j} k(\mathcal{D}_j)$

(2) demand lengths,

(3) demand widths,

(4) "normalized load";

all of them also affect the schedule length objective. Demand length has no impact on the number of configurations but can significantly influence the total length as L/l grows. The impact of input capacity is interesting even for unit-sized demand widths; the number of configurations is related to utilization of input capacities. The underlying problem is NP-hard: even optimal scheduling with a single port encompasses the knapsack problem. In our study, we explore the impact of each parameter on the performance of scheduling policies. We begin with a relation between the two objective functions.

Theorem 4.3.1. *For an input with maximal demand length L and minimal demand length l:*

(1) an SG *algorithm with approximation ratio $\leq \alpha$ with respect to configurations has approximation ratio $\leq \frac{\alpha L h}{L+(l-1)h}$ with respect to length, where h is the optimal number of configurations;*

(2) an SG *algorithm with approximation ratio $\leq \alpha$ with respect to length has approximation ratio $\leq \frac{L}{t}\left(1 + \frac{\alpha t - L}{l}\right)$ with respect to configurations, where t is the optimal length.*

We begin with the case of unit capacities: all input ports have $c_i = 1$ and all demands have $w(d) = 1$; in this case $k(\mathcal{D}_i) = |\mathcal{D}_i| = n_i$. Now the ith input port is fully utilized if $\mathcal{D}_i$ is not empty at the end of a configuration, and algorithms differ in how they reuse the shared port. Intuitively, assigning demands from $\mathcal{D}_i$ with currently longest n_i should optimize the number of configurations, and indeed, we will show that SLP has the optimal number of configurations even in a more general setting. Furthermore, it turns out that we can show an upper bound on the approximation factor of *any* algorithm following the SG general scheme.

Theorem 4.3.2. *For a single shared port system with unit capacities, any scheduling policy* ALG *from the* SG *family has approximation ratio at most* $\frac{3}{2}$ *w.r.t. configurations.*

We will see that for some algorithms, this bound is tight. Theorems 4.3.2 and 4.3.1 also imply a general upper bound of $\frac{3Lh}{2(L+(l-1)h)} \leq \frac{3L}{2l}$ on the approximation ratio w.r.t. length. Since OPT also falls into the SG family, we cannot show nontrivial general lower bounds; thus, we turn to lower bounds for specific algorithms. The tradeoff is between the number of configurations (optimal for SLP, worse for SLD) and their total length (good for SLD, worse for SLP).

Theorem 4.3.3. *For a system with unit capacities and I ports,* SLD *has approximation ratio at least* $\left(\frac{3}{2} - 2^{-(I-1)}\right)$ *w.r.t. configurations and length.*

Note that this approximation ratio tends to $\frac{3}{2}$ as $I \to \infty$, so the bound from Theorem 4.3.2 is tight for SLD.

Next we turn to the case of heterogeneous input port capacities (but still unit demand widths). The first interesting question is whether it is possible to break the upper bound of SG stated in Theorem 4.3.2 in case of heterogeneous input capacities. For this purpose, we introduce additional notation. Suppose that OPT schedules all demands over h configurations. We classify all input ports except the shared one into two groups: $\mathcal{I}_1$ and $\mathcal{I}_2$. $\mathcal{I}_1$ contains all input ports such that $k(\mathcal{D}_i) > h$, $2 \leq i \leq I$; $\mathcal{I}_2$, all other ports (excluding the first). We prove that SG performance depends on the relationship between capacity of the shared port and the capacities of $\mathcal{I}_1$ and $\mathcal{I}_2$. As a special case of SG we consider SLD. Moreover, at least for the case when $c_1 > c(\mathcal{I}_1)$ and $c_1 \leq c(\mathcal{I}_2)$ and non-unit-sized capacity constraints, the upper bound of $\frac{3}{2}$ for unit-capacity constraints can be broken. Intuitively, SG can mistakenly reuse the first port for the demands of $\mathcal{D}(\mathcal{I}_2)$, so that it cannot reuse the first port for the demands of $\mathcal{D}(\mathcal{I}_1)$ enough times.

Theorem 4.3.4. *For a system with capacities* $c_1 = 2c(\mathcal{I}_1)$ *and* $c(\mathcal{I}_2) = 2c_1$*, the number of configurations in* SLD*'s schedule is at least* $\frac{5}{3}$ *times more than optimal.*

The result stated in Theorem 4.3.4 demonstrates that generalizing the results for unit capacity constraints above is non-trivial even for unit-sized demand width. The following theorem specifies an upper bound on a number of configurations of SG in the case of non-unit-capacity constraints. Recall that we still consider unit-sized width of demands.

Theorem 4.3.5. *The number of configurations in the schedule of* SG *is at most twice more than in the optimal schedule if one of the following conditions holds:*

(1) $c_1 \leq c(\mathcal{I}_1)$*;*

(2) $c_1 > c(\mathcal{I}_1)$ *and* $c_1 > c(\mathcal{I}_2)$*.*

We show that Algorithm 19 (SLP) that prioritizes the $k(\mathcal{D}_i)$ characteristic is optimal.

Theorem 4.3.6. SLP *has the optimal number of configurations for the case of heterogeneous input capacities and unit-sized demands width.*

SLD is a special case of SG, so Theorems 4.3.5 and 4.3.4 imply the following corollaries.

Corollary 4.3.1. *For a system with heterogeneous input capacities,* SLD*'s schedule is at most twice longer than the optimal schedule if one of the following conditions holds:*

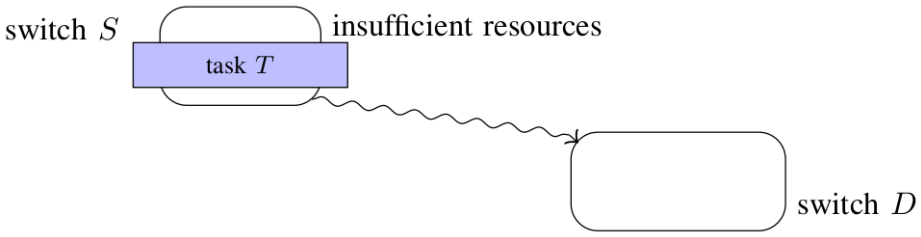
(1) $c_1 \leq c(\mathcal{I}_1)$*;*

(2) $c_1 > c(\mathcal{I}_1)$ *and* $c_1 > c(\mathcal{I}_2)$*.*

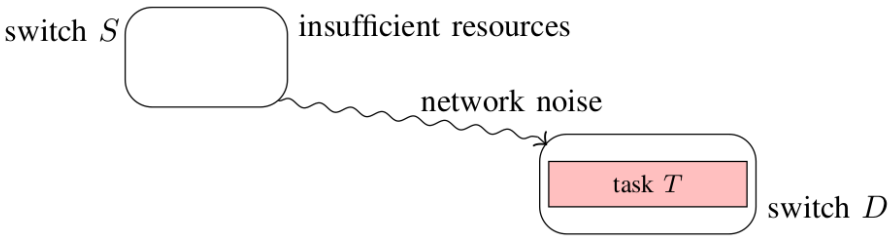
The following corollary obviously follows since SLP is optimal; a better bound remains an interesting open problem.

Corollary 4.3.2. *Under heterogeneous input capacities,* SLP *schedule is at most* $\frac{L}{l}$ *longer than* OPT *schedule.*

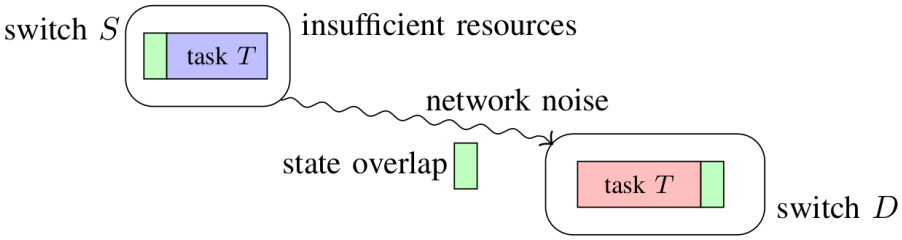
All of the above results have been supported by a simulation study that validate the performance of proposed algorithms. Results of this simulation study, as well as proofs of all the above theorems, are shown in the works [40, 41].



(a) A designated switch S that has insufficient resources.



(b) Relocation of the task to switch D and network noise.



(c) Robust distributed execution by RoDiC at both S and D.

Figure 4.12: Execution of a stateful monitoring task.

4.4 Robust counting of traffic flows

Even simply computing the sizes of all flows at a network element can overload scarce data-plane resources when the number of flows becomes large. Figure 4.12a depicts an example where task T overwhelms the resources available at switch S. A prominent body of related work explores approximate solutions that give up accuracy of flow-size computation to reduce memory required in a single element. Such approximate solutions include estimators [90, 117, 189, 194] and sketches: CM [87], CU [103], Pyramid Sketch [208], UnivMon [157], and Elastic Sketch [206]. A trace-driven evaluation of CEDAR [194], SAC [189], and DISCO [117] shows their average relative errors in excess of 12% for 8-bit per-flow estimators [194]. The average relative error of Elastic Sketch, one of the most advanced current proposals, can be equal to 4 even when using 0.2 MB to represent 110K flows, i.e., around 15 bits per flow, as Figure 9a shows in [206]. Such accuracy might be insufficiently low.

Moving the entire execution of a monitoring task from a designated overloaded element to another element on the flow path can endanger the result accuracy due to *network noise* in the form of packet reordering and loss. As depicted in Figure 4.12b, network noise might prevent switch D from executing task T as accurately as at switch S because the network might lose or reorder packets while delivering them from S to D. Such imprecision can arise regardless of whether the two network elements communicate over an unreliable or reliable transport protocol. Hence, the task execution should involve the originally designated element to at least some extent and be made resilient to network noise.

We propose Robust Distributed Computation (RoDiC), a general technique that executes a monitoring task in multiple network elements correctly despite network noise on the paths between the elements. For stateful tasks, the proposed technique not only maintains state in different elements but also resiliently communicates some state between these elements. RoDiC supports open-loop monitoring of flow metrics without introducing any control packets. Data packets of each monitored flow piggyback few (e.g., three) control bits to communicate flow state. This feature makes RoDiC applicable to unidirectional communications.

While RoDiC is a general method that can be instantiated differently for different metrics, its two main design principles are as follows:

(1) *packet grouping:* RoDiC partitions each flow into groups of consecutive packets and distributes state so that a subsequent network element deals with the metric monitoring at the coarser granularity of packet groups;

(2) *ptate overlap:* RoDiC maintains distributed state across multiple network elements in overlapping chunks; sync bits in data packets communicate the chunk overlap to keep the distributed state consistent despite network noise.

Illustrating the RoDiC approach and its state-overlap principle, Figure 4.12c shows robust distributed execution of task T at switches S and D. The state chunks at S and D overlap. S communicates the state overlap to D in-band.

Flow f traverses a network via a series of switches (network elements). At the source switch S, the flow consists of $|f|$ network-layer packets numbered consecutively from 0 to $|f| - 1$. Because we consider each flow on the network layer, the original transmission and retransmission of a data segment by the transport or application layers appear in the flow as distinct packets. Some of the subsequent

Algorithm 20: Computation of the flow size on two switches in the consecutive-loss model

```
1  begin
2      Function Initialization:
3          c_1 ← 0; c_2 ← 0
4      Function Source (j):
5          h[j] ← ⌊c_1 / 2^{n−t}⌋; c_1 ← (c_1 + 1) mod 2^n
6      Function Destination (j):
7          diff ← Overshoot(h[j]);
8          if 1 ≤ diff ≤ 2^{t−1} then
9              c_2 ← c_2 + diff
10     Function Termination:
11         c_2 ← c_2 + Overshoot(⌊c_1 / 2^{n−t}⌋);
12         |f| ← c_2 · 2^{n−t} + c_1 mod 2^{n−t}
13     Function Overshoot (g):
14         return (g − c_2 mod 2^t + 2^t) mod 2^t
```

switches assist switch S in distributedly computing a metric of the flow. Whereas the network might reorder or lose packets, the objective is to design a distributed algorithm that correctly computes the metric for all monitored flows despite such network noise. Each participating switch s_m uses n_m bits to maintain its chunk c_m of the distributed state. Switch s_m piggybacks at most t_m *sync bits* on each data packet to communicate control information to the subsequent participating switch. We refer to n_1 and t_1 as simply n and t respectively.

First, we consider a *consecutive-loss model* that constrains network noise via two parameters. *Loss parameter* L is the limit on *consecutive* packet losses, i.e., switch D receives at least one of packets i through $i + L$. *Reordering parameter* R captures the maximal distance of packet reordering, i.e., switch D can receive packet i before packet j only if $i \leq j + R$. We narrow down the feasibility limits on resilience in the consecutive-loss model as follows.

Theorem 4.4.1. *A deterministic RoDiC algorithm cannot guarantee correct distributed computation of the flow size on two switches if* $L \geq 1$ *packet and* $R \geq 2^{n-1} + 1$ *packets.*

While Theorem 4.4.1 and other results (see the works [2, 15, 29]) detect feasibility limits for robust distributed flow-size computation, we now derive a RoDiC

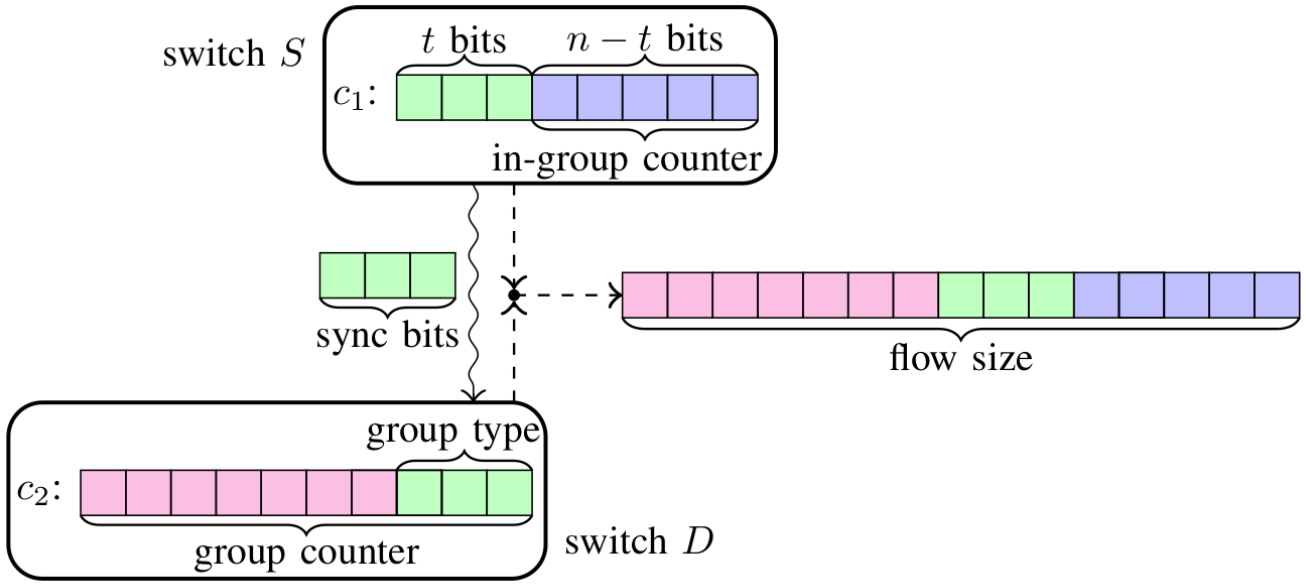


Figure 4.13: Applying the RoDiC principles of packet grouping and state overlap to the problem of flow-size computation.

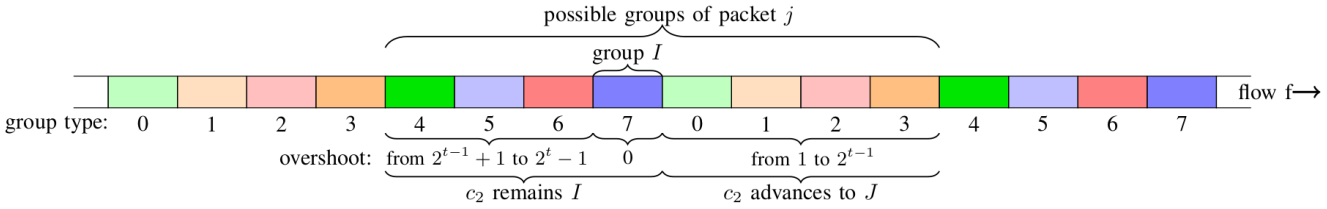
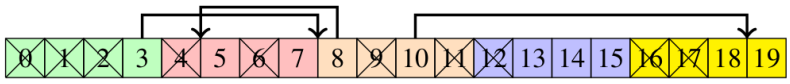


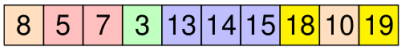
Figure 4.14: Operation of the DESTINATION procedure in algorithm 20 when $t = 3$, $c_2 = I$, and packet j arrives to D from group J.

algorithm that approaches these limits. Algorithm 20 follows both principles of packet grouping and state overlap as depicted in Figure 4.13. The t most significant bits of c_1 and t least significant bits of c_2 form a t-bit overlap between the counting states at switches S and D. These t bits specify the type of a group and numbers the group types sequentially from 0 to $2^t - 1$. The group size becomes $n - t$ bits. Switch S uses its own $n - t$ least significant bits of c_1 to count packets within a group. Counter c_2, including its t least significant bits which capture the group type, tracks the largest group number among the groups that have delivered a packet to D.

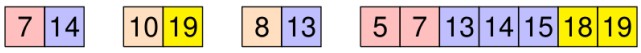
In Algorithm 20, S copies the group type from the t most significant bits of c_1 into sync bits $h[j]$ of each packet j and then increments c_1 to count this packet. Upon receiving packet j, D uses arithmetic modulo 2^t to compute by how many groups the group type in $h[j]$ overshoots the group type in the t least significant bits of c_2. For example, with $t = 3$ bits, there are eight group types numbered



(a) The flow at switch S and network noise.



(b) The packet sequence received by switch D.



(c) Four instances of a span$(2, 2)$.

Figure 4.15: Span examples for a flow of 20 packets.

from 0 to 7, and type 4 overshoots type 7 by 5 groups. In the absence of network noise, the overshoot is either 0 or 1 group. With network noise, the overshoot can become as large as $2^t - 1$ groups. When the overshoot is 0 or between $2^{t-1} + 1$ and $2^t - 1$ groups, D ignores packet j because Algorithm 20 assumes that c_2 has already accounted for the group of packet j. When the overshoot is between 1 and 2^{t-1} groups, the algorithm assumes that packet j arrives from a later, yet unaccounted group, and D advances c_2 by the amount of the overshoot. For instance, if t equals 3 bits, the group type in $h[j]$ is 1, and $c_2 = 7$, the overshoot is 2 groups, and D advances c_2 to 9. When the flow terminates, Algorithm 20 calculates by how many groups the group type in c_1 overshoots the group type in c_2. Then, the algorithm advances c_2 by the amount of the overshoot and concatenates all bits of c_2 and $n - t$ own bits of c_1 to compute the flow size.

Theorem 4.4.2. *Algorithm 20 is guaranteed to compute the flow size on two switches correctly if*

$$R \leq 2^{n-1} - 2^{n-t} \text{ packets and } L + R \leq 2^{n-1} - 1 \text{ packets.}$$

The consecutive-loss model represents maximum network noise in terms of its local impact on a flow. Next, we consider an alternative representation linked to global properties of the packet sequence received by switch D from the flow. Inheriting the constraint of reordering by at most R packets, the new model does not impose an explicit limit on consecutive packet loss and instead characterizes the received packet sequence via a *span* notion. Defined in regard to the partition of flow f at switch S into groups of 2^k consecutive packets, a span(γ, k) refers

Algorithm 21: Computation of the flow size on two switches in the span-based model.

```
1 Function Destination (j):
2     diff ← Overshoot(h[j])
3     if 1 ≤ diff ≤ γ then c2 ← c2 + diff
```

to a subsequence of the packets at S that arrive to D in order, omit at most γ full groups at the beginning of the flow, and miss at most $\gamma - 1$ full consecutive groups after any packet in this subsequence. Figure 4.15 depicts four instances of a span$(2, 2)$ for a flow that consists of 20 packets. The *span-based model* constrains network noise via reordering parameter R and span existence.

A span(γ, k) is also a span$(\gamma + 1, k)$. Hence, infeasibility of assuredly correct flow-size computation despite existence of a span(γ, k) implies such infeasibility with existing spans that have larger γ values. It also implies that enabling of assuredly correct computation necessitates existence of a span with a smaller γ value. Without packet reordering, the span-based model of maximum network noise comes with the following feasibility limits.

It turns out that the existence of a span(γ, k) is insufficient for a deterministic RoDiC algorithm to guarantee correct distributed computation of the flow size if $R \geq 2^n - \gamma \cdot 2^k + 1$ packets (a result proven in [2, 15]).

Parameter γ controls a trade-off between span existence and reordering constraint. Increasing t relaxes the feasibility limits in both γ and R dimensions. With $\gamma = 2^{t-1}$ groups, the reordering constraint for the span-based model is $R \geq 2^{n-1} + 1$ packets, i.e., the same as in Theorem 4.4.1 for the consecutive-loss model. The span-based model not merely generalizes the consecutive-loss model via parameter γ but characterizes maximum network noise more effectively.

We now present and analyze Algorithm 21 which meets the feasibility limits established above. Algorithm 21 uses $k = n - t$ and differs from algorithm 20 only in its updated DESTINATION procedure that advances c_2 if the overshoot is between 1 and γ groups.

Theorem 4.4.3. *When γ is at most $2^t - 1$ groups, Algorithm 21 always computes the flow size on two switches correctly if there exists a* span$(\gamma, n - t)$ *and $R \leq 2^n - (\gamma + 1) \cdot 2^{n-t}$ packets.*

We conclude the section by quantitatively illustrating the resilience of Algo-

rithm 21 to network noise provided by Theorem 4.4.3. When $n = 8$ bits, $t = 2$ bits, and $\gamma = 1$ group, Algorithm 21 is guaranteed to compute the flow size correctly if a span(1, 6) exists with reordering by at most 128 packets. With t increased to 3 bits, the computation correctness is assured if a span(1, 5) exists, and reordering is at most by 192 packets. When n and t remain at 8 and 3 bits respectively, and γ increases to 6 groups, the assuredly correct computation comes with existence of a span(6, 5) and reordering by at most 32 packets.

All of the above results have been supported by a simulation study that validate the performance of proposed algorithms. Results of this simulation study, as well as proofs of all the above theorems, are shown in the works [2, 15, 29].

Chapter 5 Conclusion

5.1 Main results

In this section, we provide a brief review of **key results** for this thesis that are presented for defense. All of these results have been presented and considered in detail through Chapters 2, 3, and 4.

5.1.1 Algorithms for buffer management and their analysis

Buffer management with heterogeneous processing for a single queue. For packets that require heterogeneous processing (variable number of processing cycles), we have explored a number of online algorithms in the single queue setting. We prove a lower bound of k on the competitive ratio for NPO (non-push-out FIFO), $2(1-1/B)$ for $k \geq B$, $\frac{2k}{k+1}$ for $k < B$, and $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$ for large k for PO (push-out FIFO). We introduce and study a new class of lazy online algorithms that allows for proving upper bounds on the competitive ratio, proving incomparability of PO and LPO in the worst case, an upper bound of $\max\{1, \ln k\} + 2 + o(1)$ and a lower bound of $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$ for LPO (lazy push-out), and an upper bound $2 + \frac{1}{B}$ for LPO in the special case of two required processing values. Moreover, we generalized these ideas and introduced the family of Semi-FIFO algorithms that includes the family of lazy algorithms; we proved a general upper bound of $\frac{1}{B} \log_{B/(B-1)} k + 3$ for every lazy algorithm, matching lower bound of $(1 + \frac{1}{B} \log_{B/(B-1)} k)$ for LRFIFO, LRevPQ, and RFIFO, upper bound 2 and lower bound $2 - \frac{1}{B} \lceil \frac{B}{k} \rceil$ for LPQ (lazy priority queue), a general lower bound of $\left(1 + \frac{\min\{k,B\}-1}{B}\right)$ for any greedy Semi-FIFO policy, and several incomparability results in the worst case. We have also generalized these results to the β-push-out case, proving an upper bound of $\left(3 + \frac{1}{B} \log_{\frac{\beta B}{\beta B - 1}} k\right) \frac{1-\alpha}{1-\alpha \log_\beta k}$ and a lower bound of $\frac{1-\alpha}{1-\alpha \log_\beta k}$ for copying cost $0 < \alpha < \frac{1}{\log_\beta k}$ for any lazy β-push-out algorithm LA_β. These results have been published in the works [12, 28, 32].

Buffer management with multiple separated queues. For a multi-queued system where each queue accepts packets with a given processing requirement, and the total size of all queue buffers is the same as the buffer size in the single queue architecture, we have introduced and studied a number of online algorithms, in particular the Longest-Queue-First (LQF), Shortest-Queue-First (SQF), Minimal-Queue-First (MQF), MaxQF, Packet-Round-Robin (PRR), Cycle-Round-Robin (CRR) online policies. As for lower bounds, we proved that LQF is at least $\frac{m}{2}$-competitive for $m = \min\{k, B\}$, SQF is at least k-competitive, MQF is at least $\left(1 + \frac{k-1}{2k}\right)$-competitive, MaxQF is at least k-competitive, PRR is at least $\frac{3k(k+2)}{4k+16}$-competitive, and CRR is at least $\frac{k}{H(k)}$-competitive for $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$. The main positive result is an upper bound: MQF is at most 2-competitive. For the special case of two queues, we proved that the competitiveness of MQF is at least $\left(1 + \frac{1+\lfloor \frac{aB-1}{b} \rfloor}{B+\lceil \frac{1}{a}\left(b\lfloor \frac{aB-1}{b} \rfloor+1\right)\rceil}\right)$ and at most $\left(1 + \frac{1+\lfloor \frac{aB-1}{b} \rfloor}{B+\lceil \frac{1}{a}\left(b\lfloor \frac{aB-1}{b} \rfloor+1\right)\rceil}\right)$. We also proved a number of incomparability results and studied the possibilities for simulations between multi-queue and single-queue settings, showing the possibilities and constraints in emulating multi-queue architectures by a single queue. These results have been published in the works [5, 22, 39].

Buffer management with heterogeneous processing and multiple output ports. In this part we consider a shared memory switch where the buffer is shared among all types of traffic but in contrast to the previous single queue model each core processes only one type of traffic. We introduce and consider several online policies: Non-Push-Out-Harmonic-Static-Threshold (NHST), Non-Push-Out-Equal-Static-Threshold (NEST), Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT), Longest-Queue-Drop (LQD), Biggest-Packet-Drop (BPD), and Longest-Work-Drop (LWD). First we consider the case where each queue is restricted to a single processing requirement which is the same for all packets in the queue. For non-push-out policies, we have shown that NHST is $(kZ + o(kZ))$-competitive, NEST is $(n + o(n))$-competitive, and NHDT is at least $\left(\frac{1}{2}\sqrt{k \ln k} - o(\sqrt{k \ln k})\right)$-competitive if B is asymptotically greater than k. For push-out policies, we have shown that LQD is at least $(\sqrt{k} - o(\sqrt{k}))$-competitive, BPD is at least $(\ln k + \gamma)$-competitive for $B \geq \frac{k(k+1)}{2}$, LWD is at least $\left(\frac{4}{3} - \frac{6}{B}\right)$-competitive for $k \geq 6$. The main result here is an upper bound showing that LWD is at most 2-competitive. Next, we consider the model that allows for packets with

heterogeneous processing requirements to be admitted to the same queue. This adds the Biggest Average Drop (BAD) push-out policy. As for lower bounds, we have shown that BPD and BAD are both at least $(n+1)/2$-competitive, LQD is at least $(n/2-o(n))$-competitive for $k \geq n(n-1)$ and $B \geq k+n$, and LWD with FIFO processing order is at least $(\log_{B/n} k)(1-n/B)+1$-competitive. The main result here is the upper bound for LWD: we have shown that it is at most $1+\frac{B}{T+B}$-competitive, where T is the minimal number of packets transmitted between any two consecutive iterations. Third, we have considered another generalization where each unit-sized packet with required work 1 has two parameters: an output port label and an intrinsic value; the objective is to maximize the total transmitted value. This leads to the Minimal-Value-Drop (MVD) and Maximal-Ratio-Drop (MRD) policies. We have shown that MVD is at least $\frac{m-1}{2}$-competitive for $m = \min\{k, B\}$, MRD is at least $\frac{4}{3}$-competitive if each packet's value is equal to its output port label and at least V-competitive in the general case if $n \geq B - V^2 + 1$. These results have been published in the works [9, 11, 27, 31].

Buffer management with heterogeneous processing and varying size. Here, we have considered a general model for the problem where a network processor is required to manage the admission control and scheduling modules of a single bounded size queue, where arriving traffic consists of packets such that each packet has some size (e.g., in bytes) and a processing requirement (in processor cycles). We have considered several different priority queues in this setting: Shortest Remaining Processing Time first (SRPT), Longest-Packet first (LP), and Most-Effective-Packet first (MEP), and greedy push-out (PO) and non-push-out (NPO) algorithms. We have shown that NPO has competitive ratio at least kL and at most $kL\frac{B}{B-L}$ for SRPT-based priorities, while PO has competitive ratio at least L for SRPT-based priorities for $B \geq 2L$. The main results here are upper bounds: first, if $B > 2L$, then PO is at most $\frac{4L-2}{L_a}$-competitive for SRPT-based priorities, where L_a is an average length of packets transmitted by PO. For sufficiently large buffers PO is at most $\frac{(2L-1)(N+1)}{NL_a}$-competitive for SRPT-based priorities, where L_a is an average length between all transmitted packets by PO and $N = \lceil \frac{B-2L+1}{2L-1} \rceil$, PO is at most $(k+3)$-competitive for LP-based priorities on sufficiently big buffers, and PO is $\Omega(\log k)$-competitive for MEP priorities. These results have been published in the works [13, 38].

Buffer management for packets with multiple characteristics. In this part, we again consider a single-queue switch where a buffer of size B is shared among all types of traffic. The difference in this setting is that now we are studying the impact of both packet values and required processing on weighted throughput. We introduce the general notion of a priority queue PQ_f that orders and processes packets according to the function f and consider several natural variations: $\mathrm{PQ}_{-w,v} = \mathrm{PQ}_{-w+v/(V+1)}$ orders packets in the increasing order of their required processing, breaking ties by value; $\mathrm{PQ}_{v,-w} = \mathrm{PQ}_{v-w/(W+1)}$ orders packets in the decreasing order of their value, breaking ties by required processing; $\mathrm{PQ}_{v/w}$ orders packets in the decreasing order of their value-to-work ratio, i.e., it prioritizes packets that yield the best value per one time slot of processing. We show several general lower bounds: for $B = 1$ and $V > \sqrt{W}$, any online algorithm ALG is at least $\sqrt{W}$-competitive; in the two-valued case or if $V \leq \sqrt{W}$ any online algorithm ALG is at least $\min\{V, W/V\}$-competitive for $B = 1$; for arbitrary B and $V > \sqrt{W}$, any online algorithm ALG that preserves FIFO processing and transmission order is at least $\left(\frac{\sqrt{W}}{B} + 1 - \frac{1}{B}\right)$-competitive; in the two-valued case or if $V \leq \sqrt{W}$, any online algorithm ALG with FIFO processing and transmission order is at least $\frac{V+B-1}{W+B-1}$-competitive. We also show general lower bounds specifically for priority queues: for $V > \sqrt{W}$ and any priority function f such that $f(w, v) \in \mathbb{R}$, the algorithm PQ_f is at least $\sqrt{W}$-competitive; in the two-valued case or if $V \leq \sqrt{W}$, algorithm PQ_f is at least $\min\{V, W/V\}$-competitive. We prove several lower bounds for specific algorithms: $\mathrm{PQ}^{\beta}_{-w,v}$ is at least V-competitive both in the case of arbitrary packet values and in the two-valued case; $\mathrm{PQ}^{\beta}_{v/w}$ is at least $\min\{V, W\}$-competitive in the case of arbitrary packet values; in the two-valued case, if $\beta W \geq V$ then $\mathrm{PQ}^{\beta}_{v/w}$ is at least V-competitive; $\mathrm{PQ}^{\beta}_{v,-w}$ is at least $\left(\frac{(V-1)}{V}W - o(1)\right)$-competitive in the case of arbitrary packet values and at least $\left(\frac{W}{V} + o(1)\right)$-competitive in the two-valued case. These results have been published in the works [8, 25].

Buffer management with latency constraints. In this part, we consider a single-queue switch where a buffer is shared among all types of traffic, and we assume that each arriving packet p is branded with three characteristics: (1) the number of required processing cycles (*work*) $w(p) \in \{1, \ldots, W\}$, (2) its processing *value* $v(p) \in \{1, \ldots, V\}$, (3) *slack* $s(p) \in \{1, \ldots, S\}$ that defines how long from the arrival time a packet should be transmitted before it is lost without any gain to the

final objective (weighted throughtput). We again consider mostly priority queues, in particular the following: $\mathrm{PQ}_{-w} = \mathrm{PQ}_{-w+v/(V+1)}$ orders packets in the increasing order of their required processing, breaking ties by value; $\mathrm{PQ}_{v} = \mathrm{PQ}_{v-w/(W+1)}$ orders packets in the decreasing order of their value, breaking ties by required processing; $\mathrm{PQ}_{v/w}$ orders packets in the decreasing order of their value-to-work ratio; PQ_{v/w_0} orders packets in the decreasing order of their value-to-original-work ratio; $\mathrm{PQ}_{v/s}$ and PQ_{v/s_0} order packets in the decreasing order of their value-to-slack and value-to-original-slack ratio respectively. We show a general lower bound: the competitiveness of any deterministic online algorithm A cannot be bounded from above by a constant; moreover, it cannot be bounded from above by a constant even if there is a predefined constant c such that for every packet $s(p) \geq w(p)+c$. We show a number of specific lower bounds: if w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw for some constant c, the competitiveness of priority queue algorithms PQ_{v}, $\mathrm{PQ}_{v/w}$, PQ_{v/w_0}, $\mathrm{PQ}_{s/w}$, PQ_{s/w_0} is at least

$$\frac{1-(\frac{w-2}{w})^{\frac{1}{w}\left\lfloor \log_{\frac{w}{w-2}} v \right\rfloor}}{1-\frac{w-2}{w}}\frac{1}{c},$$

which is non-constant in v and w. For a multiplicative slack constraint, we also show a general lower bound: if w, s, and v are admissible values of work, slack, and value respectively, and s is always greater than cw, then if $w > c$ the competitiveness of any non-preemptive algorithm is at least V. Another result deals with α-preemptive algorithms: the $\mathrm{PQ}_{v/w}$ algorithm with α-preemption has a lower bound on its competitiveness of $\Omega(w)$ for arbitrarily large slacks. These results have been published in the works [3, 19].

Choosing buffer management algorithms with machine learning. We have studied the design principles of a self-adjusting queueing module (QM), an essential building block in every network element, and formalized the choice between candidate policies as a multiarmed bandits problem. We have introduced several assumptions that distinguish between important practical cases: DISJ(Δ), AVGEQ(Δ), LIMIT(Δ), and ADV(Δ). We have proposed algorithms based on the multiarmed bandit formalization and studied their performance under various assumptions. These results have been published in the work [6].

5.1.2 Packet classification algorithms

Exploiting order independence for packet classification. In this part, we have identified classifier properties that guarantee zero cost for additional fields represented by ranges or prefixes transparently to the implementation scheme, proposed a technique to reduce the number of classification fields represented by prefixes or ranges, yielding semantically equivalent classification results, and presented a method to reduce a multi-field classification problem to a classification problem on at most two fields with guaranteed worst-case performance and linear memory. We have introduced the notion of order-independence for filters, classification rules, and classifiers. We have shown that $\mathcal{K}$ with a false-positive check of a single matched rule is a semantically equivalent representation of $\mathcal{K}^{+m}$, where $\mathcal{K}^{+m}$ is a classifier that results from an order-independent classifier $\mathcal{K}$ by adding m new fields of any width, and if $\mathcal{K}^{-m}$ is order-independent then $\mathcal{K}^{-m}$ with a false-positive check of a single matched rule is a semantically equivalent representation of $\mathcal{K}$, where $\mathcal{K}^{-m}$ is the classifier that results from an order-independent classifier $\mathcal{K}$ by removing m fields, i.e., each $R = (I_1, \ldots, I_k)$ is replaced with $R^{-m} = (I_1, \ldots, I_{k-m})$. Based on these results, we have introduced the Fields Subset Minimization (FSM) problem: find a maximal subset of fields M of an order-independent classifier $\mathcal{K}$ such that $\mathcal{K}^{-M}$ is order-independent; if there are several such subsets, choose M with maximal total width (to minimize lookup word width). We have introduced heuristic algorithms for solving FSM that lead to significant savings in classifier size, often saving by orders of magnitude. We have also introduced the *multi-group* representation that assigns the rules of a classifier $\mathcal{K}$ to β groups such that: (1) each rule belongs to a single group; (2) the rules of each group are order-independent on a subset of k fields of $\mathcal{K}$ (except the "catch-all" rule); (3) different groups can reuse the same fields to keep order-independence, proving that the multi-group representation of a classifier is semantically equivalent to the original classifier. This leads to the corresponding l-Groups of Rules (l-MGR), Maximum Rules Coverage (l-MRC), and (β, l)-MRC optimization problems, solving which provides even more reduction in classifier size and enables the parallelization of lookups to groups. We compare these results to Boolean minimization based on *MinDNF*. These results have been published in the works [14, 26, 42].

Multigroup representations of packet classifiers. In this part, the multigroup representation introduced above leads to the development of alternative decompositions for an IPv6 FIB, more efficient and more amenable to parallelization than standard decomposition by prefix length. We introduce two operations on packet classifiers, namely (width) reduction and partitioning. We introduce the *minimal weak (strong) order-independent reduction* problem: given a classifier $\mathcal{K}$ (not necessary order-independent) and a constant l, assign $\mathcal{K}$'s rules to a minimal number of disjoint groups such that different groups of prefixes can be based on subsets of at most l bit-identities, and each group is weak (strong) order-independent on these bits. We show the relation between this problem and the *minimum test set* problem: given a test set τ on X, find a test set $Y \subseteq X$ with minimal $|Y|$, showing that the special case of the minimum (weak or strong) order-independent reduction problem where all rules in $\mathcal{K}$ consist of 0 and 1 bits only is equivalent to the minimum test set problem. We propose several heuristics that can be used in practice to find minimal reductions based on greedily calling the OneGroup procedure that can be implemented in different ways: Earliest Deadline First (EDF), selecting the next group greedily with MaxOI, using the newly introduced MaxPair heuristic or its generalization, the MinSimilar$_\delta$ heuristic. All of these heuristics lead to large savings, with one of the main practical applications of this part being the (highly relevant for networking) problem of representing IPv6 routing tables on existing infrastructure designed for IPv4. These results have been published in the works [20, 34, 35].

Packet classifier reduction on distributed platforms. In this part, we consider the problem of representing FIBs on distributed platforms, where several line-cards (LCs) are interconnected by a switching fabric. Each ingress (RX) LC must separately maintain a FIB table (or parts of one) to forward traffic to the correct egress (TX) LC which transmits it further over an output port. The fundamental question here is how to efficiently implement a FIB table across such a distributed switching platform. We study the notion of equivalent representations: for a classifier $\mathcal{K}$ with a classification function $f : \{0,1\}^w \to \mathcal{A}$ and a set of bit indices $B \subseteq W$, we say that a classifier $\mathcal{K}^{-B}$ with classification function $g : \{0,1\}^{w-|B|} \to \mathcal{A}$ is an *equivalent* representation of $\mathcal{K}$ if $f(H) = g(H^{-B})$ for every header $H \in \{0,1\}^w$. We introduce a novel property, *match equivalence*, to

improve the efficiency of incomplete classifiers. This property relaxes the classification function by keeping only bit indices that preserve the original classifier $\mathcal{K}$'s action for any header *matched* by $\mathcal{K}$. Formally, an incomplete classifier $\mathcal{K}^B$ is *match equivalent* (ME) to a classifier $\mathcal{K}^W$, $B \subseteq W$, if the action for any matched header H^W in $\mathcal{K}^W$ coincides with the action for the corresponding header H^B in $\mathcal{K}^B$. If H^W has no match in $\mathcal{K}^W$ then H^B can either be matched or remain unmatched in $\mathcal{K}^B$. We formulate the Minimal match equivalence optimization problem: for a given classifier $\mathcal{K}$, find a match equivalent classifier to $\mathcal{K}$ whose filters require minimal memory (in bits) and propose different flavors of match equivalence based on three structural properties: filter-order independence (that pertains to classifiers whose filters are all pairwise disjoint), action-order independence (where filters with the same classification action do not have to be disjoint), and conflicting rules (where no two rules with different actions are conflicting with respect to a given subset of bits). We introduce efficient heuristic algorithms for solving the match equivalence problem under all three properties and show that they significantly reduce classifier size in the distributed setting. We find the time complexity of our heuristics, showing that for a classifier with N rules and headers of w bits, the time complexity of MINME is $O(N^2 \cdot w^3) + O(w^2) \cdot T(\text{SE})$ and the time complexity of PARMINME is $O\left(\beta N^2 \cdot w^3\right) + O(\beta w^2) \cdot T(\text{SE})$, where $T(\text{SE})$ is the time complexity of the SE procedure (that computes an equivalent representation in minimal memory). These results have been published in the works [10, 24].

Classifiers for policies reusing the same classes. In this part, we exploit other alternatives to achieve additional efficiency of policy state represented on the data plane, based on the fact that similar "classification patterns" (*classes*) are reused in different policies, where each class consists of ternary-bit filters determining a set of matched packet headers. We introduce a partial ordering $\prec_P$ of classes in C_P based on their disjointness and position in the policy (classifier). We introduce the Policy Sequence Packing (PSP) problem: given a set of policies $\mathcal{P}$, find a sequence of classes $\mathbb{S}$ compatible with all policies in $\mathcal{P}$ that minimizes $\mathrm{W}^+(\mathbb{S})$, the overhead of class duplication. We show that PSP is NP-hard even for two policies, $|\mathcal{P}| = 2$. We introduce algorithms for PSP based on the *Weighted Feedback Vertex Set* (WFVS) problem, including the AO algorithm, and show that AO correctly solves the PSP problem. These results have been published in

the works [3, 19].

Approximate packet classification. In this part, we generalize the classical packet classification problem (the exact case), where each rule is composed from a ternary bit string and an action; since rules can overlap they have priorities and the first *matched* rule is returned for an incoming packet. We introduce approximate representations of packet classifiers allowing to "multiplex" several actions. This additional level of flexibility allows to improve resource requirements by still keeping desired level of accuracy. We say that a classifier $\mathcal{K}$ is *exact* if for each rule $R_i \in \mathcal{K}$ the corresponding $\mathcal{A}_i$ (set of allowed actions) consists of a single element; otherwise, $\mathcal{K}$ is *approximate*. We define an *optimization process* consisting of a set of *basic operators* and a *heuristic* choosing a sequence of applied operations (*operation sequence*). In the approximate case, $\mathbf{O}$ (set of operations applicable on the input classifier $\mathcal{K}$) can increase compared to the exact case since generalized operations are applicable more often. In this case, we introduce the eLP algorithm for the exact case and the aLP algorithm that extends it to the approximate case. We have studied the complexity of these algorithms, showing that the time complexity of aLP is $O(|\mathcal{K}| \cdot w \cdot l_{\max})$, where $l_{\max}$ is the size of the largest action set in the original classifier $\mathcal{K}$, and that subsumptions cannot be applied on a maLPM classifier $\mathcal{K}_a$ or on the classifer $\mathcal{S}(\mathcal{K}_a)$, where $\mathcal{S}$ is an operation sequence consisting of resolutions. These results have been published in the works [1, 19].

5.1.3 Control plane algorithms

Network simplification algorithms. In this part, we consider an important direction of study intended to simplify network management: develop representations of the original network by a simpler virtual network while still implementing desired properties of the original. We define a *bandwidth allocation* as a set of routing paths together with specific bandwidths allocated along these paths and introduce the notion of *bandwidth equivalence* between two networks: the resulting set of link capacities must allow for any bandwidth allocation and routing the original did, and the notion of *routing equivalence*, a property that allows to map coherent sets of routes (called *bandwidth allocations* in what follows) between the networks while preserving capacities. We show that for every input network G, the brute force algorithm computes a minimal bandwidth net-

work G^* that is unique. We introduce the DAG-OPT algorithm and show that it outputs the minimal bandwidth network G^* equivalent to a given network (DAG) G in time $O(|V||E|^2)$. We also introduce a more efficient heuristic algorithm WPP based on a local capacity adjustment called *weight propagation* (WP). We show that the algorithm WPP, after processing a graph $G = (V, E, w, S, D)$, outputs a graph $G' = (V, E, w', S, D)$ such that $G' \simeq G$ and for all $e \in E$ it holds that $w'(e) \leq w(e)$ and $w'(e) \leq \min\{I'_e, O'_e\}$, in time $O(|E| + |V| \log |V|)$. Moreover, if $G = (V, E, w, S, D)$ is a graph such that its undirected version is a forest, then WPP applied to G outputs the minimal bandwidth network G^* equivalent to G. We introduce a number of network topology transformations that preserve routing equivalence: shrinking a non-bottleneck incoming edge, shrinking a non-bottleneck outgoing edge, merging parallel edges, removing self-loops. We show that a bandwidth preserving routing transformation applied to a forest $G = (V, E, w, S, D)$ generates a graph $G' = (V', E', w', S, D)$ that is routing equivalent to G. We introduce a greedy algorithm for applying these transformations that shows excellent results in practice for reducing network topologies. We also find and study a special case of the Braess' paradox in our setting. These results have been published in the work [18].

Formalization and taxonomy of compute-aggregate problems for cloud computing applications. In this part, we consider the problem of optimizing compute-aggregate problems in distributed settings under the assumption that each compute-aggregate task should conform to a budget constraint since different cloud tenants are able to invest different economic resources to compute their aggregations. To avoid oversubscription of "fastest" links, they can also have different costs of sending data over a link. The problem now divides into two completely decoupled phases: (1) find a "cheapest" plan given a distribution of data over the network, an *aggregation function* (that computes the size of aggregating two different pieces of data), and the cost of sending a unit of data over a link, and (2) actually redistribute aggregations computed on the first phase while optimizing desired objectives. For the first part, we introduce the notion of an aggregation plan and an aggregation size function μ, which provides a key for our taxonomy. We introduce the compute-aggregate minimization problem (CAM): given an undirected connected graph $G = (V, E)$, cost function c, a target vertex

t, a set of initial data chunks C, and an aggregation size function μ, the CAM[μ] problem is to find an aggregation plan P such that cost(P) is minimized. We also introduce two special cases: TCAM (for trees) and CCAM (for complete graphs). We show that unless $P = NP$, there is no polynomial time constant approximation algorithm for CAM without associativity constraint on μ even if G is restricted to two vertices. We show that these problems often reduce to finding a minimum weight tree that connects a given set of vertices, the minimum Steiner tree problem MStT. We show that if every vertex in G contains a data chunk, then MStT can be solved exactly in polynomial time, and if G is a tree then MStT can be solved exactly in polynomial time. We prove that there exist polynomial algorithms that solve CCAM[μ] and TCAM[μ] on a set of chunks C with approximation factor $\frac{W_C[\mu]}{w_C[\mu]}$. We show that there exists an aggregation size function μ such that $\forall a, b$ $\mu(a, b) \leq \min(a, b)$, and no polynomial time constant approximation algorithm for CCAM[μ] or TCAM[μ] exists unless $P = NP$. We prove that there exists an optimal algorithm for TCAM[min] running in polynomial time, and if there exists a polynomial α-approximate algorithm for MStT, then there exists a polynomial 4α-approximate algorithm for CAM[max], which we call `RECH_MStTMax`. We show that if there exists $a > 0$ such that $\mu(a, a) = a$ then CAM[μ] is NP-hard and does not have less than $\frac{19}{18}$-approximate polynomial algorithms even if all edge weights are equal, unless P=NP. All of these results combined lead to a full taxonomy of approximation factors for different μ and different variations of the CAM problem. These results have been published in the works [4, 21, 23, 30].

Efficient demand assignment in multi-connected microgrids. The focus of this part is on efficient demand satisfaction in the context of multi-connected microgrids, where a demand can be met by different generation resources: local, nearby, or on a regional grid that defines a *load balancing vector* for each demand that determines which generators are connected/available to satisfy this demand. We model a set of multi-connected microgrids with a switching system $(I, \mathcal{D})$ that consists of a set of inputs I (generators) with port capacities c_i (their nominal power) and a set of demands $\mathcal{D}$ that are to be scheduled; a demand d is characterized by its length $l(d)$ (how long it lasts), width $w(d)$ (power level), and a *load balancing vector* $v(d)$ that contains the set of input ports available to process d (set of generators that can meet this demand). We assume that $(I, \mathcal{D})$ is a directed

acyclic bipartite graph. We introduce the ChoosePortDemand procedure that takes the current state (remaining demands and leftover capacities) as input and outputs the input-demand pair (i, d) for the next assignment; algorithms differ in their ChoosePortDemand procedures. We introduce the Shared Greedy (SG) family of algorithms, including the Shared Longest Demand (SLD) and Shared Longest Port (SLP) algorithms. We prove several lower and upper bounds on the approximation ratios of these algorithms. In particular, we show that for an input with maximal demand length L and minimal demand length l: (1) an SG algorithm with approximation ratio $\leq \alpha$ w.r.t. configurations has approximation ratio $\leq \frac{\alpha L h}{L+(l-1)h}$ w.r.t. length, where h is the optimal number of configurations; (2) an SG algorithm with approximation ratio $\leq \alpha$ w.r.t. length has approximation ratio $\leq \frac{L}{t}\left(1+\frac{\alpha t-L}{l}\right)$ w.r.t. configurations, where t is the optimal length. We show that for a single shared port system with unit capacities, any scheduling policy ALG from the SG family has approximation ratio at most $\frac{3}{2}$ w.r.t. configurations, and for a system with unit capacities and I ports, SLD has approximation ratio at least $\left(\frac{3}{2}-2^{-(I-1)}\right)$ w.r.t. configurations and length. For a system with capacities $c_1 = 2c(\mathcal{I}_1)$ and $c(\mathcal{I}_2) = 2c_1$, we show that the number of configurations in SLD's schedule is at least $\frac{5}{3}$ times more than optimal. The number of configurations in the schedule of SG is at most twice more than in the optimal schedule if one of the following conditions holds: (1) $c_1 \leq c(\mathcal{I}_1)$; (2) $c_1 > c(\mathcal{I}_1)$ and $c_1 > c(\mathcal{I}_2)$. We prove that SLP has the optimal number of configurations for the case of heterogeneous input capacities and unit-sized demands width. These results have been published in the works [40, 41].

Robust counting of traffic flows. In this part, we consider the problem of computing the sizes of all flows at a network element, which can overload scarce data-plane resources when the number of flows becomes large. Distributing this monitoring task can endanger the accuracy due to *network noise* in the form of packet reordering and loss. We introduce Robust Distributed Computation (RoDiC), a general technique that executes a monitoring task in multiple network elements correctly despite network noise on the paths between the elements. For stateful tasks, the proposed technique not only maintains state in different elements but also resiliently communicates some state between these elements. We show feasibility limits: we prove that a deterministic RoDiC algorithm can-

not guarantee correct distributed computation of the flow size on two switches if $L \geq 1$ packet and $R \geq 2^{n-1}+1$ packets. On the positive side, we prove that the proposed algorithm is guaranteed to compute the flow size on two switches correctly if $R \leq 2^{n-1} - 2^{n-t}$ packets and $L + R \leq 2^{n-1} - 1$ packets. We introduce the notion of span(γ, k): a subsequence of the packets at S that arrive to D in order, omit at most γ full groups at the beginning of the flow, and miss at most $\gamma - 1$ full consecutive groups after any packet in this subsequence. We present and analyze the algorithm for computing the flow size on two switches in the span-based model and prove that when γ is at most $2^t - 1$ groups, the propose algorithm always computes the flow size on two switches correctly if there exists a span$(\gamma, n-t)$ and $R \leq 2^n - (\gamma+1) \cdot 2^{n-t}$ packets. These results have been published in the works [2, 15, 29].

5.2 Open questions for further research

We finish the thesis with a brief overview of research questions that remain open, are directly related to the thesis, are of both theoretical and practical interest in networking, and may be resolved in further studies.

One question that still remains open is how to formalize *latency* in the form of a final objective function. The first option is not to optimize latency but rather satisfy it by setting a "slack" (maximal delay) on per-packet basis. On the switch level, this model is called *bounded delay*, studied analytically in [134]. This lets one optimize other objectives such as throughput by incorporating multiple packet characteristics (value, required processing etc.). A major drawback of this approach that it is not conducive to end-to-end management decisions. In particular, it is unclear how to divide the slack of a given flow among network elements and packets in the flow. Note that the slack-based formalization presented in this thesis and in the works [3, 19] deals only with individual packets, while the problem of dealing with flows calls for other objective functions.

Optimizing greedily for latency at every switch addresses this issue. Such recently proposed transports as *pFabric* and *pHost* optimize for the so-called *flow completion time* (FCT) [49,107]. Under this approach, there is still some variation on how to represent FCT in the final objective and which additional information is required on the flow level. FCT is defined in [49] as the difference between the

time e_f when a flow is fully processed at the last packet's destination and the time b_f when the first packet arrives at the source. If there is no multiplexing of multiple flows (i.e., the ith packet of a flow f arrives at the source at time $b_f + i$), and the flow sizes (in packets) are known to the scheduling algorithm *a priori*, there is an algorithm with SRPT (shortest remaining processing time) priorities that optimizes average FCT represented by $e_f - b_f$ for every flow f, and *pFabric* is based on this algorithm [49]. However, analytic results in realistic settings with respect to FCT remain elusive; the *pFabric* work [49] is inspired by [65] that makes two key assumptions: that flows are *atomic*, i.e., a flow is an uninterrupted stream of packets available all at once, and that all flow sizes are available in advance. Both these assumptions can be problematic in practice: in a network of switches, delays on a previous network element can result in interruptions in flows arriving on the next one, and flow sizes are often unknown when their first packets arrive. It is an important and challenging open problem to find suitable theoretical models of latency in buffer management.

Another interesting future research direction is to automate the design of management policies, given desired objectives and involved traffic characteristics that will allow to adapt automatically to desired network behaviors by using machine learning techniques. Moreover, it would be interesting to define a virtualization layer on top of heterogeneous computing resources that will distribute processing given service characteristics and a desired objective. In this thesis, we have made the first steps towards this goal in the works [5, 22, 36], but there still remains much to be done.

Bibliography

[1] V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Packet Classifiers With Controlled Accuracy. IEEE Transactions on Networking, vol. 29, no. 3, 2021, pp. 1141–1154.

[2] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. IEEE Transactions on Networking, vol. 29, no. 1, 2021, pp. 275–288.

[3] A. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Competitive buffer management for packets with latency constraints. Computer Networks, vol. 189, no. 22, Article ID 107942, 2021.

[4] P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalization and taxonomy of compute-aggregate problems for cloud computing applications. Computer Networks, vol. 189, no. 22, Article ID 107915, 2021.

[5] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. Towards Software-Defined Buffer Management. IEEE Transactions on Networking, vol. 28, no. 5, 2020, pp. 2337–2349.

[6] P. Chuprikov, S.I. Nikolenko, K. Kogan. Towards Declarative Self-Adapting Buffer Management. ACM SIGCOMM Computer Communication Review, vol. 50, no. 3, 2020, pp. 30–37.

[7] V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. IEEE Transactions on Networking, vol. 28, no. 3, 2020, pp. 1088–1101.

[8] P. Chuprikov, S.I. Nikolenko, K. Kogan, A. Davydow. Priority Queueing for Packets with Two Characteristics. IEEE Transactions on Networking, vol. 26, no. 1, 2018, pp. 342–355.

[9] P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Admission Control in Shared Memory Switches. Journal of Scheduling, vol. 21, no. 5, 2018, pp. 533–543.

[10] K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. Efficient FIB Representations on Distributed Platforms. IEEE Transactions on Networking, vol. 25, no. 6, 2017, pp. 3309–3322.

[11] P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Heterogeneous Packet Processing in Shared Memory Buffers. Journal of Parallel and Distributed Computing, vol. 99, 2017, pp. 1–12.

[12] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. The Impact of Processing Order on Performance: A Taxonomy of Semi-FIFO Policies. Journal of Computer and System Sciences, vol. 88, 2017, pp. 220–235.

[13] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub, M. Segal. Large profits or fast gains: A Dilemma in Maximizing Throughput with Applications to Network Processors. Journal of Network and Computer Applications, vol. 74, 2016, pp. 31–43.

[14] K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. Exploiting Order Independence for Scalable and Expressive Packet Classification. IEEE Transactions on Networking, vol. 24, no. 2, 2016, pp. 1251–1264.

[15] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. 27th IEEE International Conference on Network Protocols (ICNP 2019), 2019, pp. 1–11.

[16] V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Classifiers with Controlled Accuracy. 38th IEEE International Conference on Computer Communications (INFOCOM 2019), 2019, pp. 2044–2052.

[17] V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. 26th IEEE International Conference on Network Protocols (ICNP 2018), 2018, pp. 121–131

[18] S.I. Nikolenko, K. Kogan, A. Fernàndez Anta. Network Simplification Preserving Bandwidth and Routing Capabilities. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

[19] A.P. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Throughput Optimization with Latency Constraints. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

[20] P. Chuprikov, K. Kogan, S.I. Nikolenko. General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[21] P. Chuprikov, A.P. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Planning in Compute-Aggregate Problems as Optimization Problems on Graphs. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[22] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. A Programmable Buffer Management Platform. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[23] P. Chuprikov, S.I. Nikolenko, K. Kogan. On Demand Elastic Capacity Planning for Service Auto-scaling. Proceedings of 2016 IEEE International Conference on Computer Communications (INFOCOM 2016), 2016.

[24] K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. FIB Efficiency in Distributed Platforms. Proc. 24th IEEE International Conference on Network Protocols (ICNP 2016), 2016.

[25] P. Chuprikov, K. Kogan, S.I. Nikolenko. Priority Queueing with Multiple Packet Characteristics. Proceedings of IEEE INFOCOM 2015, pp. 1418–1426.

[26] K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). ACM SIGCOMM 2014, ACM Press, 2014, pp. 15–26.

[27] P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Shared-Memory Buffer Management for Heterogeneous Packet Processing. 34th International Conference on Distributed Computing Systems (ICDCS 2014).

[28] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Online Scheduling FIFO Policies with Admission and Push-Out. Theory of Computing Systems, vol. 58, no. 2, 2016, pp. 322-344.

[29] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Distributed Counting along Lossy Paths Without Feedback. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), Lecture Notes in Computer Science, vol. 11685, 2018, pp. 30–33 (CORE rank B).

[30] P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalizing Compute-Aggregate Problems in Cloud Computing. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), 2018, pp. 377–391 (CORE rank B).

[31] P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Essential Traffic Parameters for Shared Memory Switch Performance. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015), Lecture Notes in Computer Science, vol. 9439, Springer, 2015, pp. 61–75. (CORE rank B).

[32] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. A Taxonomy of Semi-FIFO Policies. Proc. 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), 2012, pp. 295–304. (CORE rank B).

[33] K. Kogan, S.I. Nikolenko, V. Demianiuk, P. Chuprikov, A.P. Davydow. Personal Insights on Three Research Directions in Networked Systems. Proc. 10th International Conference on Communication Systems and Networks (COMSNETS 2018), IEEE, 2018.

[34] P. Chuprikov, K. Kogan, S.I. Nikolenko. How to implement complex policies on existing network infrastructure. Proc. 4th Symposium on SDN Research (SOSR 2018), ACM, 2018, pp. 1–8.

[35] S.I. Nikolenko, K. Kogan, G. Retvari, E.R. Berczi-Kovacs, A. Shalimov. How to Represent IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes. Proc. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): GI 2016: 9th IEEE Global Internet Symposium (IEEE GI 2016), 2016.

[36] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, P. Eugster. BASEL (Buffering Architecture SpEcification Language). Proc. 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2016), 2016, pp. 69–74.

[37] K. Kogan, S.I. Nikolenko, P. Eugster, E. Ruan. Strategies for Mitigating TCAM Space Bottlenecks. 22nd IEEE Annual Symposium on High-Performance Interconnects (HOTI 2014), IEEE, 2014, pp. 25–32.

[38] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub. Balancing Work and Size with Bounded Buffers. Proc. 6th International Conference on Communication Systems and Networks (COMSNETS 2014), 2014, pp. 1–8.

[39] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Multi-Queued Network Processors for Packets with Heterogeneous Processing Requirements. Proc. 5th International Conference on Communication Systems and Networks (COMSNETS 2013), 2013, pp. 1–10.

[40] K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient Demand Assignment in Multi-Connected Microgrids with a Shared Central Grid. Proc.

3rd IFIP Conference on Sustainable Internet and ICT for Sustainability 2013 (SustainIT 2013), 2013.

[41] K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient demand assignment in multi-connected microgrids.Proc. 4th Energy-Efficient Computing and Networking (e-Energy 2013), pp. 277–278.

[42] K. Kogan, S.I. Nikolenko, W. Culhane, P. Eugster, E. Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013), pp. 153–154.

[43] R. Abe, H. Taoka, and D. McQuilkin. Digital grid: Communicative electrical grids of the future. *IEEE Trans. Smart Grid*, 2(2):399–410, 2011.

[44] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Transactions on Algorithms*, 5(1), 2008.

[45] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Trans. on Algorithms*, 5(1), 2008.

[46] W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén. Competitive queue policies for differentiated services. In *INFOCOM*, pages 431–440, 2000.

[47] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX*, pages 281–296, 2010.

[48] S. Albers and M. Schmidt. On the performance of greedy algorithms in packet buffering. *SIAM Journal on Computing*, 35(2):278–304, 2005.

[49] M. Alizadeh and et al. pfabric: minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446, 2013.

[50] E. Allender, L. Hellerstein, P. McCabe, T. Pitassi, and M. E. Saks. Minimizing DNF formulas and $\boldsymbol{AC}^0_d$ circuits given a truth table. In *Conference on Computational Complexity*, 2006.

[51] E. Allender, L. Hellerstein, P. McCabe, T. Pitassi, and M. E. Saks. Minimizing disjunctive normal form formulas and $\boldsymbol{AC}^0$ circuits given a truth table. *SIAM J. Comput.*, 38(1):63–84, 2008.

[52] N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies for qos switches. In *SODA*, pages 761–770, 2003.

[53] D. Applegate, G. Călinescu, D. S. Johnson, H. J. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *ACM-SIAM SODA*, 2007.

[54] J. Audibert and S. Bubeck. Best arm identification in multi-armed bandits. In *COLT*, 2010.

[55] J. Audibert, R. Munos, and C. Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876 – 1902, 2009. Algorithmic Learning Theory.

[56] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[57] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[58] Y. Azar and A. Litichevskey. Maximizing throughput in multi-queue switches. *Algorithmica*, 45(1):69–90, 2006.

[59] Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. *Algorithmica*, 43(1-2):81–96, 2005.

[60] Y. Azar and Y. Richter. An improved algorithm for CIOQ switches. *ACM Transactions on Algorithms*, 2(2):282–295, 2006.

[61] V. Bajaj. Ten interesting things about India power, 2012.

[62] M. Bando and J. H. Chao. Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps. In *IEEE INFOCOM*, pages 821–829, 2010.

[63] T. Banerjee-Mishra, S. Sahni, and G. S. Seetharaman. Pc-duos+: A tcam architecture for packet classifiers. *IEEE Trans. Computers*, 63(6):1527–1540, 2014.

[64] T. Banerjee-Mishra, S. Sahni, and G. S. Seetharaman. Pc-trio: A power efficient tcam architecture for packet classifiers. *IEEE Trans. Computers*, 64(4):1104–1118, 2015.

[65] A. Bar-Noy, M. M. Halldórsson, G. Kortsarz, R. Salman, and H. Shachnai. Sum multicoloring of graphs. *J. Algorithms*, 37(2):422–450, 2000.

[66] P. Berman, B. DasGupta, and M. Kao. Tight approximability results for test set problems in bioinformatics. *JCSS*, 71:145–162, 2005.

[67] S. Biswas. India struggles to deliver enough power, 2012.

[68] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[69] A. Bremler-Barr, D. Hay, and D. Hendler. Layered interval codes for TCAM-based classification. In *IEEE Conference on Computer Communications*, 2009.

[70] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using Gray coding. *IEEE Trans. Computers*, 61(1):18–30, 2012.

[71] S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in multi-armed bandits problems. In *Algorithmic Learning Theory*, pages 23–37, Berlin, Heidelberg, 2009.

[72] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[73] T. Carpenter, S. Singla, P. Azimzadeh, and S. Keshav. The impact of electricity pricing schemes on storage adoption in Ontario. In *e-Energy*, page 18, 2012.

[74] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[75] Y. Chang, C. Lee, and C. Su. Multi-field range encoding for packet classification in TCAM. In *IEEE Infocom Mini-Conference*, 2011.

[76] H. Che, Z. Wang, K. Zheng, and B. Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Trans. Computers*, 57(7):902–915, 2008.

[77] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz. The case for evaluating MapReduce performance using workload suites. In *MASCOTS*, pages 390–399, 2011.

[78] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN*, pages 73–82, 2009.

[79] Cisco 12000 series internet router architecture: Line card design. http://www.cisco.com/c/en/us/support/docs/routers/12000-series-routers/47242-arch12000-lcdesign.html.

[80] CISCO 12000 Series routers. http://www.cisco.com/en/US/products/hw/routers/ps167/index.html.

[81] Cisco CRS forwarding Processor Cards. http://www.cisco.com/c/en/us/products/collateral/routers/carrierroutingsystem/datasheetc78730790.pdf.

[82] Class of service configuration guide. https://www.juniper.net/documentation/en_US/junos11.1/information-products/topic-collections/security/software-all/ class-of-service/index.html.

[83] ClassBench: A packet classification benchmark. http://www.arl.wustl.edu/classbench/.

[84] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7(2):202–220, 1986.

[85] Configuring IP ACLs. http://www.cisco.com/en/US/docs/switches/datacenter/sw/4_1/nx-os/security/configuration/guide/sec_ipacls.pdf.

[86] Configuring modular QoS packet classification. http://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-2/qos/configuration/guide/qc42clas.html.

[87] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[88] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. LOOM: Optimal aggregation overlays for in-memory big data processing. In *HotCloud*, Philadelphia, PA, 2014. USENIX Association.

[89] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. Optimal communication structures for big data aggregation. In *INFOCOM*, pages 1643–1651, April 2015.

[90] A. Cvetkovski. An algorithm for approximate counting using limited memory resources. In *SIGMETRICS*, pages 181–190, 2007.

[91] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[92] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[93] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Inf. Process. Lett.*, 86(3):129–136, 2003.

[94] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood. Fast packet classification using bloom filters. In *ACM/IEEE ANCS*, 2006.

[95] A. Dixit, K. Kogan, and P. Eugster. Composing heterogeneous SDN controllers with flowbricks. In *ICNP*, pages 287–292, 2014.

[96] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *SIGMETRICS*, pages 311–322, 2006.

[97] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing optimal IP routing tables. In *INFOCOM*, pages 88–97, 1999.

[98] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *Computer Communication Review*, 34(2):97–122, 2004.

[99] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[100] A. Elmokashfi and A. Dhamdhere. Revisiting BGP churn growth. *CCR*, 44(1):5–12, 2014.

[101] A. Elmokashfi, A. Kvalbein, and C. Dovrolis. BGP churn evolution: A perspective from the core. *Trans. Networking*, 20(2):571–584, 2012.

[102] M. Englert and M. Westermann. Lower and upper bounds on FIFO buffer management in QoS switches. *Algorithmica*, 53(4):523–548, 2009.

[103] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, Aug. 2003.

[104] G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, Feb 1998.

[105] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[106] D. E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artif. Intell.*, 57(2-3):143–181, 1992.

[107] P. Gao and et al. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, pages 1–12, 2015.

[108] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to np-completeness, 1979.

[109] M. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.

[110] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 147–160, New York, NY, USA, 1999. ACM.

[111] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[112] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro*, 20(1):34–41, 2000.

[113] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[114] E. L. Hahne, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. In *SPAA*, pages 53–58, 2001.

[115] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[116] C. Hong and et al. Achieving high utilization with software-driven WAN. In *SIGCOMM*, pages 15–26, 2013.

[117] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, Y. Cheng, and H. Wu. Discount counting for fast flow statistics on flow size and flow volume. *IEEE/ACM Trans. Netw.*, 22(3):970–981, June 2014.

[118] S. Jain and et al. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, pages 3–14, 2013.

[119] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *AISTATS*, volume 51, pages 240–248, Cadiz, Spain, 09–11 May 2016.

[120] C. Kaklamanis, M. Chlebík, and J. Chlebíková. Algorithmic aspects of global computing the steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207 – 214, 2008.

[121] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*, pages 13–24, 2013.

[122] Y. Kanizo, D. Hay, and I. Keslassy. Optimal fast hashing. In *IEEE Infocom*, 2009.

[123] N. Kathleen and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[124] J. Kawahara, K. Kobayashi, and T. Maeda. Tight analysis of priority queuing policy for egress traffic. *CoRR*, abs/1207.5959, 2012.

[125] S. Keshav and C. Rosenberg. On load elasticity. In *Proc. IEEE COMSOC MMTC E-Letter*, To Appear.

[126] I. Keslassy and et al. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[127] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. In *INFOCOM*, pages 3191–3199, 2011.

[128] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[129] A. Kesselman and K. Kogan. Nonpreemptive scheduling of optical switches. *IEEE Transactions on Communications*, 55(6):1212–1219, 2007.

[130] A. Kesselman, K. Kogan, and M. Segal. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.

[131] A. Kesselman, K. Kogan, and M. Segal. Improved competitive performance bounds for CIOQ switches. *Algorithmica*, 63(1-2):411–424, 2012.

[132] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *STOC*, pages 520–529, 2001.

[133] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. *SIAM J. Comput.*, 33(3):563–583, 2004.

[134] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[135] A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. *J. Algorithms*, 46(1):79–95, 2003.

[136] A. Kesselman and Y. Mansour. Harmonic buffer management policy for shared memory switches. *Theor. Comput. Sci.*, 324(2-3):161–182, 2004.

[137] A. Kesselman and Y. Mansour. Harmonic buffer management policy for shared memory switches. *Theor. Comput. Sci.*, 324(2-3):161–182, 2004.

[138] A. Kesselman, Y. Mansour, and R. van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43(1-2):63–80, 2005.

[139] A. Kesselman and A. Rosén. Scheduling policies for CIOQ switches. *Journal of Algorithms*, 60(1):60–83, 2006.

[140] A. Kesselman and A. Rosén. Controlling CIOQ switches with priority queuing and in multistage interconnection networks. *Journal of Interconnection Networks*, 9(1/2):53–72, 2008.

[141] S. Khot and R. Saket. Hardness of minimizing and learning DNF expressions. In *FOCS*, pages 231–240, 2008.

[142] S. Khot and R. Saket. Hardness of minimizing and learning DNF expressions. In *IEEE FOCS*, 2008.

[143] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[144] J. M. Kleinberg and É. Tardos. *Algorithm design.* Add.-Wesley, 2006.

[145] K. Kobayashi, S. Miyazaki, and Y. Okabe. Competitive buffer management for multi-queue switches in QoS networks using packet buffering algorithms. In *Proceedings of the* 21st *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 328–336, 2009.

[146] K. Kogan, A. Dixit, and P. Eugster. Serial composition of heterogeneous control planes. In *ONS*, 2014.

[147] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin, and D. Tugaryov. FIFO Queueing Policies for Packets with Heterogeneous Processing. In *MedAlg 2012*, LNCS vol. 7659, pages 248–260, 2012.

[148] A. Korösi, J. Tapolcai, B. Mihalka, G. Mészáros, and G. Rétvári. Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously. In *IEEE ICNP*, 2014.

[149] D. V. Krioukov, K. C. Claffy, K. R. Fall, and A. Brady. On compact routing for the internet. *Comput. Commun. Rev.*, 37(3):41–52, 2007.

[150] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[151] N. Lesh and M. Mitzenmacher. Bubblesearch: A simple heuristic for improving priority-based greedy algorithms. *Inf. Process. Lett.*, 97(4):161–169, 2006.

[152] K. Lim, K. Park, and H. Lim. Binary search on levels using a bloom filter for ipv6 address lookup. In *ACM/IEEE ANCS*, pages 185–186, 2009.

[153] Cisco XR 12000 series gigabit ethernet line cards. https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html.

[154] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 18(2):490–500, 2010.

[155] A. X. Liu, C. R. Meiners, and Y. Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *IEEE Infocom*, 2008.

[156] H. Liu. Efficient mapping of range classifier into Ternary-CAM. In *IEEE Hot Interconnects*, 2002.

[157] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[158] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[159] Z. Lotker and B. Patt-Shamir. Nearly optimal FIFO buffer management for two packet classes. *Computer Networks*, 42(4):481–492, 2003.

[160] B. Malakooti. *Operations and Production Systems with Multiple Objectives*. John Wiley & Sons, 2013.

[161] Y. Mansour, B. Patt-Shamir, and O. Lapid. Optimal smoothing schedules for real-time streams (extended abstract). In *PODC*, pages 21–29, 2000.

[162] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 20(2):488–500, 2012.

[163] D. Meyer, L. Zhang, and K. Fall. Report from the IAB workshop on routing and addressing. http://www.ietf.org/internet-drafts/draft-iab-raws-report-02.txt, 2007.

[164] A. K. Mishra, D. E. Irwin, P. J. Shenoy, J. Kurose, and T. Zhu. SmartCharge: cutting the electricity bill in smart homes with energy storage. In *e-Energy*, page 29, 2012.

[165] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, 2016.

[166] Netlogic Microsystems. Content addressable memory. http://www.netlogicmicro.com.

[167] S.I. Nikolenko, K. Kogan. Single and Multiple Buffer Processing. *Encyclopaedia of Algorithms*, Springer, 2014, pp. 1–9.

[168] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE JSAC*, 17(6):1083 –1092, 1999.

[169] OpenFlow 1.3 specification, 2012. http://www.openflow.org/wk/index.php/OpenFlow_1_3_proposal.

[170] J. B. Orlin. Max flows in o(nm) time, or better. In *STOC*, pages 765–774, 2013.

[171] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21(3):629–656, 1996.

[172] Pipelined packet switching and queuing architecture. US7643486B2, 2010.

[173] QoS: Modular QoS command-line interface configuration guide, cisco ios. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_mqc/configuration/ 15-mt/qos-mqc-15-mt-book/qos-mqc.html.

[174] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing ip forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM*, 2013.

[175] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[176] O. Rottenstreich, A. Berman, Y. Cassuto, and I. Keslassy. Compression for fixed-width memories. In *IEEE ISIT*, 2013.

[177] O. Rottenstreich and I. Keslassy. On the code length of TCAM coding schemes. In *IEEE ISIT*, 2010.

[178] O. Rottenstreich and I. Keslassy. Worst-case TCAM rule expansion. In *IEEE Infocom Mini-Conference*, 2010.

[179] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. On finding an optimal TCAM encoding scheme for packet classification. In *IEEE Infocom*, 2013.

[180] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim. Compressing forwarding tables for datacenter scalability. *IEEE Journal on Selected Areas in Communications*, 32(1):138–151, 2014.

[181] H. Saitoh and J. Toyoda. A new concept of electric power network for the effective transportation of small power of dispersed generation plants. *IEEE J. Trans. PE*, 115(6):568–575, 1995.

[182] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.

[183] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster IP route lookup. In *IEEE ICNP*, pages 358–367, 2005.

[184] H. Song and J. S. Turner. ABC: Adaptive binary cuttings for multidimensional packet classification. *IEEE/ACM Trans. Netw.*, 21(1):98–109, 2013.

[185] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[186] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perf. Eval. Rev.*, 26(1):1–10, 1998.

[187] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *SIGMETRICS*, 1998.

[188] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM*, 1998.

[189] R. Stanojevic. Small active counters. In *INFOCOM*, pages 2153–2161, 2007.

[190] R. Steinberg and W. I. Zangwill. The prevalence of braess' paradox. *Transportation Science*, 17(3):301–318, 1983.

[191] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[192] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, first edition, 1998.

[193] O. R. T. Mizrahi and Y. Moses. TimeFlip: scheduling network updates with timestamp-based TCAM ranges. In *IEEE INFOCOM*, 2015.

[194] E. Tsidon, I. Hanniel, and I. Keslassy. Estimators also need shared values to grow together. In A. G. Greenberg and K. Sohraby, editors, *INFOCOM*, pages 1889–1897. IEEE, 2012.

[195] C. Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4):597–611, 2001.

[196] B. Vamanan and T. N. Vijaykumar. TreeCAM: decoupling updates and lookups in packet classification. In *ACM CoNEXT*, 2011.

[197] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, 2010.

[198] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann, 2005.

[199] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *ACM SIGCOMM*, 1997.

[200] Z. Wang, H. Che, M. Kumar, and S. K. Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Trans. Computers*, 53(12):1602–1614, 2004.

[201] R. Wei, Y. Xu, and H. J. Chao. Block permutations in Boolean space to minimize TCAM for packet classification. In *INFOCOM*, pages 2561–2565, 2012.

[202] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[203] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *SIGCOMM*, pages 123–134, New York, NY, USA, 2013.

[204] World's first digital grid router controls electricity packets, boosts grid access for renewables in Japan, 2013.

[205] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou. Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *Companion Proc. of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 44–53, New York, NY, USA, 2014. ACM.

[206] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM 2018*, Forthcoming.

[207] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. In *SIGCOMM*, 2014.

[208] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, Aug. 2017.

[209] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[210] Y. Zhang and N. Ansari. On architecture design, congestion notification, TCP incast and power consumption in data centers. *IEEE Communications Surveys and Tutorials*, 15(1):39–64, 2013.

[211] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the aggregatability of router forwarding tables. In *IEEE INFOCOM*, 2010.

[212] A. Zhu. Analysis of queueing policies in QoS switches. *J. Algorithms*, 53(2):137–168, 2004.

[213] G. Zhu, S. Yu, and J. Dai. Binary search on prefix covered levels for IP address lookup. In *WiCOM*, pages 3859–3862, 2009.

САНКТ-ПЕТЕРБУРГСКОЕ ОТДЕЛЕНИЕ МАТЕМАТИЧЕСКОГО
ИНСТИТУТА ИМ. В. А. СТЕКЛОВА РОССИЙСКОЙ АКАДЕМИИ НАУК

На правах рукописи

Николенко Сергей Игоревич

Алгоритмы для сетевых приложений и их теоретический анализ

1.2.3. Теоретическая информатика, кибернетика

ДИССЕРТАЦИЯ

на соискание учёной степени доктора
физико-математических наук

Перевод с английского

Санкт-Петербург
2022

Посвящается памяти
Кирилла Когана

Оглавление

Глава 1 Введение **5**
1.1 Цели и задачи исследования 5
1.2 Управление буфером 6
1.3 Классификация пакетов 8
1.4 Алгоритмы на плоскости управления 11
1.5 Апробация работы: список публикаций 14

Глава 2 Алгоритмы управления буфером и их анализ **23**
2.1 Введение в конкурентный анализ для управления буфером и обзор связанных работ 23
2.2 Алгоритмы управления буфером для пакетов с различными требованиями к обработке в одной очереди 33
2.3 Алгоритмы управления буфером для нескольких очередей, разделённых по времени обработки пакетов 42
2.4 Алгоритмы управления буфером для пакетов с различными требованиями к обработке и несколькими исходящими портами . . . 48
2.5 Алгоритмы управления буфером для пакетов с различными требованиями к обработке и переменным размером 61
2.6 Алгоритмы управления буфером для пакетов с несколькими характеристиками 65
2.7 Алгоритмы управления буфером для пакетов с ограниченной задержкой 71
2.8 Выбор алгоритмов управления буфером методами машинного обучения 77

Глава 3 Алгоритмы классификации пакетов **87**
3.1 Введение в классификацию пакетов и обзор литературы 87

3.2 Использование порядковой независимости для классификации пакетов . . . 91
3.3 Мультигрупповые представления классификаторов пакетов . . . 111
3.4 Сокращение классификаторов пакетов на распределённых платформах . . . 119
3.5 Классификаторы, переиспользующие одни и те же классы . . . 140
3.6 Приближённая классификация пакетов . . . 147

Глава 4 Алгоритмы на плоскости управления: упрощение структуры сети, облачные вычисления и интеллектуальные энергосистемы . . . 155
4.1 Алгоритмы для упрощения структуры сети . . . 155
4.2 Формализация и таксономия задач типа compute-aggregate для облачных вычислений . . . 168
4.3 Эффективное распределение запросов в многосвязных микросетях 180
4.4 Устойчивый к шуму подсчёт потоков трафика . . . 189

Глава 5 Заключение . . . 197
5.1 Основные результаты . . . 197
5.1.1 Алгоритмы управления буфером и их анализ . . . 197
5.1.2 Алгоритмы классификации пакетов . . . 202
5.1.3 Алгоритмы на плоскости управления . . . 207
5.2 Открытые проблемы для дальнейших исследований . . . 211

Литература . . . 215

Глава 1 Введение

1.1 Цели и задачи исследования

Интернет и другие сети передачи данных построены на основе крайне разнообразного спектра сетевых элементов. Сетевые элементы передают информацию, агрегированную в пакеты, которые пересылаются к своим назначениям. Новые экономические модели и усложняющиеся сетевые сервисы требуют, чтобы сетевые элементы становились более выразительными и «умными», что обычно приводит к усложнению и удорожанию сетевой инфраструктуры. Уменьшение управляемого состояния сети и более эффективное использование базовой сетевой инфраструктуры без потерь выразительности позволяют улучшить экономическую эффективность сети.

Таким образом, **объектом настоящего исследования** являются сетевые параметры, модели и алгоритмы для плоскости данных и плоскости управления, а **предметом настоящего исследования** являются их теоретические свойства, которые могут предоставить гарантии качества или эффективности в худшем случае, свойства объектов, связанных с сетевыми приложениями (например пакетов в задачах управления буфером или классификации пакетов), которые приводят к новым методам и алгоритмам, и задачи оптимизации, возникающие из этих свойств и экономических целевых функций.

Цели исследования включают разработку новых алгоритмов для различных задач с сетевыми приложениями, в том числе задач управления буфером, классификации пакетов и других, и теоретический анализ, который подтверждает их эффективность. Работа носит **теоретический характер**, и основными результатами являются новые методы алгоритмы и теоретический анализ этих методов и алгоритмов, в том числе теоремы, показывающие гарантированные в худшем случае результаты посредством конкурентного анализа.

Методы. Возможность характеризовать трафик и понимать его свойства

является критически важной для эффективной и успешной работы сети. В частности, нужно определять сервисные классы для различных приложений и разных паттернов трафика, чтобы сделать возможным точное планирование нагрузки и обеспечить соответствующее использование ресурсов, нужных для поддержки реализуемых стратегий управления. Это даёт новые возможности оправдать и оптимизировать огромные инвестиции, связанные с построением сетей, от понимания поведения сетей до использования базовой инфраструктуры.

Сервисы на плоскости данных делятся на две основных категории: (1) сервисы, которые меняют свойства отдельных пакетов (например перекрашивание конкретных полей, изменение инкапсуляции, списки контроля доступа и т.д.) и (2) сервисы, которые меняют свойства наборов пакетов или их взаимодействий (например ограничение скорости или шейпинг трафика). В настоящей диссертации Глава 2 посвящена второй категории в виде разработки и анализа алгоритмов управления буфером, а Глава 3 посвящена первой категории, там вводятся и изучаются новые структурные свойства классификаторов пакетов.

На плоскости управления сетевая инфраструктура представляет собой дорогостоящий ресурс, который требует сложного управления. Сетевые провайдеры часто неспособны полностью оправдать эти огромные вложения. В Главе 4 диссертации мы обсуждаем абстракции на плоскости управления и механизмы, которые позволяют более эффективно использовать сетевую инфраструктуру.

В настоящей диссертации мы рассматриваем и плоскость данных, и плоскость управления, а ниже представляем краткий обзор и мотивацию каждого из основных направлений исследований, отражённых в диссертации. Более подробный мотивационный обзор и ряд открытых вопросов для дальнейших исследований (на некоторые из которых дан ответ в диссертации) содержится в работе [33]. **Основные результаты**, выносящиеся на защиту, приведены в разделе 5.1.

1.2 Управление буфером

Архитектуры с буфером определяют, как связаны входящие и исходящие порты сетевого элемента [109]. Их разработка и методы управления напрямую влияют на эффективность и цену каждого сетевого элемента. Если прибывает сразу

несколько пакетов (burst), во многих случаях оказывается невозможно передать их все сразу, и некоторым пакетам приходится оставаться в буфере внутри сетевого элемента (или быть выброшенными). *Стратегия управления буфером* (buffer management policy) состоит из трёх основных частей:

(1) *управление допуском* (admission control), которое определяет, какой трафик должен быть допущен в буфер,

(2) *стратегия обработки* (processing policy), которая определяет порядок обработки пакетов, и

(3) *планирование передачи пакетов* (packet scheduling), которое определяет, какой пакет передаётся следующим.

Чтобы проанализировать эффективность стратегий управления буфером, в сетевых исследованиях традиционно использовались стохастические модели оценки качества. Соответствующие определения для анализа в худшем случае были сначала введены в теоретической информатике в начале 2000-х гг. в работах [46, 132, 161], которые применили к этой задаче *конкурентный анализ* (competitive analysis) [68]. Конкурентный анализ не только помогает лучше понять свойства стратегий управления буфером, но также предоставляет гарантии качества их работы в худшем случае, что важно для реальных коммутаторов с непредсказуемыми и меняющимися в широких пределах паттернами трафика. Мы выделяем два основных направления среди недавних результатов в этой области: внедрение новых характеристик трафика в стратегии управления буфером и изучение влияния новых характеристик на целевые функции.

Характеристики пакетов, которые изучались в предыдущих исследованиях, включали *ценность* (value), т.е. то, сколько пакет добавляет к целевой функции после передачи, *требуемое время обработки* (required processing), т.е. то, сколько циклов обработки пакет должен пройти, прежде чем его можно будет передать (эта характеристика мотивирована многочисленными задачами разной сложности, которые современный сетевой элемент может выполнять для каждого пакета), *размер* (size), т.е. сколько памяти в буфере занимает пакет, и другие. Некоторые работы рассматривали пакеты с несколькими характеристиками, например ценностью и требующимся временем обработки одновременно. Другой тип характеристики — *допуск* (slack) пакета, т.е. разница между временем прибытия пакета и тем временем (дедлайном), когда пакет должен быть

передан или будет утерян. В Главе 2 мы рассматриваем несколько разных постановок задач, в которых пакетам присваиваются разные характеристики и их комбинации. В частности:

— раздел 2.1 вводит основные определения конкурентного анализа и даёт обзор предыдущих работ, применявших его к задачам управления буфером;

— раздел 2.2 посвящён архитектуре с одной очередью и пакетами с разнородным требующимся временем обработки;

— раздел 2.3 переходит к буферам с несколькими исходящими очередями, но рассматривает случай, в котором исходящие очереди имеют разные характеристики (например, пакеты в каждой исходящей очереди имеют одно и то же требуемое время обработки, но эти значения разные в разных очередях);

— раздел 2.4 переходит к общему случаю, когда несколько исходящих очередей могут получать пакеты с разнородными требованиями к времени обработки;

— начиная с раздела 2.5, мы рассматриваем пакеты с несколькими характеристиками; раздел 2.5 посвящён пакетам, которые сочетают разнородные требования к времени обработки и разный размер;

— в разделе 2.6 мы рассматриваем управление буфером с одной очередью и пакетами, у которых меняются требования к времени обработки и ценность;

— раздел 2.7 добавляет к этому ещё одну характеристику, а именно *задержку* (latency), т.е. дедлайн для обработки каждого пакета;

— наконец, раздел 2.8 посвящён новому направлению исследований, в котором алгоритмы управления буфером выбираются при помощи моделей машинного обучения в зависимости от свойств трафика.

1.3 Классификация пакетов

Обычно сервисы на уровне отдельных пакетов представляются в виде *классификаторов пакетов* (packet classifiers). Классификатор — это по сути программа, состоящая из упорядоченного множества условий. Первое по порядку условие, которому подходит входящий пакет, определяет, какие действия к нему нужно

применить. Каждое условие в классификаторе выражается как правило классификации на нескольких полях, с отдельными условиями для каждого поля. Классификаторы пакетов с более чем одним полем постоянно используются в современных сетях передачи данных.

Если правило классификации реагирует только на точное совпадение значений во всех полях, его можно записать, просто конкатенировав эти поля; такие классификаторы легко представить или в памяти с адресацией по содержимому (content-addressable memory, CAM), или простой хеш-функцией, использовав память, линейную от числа правил. Задача становится сложнее, если поле представлено префиксом (т.е. ведётся поиск конкретных значений только для старших битов в поле) или интервалом (который ограничивает значение поля), поскольку конкатенация префиксов или интервалов уже не будет префиксом или интервалом соответственно. Чтобы представить интервал в памяти с адресацией по содержимому, нужно применить ту или иную схему кодирования интервалов, каждая из которых представляет интервал несколькими записями, и если полей с интервалами значений несколько, число записей растёт мультипликативно и может быстро стать слишком большим.

Был разработан целый ряд сложных программных (software-based) подходов [198]. Оценки, полученные методами вычислительной геометрии, говорят, что программный классификатор пакетов с N правилами и $k \geqslant 3$ полями должен использовать либо память $O(kN)$ и время поиска $O(\log N)$, либо память $O(N)$ и время поиска $O(\log^{k-1} N)$ [112,171]. Поэтому программные подходы либо слишком медленны, либо требуют слишком много памяти даже для нескольких префиксных или интервальных полей.

Более сложные сетевые сервисы требуют дополнительной выразительности на существующих полях для классификации или вводят новые поля. Учитывая вышеприведённые теоретические оценки, увеличение выразительности (добавление новых полей) и рост числа полей в классификаторе могут существенно ухудшить эффективность реализаций. В Главе 3 настоящей диссертации мы вводим новый подход и несколько направлений исследований, которые позволяют найти фундаментальный баланс между масштабируемостью и выразительностью, улучшив эффективность представлений классификаторов. Мы также вводим и изучаем структурные свойства, которые позволяют эффективно представлять классификаторы пакетов.

Предшествующие исследования об эффективных классификаторах пакетов в основном концентрировались на поиске семантически эквивалентных представлений. Некоторые вводят проприетарные методы, и большинство подходов основаны на методах булевой минимизации [51, 141, 195]. Такие методы могут уменьшить как ширину классификации (число битов, участвующих в классификации), так и число правил. Но что делать, если оптимизировать классификатор дальше уже невозможно, но он всё ещё не соответствует таким желаемым метрикам эффективности, как требующаяся память или время поиска? Существенное увеличение ширины классификации и числа правил очень сильно влияет на эффективность представлений классификаторов, а значит, и их практическую реализуемость; теоретические оценки показывают, что нет эффективных представлений без экспоненциальной памяти и с подходящим для практических приложений временем поиска. Ответ на этот вопрос может прийти из структурных свойств классификаторов, допускающих эффективные представления.

В Главе 3 мы вводим и изучаем несколько структурных свойств, которые позволяют использовать более эффективные представления и более эффективные алгоритмы поиска: порядковую независимость по правилам, порядковую независимость по действиям, ослабленную семантическую эквивалентность, переупорядочиваемость префиксов и другие. В частности:

— раздел 3.1 представляет задачу классификации пакетов и даёт обзор смежных результатов;

— в разделе 3.2 мы вводим основной инструмент для наших новых методов в классификации пакетов: понятие порядковой независимости и связанные с ним задачи и результаты;

— в разделе 3.3 мы расширяем методы сокращения классификаторов на основе порядковой независимости, разделяя классификатор на несколько групп правил, вводим соответствующие задачи оптимизации, представляем и анализируем алгоритмы для их решения; в частности, здесь мы показываем, как предлагаемые подходы могут помочь разместить IPv6 классификатор в памяти, предназначенной для протокола IPv4 или даже MPLS;

— раздел 3.4 показывает, как классификация пакетов может быть реализована распределённым образом, и как таблицы классификации могут быть эффективно разделены между несколькими сетевыми элементами;

— раздел 3.5 изучает совместную оптимизацию нескольких классификаторов, которые переиспользуют одни и те же подмножества правил (классы; это очень частая на практике ситуация);

— в разделе 3.6 мы рассматриваем возможности для дальнейшего сокращения классификатора в случае, если можно ослабить некоторые из целей классификации, переходя таким образом от точной к приближённой классификации пакетов.

1.4 Алгоритмы на плоскости управления

Парадигма программно-конфигурируемых сетей (software-defined networking, SDN) позволяет сетевым операторам реализовывать сетевые сервисы через централизованный контроллер. Растущий интерес к SDN привёл к тому, что многие сетевые сервисы были реализованы на контроллерах, написанных на разных языках и поддерживающихся разными организациями, так что ни один контроллер не способен предоставить все сетевые сервисы, и даже если он предоставляет все нужные сервисы, вряд ли он может делать это наилучшим образом в каждом конкретном случае. Поиск новых абстракций может существенно упростить управление сетью и увеличить гибкость плоскости управления.

Изначально предполагалось, что стандартизация южного API может выполнить требования SDN, но затем стало ясно, что один виртуальный контроллер недостаточен для управления сетью. Разделение плоскостей управления и данных и стандартизация южного API только улучшают разрешение управления, но не решают проблему в целом: если нужно, чтобы более одного виртуального контроллера работали вместе, то восточно-западный API (east-west API) тоже должен быть стандартизован. До SDN это делалось для каждого протокола отдельно, но с ростом числа и разнообразия протоколов это становится слишком сложным решением. Стандартизация восточно-западного API должна быть независима от протоколов, и мы предполагаем, что для этой цели можно использовать инфраструктуру протоколов маршрутизации.

Чтобы упростить управление сетью, можно предложить представлять исходную сеть более простой и/или дешёвой сетью, которая всё ещё соответствует нужным свойствам исходной сети, что приводит к поиску баланса между простотой представления и эффективностью переиспользования базовой инфра-

структуры. Сохранение свойств сети позволяет сервисам работать на более простом представлении прозрачным образом, а понимание ограничений конкретной сети показывает, в какие ресурсы нужно инвестировать в первую очередь. Уже предпринимались попытки виртуализировать конкретные архитектуры сетей [49, 121], но в предыдущих работах не был представлен единый процесс построения упрощённого представления сети, которое бы сохраняло её свойства по пропускной способности и маршрутизации.

В разделе 4.1 мы делаем первые шаги в этом направлении, вводя новые структурные свойства сетей и рассматривая преобразования, которые сохраняют эти свойства.

Чтобы соблюсти ограничения на пропускную способность передачи данных между хранилищами, датацентры хранят данные в разных связанных друг с другом местах. Современные приложения больших данных являются крайне распределёнными, и запросы должны удовлетворять разным и зачастую конфликтующим целям: минимизировать время отклика, добиться ценовой эффективности и многое другое [47, 74, 150]. Задачи типа *compute-aggregate*, где несколько блоков данных (data chunks) должны быть агрегированы в стоке сети, представляют собой важный класс приложений больших данных, реализуемых в современных датацентрах. Чтобы вычислить окончательный результат такой задачи, несколько блоков данных нужно агрегировать в одном месте, выполнив при этом общие ограничения на время отклика.

В работах [88, 89] изучались оптимальные структуры для задач типа compute-aggregate, которые минимизируют время отклика. К сожалению, окончательный результат может существенно зависеть от свойств сетевых транспортов и использования сетевой инфраструктуры. Чтобы справиться с этими ограничениями, в разделе 4.2 мы предлагаем модель, которая строит расписание агрегаций с учётом бюджетных ограничений, которые можно ввести для каждой задачи типа compute-aggregate. Затем сетевой транспорт может перераспределить найденное расписание во времени, чтобы оптимизировать требующиеся целевые функции: время отклика или пропускную способность. Мы считаем, что этот подход позволит отделить задачи оптимизации от сетевых транспортов и предоставить более детальный контроль, который приведёт к более эффективному использованию сетевой инфраструктуры.

Раздел 4.3 посвящён применению методов конкурентного анализа онлайн-

алгоритмов, которые мы разрабатывали в других контекстах, к эффективному распределению запросов в «умных» энергосетях. Быстрое развитие распределённых ресурсов генерации, особенно дизельных и солнечных генераторов, позволил отдельным группам домов или бизнесов закрыть все или многие из их эластичных потребностей в электроэнергии при помощи локальной генерации. Этот паттерн использования особенно часто встречается в развивающихся странах, например Индии, которые часто страдают от дефицита электричества, и в результате миллионы пользователей часто получают недостаточную мощность [61, 67]. Там пользователи используют локальную генерацию, чтобы дополнить центральную энергосеть, которая может быть ненадёжной и ограниченной по мощности. Мы предполагаем, что в будущем географически близкие микросети будут оппортунистически образовывать связи между собой, чтобы улучшить надёжность, что естественным образом повторяет процесс самоорганизации, который изначально привёл к образованию энергосетей, до того, как централизованная генерация по сути уничтожила микрогенерацию около ста лет назад. В разделе 4.3 мы рассматриваем эффективное удовлетворение запросов в контексте многосвязных микросетей, где один и тот же запрос можно удовлетворить при помощи разных ресурсов порождения: локальных, близлежащих или на региональной сети, что определяет *вектор балансирования загрузки* для каждого запроса, который определяет, какие генераторы подключены и/или доступны для его выполнения.

Масштабируемый мониторинг сетевого трафика также становится всё сложнее из-за постоянного роста объёма трафика, разнородности сетевых устройств и неравномерности нагрузок. Во-первых, в то время как трафик растёт и по объёму, и по числу потоков, вычислительные ресурсы и память, требующиеся для мониторинга трафика на сетевых элементах, тоже растут. Во-вторых, сети состоят из всё более разнородных элементов, от базовых IoT (Internet of Things) устройств с крайне ограниченными возможностями до самых мощных базовых маршрутизаторов, способных пересылать миллионы одновременных потоков. В-третьих, нагрузка мониторинга на разных сетевых элементах неоднородна, и один элемент может оказаться перегруженным, даже если на других есть свободные ресурсы. Даже такая относительно простая задача, как подсчёт размеров всех потоков пакетов на сетевом элементе, может существенно нагрузить ограниченные ресурсы плоскости данных, когда число потоков становится

большим.

Один из способов обеспечить масштабирование, недоступное для одного сетевого элемента, состоит в том, чтобы проводить мониторинг потоков трафика при помощи объединённых ресурсов сразу нескольких сетевых элементов. Если один сетевой элемент перегружается задачей мониторинга, её исполнение можно сместить с перегруженного элемента вдоль пути потока к другому элементу, у которого ресурсов достаточно. Это перемещение задач позволяет сети эффективно использовать её глобальные ресурсы вычислений и памяти и избегать локальных перегрузок, но в то же время задача мониторинга становится подвержена проблемам, связанным с сетевым шумом. В разделе 4.4 мы вводим и изучаем различные модели сетевого шума, предлагаем и изучаем алгоритмы, которые исполняют задачу мониторинга на нескольких сетевых элементах корректно, несмотря на сетевой шум на путях между элементами.

В целом, диссертация покрывает широкий спектр алгоритмических и аналитических задач, связанных с сетевыми приложениями, вводит новые алгоритмы, новые методы анализа алгоритмов и новые теоретические результаты для сетевых алгоритмов и на плоскости данных, и на плоскости управления. Результаты, выносящиеся на защиту, перечислены в разделе 5.1. Результаты этой работы могут и уже активно используются для многих сетевых приложений, включая разработку архитектур коммутаторов и других сетевых элементов, классификацию пакетов, планирование, виртуализацию сетей, подсчёт пакетов в потоках и другие.

1.5 Апробация работы: список публикаций

Каждый из ключевых результатов, представленных в диссертации, был опубликован в одной из рецензируемых научных статей, представленных ниже. В тексте диссертации все публикации цитируются при помощи арабских цифр (например, [1]), причём работы из этого списка имеют номера от 1 до 42, а нумерация других работ начинается с 43.

Работы, опубликованные в ведущих журналах (Q1 в Web of Science / Scopus)

1. V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Packet Classifiers With

Controlled Accuracy. IEEE Transactions on Networking, vol. 29, no. 3, 2021, pp. 1141–1154.

Вклад диссертанта: теоремы 4 и 5 в соавторстве с V.D., K.K.

2. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. IEEE Transactions on Networking, vol. 29, no. 1, 2021, pp. 275–288.

   Вклад диссертанта: теорема 3; теоремы 1, 5 в соавторстве с K.K.

3. A. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Competitive buffer management for packets with latency constraints. Computer Networks, vol. 189, no. 22, Article ID 107942, 2021.

   Вклад диссертанта: теорема 1; теоремы 2, 3, 4, 5 в соавторстве с A.D.

4. P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalization and taxonomy of compute-aggregate problems for cloud computing applications. Computer Networks, vol. 189, no. 22, Article ID 107915, 2021.

   Вклад диссертанта: теоремы 3, 4, 5, 7, 15, 16, 17, 18 в соавторстве с P.C., A.D., K.K., A.S.

5. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. Towards Software-Defined Buffer Management. IEEE Transactions on Networking, vol. 28, no. 5, 2020, pp. 2337–2349.

   Вклад диссертанта: теоремы 1, 2, 3, 4.

6. P. Chuprikov, S.I. Nikolenko, K. Kogan. Towards Declarative Self-Adapting Buffer Management. ACM SIGCOMM Computer Communication Review, vol. 50, no. 3, 2020, pp. 30–37.

   Вклад диссертанта: применение задачи о многоруких бандитах в данном контексте; алгоритмы на основе задачи о многоруких бандитах.

7. V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. IEEE Transactions on Networking, vol. 28, no. 3, 2020, pp. 1088–1101.

   Вклад диссертанта: теоремы 8 и 9 в соавторстве с P.C., K.K.

8. P. Chuprikov, S.I. Nikolenko, K. Kogan, A. Davydow. Priority Queueing for Packets with Two Characteristics. IEEE Transactions on Networking, vol. 26, no. 1, 2018, pp. 342–355.

   Вклад диссертанта: теоремы 4, 5, 6, 13; теоремы 8, 9, 10 в соавторстве с P.C., A.D.

9. P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Admission Control in Shared Memory Switches. Journal of Scheduling, vol. 21, no. 5, 2018, pp. 533–543.

   Вклад диссертанта: теоремы 1, 2, 3, 7; теорема 5 в соавторстве с P.E., A.K., K.K., A.S.; экспериментальное исследование в соавторстве с A.S.

10. K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. Efficient FIB Representations on Distributed Platforms. IEEE Transactions on Networking, vol. 25, no. 6, 2017, pp. 3309–3322.

    Вклад диссертанта: алгоритмы 1, 2, 3, 4, 5 в соавторстве с K.K.; анализ сложности алгоритмов; экспериментальное исследование в соавторстве с A.S.

11. P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Heterogeneous Packet Processing in Shared Memory Buffers. Journal of Parallel and Distributed Computing, vol. 99, 2017, pp. 1–12.

    Вклад диссертанта: теоремы 1, 2, 3, 6, 7; теорема 5 в соавторстве с A.K., K.K., A.S.; экспериментальное исследование в соавторстве с A.S.

12. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. The Impact of Processing Order on Performance: A Taxonomy of Semi-FIFO Policies. Journal of Computer and System Sciences, vol. 88, 2017, pp. 220–235.

    Вклад диссертанта: теоремы 5, 7, 8, 9, 10, 11, 14; определения ленивых алгоритмов и теоремы 1, 6, 13 в соавторстве с K.K., A.L., A.S; экспериментальное исследование в соавторстве с A.S.

13. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub, M. Segal. Large profits or fast gains: A Dilemma in Maximizing Throughput with Applications to Network Processors. Journal of Network and Computer Applications, vol. 74, 2016, pp. 31–43.

Вклад диссертанта: теоремы 3, 4, 11, 13; теоремы 5, 8, 12 и следствие 10 в соавторстве с K.K., A.L., G.S., M.S.; леммы 1 и 2 в соавторстве с K.K., A.L.; экспериментальное исследование.

14. K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. Exploiting Order Independence for Scalable and Expressive Packet Classification. IEEE Transactions on Networking, vol. 24, no. 2, 2016, pp. 1251–1264.

    Вклад диссертанта: теоремы 5, 6; теоремы 1, 2 и соответствующие алгоритмы для сокращения классификаторов в соавторстве с K.K.; экспериментальное исследование.

Публикации в трудах ведущих конференций (ранг A/A* по рейтингу CORE)

15. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. 27th IEEE International Conference on Network Protocols (ICNP 2019), 2019, pp. 1–11.

    Вклад диссертанта: см. [2] (это первая статья в трудах конференций, объединённая в журнальной статье [2]).

16. V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Classifiers with Controlled Accuracy. 38th IEEE International Conference on Computer Communications (INFOCOM 2019), 2019, pp. 2044–2052.

    Вклад диссертанта: см. [1] (это её конференционная версия).

17. V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. 26th IEEE International Conference on Network Protocols (ICNP 2018), 2018, pp. 121–131

    Вклад диссертанта: см. [17] (это её конференционная версия).

18. S.I. Nikolenko, K. Kogan, A. Fernàndez Anta. Network Simplification Preserving Bandwidth and Routing Capabilities. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

    Вклад диссертанта: теоремы 3, 4; теоремы 1, 2, 5, 6 в соавторстве с K.K., A.F.A.; конструкция парадокса Браесса в разделе IV; экспериментальное исследование.

19. A.P. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Throughput Optimization with Latency Constraints. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

    Вклад диссертанта: см. [3] (это её конференционная версия).

20. P. Chuprikov, K. Kogan, S.I. Nikolenko. General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Вклад диссертанта: постановка задач, подходы к сокращению ширины классификаторов (раздел VI).

21. P. Chuprikov, A.P. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Planning in Compute-Aggregate Problems as Optimization Problems on Graphs. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Вклад диссертанта: см. [4] (это первая статья в трудах конференций, объединённая в журнальной статье [4]).

22. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. A Programmable Buffer Management Platform. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

    Вклад диссертанта: см. [5] (это вторая статья в трудах конференций, объединённая в журнальной статье [5]).

23. P. Chuprikov, S.I. Nikolenko, K. Kogan. On Demand Elastic Capacity Planning for Service Auto-scaling. Proceedings of 2016 IEEE International Conference on Computer Communications (INFOCOM 2016), 2016.

    Вклад диссертанта: постановки задач, конструкция модели, алгоритм NRAP в соавторстве с К.К.

24. K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. FIB Efficiency in Distributed Platforms. Proc. 24th IEEE International Conference on Network Protocols (ICNP 2016), 2016.

    Вклад диссертанта: см. [10] (это её конференционная версия).

25. P. Chuprikov, K. Kogan, S.I. Nikolenko. Priority Queueing with Multiple Packet Characteristics. Proceedings of IEEE INFOCOM 2015, pp. 1418–1426.

Вклад диссертанта: см. [8] (это её конференционная версия).

26. K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). ACM SIGCOMM 2014, ACM Press, 2014, pp. 15–26.

    Вклад диссертанта: см. [14] (это её конференционная версия).

27. P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Shared-Memory Buffer Management for Heterogeneous Packet Processing. 34th International Conference on Distributed Computing Systems (ICDCS 2014).

    Вклад диссертанта: см. [11] (это её конференционная версия).

Другие публикации

28. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Online Scheduling FIFO Policies with Admission and Push-Out. Theory of Computing Systems, vol. 58, no. 2, 2016, pp. 322-344.

    Вклад диссертанта: теоремы 1, 2, 3, 4, 6; теоремы 5 и 7 в соавторстве с K.K., A.L., A.S.; экспериментальное исследование.

29. V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Distributed Counting along Lossy Paths Without Feedback. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), Lecture Notes in Computer Science, vol. 11685, 2018, pp. 30–33 (CORE rank B).

    Вклад диссертанта: см. [2] (это вторая статья в трудах конференций, объединённая в журнальной статье [2]).

30. P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalizing Compute-Aggregate Problems in Cloud Computing. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), 2018, pp. 377–391 (CORE rank B).

    Вклад диссертанта: см. [4] (это вторая статья в трудах конференций, объединённая в журнальной статье [4]).

31. P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Essential Traffic Parameters for Shared Memory Switch Performance. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015), Lecture Notes in Computer Science, vol. 9439, Springer, 2015, pp. 61–75. (CORE rank B).

    Вклад диссертанта: см. [9] (это её конференционная версия).

32. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. A Taxonomy of Semi-FIFO Policies. Proc. 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), 2012, pp. 295–304. (CORE rank B).

    Вклад диссертанта: см. [12] (это её конференционная версия).

33. K. Kogan, S.I. Nikolenko, V. Demianiuk, P. Chuprikov, A.P. Davydow. Personal Insights on Three Research Directions in Networked Systems. Proc. 10th International Conference on Communication Systems and Networks (COMSNETS 2018), IEEE, 2018.

    Вклад диссертанта: большая часть текста статьи (это постановочная статья, не содержащая новых результатов).

34. P. Chuprikov, K. Kogan, S.I. Nikolenko. How to implement complex policies on existing network infrastructure. Proc. 4th Symposium on SDN Research (SOSR 2018), ACM, 2018, pp. 1–8.

    Вклад диссертанта: эвристики булевой минимизации и результаты о несравнимости (разделы 4.3, 4.4).

35. S.I. Nikolenko, K. Kogan, G. Retvari, E.R. Berczi-Kovacs, A. Shalimov. How to Represent IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes. Proc. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): GI 2016: 9th IEEE Global Internet Symposium (IEEE GI 2016), 2016.

    Вклад диссертанта: алгоритмы 1, 2, 3, 4, 5, 6 в соавторстве с K.K., G.R., E.B.K.; экспериментальное исследование в соавторстве с A.S.

36. K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, P. Eugster. BASEL (Buffering Architecture SpEcification Language). Proc. 12th

ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2016), 2016, pp. 69–74.

Вклад диссертанта: см. [5] (это первая статья в трудах конференций, объединённая в журнальной статье [5]).

37. K. Kogan, S.I. Nikolenko, P. Eugster, E. Ruan. Strategies for Mitigating TCAM Space Bottlenecks. 22nd IEEE Annual Symposium on High-Performance Interconnects (HOTI 2014), IEEE, 2014, pp. 25–32.

    Вклад диссертанта: булевы эвристики для оптимизации классификаторов (раздел VII); экспериментальное исследование.

38. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub. Balancing Work and Size with Bounded Buffers. Proc. 6th International Conference on Communication Systems and Networks (COMSNETS 2014), 2014, pp. 1–8.

    Вклад диссертанта: см. [13] (это её конференционная версия).

39. K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Multi-Queued Network Processors for Packets with Heterogeneous Processing Requirements. Proc. 5th International Conference on Communication Systems and Networks (COMSNETS 2013), 2013, pp. 1–10.

    Вклад диссертанта: теоремы 1, 2, 3, 4, 6, 8, 9, 10, 11; теоремы 5, 7 в соавторстве с K.K. и A.L.; экспериментальное исследование в соавторстве с A.S.

40. K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient Demand Assignment in Multi-Connected Microgrids with a Shared Central Grid. Proc. 3rd IFIP Conference on Sustainable Internet and ICT for Sustainability 2013 (SustainIT 2013), 2013.

    Вклад диссертанта: оценки на аппроксимационные факторы, показанные в таблице 1, в соавторстве с K.K., A.L.

41. K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient demand assignment in multi-connected microgrids. Proc. 4th Energy-Efficient Computing and Networking (e-Energy 2013), pp. 277–278.

Вклад диссертанта: теоремы 1, 2, 3, 4; теоремы 5, 6 в соавторстве с K.K., A.L.

42. K. Kogan, S.I. Nikolenko, W. Culhane, P. Eugster, E. Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013), pp. 153–154.

    Вклад диссертанта: применения MinDNF и анализ сложности минимизации TCAM (раздел 2.2).

Глава 2 Алгоритмы управления буфером и их анализ

2.1 Введение в конкурентный анализ для управления буфером и обзор связанных работ

В этом разделе мы начинаем со введения, в котором даются общие определения конкурентного анализа и краткий обзор основных результатов в конкурентном анализе для алгоритмов управления буфером, которые предшествовали выносящимся на защиту работам. Обзор литературы в этом разделе следует работе [167].

Сетевые коммутаторы выполняют две основные задачи:

(1) изменение свойств отдельных пакетов (перекрашивание, инкапсуляция);

(2) изменение свойств потоков пакетов (rate-limiting, shaping); работа с потоками пакетов может требовать буферизации.

Разработка и управление архитектурой буфера — это одна из фундаментальных задач в разработке сетевых коммутаторов. Чтобы учесть размер сети, число запросов на передачу данных и эффективное переиспользование сетевой инфраструктуры, в Интернете в настоящее время обычно реализуется «best-effort» обслуживание, которое приоритезирует некоторые типы трафика, чтобы оптимизировать заданные целевые функции (например, экономические свойства сети). К сожалению, существующие сети в основном построены исключительно для того, чтобы выполнять свойство равного доступа к сетевым ресурсам (fairness), в то время как сетевые операторы уже давно ищут способы реализовать и другие экономические модели (например, оптимизацию пропускной способности или дохода оператора). Большинство алгоритмов управления буфером основаны на одной простой характеристике (например, размер занятой

части буфера), но трафик в современных сетях также имеет важные дополнительные характеристики (требующееся время обработки, ценность и т.д.), которые при этом не учитываются. Учёт новых характеристик, помимо равного доступа, в принятии решений о размещении пакетов в буфере приводит к новым задачам для исследований и новым проблемам в оптимизации качества работы реализаций коммутационных архитектур.

Сетевой элемент с ограниченным буфером выполняет три основные задачи, а именно:

(1) *управление буфером*, т.е. принятие решений о допуске в буфер вновь прибывших пакетов (admission control) и вытеснение уже содержащихся в буфере пакетов (push-out), если такая возможность поддерживается архитектурой;

(2) *обработка*, т.е. принятие решений о том, какие пакеты из лежащих в настоящее время в буфере будут обрабатываться;

(3) *передача*, т.е. принятие решений о том, какие именно уже обработанные пакеты нужно передавать дальше, и собственно передача (при этом обработанный пакет покидает буфер).

Мы предполагаем, что время дискретно и делится на циклы обработки (processing cycles). Пакет *полностью обработан*, если он был выбран для обработки в течение по меньшей мере стольких циклов, сколько составляет его требующееся время обработки.

Каждый сетевой пакет может иметь следующие характеристики:

(1) *требующееся время обработки* (required processing), т.е. в течение скольких циклов обработки пакет должен быть выбран для этой обработки, прежде чем его можно будет передавать дальше;

(2) *ценность* (value), т.е. то, сколько пакет добавляет к целевой функции после того, как он передан;

(3) *исходящий порт*, т.е. пункт назначения пакета внутри коммутатора; в постановках задач управления буфером с несколькими исходящими портами обычно предполагается, что обработка происходит независимо в каждом порте, и становится предпочтительным иметь больше занятых исходящих портов в каждый момент времени;

(4) *размер*, т.е. сколько слотов (или байтов) пакет занимает в буфере.

Цель алгоритма управления буфером состоит в том, чтобы максимизировать общую ценность переданных дальше пакетов. В различных постановках задачи может предполагаться, что некоторые характеристики являются одинаковыми для всех пакетов. Например, в этом разделе мы не рассматриваем размер пакетов, т.е. предполагаем, что каждый пакет занимает ровно один слот в буфере общего размера B, но в разделе 2.5 мы будем рассматривать и пакеты переменного размера.

Конкурентный анализ — это методология, которая позволяет доказывать теоретические оценки, гарантирующие определённую пропускную способность для данного онлайн-алгоритма в худшем случае, для любого набора пакетов на входе.

Определение 2.1.1. Онлайн-алгоритм ALG является *α-конкурентным* (*α-competitive*) по отношению к некоторой целевой функции f для некоторого $\alpha \geqslant 1$, которое называется *конкурентностью* (*competitive ratio*), если для любой входной последовательности пакетов σ значение целевой функции на результате работы алгоритма ALG составляет по крайней мере долю $1/\alpha$ от значения целевой функции на решении, полученном оптимальным всевидящим оффлайн-алгоритмом, который мы будем называть OPT.

В конкурентном анализе есть три основных типа результатов:

(1) *нижние оценки* на конкурентность некоторого конкретного алгоритма можно доказать, просто предъявив конкретный сложный пример, на котором этот алгоритм работает значительно хуже оптимального; это самый простой тип результатов;

(2) *общие нижние оценки* на конкурентность иногда удаётся показать для всех детерминированных онлайн-алгоритмов или для некоторого конкретного подкласса алгоритмов (например, для всех приоритетных очередей); в этом случае, нижнюю оценку надо доказывать в состязательном ключе (adversarial bound), строя сложный пример в зависимости от выборов, которые может сделать данный онлайн-алгоритм;

(3) наконец, *верхние оценки* на конкурентность представляют собой общие утверждения, которые должны выполняться для всех возможных входов,

и доказательство верхней оценки обычно представляет собой гораздо более сложную математическую задачу.

На протяжении этой главы многие результаты будут представлять собой нижние, общие нижние или верхние оценки на конкурентность различных алгоритмов управления буфером в различных постановках задачи.

Задача 2.1.1 (оценки конкурентности)**.** *Для заданной архитектуры коммутатора, данного набора характеристик сетевых пакетов (например, требующееся время обработки, ценность, размер и т.д.) и данного онлайн-алгоритма* ALG *в этой постановке задачи доказать нижние и верхние оценки на его конкурентность относительно целевой функции взвешенной пропускной способности (общей ценности пакетов, переданных алгоритмом).*

Задача 2.1.2 (конкурентные алгоритмы)**.** *Для заданной архитектуры коммутатора и данного набора характеристик пакетов (например, требующееся время обработки, ценность, размер и т.д.) найти онлайн-алгоритмы, которые имели бы наилучшие (самые маленькие) возможные верхние оценки на конкурентность.*

На протяжении этого раздела (который представляет собой обзор предшествующей литературы) мы представляем алгоритмы и нижние и верхние оценки на их конкурентность относительно различных постановок задачи управления буфером; постановки задач различаются в том, какие характеристики пакетов предполагаются одинаковыми (равномерными, uniform), а какие могут варьироваться, а также иногда дополнительными ограничениями, связанными с допуском пакетов, порядком передачи и/или обработки, или характеристиками допускаемых пакетов. Часто также рассматривается важный частный случай *пакетов с двумя значениями*, когда та или иная характеристика может принимать только два возможных значения, обычно 1 и некоторое $x > 1$.

Равномерное время обработки, равномерная ценность, коммутатор с общей памятью. Поскольку в этой постановке задачи все пакеты идентичны, задача для одной очереди с одним исходящим портом становится тривиальной. Рассмотрим коммутатор $M \times N$ с общей памятью, в которую помещается B пакетов с отдельным процессором на каждом исходящем порте. Каждому пакету требуется один цикл обработки, и все пакеты имеют равную ценность;

цель состоит в том, чтобы максимизировать число переданных пакетов. Каждый пакет помечен индексом исходящего порта, где его нужно будет обработать и затем передать.

Алгоритмы без выталкивания (non-push-out policies). Кессельман и Мансур [136] показали состязательную логарифмическую нижнюю оценку: ни один онлайн-алгоритм без выталкивания не может добиться конкурентности меньше чем $d/2$ для $d = \log_d N$. С другой стороны, они также представили алгоритм Harmonic, который назначает приблизительно $1/i$ часть буфера для i-й наибольшей очереди и показали, что для одного из вариантов этого алгоритма, Parametric Harmonic, верна верхняя оценка на конкурентность $c \log_c N + 1$.

Алгоритмы с выталкиванием (push-out policies). Наилучший известный здесь алгоритм — это Longest Queue Drop (LQD): принимать пакеты жадно, пока буфер не заполнится, а если буфер полон, то принять новый пакет и вытолкнуть (drop, push out) пакет из самой длинной очереди (т.е. очереди к исходящему порту, которому назначено больше всего пакетов). Айелло с соавторами [44, 114] показали, что конкурентность LQD находится между $\sqrt{2}$ и 2; они также показали не-константные нижние оценки для других известных алгоритмов и общую состязательную нижнюю оценку $\frac{4}{3}$ на конкурентность любого онлайн-алгоритма.

Равномерное время обработки, равномерная ценность, несколько раздельных очередей. В коммутаторе $N \times 1$, где каждая из N входных очередей имеет отдельный буфер размера B, алгоритм управления буфером должен принять решение, из какой входной очереди взять пакет, а также установить политики принятия пакетов во входные очереди. Азар и Рихтер [59] доказали, что любой сохраняющий работу (work-conserving) алгоритм, т.е. алгоритм, который продолжает обработку, пока есть хотя бы одна непустая входная очередь, будет не более чем 2-конкурентным; это покрывает как случай алгоритмов с выталкиванием, так и без, поскольку с равномерной ценностью и временем обработки выталкивание не нужно. Они также доказали нижнюю оценку $2 - \frac{1}{N}$ для $B = 1$ и $1.366 - \Theta(1/N)$ для произвольного B [59]. Альберс и Шмидт [48] показали, что жадный алгоритм, который берёт пакет из самой длинной очереди, является не менее чем $\left(2 - \frac{1}{B}\right)$-конкурентным, и разработали новый алгоритм SemiGreedy, для которого доказали верхнюю оценку конкурентности в 1.889 для

произвольного B и 1.857 для $B = 2$, что в последнем случае является точной оценкой. Наконец, Азар и Литичевский [58] закрыли это направление исследований, разработав детерминированный алгоритм, чья конкурентность стремится к $\frac{e}{e-1} \approx 1.582$ для произвольного B, а соответствующая нижняя оценка уже была показана в работе [59].

Равномерное время обработки, переменная ценность, одна очередь. В этой постановке задачи исходящий порт только один (одна очередь), и каждый пакет полностью обрабатывается за один цикл; однако у пакетов различная ценность, и алгоритм управления буфером должен стараться выталкивать пакеты с малой ценностью и обрабатывать и передавать пакеты с высокой ценностью. Легко показать, что алгоритм Priority Queue (PQ), который упорядочивает пакеты по ценности и выталкивает наименьшие, является оптимальным. Поэтому исследования в этой области были посвящены моделям с дополнительными ограничениями. Рассматривались алгоритмы без выталкивания, которым нельзя выбрасывать уже принятые пакеты, и модель FIFO, где пакеты требуется передать в том же порядке, в котором они прибывали. Другой важный частный случай — это случай двух возможных ценностей, 1 и $V > 1$.

Алгоритмы без выталкивания. Айелло с соавторами [52] рассмотрели пять онлайн-алгоритмов для случая с двумя ценностями, рассмотрев в том числе случаи $V = 1$, $V = 2$ и $V = \infty$. Андельман, Мансур и Чжу разработали детерминированный алгоритм Ratio Partition, который достигает оптимального значения конкурентности $\left(2 - \frac{1}{V}\right)$ [212]. В случае произвольной ценности между 1 и $V > 1$ они показали, что оптимальная конкурентность составляет $\ln V$, и доказали асимптотически точные нижние и верхние оценки $1 + \ln V$ и $2 + \ln V + O(\ln^2 V/B)$ [52, 212].

Алгоритмы с выталкиванием. В случае с выталкиванием, как мы уже говорили выше, алгоритм PQ, которому разрешено переупорядочивать пакеты, является оптимальным. Для модели FIFO исследователи доказали целый ряд состязательных нижних оценок, кульминацией которых стала нижняя оценка 1.419, доказанная Кессельманом, Мансуром и ван Штее [138], которая применима ко всем алгоритмам, а также более сильная оценка 1.434 для $B = 2$ [52, 212]. С другой стороны, в этой простой модели жадный FIFO алгоритм с выталкиванием (принимаем каждый пакет в конец очереди, в случае переполнения бу-

фера выталкиваем пакет с наименьшей ценностью) является 2-конкурентным; это показали Кессельман с соавторами [133]. В случае с двумя ценностями они доказали состязательную нижнюю оценку 1.282, а верхняя оценка постепенно улучшалась много раз: для жадного алгоритма с выталкиванием верхнюю оценку $\left(2 - \frac{2}{V+1}\right)$ доказали в [133], Кессельман и Мансур разработали метод анализа с ограниченными потерями (loss-bounded analysis) [135], Лоткер и Патт-Шамир разработали алгоритм с верхней оценкой $\left(1 + \frac{1}{\sqrt{V}} + O(\frac{1}{V})\right)$ [159]; оптимальным же алгоритмом стал Account Strategy, разработанный Энглертом и Вестерманном [102]. Они показали состязательную нижнюю оценку $r = \frac{1}{2}(\sqrt{13}-1) \approx 1.303$ для любого $B \geqslant 2$ и $r_\infty = \sqrt{2} - \frac{1}{2}(\sqrt{5+4\sqrt{2}}-3) \approx 1.282$ для $B \to \infty$, а также показали, что Account Strategy достигает конкурентности r для произвольного B и r_∞ для $B \to \infty$. Таким образом, в случае с двумя ценностями и выталкиванием зазор между нижними и верхними оценками был закрыт полностью.

Равномерное время обработки, переменная ценность, несколько раздельных очередей. Кавахара с соавторами [124] рассматривают коммутатор $N \times 1$ с N раздельными очередями, у каждой из которых есть свой собственный буфер размера B и соответствующая ему ценность α_j, $1 = \alpha_1 \leqslant \ldots \leqslant \alpha_N = \alpha$. Алгоритм выбирает одну из N очередей, максимизируя общую ценность переданных пакетов; в работе [124] доказаны точные (соответствующие друг другу) нижняя и верхняя оценка на конкурентность алгоритма PQ в виде $1 + \frac{\sum_{j=1}^{n'} \alpha_j}{\sum_{j=1}^{n'+1} \alpha_j}$, где $n' = \arg\max_n \frac{\sum_{j=1}^{n} \alpha_j}{\sum_{j=1}^{n+1} \alpha_j}$, а также состязательная нижняя оценка $1 + \frac{\alpha^3+\alpha^2+\alpha}{\alpha^4+4\alpha^3+3\alpha^2+4\alpha+1}$ для любого онлайн-алгоритма. Азар и Рихтер [59] показали, что любой r-конкурентный алгоритм для FIFO очереди с переменной ценностью пакетов можно преобразовать в $2r$-конкурентный алгоритм для нескольких очередей. Кобаяси с соавторами [145] доказали, что любой r-конкурентный алгоритм с равномерными ценностями и несколькими очередями можно преобразовать в $\min\left\{Vr, \frac{Vr(2-r)+r^2-2r+2}{V(2-r)+r-1}\right\}$-конкурентный алгоритм для случая с двумя ценностями.

Равномерное время обработки, коммутатор CIOQ. Критически важным аспектом архитектуры коммутатора является размещение буферов. В архитектурах с очередями на выходах (output queueing, OQ) пакеты, приходящие со входов коммутатора, немедленно проходят через коммутационную матрицу

(switching fabric) и присоединяются к очереди на исходящем порте коммутатора. Таким образом, архитектуры OQ позволяют максимизировать пропускную способность и предоставляют возможность точного контроля над задержками при передаче пакетов (latency). Однако, чтобы избежать переполнений, внутренняя скорость OQ-коммутатора должна быть не меньше, чем суммарная пропускная способность всех входящих очередей, и столь высокие требования к мультипликатору увеличения скорости (speedup) оказывается нелегко удовлетворить. Это обстоятельство привело к значительному интересу к IQ-коммутаторам (input queueing), в которых прибывающие пакеты помещаются в очереди непосредственно на входных портах. Затем алгоритм управления буфером выбирает, какие пакеты из входящих очередей пересекут коммутационную матрицу и будут переданы через соответствующие исходящие порты. Хорошо известно, что использование IQ-архитектур может привести к низкой пропускной способности, и оно не позволяет управлять задержкой передачи на уровне коммутатора. Главная проблема IQ-архитектур — это блокирование головы очереди (head-of-line blocking, HOL blocking), которое происходит, когда пакеты в начале нескольких входных очередей должны отправиться в один и тот же исходящий порт коммутатора, что приводит к недоиспользованию возможностей коммутатора для обработки и передачи пакетов. Чтобы смягчить последствия этой проблемы, можно поддерживать отдельную очередь для каждого выхода на каждом входе. Этот метод известен как виртуальные исходящие очереди (virtual output queuing, VOQ). Другой способ получить гарантии на задержку в IQ-коммутаторе, близкие к гарантиям, возможным для OQ-коммутаторов, — это увеличить мультипликатор (speedup) S коммутационной матрицы. Говорят, что коммутатор имеет мультипликатор S, если коммутационная матрица работает в S раз быстрее, чем каждый входной или исходящий порт. Таким образом, OQ-коммутатор имеет мультипликатор N, где N — число входных и исходящих портов, а IQ-коммутатор имеет мультипликатор 1. Для значений S между 1 и N пакетам будут требоваться буферы на входе, где пакеты могут ожидать перехода через коммутационную матрицу, а также буферы у исходящих портов. Такая архитектура называется совмещённой (Combined-Input-and-Output Queued switch, CIOQ).

Равномерная ценность. Рассмотрим $N \times N$ CIOQ-коммутатор с мультипликатором S. Пакеты равного размера прибывают на входные порты, и каждый

пакет помечен индексом исходящего порта, через который он должен покинуть коммутатор. Каждый пакет помещается во входную очередь, соответствующую его исходящему порту; когда пакет пересекает коммутационную матрицу, он помещается в исходящую очередь и остаётся там до передачи через этот исходящий порт. Для пакетов с равномерной ценностью Кессельман и Розен [139] предложили алгоритм без выталкивания, который является 3-конкурентным для любого S и 2-конкурентным для $S = 1$.

Переменная ценность. Для не более чем m различных ценностей пакетов из интервала $[1, V]$ Кессельман и Розен [139] доказали, что два разных алгоритма с выталкиванием являются $4S$-конкурентным и $8 \min\{m, 2 \log V\}$-конкурентным соответственно. Азар и Рихтер [60] предложили алгоритм с выталкиванием β-PG с параметром β; Кессельман с соавторами [131] показали, что конкурентность β-PG составляет не более 7.5 для $\beta = 3$ и не более 7.47 для $\beta = 2.8$. Кессельман и Розен [140] рассмотрели CIOQ-коммутатор с буферами, реализующими приоритетные очереди (передаются пакеты с наибольшей ценностью), и показали, что этот алгоритм является 6-конкурентным для любого S.

Равномерные требования к обработке, перекрёстная архитектура. В коммутаторах с перекрёстной архитектурой (crossbar switching) в дополнение к входящим и исходящим очередям есть ещё небольшие буферы, располагающиеся на каждом пересечении (отдельный буфер для каждой пары входящего и исходящего портов), что существенно упрощает процесс планирования. Для пакетов с однородными размером и ценностью Кессельман с соавторами [131] ввели жадный алгоритм управления буфером с конкурентностью между $\frac{3}{2}$ и 4, а также показали общую нижнюю оценку $\frac{3}{2}$ для буферов единичного размера. Для переменной ценности и PQ буферов они предложили жадный алгоритм управления буфером с выталкиванием с фактором выталкивания β, чья конкурентность находится между $(2\beta - 1)/(\beta - 1)$ (что составляет около 3.87 для $\beta = 1.53$) и $(\beta + 2)^2 + 2/(\beta - 1)$ (что составляет около 16.24 для $\beta = 1.53$). Для переменной ценности и FIFO буферов они предложили жадный алгоритм с β-выталкиванием с конкурентностью $6+4\beta+\beta^2+3/(\beta-1)$ (19.95 для $\beta = 1.67$) [130].

Равномерная ценность, переменные требования к обработке, одна очередь. В этой постановке задачи каждый пакет добавляет одну единицу

к целевой функции, но разные пакеты могут иметь различные требования к обработке, т.е. разные пакеты требуют разное число циклов обработки у процессора. Обозначим максимально возможное требующееся время обработки через k.

Алгоритмы без выталкивания. Для одной очереди и пакетов с различными требованиями к обработке алгоритмы без выталкивания не рассматривались подробно до результатов, которые мы рассмотрим ниже, в разделе 2.2.

Алгоритмы с выталкиванием. Кесласси с соавторами [128] показали что в случае одной очереди и здесь алгоритм PQ (Priority Queue), который упорядочивает пакеты относительно требующегося времени обработки (в порядке возрастания), является оптимальным; дальнейшие исследования сконцентрировались на случае FIFO, где пакеты должны быть переданы в порядке их прибытия. Важные результаты в этой области были получены в рамках диссертации; они показаны в разделе 2.2.

Цена копирования. Важное обобщение модели с переменными требованиями к обработке было предложено Кесласси с соавторами [128]. Они добавляют штраф α, называемый ценой копирования (copying cost), за добавление пакета в очередь; иначе говоря, целевая функция теперь выглядит как $T - \alpha A$, где T — число переданных пакетов, а A — число пакетов, принятых в очередь; таким образом, становится менее выгодно выталкивать пакеты из очереди. Для постановки задачи с ценой копирования авторы предложили использовать алгоритмы с β-выталкиванием, которые выталкивают пакет, только если требующееся ему время обработки по крайней мере в $\beta > 1$ раз меньше требующегося времени обработки для пакета, который будет вытолкнут в результате. Кесласси с соавторами [128] ввели алгоритм PQ_β (Priority Queue with β-Push-Out) и показали, что он не более чем $\frac{1}{1-\alpha \log_\beta k}\left(1 + \log_{\frac{\beta}{\beta-1}} \frac{k}{2} + 2\log_\beta k)(1-\alpha)\right)$-конкурентен. В этой главе мы увидим несколько результатов для вариаций модели с нетривиальной ценой копирования в разных архитектурах.

2.2 Алгоритмы управления буфером для пакетов с различными требованиями к обработке в одной очереди

Современные сетевые элементы должны выполнять задачи с различной сложностью, в частности такие, как сложные VPN-сервисы, DPI (deep packet inspection), задачи сетевого брандмауэра, распознавание вторжений и многое другое. Каждая из этих разнородных задач может потребовать различное время выполнения на процессоре сетевого элемента, что приводит к дополнительным сложностям для традиционных архитектур буферов, как в реализации, так и в результирующем качестве работы. Все эти задачи напрямую влияют на задержку обработки пакетов. В результате порядок обработки пакетов и то, как именно они обрабатываются, может оказывать существенное влияние не только на задержку и пропускную способность, но и может привести к переупорядочиванию пакетов при передаче.

В этом разделе мы рассматриваем постановку задачи с одной очередью для пакетов с различными требованиями к обработке. Обозначим пакет с требующимся временем обработки r через $\boxed{r}$, а последовательность из n пакетов с одинаковым требующимся временем обработки r — через $n \times \boxed{r}$. Если пакет *выброшен* до того, как он был передан (пока у него ещё остаётся положительное число требующихся циклов обработки), он теряется. Пакет может быть выброшен либо по прибытию, либо в результате принятия решения о выталкивании в то время, пока он ещё находится в буфере. Пакет добавляет одну единицу к целевой функции, только если он успешно передан. Задача состоит в том, чтобы разработать алгоритмы управления буфером, которые максимизируют общую пропускную способность, т.е. общее число пакетов, переданных из очереди.

В этой архитектуре одна очередь выполняет все три основные задачи буфера, а именно:

(1) *управление буфером*, т.е. выбор того, какие из входящих пакетов допускать и какие уже находящиеся в буфере пакеты выталкивать (если выталкивание поддерживается архитектурой);

(2) *обработка*, т.е. принятие решений о том, какие пакеты из лежащих в настоящее время в буфере будут обрабатываться;

(3) *передача*, т.е. выбор того, какие из обработанных пакетов должны быть переданы дальше, и собственно их передача.

Пакет называется *полностью обработанным*, если он уже был назначен на обработку столько раз (циклов), сколько составляет его требующееся время обработки. В этой модели мы также будем рассматривать многоядерную архитектуру с C процессорами (ядрами), в которой не более C пакетов могут быть выбраны для обработки в каждом конкретном цикле (два ядра не могут обрабатывать один и тот же пакет одновременно).

Мы предполагаем дискретное время, разбитое на слоты (циклы обработки), где каждый временной слот состоит из трёх фаз (см. рис. 2.1, который иллюстрирует это определение):

(1) *прибытие*: прибывают новые пакеты, и алгоритм управления буфером решает для каждого нового пакета, нужно ли его принять или отбросить, а если принять, то вытолкнуть ли ранее принятый пакет;

(2) *обработка*: алгоритм управления буфером выбирает C пакетов для обработки;

(3) *передача*: не более чем один полностью обработанный пакет выбирается для передачи и покидает очередь.

Для некоторого алгоритма ALG и временного слота t обозначим через IB_t^{ALG} множество пакетов, которое хранится в буфере алгоритма ALG в момент времени t после прибытия новых пакетов, но до обработки (т.е. имеется в виду состояние буфера, показанное во второй строке рис. 2.1). Число *циклов обработки* пакета является ключевым для рассматриваемых алгоритмов. Формально говоря, для каждого временного слота t и каждого пакета p, который в настоящее время находится в очереди, мы определяем число его *оставшихся циклов обработки* (residual processing cycles) $r_t(p)$ как число циклов обработки, которые ему ещё требуются, прежде чем этот пакет можно будет успешно передать. Определим *жадный* (greedy) алгоритм управления буфером как алгоритм, принимающий все пришедшие пакеты, если в очереди есть свободное место. Алгоритм называется *сохраняющим работу* (work-conserving), если он всегда обрабатывает пакеты, пока очередь не пуста. Будем говорить, что входящий пакет p *выталкивает* пакет q, который был ранее принят в буфер, если

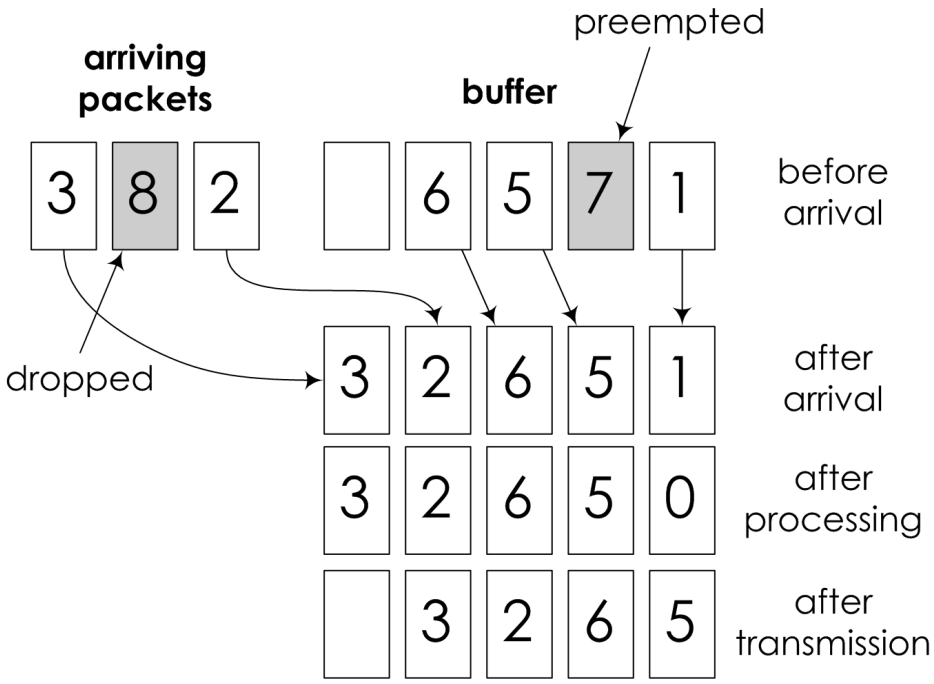


Рис. 2.1: Структура одного временного слота FIFO алгоритма с выталкиванием. Входящий пакет с $r = 8$ выбрасывается (не принимается), а пакет с $r = 7$ выталкивается из очереди, чтобы освободить место для прибывшего пакета с $r = 3$.

q выбрасывается из очереди, чтобы освободить место в буфере для p, и p добавляется в буфер в FIFO порядке. Алгоритм управления буфером называется алгоритмом *с выталкиванием*, если он допускает выталкивание пакетов из очереди. Рис. 2.1 показывает пример структуры временного слота в нашей модели для жадного алгоритма с выталкиванием.

Начнём с результатов о трёх алгоритмах. Алгоритм NPO (Non-Push-Out) — это простой жадный сохраняющий работу алгоритм без выталкивания, который принимает пакеты, пока в буфере есть свободное место. Алгоритм PO (Push-Out) тоже жадный и сохраняющий работу, но теперь, если прибывающий пакет p требует меньше циклов обработки, чем по крайней мере один пакет из буфера, PO выталкивает первый пакет с максимальным числом требуемых циклов обработки из своего буфера и принимает p согласно порядку FIFO. Для обработки PO всегда выбирает первый пакет в очереди. Мы также вводим новый алгоритм LPO (Lazy-Push-Out), который работает так же, как PO, за исключением двух важных различий:

(i) LPO не обрабатывает и не передаёт пакет из начала очереди, у которого остался только один цикл обработки, если в буфере есть ещё хотя бы один

пакет, у которого этих циклов больше, и переходит к обработке следующего пакета в очереди, пока буфер не содержит только пакеты с одним оставшимся циклом обработки, $r(p) = 1$;

(ii) когда все пакеты в буфере LPO (пусть там содержится m пакетов) имеют ровно один оставшийся цикл обработки, LPO заканчивает их обработку и передаёт их все в течение следующих m циклов обработки.

Выталкивание у алгоритма LPO работает так же, как у PO: если прибывающий пакет p требует меньше циклов обработки, чем первый находящийся в буфере пакет q с максимальным числом циклов обработки, p выталкивает q. Интуитивно можно сказать, что LPO является ослабленной версией PO, поскольку PO стремится опустошить свой буфер быстрее.

Мы провели подробный конкурентный анализ этих алгоритмов и получили следующие результаты (доказательства и более детальные описания даны в работах [28] и [147].

Теорема 2.2.1 (несравнимость PO и LPO в худшем случае)**.**

1. *Существует последовательность входов, на которой* PO *передаёт в* $\geqslant \frac{3}{2}$ *раз больше пакетов, чем* LPO*.*

2. *Существует последовательность входов, на которой* LPO *передаёт в* $\geqslant \frac{5}{4}$ *раз больше пакетов, чем* PO*.*

Теорема 2.2.2 (точные нижние и верхние оценки на конкурентность NPO)**.** *Для достаточно длинной последовательности входов конкурентность* NPO *составляет не менее* k *и не более* k*.*

Теорема 2.2.3 (нижние оценки для LPO и PO)**.**

1. *Конкурентность как* LPO*, так и* PO *составляет не менее* $2\left(1 - \frac{1}{B}\right)$ *для* $k \geqslant B$*.*

2. *Конкурентность для* $k < B$ *составляет не менее* $\frac{2k}{k+1}$ *для* PO *и не менее* $\frac{2k-1}{k}$ *для* LPO*.*

Теорема 2.2.4 (логарифмическая нижняя оценка для больших k)**.** *Конкурентность* PO *(*LPO*) составляет не менее* $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$*.*

Наш первый основной результат — верхняя оценка на конкурентность LPO.

Теорема 2.2.5 (верхняя оценка для LPO)**.** LPO *является не более чем* $(\max\{1, \ln k\} + 2 + o(1))$*-конкурентным.*

Мы также рассмотрели частный случай с двумя значениями, когда есть только два различных возможных значения для требующегося времени обработки, 1 и k, так что в систему могут поступать только два вида пакетов, $\boxed{1}$ и $\boxed{k}$. Этот случай часто встречается на практике; например, $\boxed{1}$ могут представлять собой «commodity» пакеты, которые обрабатываются сразу, в то время как $\boxed{k}$ соответствует тому или иному более сложному способу обработки пакетов, например DPI (deep packet inspection).

Теорема 2.2.6 (нижние оценки для LPO и PO в случае с двумя значениями)**.**

1. *Конкурентность* LPO *в случае с двумя значениями требующегося времени обработки составляет не менее* $2 - \frac{1}{k}$ *для* $B \geqslant k$*.*

2. *Конкурентность* PO *в случае с двумя значениями требующегося времени обработки составляет не менее* $\frac{2}{1+\frac{1}{k}}$ *для* $B \geqslant k$*.*

В этом случае становится возможным доказать константную верхнюю оценку для LPO.

Теорема 2.2.7 (верхняя оценка для LPO в случае с двумя значениями)**.** *В случае с двумя значениями требующегося времени обработки* LPO *является не более чем* $\left(2 + \frac{1}{B}\right)$*-конкурентным.*

Далее рассмотрим семейство алгоритмов *Semi-FIFO*, которые могут обрабатывать пакеты в любом порядке, но передают их всегда в порядке прибытия (чтобы сохранить порядок пакетов в потоке). Алгоритм обычно определяется набором свойств (ленивый, сохраняющий работу и т.д.) и его порядком обработки. В дальнейшем мы рассмотрим, в частности, такие порядки обработки:

(1) *приоритетная очередь* (PQ), где первым обрабатывается пакет с минимальной оставшейся требуемой обработкой;

(2) *обратная приоритетная очередь* (RevPQ), где первым обрабатывается пакет с максимальной оставшейся требуемой обработкой;

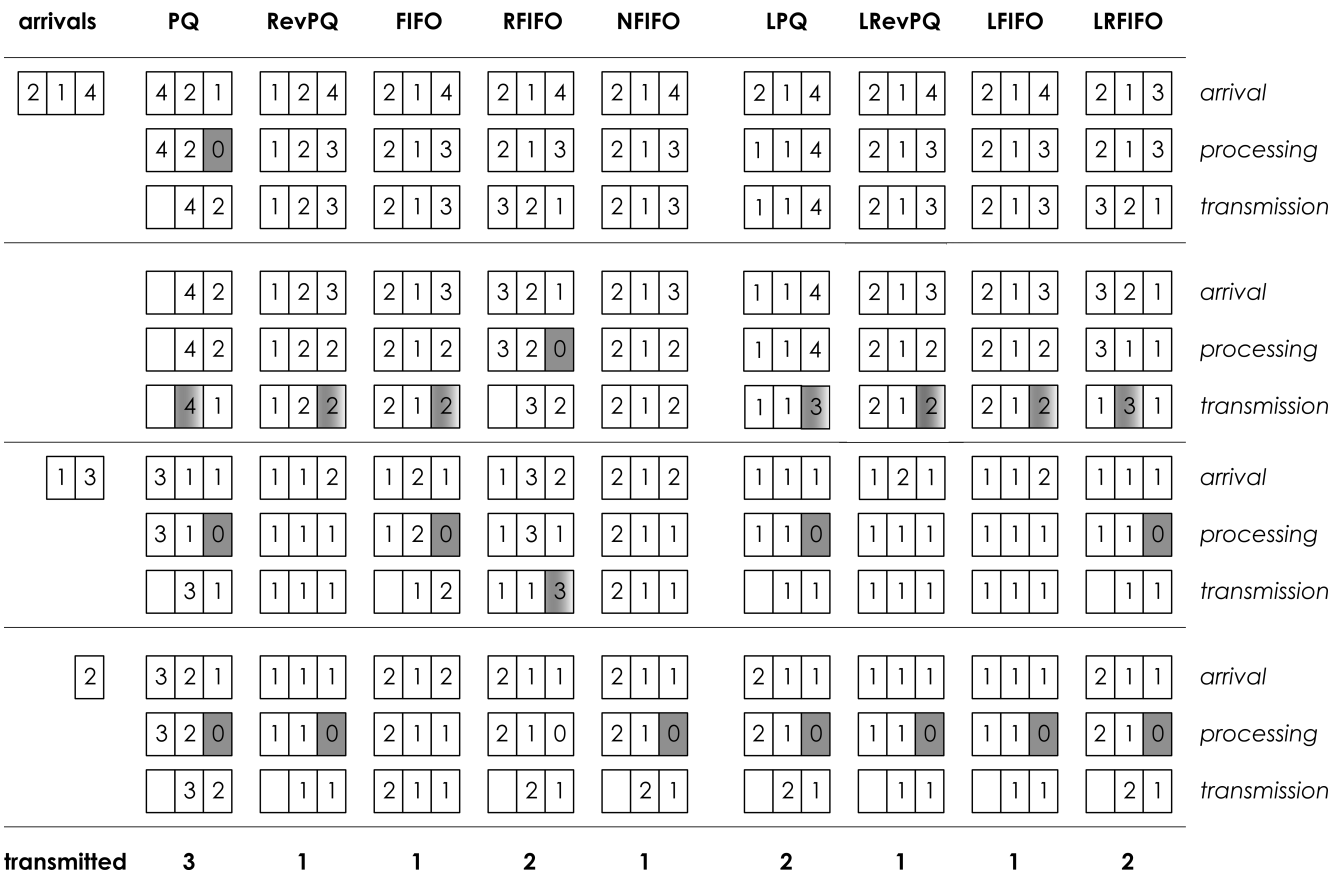


Рис. 2.2: Пример работы алгоритмов. Слева: обычные алгоритмы; справа: ленивые Semi-FIFO алгоритмы. Время течёт сверху вниз, первые пакеты в очередях (head-of-line) показаны с правого края каждого буфера, все алгоритмы начинают с пустого буфера и имеют $B = 3$. Заштрихованные прямоугольники показывают переданные пакеты; прямоугольники с градиентной заливкой показывают пакеты, которые будут вытолкнуты (ради пакета с $r(p) = 1$, прибывающего в третьем временном слоте, или пакета с $r(p) = 2$ на четвёртом). В каждом временном слоте показаны три состояния буфера: после прибытия, после обработки и после передачи.

(3) *FIFO*, обычный first-in-first-out порядок обработки;

(4) *FIFO с возвратами* (RFIFO), где не до конца обработанные пакеты возвращаются в конец очереди после каждого цикла обработки.

В случае, когда порядок передачи и обработки совпадают для любого входа, алгоритм называется по своему порядку обработки (например, алгоритм PQ обрабатывает и передаёт пакеты в порядке PQ). Рис. 2.2 показывает пример работы алгоритмов, которые мы рассматриваем ниже; пять алгоритмов в левой части рисунка соответствуют этим четырём порядкам обработки плюс алгоритм без выталкивания.

Далее мы вводим более общее понятие ленивых алгоритмов.

Определение 2.2.1. Алгоритм управления буфером LA называется *ленивым* (lazy), если он удовлетворяет следующим требованиям:

(1) LA является жадным, т.е. принимает пакеты, пока его буфер не заполнится;

(2) LA выталкивает первый пакет с максимальным числом циклов обработки в случае переполнения;

(3) LA не обрабатывает и не передаёт пакеты с одним оставшимся циклом обработки, если в его буфере есть по крайней мере один пакет с более чем одним оставшимся циклом обработки;

(4) когда все пакеты в буфере LA (пусть их m) имеют только один оставшийся цикл обработки, LA передаёт их в порядке прибытия в течение последующих m временных слотов, даже если в это время прибывают новые пакеты, которые также принимаются в буфер.

Определение 2.2.1 вводится для того, чтобы получить возможность организовать инфраструктуру подсчёта пакетов и оставшихся циклов обработки. В частности, оно позволяет определить понятие *итерации*, а затем считать переданные оптимальным алгоритмом пакеты и сравнивать их с содержимым буфера LA в рамках каждой итерации. Первая итерация начинается с прибытием первого пакета. Итерация заканчивается, когда все пакеты в буфере LA имеют только один оставшийся цикл обработки. Каждая последующая итерация начинается после того, как LA передал все пакеты с предыдущей итерации. Поскольку все пакеты в конце итерации имеют один цикл обработки, их можно передавать в любом порядке (в нашем случае FIFO). Например, на рис. 2.2 LPQ и LRFIFO заполняют свой буфер пакетами $\boxed{1}$ и начинают передавать их на третьем цикле обработки, а LRevPQ и LRFIFO — на четвёртом цикле.

Определение 2.2.1 покрывает широкий спектр разных алгоритмов, потому что оно не определяет конкретный порядок обработки пакетов в течение итерации. Порядок обработки имеет большое значение, поскольку он определяет, как много циклов обработки будут «потрачены зря» на пакеты, которые всё равно в результаты придётся вытолкнуть. Однако ленивые алгоритмы — это лишь подкласс Semi-FIFO алгоритмов.

Второй основной результат этой части — общая верхняя оценка на конкурентность всех ленивых алгоритмов управления буфером.

Теорема 2.2.8 (общая верхняя оценка для LA). *Конкурентность* LA *составляет не более*

$$\frac{1}{B}\log_{B/(B-1)} k + 3.$$

Заметим, что асимптотически, для $k, B \to \infty$, это логарифмическая оценка по k:

$$\frac{1}{B}\log_{B/(B-1)} k = \frac{\ln k}{B\ln\left(1+\frac{1}{B}+o(\frac{1}{B})\right)} = \ln k + o(\ln k).$$

Существуют порядки обработки, для которых это точная оценка.

Теорема 2.2.9 (нижние оценки для LRFIFO, LRevPQ, RFIFO). *Конкурентность* LRFIFO, LRevPQ *и* RFIFO *составляет не менее* $(1 + \frac{1}{B}\log_{B/(B-1)} k)$.

Возникает естественный вопрос: существует ли порядок обработки, у которого конкурентность была бы существенно лучше этой оценки? Чтобы показать, что порядок обработки действительно имеет большое значение для поведения алгоритма управления буфером, рассмотрим ленивый Semi-FIFO алгоритм с порядком обработки PQ в течение итерации (LPQ); для него нам удалось доказать константную верхнюю оценку.

Теорема 2.2.10 (точные нижняя и верхняя оценки для LPQ).

1. LPQ *является не более чем* 2*-конкурентным.*

2. LPQ *является не менее чем* $\left(2 - \frac{1}{B}\left\lceil\frac{B}{k}\right\rceil\right)$*-конкурентным.*

На самом деле мы можем использовать конструкцию из теоремы 2.2.10(2), чтобы получить общую нижнюю оценку для любого Semi-FIFO алгоритма.

Теорема 2.2.11. *Любой жадный Semi-FIFO алгоритм является не менее чем* $\left(1 + \frac{m-1}{B}\right)$*-конкурентным для* $m = \min\{k, B\}$.

Мы также показали ряд результатов о несравнимости между различными Semi-FIFO алгоритмами.

Теорема 2.2.12 (несравнимость между Semi-FIFO алгоритмами). *Любой ленивый алгоритм* LA (*в том числе* LRevPQ) *несравним как с* FIFO, *так и с* RFIFO *в худшем случае для любого* $k > 2$ *и* $B > 2$.

Для алгоритмов без выталкивания можно также доказать линейную нижнюю оценку.

Теорема 2.2.13 (общая нижняя оценки для алгоритмов без выталкивания)**.**

1. *Любой жадный Semi-FIFO алгоритм без выталкивания* NPO *является не менее чем* $\frac{k+1}{2}$*-конкурентным.*

2. *Любой жадный ленивый алгоритм без выталкивания* NLPO *является не менее чем* $(k-1)$*-конкурентным.*

В некоторых системах также требуется контролировать число вытолкнутых пакетов. Это делается по двум причинам: во-первых, каждый принятый пакет приводит к некоторой *цене копирования* α, а во-вторых, выталкивание пакета из очереди также не является бесплатной операцией. Добавление цены копирования α покрывает оба случая. Для этого Кесласси с соавторами [127] ввели понятие цены копирования: если для последовательности входов σ онлайн-алгоритм принимает A пакетов и передаёт T пакетов, то общая ценность (целевая функция) составляет $\max(0, T - \alpha A)$.

Далее мы представляем третий главный результат этого раздела: общую верхнюю оценку для любого ленивого Semi-FIFO алгоритма с β-выталкиванием. Эта оценка представляет собой ещё одно применение мощной инфраструктуры итераций, которую предоставляют введённые нами ленивые алгоритмы.

Теорема 2.2.14 (верхняя оценка для β-push-out LA)**.** *Для цены копирования* α*, удовлетворяющей* $0 < \alpha < \frac{1}{\log_\beta k}$*, конкурентность алгоритма* LA_β *составляет не более чем*

$$\left(3 + \frac{1}{B} \log_{\frac{\beta B}{\beta B - 1}} k\right) \frac{1-\alpha}{1 - \alpha \log_\beta k}.$$

В этом случае также можно доказать и общую нижнюю оценку.

Теорема 2.2.15 (нижняя оценка для LA с β-выталкиванием)**.** *Для цены копирования* α*, удовлетворяющей* $0 < \alpha < \frac{1}{\log_\beta k}$*, конкурентность любого алгоритма с* β*-выталкиванием составляет не менее*

$$\frac{1-\alpha}{1 - \alpha \log_\beta k}.$$

Мы также распространили результаты, полученные выше в случае с двумя возможными значениями для алгоритмов LPO и PO, на более общий случай ленивых алгоритмов, в том числе алгоритмов с β-выталкиванием.

Теорема 2.2.16 (верхняя оценка для LA в случае с двумя значениями)**.** *В случае с двумя возможными значениями требующегося числа циклов обработки* LA *является не более чем* $\left(2 + \frac{1}{B}\right)$*-конкурентным.*

Теорема 2.2.17 (верхняя оценка для β-push-out LA в случае с двумя значениями)**.** *Для цены копирования* $0 < \alpha < \frac{1}{2}$ *конкурентность алгоритма* LA_β *для* $\beta < k$ *в случае с двумя значениями (когда в систему поступают только пакеты видов* $\boxed{1}$ *и* $\boxed{k}$*) составляет не более*

$$\left(2 + \frac{1}{B}\right) \frac{1 - \alpha}{1 - 2\alpha}.$$

Теорема 2.2.18 (нижняя оценка для β-push-out LA в случае с двумя значениями)**.** *Для цены копирования* $0 < \alpha < \frac{1}{2}$ *конкурентность алгоритма* LA_β *для* $\beta < k$ *в случае с двумя значениями составляет не менее*

$$\frac{(2B - 2)(1 - \alpha)}{(B - 1)(1 - 2\alpha) + (1 - \alpha)}.$$

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы, а также позволили выделить конкретные ситуации, где лучше применять те или иные алгоритмы, что открывает возможности для практического внедрения. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [12, 28, 32].

2.3 Алгоритмы управления буфером для нескольких очередей, разделённых по времени обработки пакетов

В разделе 2.2 мы рассмотрели одну очередь, которая принимает пакеты с различными требованиями к обработке. В таких случаях для оптимизации пропускной способности требуется алгоритм, который может менять порядок об-

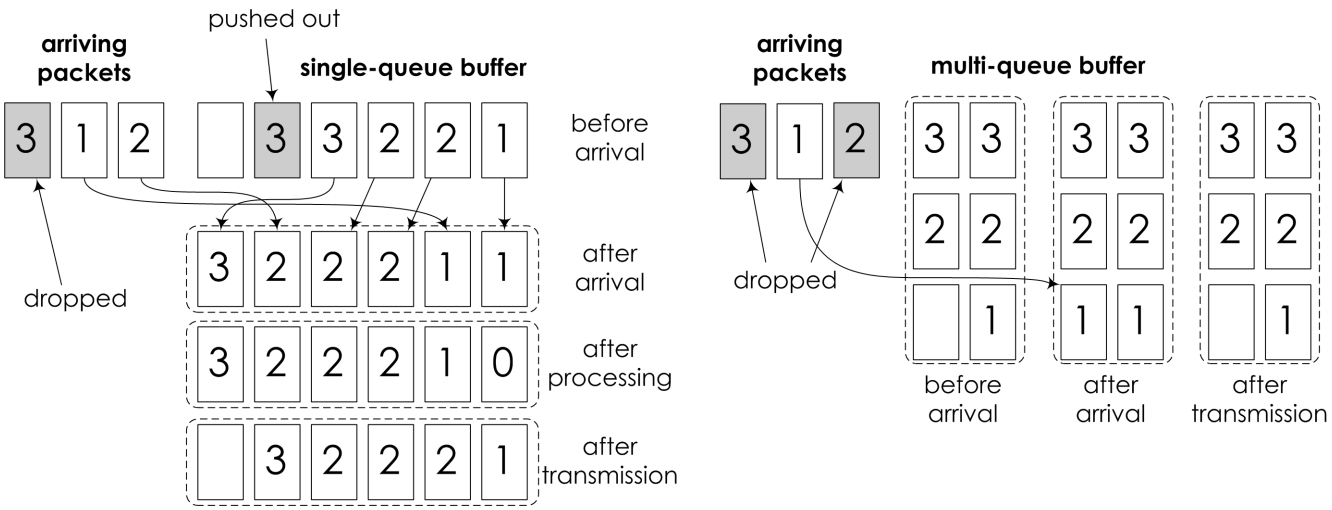


Рис. 2.3: Пример временного слота. Слева: одна приоритетная очередь с буфером размера $B = 6$; справа: коммутатор с тремя очередями (здесь $k = 3$) и буфер размера $B = 2$ у каждой. Буферы заключены в пунктирные линии. Заметим, что множества пакетов в буферах в конце каждого временного слота различаются.

работки пакетов и который реализует механизм выталкивания (т.е. может удалять из буфера уже принятые туда пакеты); в случае одной очереди жадный алгоритм с выталкиванием PQ, который в первую очередь обрабатывает пакет с наименьшим требующимся временем обработки, является доказуемо оптимальным относительно пропускной способности [127]. Чтобы изучить влияние свойства выталкивания на пропускную способность, в разделе 2.2 мы показали, что жадный алгоритм с выталкиванием, который реализует порядок обработки FIFO или FIFO с повторами, является не менее чем $\mathbf{\Omega}(\log k)$-конкурентным по сравнению с оптимальным всевидящим алгоритмом, где k — это максимально возможное требующееся время обработки одного пакета, и в то же время алгоритмы без выталкивания с одинаковым порядком обработки плохо работают в худшем случае (являются $\mathbf{\Omega}(k)$-конкурентными).

В этом разделе мы исследуем возможность построения упрощённой архитектуры, которая, с одной стороны, даёт гарантии на пропускную способность, сравнимые с архитектурами, основанными на одной очереди, реализующей PQ порядок, но, с другой стороны, не требует достаточно сложно реализуемых на практике механизмов выталкивания или нетривиального порядка обработки. В качестве такой альтернативы мы рассматриваем систему с несколькими очередями, где каждая очередь принимает пакеты с некоторым заданным требованием к обработке, и общий размер всех очередей в буфере тот же, что размер

буфера в архитектуре с одной очередью (см. рис. 2.3). Одно важное преимущество такой системы — это простота в реализации: поскольку каждая очередь содержит пакеты с одинаковым требущимся временем обработки, механизм выталкивания не нужен, и каждая очередь может просто обрабатывать пакеты в FIFO порядке.

Это разделение также даёт дополнительную возможность выделить разные ресурсы обработки на пакеты из разных очередей, чтобы, например, реализовать более сложные QoS-механизмы (quality of service); алгоритм управления буфером может применяться к каждой очереди независимо, в полностью распределённом виде. Более того, если очереди реализуются как отдельные буферы, общая пропускная способность памяти увеличивается пропорционально числу очередей. Очевидный недостаток такой архитектуры — фиксированный верхний предел на размер каждой очереди, поскольку буферы не делятся между пакетами с разными требованиями к времени обработки. Однако во многих приложениях это не недостаток, а скорее даже дополнительное преимущество: желательно ограничить число пакетов из каждого класса обслуживания. В этом разделе мы исследуем, как такое упрощение системы влияет на пропускную способность.

Сначала давайте рассмотрим характеристику занятости буфера; это приводит к онлайн-алгоритму Longest-Queue-First (LQF), который обрабатывает первый (HOL) пакет очереди, которая в настоящий момент является самой длинной, чтобы освободить очереди для потенциально возможного прихода пакетов в будущем.

Теорема 2.3.1 (нижняя оценка для LQF)**.** *Алгоритм* LQF *является не менее чем* $\frac{m}{2}$*-конкурентным для* $m = \min\{k, B\}$.

Далее рассмотрим другой алгоритм, который тоже использует занятость буфера, но обрабатывает пакеты из самой короткой очереди, а не самой длинной. Этот алгоритм мы назовём Shortest-Queue-First (SQF).

Теорема 2.3.2 (нижняя оценка для SQF)**.** *Алгоритм* SQF *является не менее чем* k*-конкурентным.*

Теоремы 2.3.1 и 2.3.2 показывают, что в худшем случае между алгоритмами, обрабатывающими пакеты в зависимости от заполненности очередей, и алгоритма, который просто обрабатывает любой доступный пакет, разница невелика:

у любого сохраняющего работу алгоритма есть тривиальная верхняя оценка конкурентности k, потому что пока в буфере есть пакеты, он всегда может обработать не менее одного пакета за k временных слотов.

Далее рассмотрим другую характеристику и введём другой алгоритм, который, в отличие от LQF и SQF, не обращает внимания на текущую заполненность буфера, а обрабатывает пакет из непустой очереди с наименьшим требующимся временем обработки. Назовём этот алгоритм Minimal-Queue-First (MQF). Для нижней оценки заметим, что MQF может потратить существенные усилия на обработку пакетов из очереди, которая никогда не будет переполнена.

Теорема 2.3.3 (нижняя оценка для MQF)**.** *Алгоритм* MQF *является не менее чем* $\left(1 + \frac{k-1}{2k}\right)$*-конкурентным.*

Теорема 2.3.3 показывает, что требущееся время обработки потенциально имеет большее влияние на пропускную способность, чем заполненность буфера. Противоположный MQF алгоритм MaxQF, который обрабатывает сначала очередь с пакетами максимального требующегося времени обработки, очевидно является худшим алгоритмом из всех, что мы рассматриваем в этом разделе.

Теорема 2.3.4 (нижняя оценка для MaxQF)**.** *Алгоритм* MaxQF *является не менее чем* k*-конкурентным.*

Основной результат этого раздела — следующая константная верхняя оценка.

Теорема 2.3.5 (верхняя оценка для MQF)**.** *Алгоритм* MQF *является не более чем* 2*-конкурентным.*

Мы также рассматриваем частный случай, в котором есть только два вида пакетов, $\boxed{a}$ и $\boxed{b}$, то есть система состоит из двух очередей. В этом частном случае нам удалось показать точно соответствующие друг другу нижние и верхние оценки.

Теорема 2.3.6 (нижняя оценка для MQF в случае с двумя значениями времени обработки)**.** *Конкурентность* MQF *с двумя очередями составляет не менее чем*

$$\left(1 + \frac{1 + \left\lfloor \frac{aB-1}{b} \right\rfloor}{B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil}\right).$$

Теорема 2.3.7 (верхняя оценка для MQF в случае с двумя значениями времени обработки)**.** *Конкурентность* MQF *с двумя очередями составляет не более чем*

$$\left(1+\frac{1+\left\lfloor\frac{aB-1}{b}\right\rfloor}{B+\left\lceil\frac{1}{a}\left(b\left\lfloor\frac{aB-1}{b}\right\rfloor+1\right)\right\rceil}\right).$$

Далее мы рассматриваем и оцениваем влияние реализации *равного доступа* к сетевым ресурсам (fairness) на пропускную способность. Мы рассмотрим равный доступ как на уровне пакетов, так и на уровне отдельных циклов обработки. Поскольку LQF не менее чем $\frac{m}{2}$-конкурентен, где $m = \min\{k, B\}$, мы не ожидаем получить более высокую пропускную способность от алгоритмов, реализующих равный доступ к ресурсам.

Сначала рассмотрим равный доступ на уровне отдельных пакетов. Алгоритм *Packet-Round-Robin* (PRR) последовательно и циклично обрабатывает активные очереди, переключаясь к следующей активной очереди, когда первый (HOL) пакет в текущей очереди обработан до конца и передан.

Теорема 2.3.8 (нижняя оценка для PRR)**.** *Алгоритм* PRR *является не менее чем* $\frac{3k(k+2)}{4k+16}$*-конкурентным.*

Хотя этот результат весьма предсказуем, интересно, что PRR может работать лучше, чем SQF. Далее, рассмотрим алгоритм *Cycle-Round-Robin* (CRR), который циклически перебирает активные очереди, переходя к следующей активной очереди после каждого цикла обработки.

Теорема 2.3.9 (нижняя оценка для CRR)**.** *Алгоритм* CRR *является не менее чем* $\frac{k}{H(k)}$*-конкурентным для* $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$.

Кроме того, мы доказали несколько результатов о несравнимости в худшем случае между рассматриваемыми алгоритмами.

Теорема 2.3.10. *Алгоритм* MQF *несравним с* SQF*,* LQF*,* CRR *и* PRR*.*

Теорема 2.3.11.

1. Алгоритм PRR *несравним с* SQF*,* LQF *и* CRR*.*

2. Алгоритм CRR *несравним с* SQF *и* LQF*.*

3. Алгоритмы SQF *и* LQF *несравнимы.*

В дополнение к этим результатам, мы также исследовали возможности взаимной эмуляции между архитектурами с одной и несколькими очередями. Это важный вопрос для того, насколько сложной должна быть абстракция очереди и её реализация в программном (software-defined) управлении буфером.

Рассмотрим сначала архитектуру буфера с одной очередью размера $\boldsymbol{B}$, пакетами одного размера с разными ценностями, и целевую функцию максимизации общей переданной ценности (weighted throughput); эту постановку задачи мы рассматривали в разделе 2.2. В этом случае, как мы уже видели в разделе 2.2, не существует детерминированных онлайн-алгоритмов, которые были бы оптимальны для одной очереди с этой целевой функцией и FIFO обработкой. Более того, можно показать, что PQ является оптимальным в этой архитектуре, и можно построить примеры, где он работает строго лучше, чем любой FIFO алгоритм.

Теорема 2.3.12 (строгая оптимальность PQ)**.** *В архитектуре с одной очередью PQ на любом входе работает по крайней мере так же хорошо, как любой детерминированный онлайн-алгоритм с FIFO порядком обработки, и существуют входы, на которых он работает строго лучше.*

Далее рассмотрим архитектуру с несколькими очередями с одинаковым размером буфера, где каждая очередь имеет FIFO порядок обработки и обрабатывает пакеты с одинаковой ценностью (при этом несколько очередей могут содержать пакеты одной и той же ценности). Далее мы рассматриваем именно эту постановку задачи, которая упрощает обработку и переносит сложные решения на уровень выбора очереди для обработки. Теперь алгоритм, который выталкивает наименее ценный и обрабатывает наиболее ценный пакет, передаёт ту же общую ценность, что и PQ.

Теорема 2.3.13 (архитектура с несколькими очередями, эмулирующая одну очередь)**.** *Существует детерминированный онлайн-алгоритм для архитектуры с несколькими очередями с FIFO порядком обработки в каждой очереди, который работает так же, как алгоритм PQ в одной очереди с одинаковыми входами, и, как следствие, работает лучше, чем любой детерминированный онлайн-алгоритм для одной очереди с FIFO обработкой.*

Оказывается, что для одного и того же размера буфера B эти архитектуры одинаково выразительны.

Теорема 2.3.14 (архитектура с одной очередью, эмулирующая несколько очередей)**.** *Для любого детерминированного (соответственно, стохастического) алгоритма ALG в архитектуре с несколькими очередями существует такой алгоритм с одной очередью, который для любой последовательности входов передаёт в точности то же множество пакетов (соответственно, множество пакетов, полученное из того же распределения), что и ALG.*

Теорема 2.3.14 показывает, что теоретически архитектура с одной очередью столь же выразительна, как архитектура с несколькими очередями для любой целевой функции и любой комбинации характеристик пакетов, причём как для детерминированных, так и для стохастических алгоритмов. Однако дополнительные ограничения и параметры, например вычислительная сложность операций с пакетами, могут сделать одну архитектуру буфера более предпочтительной.

Теорема 2.3.15 (разница в сложности между одной очередью и несколькими очередями)**.** *В худшем случае архитектура с m очередями имеет вычислительную сложность операций по крайней мере в $O(m/\log B)$ раз меньше, чем архитектура с одной очередью, эмулирующая тот же алгоритм.*

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [5, 22, 36].

2.4 Алгоритмы управления буфером для пакетов с различными требованиями к обработке и несколькими исходящими портами

До сих пор мы рассматривали только алгоритмы управления буфером с одной исходящей очередью, т.е. с одним возможным пунктом назначения для всех входящих пакетов; даже архитектура с несколькими очередями, рассмотренная

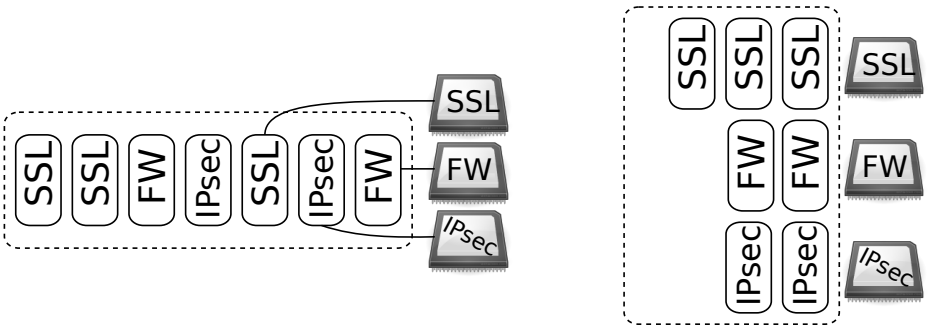


Рис. 2.4: Две различные модели управления буфером. Слева: каждое ядро может обработать любой тип трафика, IPsec, SSL, или firewall (FW); справа: каждое ядро обрабатывает конкретный тип трафика из своей собственной очереди, но все очереди делят один и тот же буфер.

в разделе 2.2, по сути представляет собой лишь вспомогательную конструкцию, которая нужна для упрощения алгоритмов управления буфером, а сами пакеты всё ещё имеют один и тот же пункт назначения.

Действительно, лучший способ справиться с неравномерностью трафика (burstiness) — это использовать архитектуру с одной очередью, где весь буфер делится между всеми типами трафика, и каждое ядро может обрабатывать любой тип трафика (см. левую часть рис. 2.4). В этой архитектуре жадный онлайн-алгоритм, который реализует приоритетную очередь (PQ), в которой пакеты упорядочены по неубыванию их требущегося времени обработки и могут быть вытеснены по прибытию новых пакетов, имеет оптимальную пропускную способность [126].

Однако у этого простого подхода есть ряд недостатков. Во-первых, он требует PQ порядка обработки: мы уже видели, что FIFO порядок обработки, который проще в реализации, приводит к ухудшению конкурентности до $\Omega(\log k)$ относительно оптимального всевидящего алгоритма (раздел 2.3), где k — это максимально возможное требуемое время обработки одного пакета. Во-вторых, это может вызвать недостаточную обработку пакетов, у которых требования к обработке более высокие, что приводит к неявной приоритезации разных типов трафика, то есть получается, что требования к обработке служат обратным приоритетом, что нежелательно. В-третьих, обработка нескольких типов трафика на каждом отдельном ядре приводит к более сложным программам, загружаемым в эти ядра, так как теперь они должны реализовывать сразу несколько сервисов.

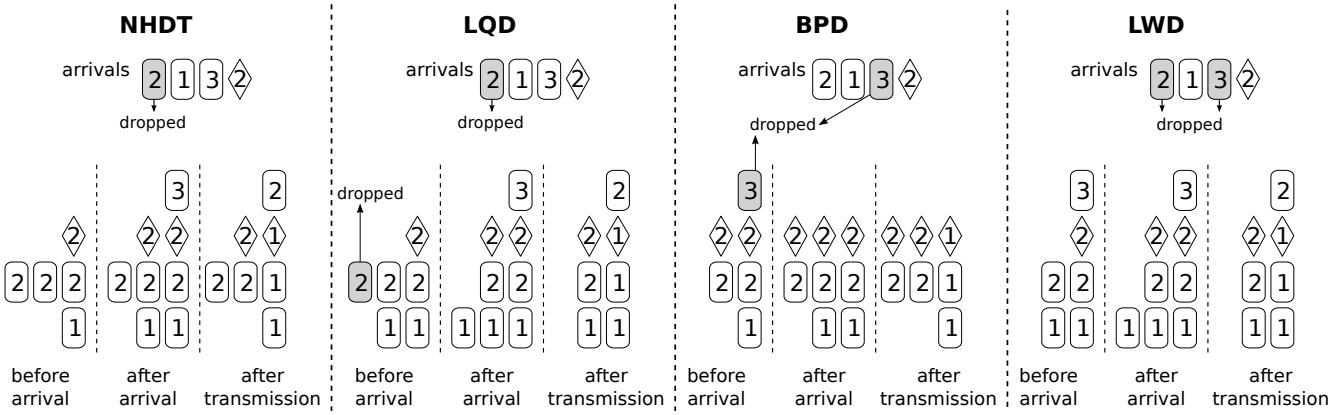


Рис. 2.5: Пример временного слота алгоритмов Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT), Longest-Queue-Drop (LQD), Biggest-Packet-Drop (BPD) и Longest-Work-Drop (LWD) с максимальным требующимся временем обработки $k = 3$, 4 исходящими портами (с двумя исходящими портами с одинаковым требущимся временем обработки 2, которые обозначены прямоугольными и ромбовидными пакетами соответственно) и общим буфером размера $B = 8$.

Таким образом, мы рассматриваем коммутатор с общей памятью, где один буфер разделяется между всеми типами трафика, но в отличие от предыдущей модели с одной очередью каждое ядро обрабатывает только один тип трафика. Архитектура буфера с одной очередью является частным случаем коммутатора с общей памятью (см. правую часть рис. 2.4). Поскольку все прибывающие пакеты с одной и той же меткой исходящего порта имеют один и тот же тип трафика, нам не требуется реализовывать сложные порядки обработки (вполне достаточно обычного FIFO). Более того, ни один конкретный вид трафика не остаётся необработанным. Мы рассматриваем пакеты единичного размера, помеченные номером исходящего порта и требущимся временем обработки (т.е. пакеты с различными требованиями к обработке) и максимизируем число переданных пакетов; качество работы различных алгоритмов управления буфером анализируются посредством конкурентного анализа.

Рассмотрим $l \times n$ коммутатор с общей памятью с буфером размера B; l и n обозначают число входных и исходящих портов соответственно. Предположим, что $B \geqslant n$. Каждый исходящий порт i управляет одной выходной очередью Q_i, $1 \leqslant i \leqslant n$; число пакетов в Q_i обозначается как $|Q_i|$. Каждая очередь Q_i реализует порядок обработки FIFO. Каждый пакет $p(d, w)$, прибывающий на входной порт, содержит метку исходящего порта номер d и требущееся время обработки w в циклах обработки ($1 \leqslant d \leqslant n$ и $1 \leqslant w \leqslant k$), где k обознача-

ет общую верхнюю границу на требущееся время обработки одного пакета. В дальнейшем будем обозначать через $\boxed{w}$ пакет с требующимся временем обработки w; через $h \times \boxed{w}$, набор из h пакетов с требующимся временем обработки w у каждого.

Будем называть *последовательным случаем* ситуацию, когда в каждой очереди Q_i находятся пакеты с требуемым временем обработки i, то есть каждое возможное время обработки встречается ровно в одной очереди; это частный случай нашей модели, но уже в нём зачастую можно доказать большие нижние оценки на конкурентность.

Время дискретно; каждый временной слот разделяется на две фазы (см. рис. 2.5). Во время *фазы прибытия* на каждый входной порт поступают новые пакеты, и алгоритм управления буфером решает, какие из них принимаются в буфер; мы предполагаем, что во время фазы прибытия входные порты обрабатываются в некотором фиксированном порядке, от 1 до l, и нет никаких ограничений на число прибывающих пакетов (burst size), т.е. более чем l пакетов могут поступить на вход за один временной слот. Прибытия являются *состязательными* (adversarial, т.е. они могут выбираться наихудшим для текущего алгоритма образом), и мы не предполагаем никакого конкретного распределения трафика, за исключением одного ограничения: все прибывающие пакеты с исходящим портом i имеют одно и то же требующееся время обработки w_i, $1 \leqslant w_i \leqslant k$. Заметим, что две разные исходящие очереди Q_i и Q_j, $1 \leqslant i, j \leqslant n$, могут принимать пакеты с одним и тем же требущимся временем обработки, $w_i = w_j$. Без потери общности предположим, что очереди отсортированы в порядке требущегося времени обработки их пакетов: если $i < j$, то $w_i \leqslant w_j$. Принятый пакет может быть позднее вытеснен из буфера, и на его место может быть принят другой пакет; в таком случае будем говорить, что пакет p *вытолкнут* (is pushed out) другим пакетом q, и позволяющий такие операции алгоритм называется алгоритмом *с выталкиванием*.

Во время *фазы передачи* требущееся время обработки первого (head-of-line) пакета в FIFO порядке в каждой непустой очереди уменьшается на единицу, и все пакеты, у которых оставшееся время обработки равно нулю, передаются дальше.

Начнём с алгоритмов без выталкивания. В случае одной очереди, как мы видели в разделе 2.3, жадные алгоритмы без выталкивания работают плохо (они

k-конкурентны) на пакетах с различными требованиями к обработке, независимо от порядка обработки. В этой модели мы начинаем с двух простых случаев жадных алгоритмов управления буфером без выталкивания со статическими порогами. Первый алгоритм устанавливает для каждой очереди статические пороги, обратно пропорциональные требующемуся времени обработки, а второй просто использует одинаковые пороги для каждой очереди. Интересно, что второй алгоритм имеет конкурентность лучше первого. Обозначим $Z = \sum_{i=1}^{n} \frac{1}{r_i}$ (сумма величин, обратных значениям требующегося времени обработки для каждого порта); в дальнейшем мы опускаем $\lfloor\cdot\rfloor$ и $\lceil\cdot\rceil$ для упрощения записи, предполагая, что B делит все встречающиеся в определениях и оценках числа.

Non-Push-Out-Harmonic-Static-Threshold (NHST): по прибытии пакета p с исходящим портом i, если $|Q_i| < \frac{B}{r_i Z}$, то принять p в очередь Q_i; иначе выбросить p.

Теорема 2.4.1 (верхняя оценка для NHST)**.** *Алгоритм NHST является* $(kZ + o(kZ))$*-конкурентным.*

Non-Push-Out-Equal-Static-Threshold (NEST): по прибытии пакета p с исходящим портом i, если $|Q_i| < B/n$ пакетов, то принять p в очередь Q_i, иначе выбросить p.

Теорема 2.4.2 (верхняя оценка для NEST)**.** *Алгоритм NEST является* $(n + o(n))$*-конкурентным.*

Далее рассмотрим алгоритм Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT), который был ранее введён в работе [137]. В алгоритме NHDT пороги для очередей динамические и зависят от числа пакетов в очередях. Идея состоит в том, что для каждого $1 \leqslant m \leqslant k$ m очередей с наибольшим числом пакетов должны содержать в общей сложности не более чем $\frac{B}{H_k}\left(1 + \frac{1}{2} + \ldots + \frac{1}{m}\right)$ пакетов, где $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ — это k-е гармоническое число.

Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT): по прибытии пакета p с исходящим портом i, обозначим через j_1, j_2, …, $j_m = i$ очереди, для которых $|Q_{j_s}| \geqslant |Q_i|$; если

$$\sum_{s=1}^{m} |Q_{j_s}| < \frac{B}{H_k}\left(1 + \frac{1}{2} + \ldots + \frac{1}{m}\right),$$

то принять p в очередь Q_i, иначе выбросить p.

Алгоритм NHDT проиллюстрирован примером на рис. 2.5; это первый алгоритм слева.

Для пакетов с одинаковым временем обработки известно, что NHDT является $O(\log n)$-конкурентным для n исходящих портов [137] (в нашей модели k исходящих портов). Оказывается, что для разнородных требований к обработке NHDT всё ещё работает не слишком хорошо: хотя нам не удалось доказать линейную нижнюю оценку, подобную теореме 2.4.1 для NHST, доказанная ниже нижняя оценка всё ещё растёт слишком быстро.

Теорема 2.4.3 (нижняя оценка для NHDT)**.** *Если B асимптотически растёт быстрее, чем k, алгоритм NHDT является не менее чем $\left(\frac{1}{2}\sqrt{k \ln k} - o(\sqrt{k \ln k})\right)$-конкурентным в последовательном случае.*

Теперь перейдём к алгоритмам с выталкиванием. Начнём с хорошо известного алгоритма Longest-Queue-Drop, который не обращает внимания на требующееся время обработки и рассматривает только размеры очередей.

Longest-Queue-Drop (LQD): по прибытии пакета p с исходящим портом i найти $j^* = \arg\max_j\{|Q_j| + [i = j]\}$, где $[i = j] = 1$ при $i = j$ и 0 в противном случае (т.е. Q_{j^*} — это самая длинная очередь после того, как мы виртуально добавим пакет p к очереди Q_i; если таких несколько, выберем очередь с наибольшим общим требующимся временем обработки); затем сделать следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и $i \neq j^*$, вытолкнуть последний пакет из Q_{j^*} и принять p в очередь Q_i;

(3) иначе выбросить p.

Алгоритм LQD проиллюстрирован на рис. 2.5; это второй алгоритм слева. В случае с одинаковым временем обработки LQD является не менее чем $\sqrt{2}$- и не более чем 2-конкурентным [45]. Для разнородного требующегося времени обработки ситуация значительно хуже.

Теорема 2.4.4 (нижняя оценка для LQD)**.** *Для достаточно большого B алгоритм LQD является не менее чем $(\sqrt{k} - o(\sqrt{k}))$-конкурентным в последовательном случае.*

Далее мы вводим алгоритм, который пытается минимизировать общее требующееся время обработки в случае переполнения, выталкивая пакеты с максимальными требованиями к обработке.

Biggest-Packet-Drop (BPD): пусть Q_j — это непустая очередь с наибольшим требующимся временем обработки; по прибытии пакета p с исходящим портом i сделать следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и $i \leqslant j$, вытолкнуть последний пакет из Q_j и принять p в очередь Q_i;

(3) если буфер полон и $i > j$, выбросить p.

Рис. 2.5 показывает пример одного временного слота алгоритма BPD; это второй алгоритм справа.

Теорема 2.4.5 (нижняя оценка для BPD)**.** *Для* $B \geqslant \frac{k(k+1)}{2}$ *BPD является не менее чем* $(\ln k + \gamma)$*-конкурентным в последовательном случае, где* $\gamma = 0.5772\ldots$ *— константа Эйлера.*

В качестве улучшения для LQD мы предлагаем алгоритм, который вытесняет пакеты из очереди с наибольшим требущимся временем обработки; обозначим общее время обработки очереди Q_i (общее оставшееся число циклов обработки всех пакетов в Q_i) через W_i.

Longest-Work-Drop (LWD): по прибытии пакета p с исходящим портом i, пусть $j^* = \arg\max_j\{W_j + [i = j]r_i\}$ (т.е. W_{j^*} максимально после того, как мы виртуально добавим пакет p к очереди Q_i; выберем самую длинную из таких очередей, если их несколько); затем сделать следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и $i \neq j^*$, вытолкнуть последний пакет из Q_{j^*} и принять p в очередь Q_i;

(3) иначе выбросить p.

Алгоритм LWD проиллюстрирован справа на рис. 2.5.

Теорема 2.4.6 (нижняя оценка для LWD). *Если $k \geqslant 6$, то LWD является не менее чем $\left(\frac{4}{3} - \frac{6}{B}\right)$-конкурентным в последовательном случае.*

Поскольку LWD эквивалентен LQD в случае, когда все пакеты имеют одно и то же требуещееся время обработки, LWD является не менее чем $\sqrt{2}$-конкурентным в этой постановке задачи [45]. Заметим, что наша нижняя оценка для LWD в последовательном случае меньше, чем нижняя оценка $\sqrt{2}$ для LQD в модели с одинаковыми требования к обработке. LWD оптимизирует «локальное» состояние буфера, выбрасывая пакеты из очередей с максимальной задержкой. С другой стороны, OPT может оставить эти пакеты и тем самым оставить эти очереди активными, если OPT знает заранее, что в очереди с высокой задержкой больше не будут приходить новые пакеты, в то время как соответствующие порты у LWD будут простаивать; то же соображение объясняет субоптимальность LQD в модели с одинаковым требующимся временем обработки. Но в последовательном случае обобщённой модели для алгоритма OPT гораздо сложнее много выиграть в этом случае, поскольку LWD вытеснит меньше пакетов из очередей с наивысшей задержкой (пропорционально требующемуся времени обработки).

Основной результат этого раздела — константная верхняя оценка на конкурентность LWD. Заметим, что хотя эта оценка совпадает с верхней оценкой для LQD в модели с одинаковым временем обработки [45], доказательства теорем и построение соответствий между пакетами LQD и OPT в этих доказательствах на самом деле существенно различаются.

Теорема 2.4.7 (верхняя оценка для LWD). *Алгоритм LWD является не более чем 2-конкурентным.*

В качестве второй постановки задачи в этом разделе рассмотрим модель, которая допускает пакеты с различными требованиями к обработке в одну и ту же очередь. Это обобщение оказывает существенное влияние на эффективность рассматриваемых алгоритмов и имеет большое практическое значение. Описанная выше модель аллоцирует отдельную очередь для каждого значения требущегося времени обработки и исходящего порта, но этот случай накладывает серьёзные ограничения на максимальное число поддерживаемых очередей, что ограничивает его применимость при растущих k и n.

Мы снова рассматриваем $1 \times n$ коммутатор с общей памятью с одним входящим и n исходящими портами и буфером размера B, т.е. общая длина всех

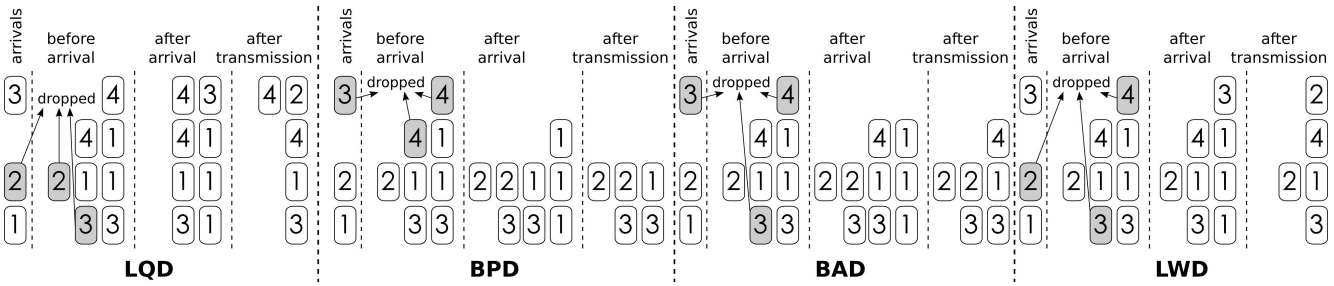


Рис. 2.6: Пример временного слота алгоритмов Longest Queue Drop (LQD), Biggest Packet Drop (BPD), Biggest Average Drop (BAD) и Largest Work Drop (LWD) с максимальным требующимся временем обработки $k = 4$, $n = 4$ исходящих портов и общим буфером размера $B = 8$. Очереди для каждого исходящего порта показаны по горизонтали. Заштрихованные пакеты выбрасываются при прибытии.

очередей ограничена B. Предположим, что $B \geqslant n$. Число входящих портов не важно для этой модели, все входящие пакеты обрабатываются в одно и то же время, одной пачкой (burst). Каждый исходящий порт, с другой стороны, управляет отдельной выходной очередью, обозначенной Q_i для порта i, $1 \leqslant i \leqslant n$; обозначим число пакетов в Q_i через $|Q_i|$. Каждый пакет, прибывающий на один из входящих портов, помечен номером исходящего порта d и требущимся временем w в циклах обработки ($1 \leqslant d \leqslant n$ и $1 \leqslant w \leqslant k$), где k обозначает общую верхнюю оценку на требущееся время обработки одного пакета. Каждая очередь Q_i реализует

(1) либо приоритетный порядок обработки (priority queueing, PQ), где пакеты обрабатываются в порядке неубывания требующегося времени обработки,

(2) либо порядок обработки по очереди (first-in-first-out, FIFO), где пакеты обрабатываются в порядке прибытия.

В дальнейшем будем обозначать через $\boxed{w \mid i}$ пакет с требованиями времени обработки w и исходящим портом i; будем обозначать пачку из h пакетов $\boxed{w \mid i}$, прибывающих в одно и то же время, через $h \times \boxed{w \mid i}$. Также обозначим через $r_t(p)$ оставшееся требующееся время обработки пакета p в момент времени t.

Рассмотрим в этой модели несколько различных алгоритмов, проиллюстрированных на рис. 2.6. Ниже мы определяем эти алгоритмы и показываем результаты нашего теоретического анализа.

Longest-Queue-Drop (LQD): по прибытии пакета p с исходящим портом i, обозначим через $j^* = \arg\max_j\{|Q_j| + [i = j]\}$, где $[i = j] = 1$, если $i = j$, и 0 в противном случае (т.е. Q_{j^*} — это самая длинная очередь после того, как мы виртуально добавим пакет p к очереди Q_i; выберем очередь с наибольшим общим требующееся время обработки, если их несколько); затем сделать следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон, принять p в очередь Q_i и вытолкнуть последний пакет из Q_{j^*};

(3) иначе выбросить p.

Заметим, что в случае (2), если $i = j^*$ и p является самым большим пакетом в Q_{j^*}, пакет p сам будет вытеснен (выброшен).

Теорема 2.4.8 (нижняя оценка для LQD)**.** *Для $k \geqslant n(n-1)$ и $B \geqslant k+n$ алгоритм LQD является не менее чем $(n/2 - o(n))$-конкурентным.*

Biggest-Packet-Drop (BPD): по прибытии пакета p с требующимся временем обработки w и исходящим портом i, обозначим через Q_j непустую очередь, которая содержит пакет $p_{\max}$ с наибольшим требущимся временем обработки $w_{\max}$; затем алгоритм делает следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и $w < w_{\max}$, вытеснить $p_{\max}$ из Q_j и принять p в очередь Q_i;

(3) если буфер полон и $w > w_{\max}$, выбросить p.

Biggest-Average-Drop (BAD): по прибытии пакета p с требующимся временем обработки w и исходящим портом i, обозначим через Q_j непустую очередь с наибольшим средним требущимся временем обработки $\bar{w}_{\max}$; затем алгоритм делает следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и $w < \bar{w}_{\max}$, вытеснить пакет с максимальной требующейся работой из Q_j и принять p в очередь Q_i;

(3) если буфер полон и $w > \bar{w}_{\max}$, выбросить p.

Теорема 2.4.9 (нижняя оценка для BPD и BAD)**.** *Алгоритмы BPD и BAD оба являются не менее чем $(n+1)/2$-конкурентными.*

Largest-Work-Drop (LWD): по прибытии пакета p с исходящим портом i и требующимся временем обработки w, обозначим $j^* = \arg\max_j\{W_j + [i = j]w\}$, где $[i = j] = 1$ при $i = j$ и 0 в противном случае, и W_j — это общее требующееся время обработки всех пакетов в очереди Q_j (т.е. Q_{j^*} — это очередь с наибольшим общим требующимся временем обработки после того, как мы виртуально добавим p к очереди Q_i; если в буфере несколько очередей с одинаковой наибольшей общей работой, выберем очередь с наибольшим требованием обработки у одного пакета). Затем алгоритм делает следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и w меньше, чем требующееся время обработки по крайней мере одного пакета в Q_{j^*}, вытеснить наибольший пакет из Q_{j^*} и принять p в очередь Q_i;

(3) иначе выбросить p.

Теорема 2.4.10 (нижняя оценка для LWD)**.** *Алгоритм LWD с FIFO порядком обработки является не менее чем* $\left(1 + \log_{B/n} k)(1 - n/B)\right)$*-конкурентным.*

Основной теоретический результат этого раздела — следующая верхняя оценка на конкурентность алгоритма LWD.

Теорема 2.4.11 (верхняя оценка для LWD)**.** *Для* $n \times n$ *коммутатора с общей памятью и буфером размера* B *алгоритм LWD является не более чем* $\left(1 + \frac{B}{T+B}\right)$*-конкурентным, где* T *— это наименьшее число пакетов, переданных между любыми двумя последовательными итерациями.*

В третьей части этого раздела мы рассматриваем ещё одно обобщение, которое было выделено как открытая проблема в работе [109]. В этой модели каждый пакет единичного размера с требующимся временем обработки 1 имеет два параметра: метку исходящего порта и внутреннюю *ценность*; целью алгоритмов управления буфером является максимизация общей переданной ценности. В остальном модель та же, только каждый пакет теперь требует ровно один цикл обработки, зато пакеты имеют различную ценность. Заметим, что для единичных ценностей это сводится к модели из работы [45], так что показанная там общая нижняя оценка 4/3 на любой онлайн-алгоритм верна и в этом случае.

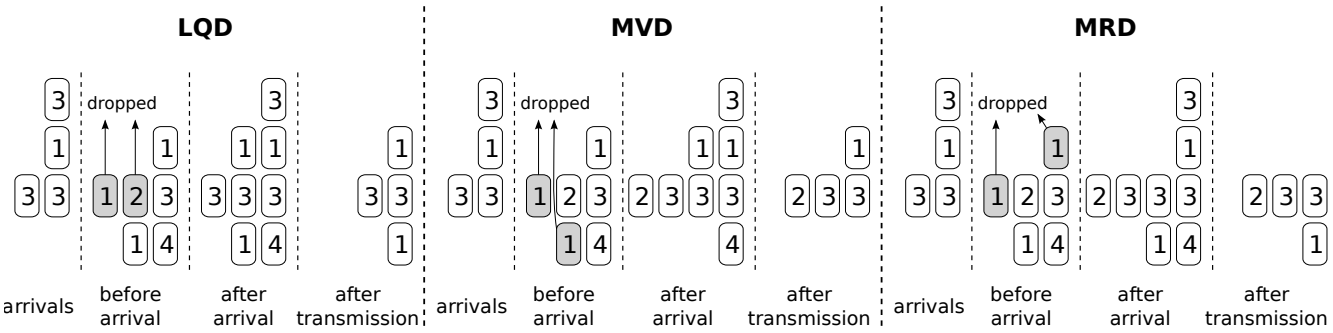


Рис. 2.7: Пример временного слота алгоритмов Longest-Queue-Drop (LQD), Minimal-Value-Drop (MVD) и Maximal-Ratio-Drop (MRD) с максимальной ценность $k = 4$, 4 исходящими портами и общим буфером размера $B = 8$.

Даже если порядок обработки зафиксирован, неясно, какой именно трафик нужно принимать в буфер; здесь может быть нужно достичь баланса между трафиком, который открывает новые активные порты, и трафиком, который увеличивает общую принятую ценность. Мы не будем рассматривать алгоритмы без выталкивания, поскольку очевидно, что жадный алгоритм без выталкивания, который принимает пакеты, пока есть свободное место, является не менее чем k-конкурентным (нужно заполнить буфер пакетами вида $\boxed{1}$, а затем присылать $\boxed{k}$). Определим несколько алгоритмов с выталкиванием, которые жадно оптимизируют число активных портов, общую принятую ценность, или комбинацию этих величин. Алгоритмы проиллюстрированы на рис. 2.7.

Сначала снова рассмотрим алгоритм Longest-Queue-Drop (LQD); в этой модели LQD выбрасывает последний (с наименьшей ценностью) пакет из самой длинной очереди при переполнении буфера, балансируя тем самым размеры очередей. Он проиллюстрирован слева на рис. 2.7. Однако в модели с ценностями алгоритму LQD снова не удаётся достичь константной конкурентности.

Теорема 2.4.12 (нижняя оценка для LQD)**.** *Конкурентность алгоритма LQD составляет не менее чем* $\left(\sqrt[3]{k} - o\left(\sqrt[3]{k}\right)\right)$.

Следующий алгоритм жадно максимизирует общую принятую ценность, выталкивая пакеты с наименьшей ценностью.

Minimal-Value-Drop (MVD): по прибытии пакета p с ценностью m и i-м исходящим портом, обозначим через Q_j непустую очередь, которая содержит пакет с наименьшей ценностью (если таких очередей несколько, выберем самую длинную из них); затем алгоритм делает следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон и наименьшая ценность находящихся в буфере пакетов меньше m, вытолкнуть последний пакет из Q_j и принять p в очередь Q_i;

(3) если буфер полон и наименьшая ценность находящихся в буфере пакетов не более m, выбросить пакет p.

MVD — это средний алгоритм из показанных на рис. 2.7. Алгоритм MVD приоритезирует пакеты с максимальной ценностью. К сожалению, MVD тоже имеет линейную нижнюю оценку на конкурентность в худшем случае.

Теорема 2.4.13 (нижняя оценка для MVD)**.** MVD *является не менее чем* $\frac{m-1}{2}$*-конкурентным для* $m = \min\{k, B\}$.

Как потенциально лучший алгоритм мы предлагаем комбинацию этих двух характеристик; эта идея похожа на алгоритм LWD, для которого в предыдущей модели нам удалось доказать верхнюю оценку 2 на конкурентность (Теорема 2.4.7). Естественной идеей было бы объединить эти две эвристики, приоритезируя пакеты с высокой ценностью, только если они приходят из более коротких очередей. Одна возможная реализация этой идеи даётся в алгоритме Maximal-Ratio-Drop.

Maximal-Ratio-Drop (MRD): по прибытии пакета p с ценностью m и исходящим портом i, обозначим через Q_j непустую очередь с максимальным значением $\frac{|Q_j|}{a_j}$, где a_j — средняя ценность пакетов в Q_j (если таких очередей несколько, выберем в качестве Q_j очередь, содержащую пакет с наименьшей ценностью); затем алгоритм делает следующее:

(1) если буфер не полон, принять p в очередь Q_i;

(2) если буфер полон, и очередь Q_j содержит пакет с ценностью меньше m, вытолкнуть последний пакет из Q_j и принять p в очередь Q_i;

(3) если буфер полон, и минимальная ценность пакетов в буфере больше m, выбросить p.

Пример временного слота алгоритма MRD показан справа на рис. 2.7. Заметим, что MRD работает так же, как LQD, в случае, если все пакеты имеют единичную ценность, и нижняя оценка $\sqrt{2}$, показанная в [45], верна и здесь. Мы также можем показать константную нижнюю оценку для MRD в частном случае, когда ценность каждого пакета равна метке его исходящего порта.

Теорема 2.4.14 (нижняя оценка для MRD с равными ценностями в каждой очереди)**.** MRD *является не менее чем* $\frac{4}{3}$*-конкурентным в случае, когда ценность каждого пакета равна метке его исходящего порта.*

К сожалению, в общем случае, по крайней мере для больших n, MRD тоже не является конкурентным.

Теорема 2.4.15 (нижняя оценка для MRD для больших n)**.** *Если* $n \geqslant B - V^2 + 1$*, алгоритм MRD является не менее чем* V*-конкурентным.*

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [9, 11, 27, 31].

2.5 Алгоритмы управления буфером для пакетов с различными требованиями к обработке и переменным размером

В разделах 2.2, 2.3 и 2.4 мы рассмотрели управление буфером для пакетов с различными требованиями к обработке с различными архитектурами очередей. В этом разделе мы рассмотрим общую модель для задачи, в которой сетевой элемент должен управлять принятием и обработкой пакетов в одной очереди ограниченного размера, а входящий трафик состоит из пакетов, каждый из которых характеризуется *размером* (size) (например, в байтах) и *требующимся временем обработки* (в циклах процессора). Пакет успешно *передаётся*, когда модуль планирования назначил его для обработки на столько циклов, сколько составляет его требующееся время обработки. Если пакет выброшен из буфера — либо не принят по прибытии, либо вытолкнут после принятия и, возможно, частичной обработки — он считается безвозвратно утерянным. Мы рассматриваем задачу максимизации *пропускной способности* (throughput) очереди, которая измеряется общим размером (числом байтов) успешно переданных пакетов.

Эта простая модель подчёркивает естественно возникающее противоречие между двумя (потенциально) конфликтующими целями: с одной стороны, хо-

чется сконцентрироваться на пакетах, которым требуется мало циклов обработки, поскольку они могут быть быстро переданы. С другой стороны, хочется также давать более высокий приоритет большим пакетам, потому что они больше вносят в целевую функцию после передачи. В этом разделе мы рассматриваем это противоречие и доказываем гарантии качества работы различных алгоритмов в этой постановке.

Рассмотрим буфер ограниченного размера в B байтов, в который прибывают последовательности пакетов. Каждый прибывающий пакет p помечен размером $\ell(p) \in \{1, \ldots, L\}$ (в байтах) и некоторым числом требующихся циклов обработки $r(p) \in \{1, \ldots, k\}$. Оба этих числа считаются известными для каждого прибывающего пакета. Мы рассматриваем управление буфером и планирование на основе приоритетов, как с выталкиванием (push-out), когда принятые пакеты можно удалить из буфера до окончания их обработки (и тогда они ничего не добавляют к целевой функции), так и без выталкивания (non-push-out), когда управление буфером ограничивается управлением принятием пакетов.

В частности, мы рассмотрим следующие порядки для приоритетных очередей:

(1) по кратчайшему оставшемуся времени обработки (Shortest Remaining Processing Time, SRPT), что часто встречается в задачах календарного планирования;

(2) по длине пакета (Longest-Packet, LP);

(3) по эффективности пакета (Most-Effective-Packet, MEP), где пакеты приоритезируются по отношению оставшегося времени обработки к их размеру.

В этом разделе мы представляем алгоритмы управления буфером для этих схем с доказуемыми результатами о конкурентности, а далее показываем нижние оценки на качество работы алгоритмов для таких приоритетных очередей. Мы покажем, что конкурентность, полученная нашими алгоритмами, зависит от двух фундаментальных параметров задачи, а именно

(1) максимального размера пакета и

(2) максимального числа циклов обработки, требующихся одному пакету.

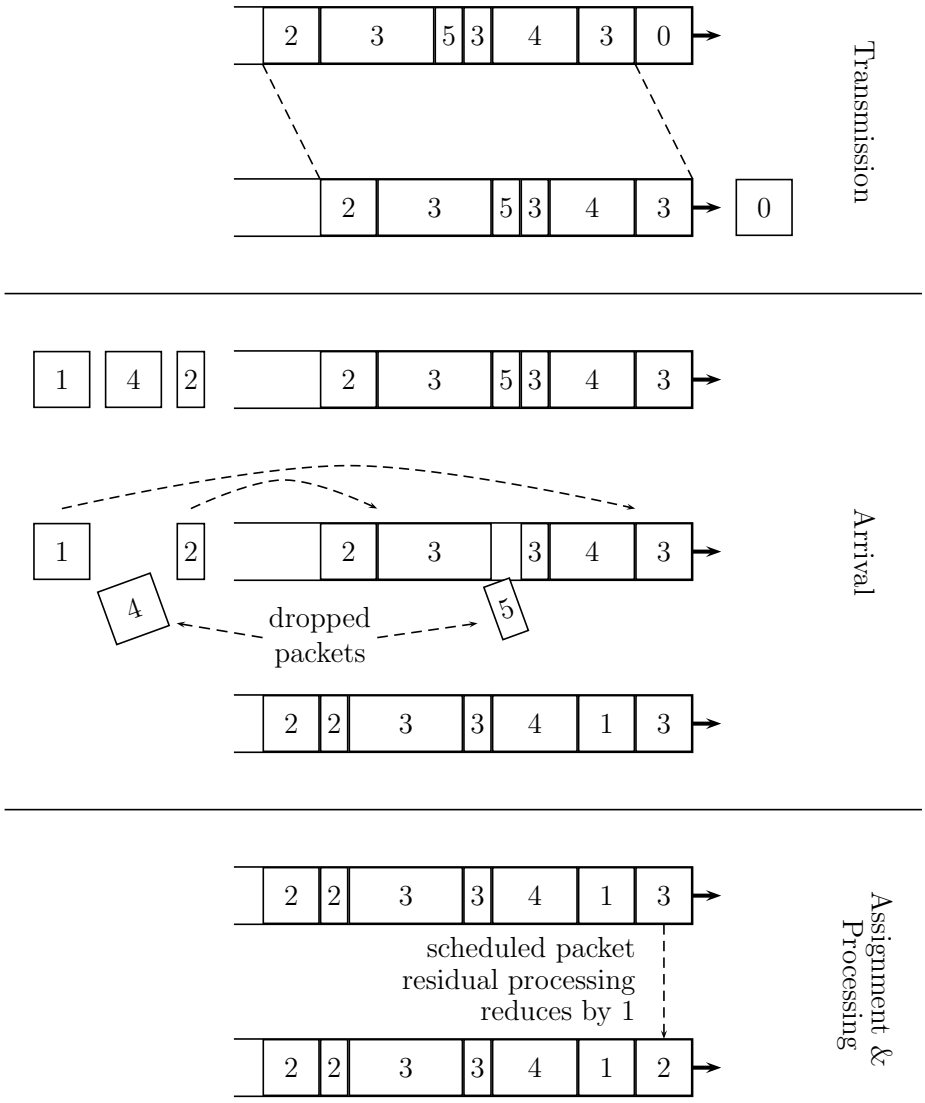


Рис. 2.8: Общая схема модели. Верхняя часть рисунка показывает фазу передачи, средняя — фазу прибытия, когда пакеты могут быть выброшены (не приняты), а нижняя часть показывает фазу назначения и обработки. Длина пакета показывает его размер, число на пакете — требующееся число циклов обработки.

Заметим, что ни один из представленных ниже алгоритмов не требует априорного знания максимального числа циклов обработки (хотя некоторые из наших алгоритмов требуют априорного знания максимально разрешённого размера пакета). Рис. 2.8 графически иллюстрирует нашу общую модель (вне зависимости от конкретной схемы приоритетов данной системы).

В доказательствах основных результатов используются свойства упорядоченных (мульти)множеств, которые также были доказаны нами. Мы рассматриваем мультимножества вещественных чисел, предполагая, что каждое мультимножество упорядочено по неубыванию. Будем называть такие мультимно-

жества *упорядоченными множествами.* Для каждого $1 \leqslant i \leqslant |A|$, элемент $a_i \in A$ или $A[i]$ обозначает i-й элемент в множестве A по порядку. Для двух упорядоченных множеств A, B будем говорить, что $A \geqslant B$, если $a_i \geqslant b_i$ для каждого i, для которого существуют a_i и b_i.

Теорема 2.5.1. *Для любых двух упорядоченных множеств A, B, для которых $A \geqslant B$, и любых двух вещественных чисел a, b, для которых $a \geqslant b$, если $b \leqslant b_{|B|}$ или $|A| \leqslant |B|$, то упорядоченные множества $A' = A \cup \{a\}$, $B' = B \cup \{b\}$ удовлетворяют $A' \geqslant B'$.*

Теорема 2.5.2. *Для любых двух упорядоченных множеств A, B, для которых $A \geqslant B$, и любого вещественного числа b, если $b \leqslant b_{|B|}$ или $|A| \leqslant |B|$, то упорядоченное множество $B' = B \cup \{b\}$ удовлетворяет $A \geqslant B'$.*

Начнём с оценки качества работы жадного алгоритма NPO, где нам удалось показать точные (соответствующие друг другу) нижнюю и верхнюю оценки.

Теорема 2.5.3 (нижняя оценка для NPO)**.** *Алгоритм* NPO *имеет конкурентность не менее kL для SRPT приоритетов.*

Теорема 2.5.4 (верхняя оценка для NPO)**.** *Алгоритм* NPO *является не более чем $kL\frac{B}{B-L}$-конкурентным для SRPT приоритетов.*

Для алгоритма PO ситуация с пакетами разного размера хуже, чем было в предыдущих разделах.

Теорема 2.5.5 (нижняя оценка для PO)**.** *Алгоритм* PO *имеет конкурентность не менее L для SRPT приоритетов для $B \geqslant 2L$.*

Однако нам также удалось показать первую известную верхнюю оценку для PO в этой постановке задачи, первый основной результат этого раздела.

Теорема 2.5.6 (верхняя оценка для PO в случае $B > 2L$)**.** *Если $B > 2L$, то алгоритм* PO *является не более чем $\frac{4L-2}{L_a}$-конкурентным для SRPT приоритетов, где L_a — средний размер пакетов, переданных алгоритмом* PO*.*

Для достаточно больших буферов соответствие, построенное в теореме 2.5.6, можно обобщить.

Теорема 2.5.7 (верхняя оценка для PO). *Алгоритм* PO *является не более чем* $\frac{(2L-1)(N+1)}{NL_a}$*-конкурентным для SRPT приоритетов, где* L_a *— средний размер всех пакетов, переданных алгоритмом* PO*, и* $N = \lceil \frac{B-2L+1}{2L-1} \rceil$.

Следующая теорема показывает, что можно улучшить верхнюю оценку на конкурентность PO для больших B при помощи этого обобщения.

Теорема 2.5.8 (верхняя оценка для PO для больших B). *Алгоритм* PO *является не более чем* $(k+3)$*-конкурентным для LP приоритетов и достаточно больших размеров буфера.*

Далее мы рассматриваем качество работы приоритетной очереди (priority queue, PQ), где пакеты упорядочиваются в невозрастающем порядке числа циклов обработки, разделённых на размер пакета. Назовём этот приоритет *Most Effective Packet* (MEP), или приоритет *удельной эффективности*. Напомним, что нашей целью является максимизировать число пересланных байтов. Мы рассматриваем только случай с выталкиванием, поскольку случай без выталкивания имеет результаты, аналогичные результатам для NPO. Следующая теорема является вторым основным результатом этого раздела.

Теорема 2.5.9 (верхняя оценка для PO). *Алгоритм* PO *является* $\Omega(\log k)$*-конкурентным.*

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [13, 38, 39].

2.6 Алгоритмы управления буфером для пакетов с несколькими характеристиками

Как мы уже обсуждали в предыдущих разделах, внедрение новых характеристик (например требующегося времени обработки или ценности пакетов) в принятие решений об управлении пакетами, а также реализация дополнительных целевых функций, помимо равного доступа к сети, приводят к новым задачам в дизайне и реализации традиционных сетевых элементов. В этом разделе мы

снова рассматриваем коммутатор с одной очередью, где буфер размера B делится между всеми типами трафика. Разница в том, что в этом разделе мы изучаем влияние *двух* характеристик, ценности и требующегося времени обработки, на взвешенную пропускную способность.

А именно, мы предполагаем, что для каждого прибывающего пакета p известны:

(1) число требующихся циклов обработки (требущееся время обработки, вес) $w(p) \in \{1, \dots, W\}$;

(2) ценность $v(p) \in \{1, \dots, V\}$.

Хотя значения W и V будут играть важную роль в нашем анализе, рассматриваемые алгоритмы не должны знать W или V заранее. Заметим, что для $W = 1$ модель вырождается в одну очередь одинаковых пакетов с различной ценностью; эта модель ранее рассматривалась, например, в работах [52, 212]. А для $V = 1$ она превращается в одну очередь пакетов равной ценности с различным требующимся временем обработки, как в разделе 2.2. Обозначим пакет с требующимся временем обработки w и ценностью v через $\boxed{w \mid v}$, а пачку из n пакетов с одними и теми же параметрами w и v — через $n \times \boxed{w \mid v}$.

Конечная цель всего этого направления исследований состоит в том, чтобы выбрать правильный алгоритм управления буфером для каждой постановки задачи, т.е. для каждой возможной конфигурации коммутатора и каждой целевой функции. Например, предыдущие работы (и предыдущие разделы этой главы) детально изучали взаимодействие между алгоритмами с выталкиванием и без, т.е. влияние возможности выбрасывать пакеты, которые уже были приняты в буфер ранее. Алгоритмы без выталкивания проще в реализации, но они часто оказываются неконкурентными по отношению к взвешенной пропускной способности, т.е. нижние оценки на их конкурентность оказываются линейными по таким параметрам задачи, как размер буфера B, максимальная возможная ценность V или максимальное требующееся время обработки W. Заметим, что даже это ещё не совсем вся история: хотя алгоритмы без выталкивания обычно явно хуже относительно (взвешенной) пропускной способности, они всё ещё могут оказаться впереди по другим целевым функциям, например с точки зрения минимизации общего потребления энергии (выталкивание может оказаться дорогостоящей процедурой).

Однако в этом разделе мы рассматриваем именно взвешенную пропускную способность и вводим постановку задачи, в которой нижние оценки в худшем случае, т.е. сложные примеры для конкретных алгоритмов, обычно относительно легко найти: у нас сразу две характеристики, которые можно менять для каждого пакета, — ценность и требующееся время обработки — вместо одной, как в большинстве предыдущих работ. Поэтому мы концентрируемся на изучении самого лучшего естественного класса алгоритмов, доступных в этой постановке с одной очередью; предыдущие исследования показывают, что *приоритетные очереди с выталкиванием* являются наилучшими инструментами, которые часто приводят к хорошим результатам. В частности, Кесласси с соавторами показали, что одна приоритетная очередь с выталкиванием является оптимальной для пакетов с различным требующимся временем обработки и единичной ценностью [128].

Отметим, однако, что хотя в предыдущих работах одна приоритетная очередь обычно представляла собой самый лучший алгоритм, иногда буквально оптимальный, и цель обычно состояла в том, чтобы достичь сравнимой пропускной способности с дополнительными ограничениями, например при FIFO порядке передачи пакетов или несколькими отделёнными друг от друга очередями, в модели с двумя различными характеристиками не вполне ясно даже, по какому критерию приоритетная очередь должна сортировать пакеты: если у одного пакета ценность меньше, но и требующееся время обработки тоже меньше, чем у другого, какой пакет нужно предпочесть? Чтобы рассмотреть разные приоритетные очереди, введём следующее определение.

Определение 2.6.1. Пусть f — некоторая функция пакетов, $f(w, v) \in \mathbb{R}$, смысл которой в том, что лучшим пакетам присваиваются более высокие значения f. Тогда алгоритм управления буфером PQ_f определяется следующим образом:

(1) PQ_f является жадным, т.е. он принимает входящие пакеты, пока в буфере есть место;

(2) PQ_f сохраняет работу, т.е. он обрабатывает пакеты, пока его буфер непуст;

(3) PQ_f упорядочивает и обрабатывает пакеты в очереди в порядке невозрастания значения f;

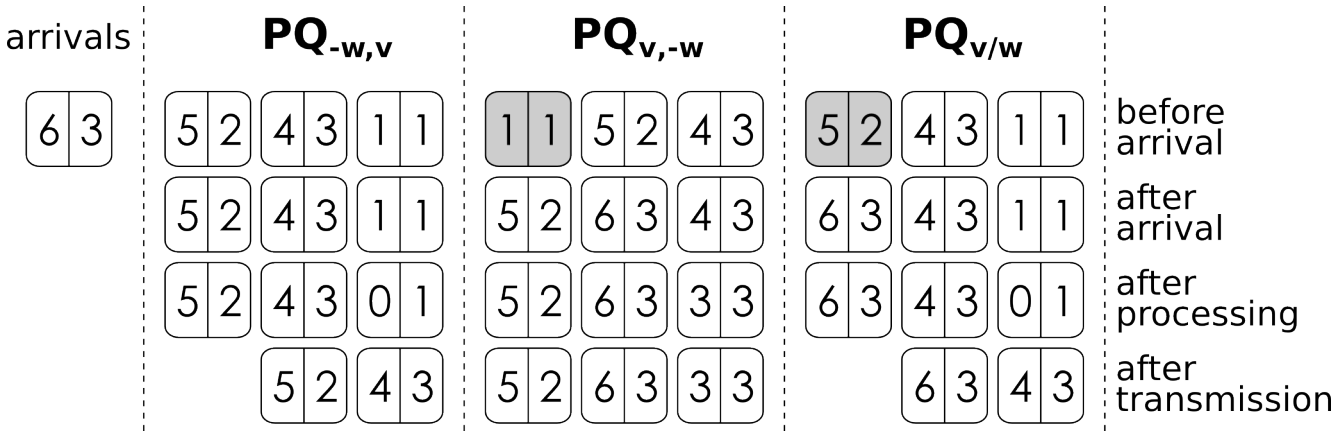


Рис. 2.9: Пример временного слота алгоритмов $\mathrm{PQ}_{-w,v}$, $\mathrm{PQ}_{v,-w}$ и $\mathrm{PQ}_{v/w}$.

(4) PQ_f выталкивает пакет p и добавляет новый пакет p' к очереди в момент времени t, если буфер полон, p — это в настоящее время наихудший пакет в буфере относительно значений f, и p' лучше p:

$$f(p) = \min_{q \in \mathrm{IB}^{\mathrm{PQ}_f}} f(q), \quad \text{и} \quad f(p') > f(p).$$

Здесь $\mathrm{IB}^{\mathrm{PQ}_f}$ — это $\mathrm{IB}^{\mathrm{PQ}_f}(t)$ для текущего временного слота t.

Другими словами, PQ_f упорядочивает и обрабатывает пакеты согласно значениям функции f.

В частности, рассмотрим три конкретных приоритетных очереди (здесь w обозначает *текущее* оставшееся время обработки и v обозначает ценность пакета):

(1) $\mathrm{PQ}_{-w,v} = \mathrm{PQ}_{-w+v/(V+1)}$ упорядочивает пакеты по возрастанию их требующегося времени обработки, выбирая в случае равенства пакеты более высокой ценности;

(2) $\mathrm{PQ}_{v,-w} = \mathrm{PQ}_{v-w/(W+1)}$ упорядочивает пакеты по убыванию их ценности, выбирая в случае равенства пакеты с меньшим требующимся временем обработки;

(3) $\mathrm{PQ}_{v/w}$ упорядочивает пакеты по убыванию их отношения ценности и работы, т.е. эта очередь приоритезирует пакеты, которые дают наибольшую ценность на один временной слот обработки. Заметим, что здесь есть две возможности разрешать «ничьи», $\mathrm{PQ}_{v/w,-w} = \mathrm{PQ}_{v/w-w/(W^2+1)}$ и

$\mathrm{PQ}_{v/w,v} = \mathrm{PQ}_{v/w+v/(WV+1)}$, но в данном случае конкретный способ их разрешать не важен ни для одного из дальнейших утверждений, поэтому мы объединим их одним и тем же обозначением.

Рис. 2.9 показывает пример одного временного слота этих приоритетных очередей.

Первый набор результатов касается общих нижних оценок в этой постановке задачи. Они доказываются *состязательным* (adversarial) образом: мы строим последовательность входов, в которой последующие входы зависят от решений, которые ранее принял онлайн-алгоритм, так что в конце концов мы строим «плохой» вход для каждого возможного выбора. Мы доказываем сильные оценки для нескольких частных случаев и более слабую общую нижнюю оценку для произвольной приоритетной очереди.

Теорема 2.6.1 (общая нижняя оценка для $B = 1$)**.** *Для $B = 1$ и $V > \sqrt{W}$ любой онлайн-алгоритм ALG является не менее чем $\sqrt{W}$-конкурентным. Более того, в случае с двумя ценностями, или если $V \leqslant \sqrt{W}$, любой онлайн-алгоритм ALG является не менее чем $\min\{V, W/V\}$-конкурентным для $B = 1$.*

Теорема 2.6.2 (общая нижняя оценка для FIFO порядка)**.** *Для произвольного B и $V > \sqrt{W}$ любой онлайн-алгоритм ALG, который сохраняет FIFO порядок обработки и передачи пакетов, является не менее чем $\left(\frac{\sqrt{W}}{B} + 1 - \frac{1}{B}\right)$-конкурентным. В случае с двумя возможными значениями требующегося числа циклов обработки, или если $V \leqslant \sqrt{W}$, любой онлайн-алгоритм ALG с FIFO порядком обработки и передачи пакетов является не менее чем $\frac{V+B-1}{W+B-1}$-конкурентным.*

Теорема 2.6.3 (произвольная приоритетная очередь)**.** *Для $V > \sqrt{W}$ и любой функции приоритета f, для которой $f(w, v) \in \mathbb{R}$, алгоритм PQ_f является не менее чем $\sqrt{W}$-конкурентным. В случае с двумя ценностями, или если $V \leqslant \sqrt{W}$, алгоритм PQ_f является не менее чем $\min\{V, W/V\}$-конкурентным.*

Основной результат в этой постановке, доказанный в работах [8, 25], — это верхняя оценка на конкурентность $\mathrm{PQ}_{v,-w}$ для случая с двумя ценностями, где требующееся время обработки может принимать произвольное значение от 1 до W, но есть только два возможных значения ценности, 1 и V. Среди других результатов также есть ряд нижних оценок на конкурентность различных

приоритетных очередей. Эти результаты не выносятся на защиту, но мы также доказали обобщение нескольких нижних оценок на случай алгоритмов с β-выталкиванием. В этом случае приоритетные очереди определяются следующим образом.

Определение 2.6.2. Пусть f — некоторая функция пакетов, $f(w, v) \in \mathbb{R}$, смысл которой в том, что лучшим пакетам присваиваются более высокие значения f. Тогда алгоритм управления буфером PQ_f^β определяется как алгоритм PQ_f со следующей модификацией: PQ_f^β может вытеснить пакет p и добавить новый пакет p' в очередь во время t, если p является худшим пакетом в буфере в это время, и p' лучше p по крайней мере в β раз:

$$f(p) = \min_{q \in \mathrm{IB}^{\mathrm{PQ}_f}} f(q), \quad \text{и} \quad f(p') > \beta f(p).$$

Теорема 2.6.4. *Рассмотрим буфер размера B с максимальным требующимся временем обработки W и максимальной ценностью пакета V. Тогда:*

(1) алгоритм $\mathrm{PQ}_{-w,v}^\beta$ является не менее чем V-конкурентным как в случае произвольных ценностей пакетов, так и в случае с двумя ценностями;

(2) алгоритм $\mathrm{PQ}_{v/w}^\beta$ является не менее чем $\min\{V, W\}$-конкурентным в случае произвольных ценностей пакетов;

(3) в случае с двумя ценностями, если $\beta W \geqslant V$, то алгоритм $\mathrm{PQ}_{v/w}^\beta$ является не менее чем V-конкурентным;

(4) алгоритм $\mathrm{PQ}_{v,-w}^\beta$ является не менее чем $\left(\frac{(V-1)}{V} W - o(1)\right)$-конкурентным в случае произвольных ценностей пакетов и по крайней мере $\left(\frac{W}{V} + o(1)\right)$-конкурентным в случае с двумя ценностями.

Все приведённые выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [8, 25].

2.7 Алгоритмы управления буфером для пакетов с ограниченной задержкой

Современные датацентры всё чаще сталкиваются с приложениями, в которых нужно контролировать сетевую задержку (latency). Низкая задержка критически важна для интерактивных приложений, а достичь требуемой задержки часто очень сложно. Современные приложения часто являются распределёнными; более того, чтобы получить конечный результат, соединяющий результаты разных запросов, ограничения на задержку часто накладываются на комплексные операции [47,74,150]. Задержка — это важный фактор, и есть два основных способа представлять требования к задержке для задач оптимизации, которые возникают в дизайне сетевых элементов. Один подход — включить задержку в целевую функцию и оптимизировать её; например, часто встречающийся способ учесть задержку в целевой функции — это минимизация *среднего времени завершения потоков* (average flow completion time, AFCT) [49, 107]. Однако это требует априорных знаний о длительностях потоков, которые могут быть недоступны, и всё ещё оставляет проблему того, как сбалансировать задержку и другие желаемые цели, например пропускную способность или равный доступ к сети.

Помимо явной оптимизации задержки, другой способ рассмотреть ту же проблему — это ввести ограничения на задержку, которые должны выполняться на уровне отдельных пакетов; этот основанный на ограничениях подход позволяет оптимизировать дополнительные целевые функции и не требует априорных знаний о длительностях потоков. Изначально оптимизация взвешенной пропускной способности для пакетов с ценностью и допуском по задержке была введена в работе [133] и названа *моделью с ограниченной задержкой* (bounded-delay model). В этой модели *допуск* (slack) определяет ограничение на задержку, т.е. разницу между временем прибытия пакета и тем временем (дедлайном), когда пакет должен быть передан или будет утерян; кроме того, в этой модели каждый успешно переданный пакет добавляет свою внутреннюю ценность (value) в целевую функцию взвешенной пропускной способности. Однако модель в работе [133] не включает другой важной характеристики, требований к времени обработки каждого пакета, которую мы подробно рассматривали уже в нескольких разделах этой главы.

Использование сразу нескольких характеристик трафика (например, ценность и требующееся время обработки пакета) существенно усложняет алгоритмы управления буфером. В этом контексте возникают два основных вопроса: как представить задержку в целевых функциях и как предоставить гарантии для алгоритмов управления буфером, которые бы выполнялись на широком спектре паттерном трафика. В этом разделе мы рассматриваем архитектуру буфера с одной очередью, где каждый входящий пакет имеет внутреннюю ценность, требующееся время обработки и допуск (slack), т.е. число временных слотов с момента прибытия, в течение которых этот пакет нужно передать. Размер буфера ограничен неявно, значениями допусков. Наша цель — максимизация общей переданной ценности (взвешенная пропускная способность). В этой постановке мы исследуем результаты о качестве работы онлайн-алгоритмов в худшем случае посредством конкурентного анализа, сравнивая с оптимальным всевидящим оффлайн-алгоритмом.

Формально говоря, мы рассматриваем коммутатор с одной очередью, где буфер делится между всеми типами трафика, и предполагаем, что каждый прибывающий пакет p снабжён тремя характеристиками:

(1) число требующихся циклов обработки (*работа*, work) $w(p) \in \{1, \dots, W\}$,

(2) *ценность* (value) пакета $v(p) \in \{1, \dots, V\}$,

(3) *допуск* (slack) $s(p) \in \{1, \dots, S\}$, который определяет, через какое время после прибытия пакет должен быть передан, прежде чем он будет потерян и ничего не добавит в целевую функцию (взвешенную пропускную способность).

Обозначим через $[\![w \mid v \mid s]\!]$ пакет с работой w, ценностью v и допуском s; обозначим пачку из n пакетов с одними и теми же параметрами через $n \times [\![w \mid v \mid s]\!]$. Подобно работе [133], мы предполагаем, что размер очереди не ограничен; заметим, что максимальное значение допуска S по сути его ограничивает. Во введённых выше обозначениях допуск пакета s уменьшается после каждого прошедшего слота времени, т.е. мы говорим и «текущем» оставшемся допуске. Пакет *полностью обработан*, если он был назначен для обработки в течение требующегося для него числа циклов обработки. Хотя максимальная работа W, максимальная ценность V и максимальный допуск S будут играть

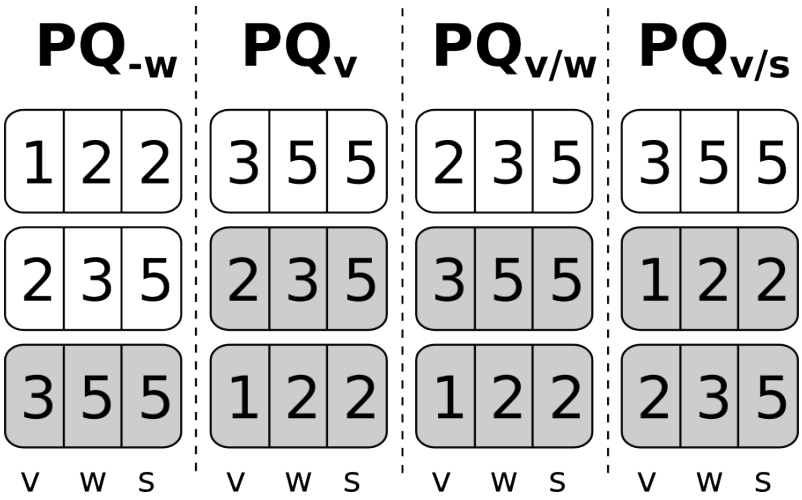


Рис. 2.10: Пример временного слота алгоритмов PQ_{-w}, PQ_{v}, $PQ_{v/w}$ и $\mathrm{PQ}_{v/s}$. Первый пакет очереди находится сверху. Заштрихованные пакеты не будут переданы из-за недостатка времени.

важную роль в анализе алгоритмов, сами предложенные онлайн-алгоритмы не должны знать их заранее. Заметим, что для $W = 1$ модель вырождается в одну очередь пакетов единичного размера с гетерогенными внутренними ценностями, как в работе [133].

Мы снова рассматриваем различные приоритетные очереди. Пусть f — некоторая функция пакетов, $f(w, v) \in \mathbb{R}$, смысл которой в том, что лучшим пакетам присваиваются более высокие значения f. Тогда алгоритм управления буфером PQ_f (*приоритетная очередь*) определяется как алгоритм, который обрабатывает пакеты в очереди в порядке убывания значений функции f на этих пакетах. В частности, рассмотрим следующие конкретные приоритетные очереди; ниже w обозначает текущую оставшуюся работу, v обозначает ценность и s обозначает текущий допуск:

(1) $\mathrm{PQ}_{-w} = \mathrm{PQ}_{-w+v/(V+1)}$ упорядочивает пакеты по возрастанию их требующегося времени обработки, разрешая «ничьи» по ценности;

(2) $\mathrm{PQ}_{v} = \mathrm{PQ}_{v-w/(W+1)}$ упорядочивает пакеты по убыванию их ценности, разрешая «ничьи» по требующемуся времени обработки;

(3) $\mathrm{PQ}_{v/w}$ упорядочивает пакеты по убыванию их отношения ценности к работе, т.е. она приоритезирует пакеты, который дают наивысшую ценность на один цикл обработки; разрешать «ничьи» можно или по работе, или по ценности, и все приведённые ниже результаты сохраняются в обоих случаях, так что мы не различаем эти варианты в обозначениях;

(4) PQ_{v/w_0} упорядочивает пакеты по убыванию их отношения ценности к исходной работе, т.е. в отличие от $\mathrm{PQ}_{v/w}$ приоритет пакета не увеличивается по мере того, как он постепенно обрабатывается и его значение w уменьшается;

(5) $\mathrm{PQ}_{v/s}$ и PQ_{v/s_0} упорядочивают пакеты по убыванию их отношения ценности к допуску и ценности к исходному допуску соответственно; эта идея уже была известна в теории планирования, где было показано, что она хорошо работает как вариант эвристики самого раннего дедлайна (earliest due date, EDD) [160].

Рис. 2.10 иллюстрирует пример временного слота с четырьмя различными порядками обработки; очереди показаны вертикально, и их первые элементы расположены сверху. Заштрихованы те пакеты, которые соответствующий алгоритм не успеет обработать; это показывает разницу между четырьмя порядками обработки.

Начнём с того, что покажем несколько *нижних* оценок на конкурентность онлайн-алгоритмов, т.е. результатов, которые показывают, насколько плох тот или иной онлайн-алгоритм в худшем случае по сравнению с всевидящим оффлайн-алгоритмом. Сначала сосредоточимся на *общих* нижних оценках, т.е. оценках, которые выполняются для всех детерминированных онлайн-алгоритмов.

Теорема 2.7.1. *Конкурентность любого детерминированного онлайн-алгоритма A не может быть ограничена сверху константой.*

К сожалению, даже если мы попытаемся отделить допуски от требующегося времени обработки некоторой *аддитивной* константой c (т.е. предположить, что $s(p) \geqslant w(p) + c$ у всех пакетов), это не поможет нам найти детерминированный онлайн-алгоритм с константной конкурентностью.

Теорема 2.7.2. *Конкурентность любого детерминированного онлайн-алгоритма* ALG *не может быть ограничена сверху константой, даже если существует такая фиксированная константа c, что для любого пакета $s(p) \geqslant w(p) + c$.*

Однако в нашем основном результате, приведённом ниже, мы покажем, что если потребовать, чтобы допуск s был отделён от w некоторой *мультипликативной* константой (т.е. $s(p) > cw(p)$ для некоторой константы c для любого пакета p), то константную конкурентность удастся получить.

Теорема 2.7.2 уже показывает, что если «just-in-time» пакеты разрешены, то ни один онлайн-алгоритм не является конкурентным. Однако оказывается, что базовые приоритетные очереди не будут конкурентными даже для больших допусков (т.е. для допусков, которые всегда больше требующегося времени обработки на некоторую константу c).

Теорема 2.7.3. *Если w, s и v являются допустимыми значениями требующегося времени обработки, допуска и ценности соответственно, и s всегда составляет не менее cw для некоторой константы c, то конкурентность основанных на приоритетных очередях алгоритмов* PQ_v, $\mathrm{PQ}_{v/w}$, PQ_{v/w_0}, $\mathrm{PQ}_{s/w}$, PQ_{s/w_0} *составляет не менее чем*

$$\frac{1-(\frac{w-2}{w})^{\frac{1}{w}\left\lfloor \log_{\frac{w}{w-2}} v\right\rfloor}}{1-\frac{w-2}{w}}\frac{1}{c},$$

что не является константой по v и w.

С другой стороны, для алгоритмов без выталкивания ситуация ещё хуже.

Теорема 2.7.4. *Если w, s и v являются допустимыми значениями требующегося времени обработки, допуска и ценности соответственно, и s всегда составляет не менее cw для некоторой константы c, то для $w > c$ конкурентность любого онлайн-алгоритма без выталкивания составляет не менее чем V.*

Однако в худшем случае α-выталкивание тоже не совсем помогает; мы иллюстрируем это ниже для $\mathrm{PQ}_{v/w}$, и подобные примеры могут быть также построены и для других приоритетных очередей.

Теорема 2.7.5. *Алгоритм* $\mathrm{PQ}_{v/w}$ *с α-выталкиванием имеет конкурентность не менее $\Omega(w)$ для произвольно больших допусков.*

Основной результат этого раздела — новая верхняя оценка для модели с тремя характеристиками, которая достигается введением нового алгоритма. Основная идея этого алгоритма состоит в том, чтобы улучшить алгоритм $\mathrm{PQ}_{v/w}$ с α-выталкиванием двумя новыми идеями, которые призваны решать проблемы, выделенные в теореме 2.7.5:

- специальные правила назначения приоритетов, которые уменьшают приоритет новых пакетов и/или увеличивают приоритет пакетов, обработка которых уже началась (чтобы избежать потерь от слишком агрессивного выталкивания);

- дополнительное правило (мы называем его β-*пессимистичностью*), которое запрещает алгоритму обрабатывать пакеты с заканчивающимися дедлайнами, так что алгоритм никогда не попадёт в «just-in-time» ситуацию.

Полученный алгоритм можно рассматривать как приоритетную очередь с двумя новыми модификациями:

- у каждого пакета есть флаг, который показывает, была ли уже начата обработка этого пакета;

- если обработка пакета ещё не началась (это «свежий» пакет), то для целей выталкивания (т.е. порядка приоритетной очереди) его ценность уменьшается в α раз (это свойство α-выталкивания);

- более того, чтобы новые пакеты начали обрабатываться, им нужен лишний «зазор» в их дедлайне: алгоритм предполагает, что доля β от времени обработки нового пакета может быть потрачена на другие пакеты в результате выталкивания, так что обработка нового пакета может начаться, только если его допуск составляет не менее чем $(1+\beta)w$ (это свойство β-*пессимистичности*).

Новый алгоритм — это приоритетная очередь $PQ_{v/w}$ с двумя вышеописанными модификациями.

Главный результат работ [3, 19] (он не выносится на защиту, поэтому мы не формулируем его в виде теоремы) говорит, что если $s \geqslant cw$ для любого пакета в момент его прибытия, то алгоритм SPQ, определённый выше, имеет конкурентность не более

$$\left(1+\frac{1}{\beta(\alpha-1)-1}\right)\left(1+\alpha+\frac{\alpha c}{(c-\beta-1)}\right).$$

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных

исследований, а также доказательства всех приведённых выше теорем, показаны в работах [3, 19].

2.8 Выбор алгоритмов управления буфером методами машинного обучения

В этом разделе мы рассматриваем принципы построения модуля очередей (queueing module, QM), который является одной из основных составных частей каждого сетевого элемента. Традиционное управление сетями позволяет разместить только конкретный фиксированный набор алгоритмов управления буфером, оптимизирующих заранее определённые целевые функции и использующих заранее определённые параметры трафика. Поиск «идеального» алгоритма — это сложная задача, которая требует высокой квалификации от операторов сети. Более того, даже хорошо выбранные параметры отражают только наше текущее понимание паттернов трафика; обычно нет никакой *обратной связи* между операторами сети и QM, и в результате качество работы даже хорошо выбранных параметров алгоритмов управления буфером будет ухудшаться, когда паттерны трафика изменятся со временем.

Вместо того чтобы реализовывать сложный цикл обратной связи, мы предлагаем декларативный подход, в котором QM получает множество «реализуемых» алгоритмов управления буфером и метрик для сравнения качества их работы. Каждый алгоритм является «чёрным ящиком», и операторы сети могут предоставить QM несколько «разумных» алгоритмов управления буфером. Инфраструктура QM использует методы машинного обучения для того, чтобы выбрать наилучший алгоритм из этого множества для текущей ситуации. Этот подход основан на модели многорукого бандита, где наилучший алгоритм («руку» бандита) нужно выбрать на основе неполной информации, минимизируя общую цену обучения (regret). Однако для того, чтобы сделать модель многоруких бандитов полезной в реальных условиях, нужно выбрать, какая из моделей подходит лучше: разные постановки задач и предположения приводят к разным алгоритмам для задачи о многоруких бандитах. Также нужно решить проблемы, связанные с дискретизацией и представлением метрик качества, например выбрать интервал для измерений; в этом разделе мы обсудим эти проблемы, а показанные здесь результаты были успешно подтверждены на

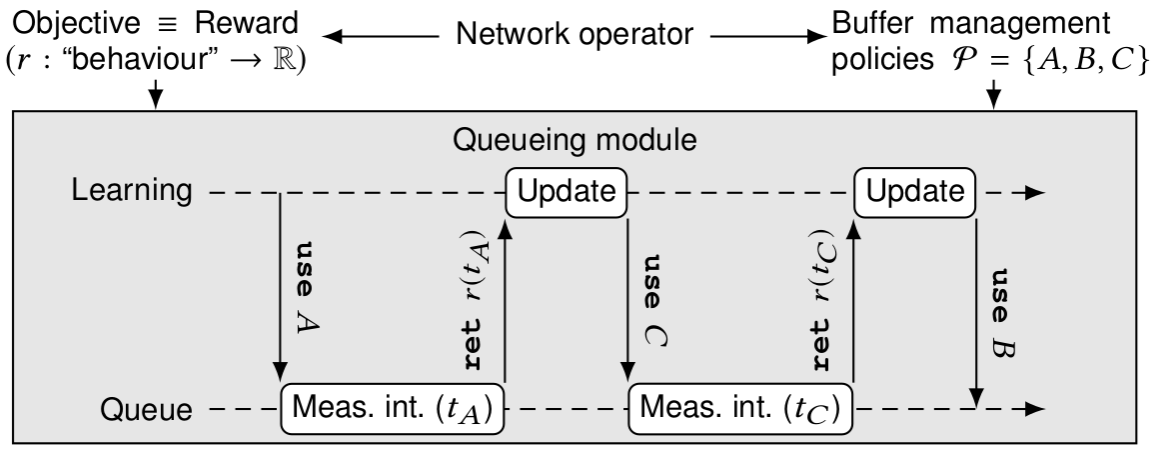


Рис. 2.11: Структура системы: оператор сети определяет целевую функцию и множество алгоритмов. Алгоритм обучения взаимодействует с очередью, чтобы выбрать наилучший алгоритм.

практике. Более того, недавно были разработаны различные программные абстракции, которые ввели атомарные примитивы, которые можно использовать, чтобы составить алгоритмы управления буфером. В этом случае оператор сети может предоставить QM такие атомарные примитивы (вместо целых алгоритмов), а инфраструктура QM построит множество алгоритмов-кандидатов, что сделает QM ещё менее зависимой от способностей оператора сети к разработке новых алгоритмов.

Предположим, что оператор сети конфигурирует QM *декларативным* образом, определяя множество *алгоритмов-кандидатов* $\mathcal{P}$ и *функцию вознаграждения* r, $0 \leqslant r(t) \leqslant 1$, которая выражает желаемую целевую функцию (больше — лучше) и позволяет оценить качество работы QM на любом временном интервале t; примеры алгоритмов-кандидатов могут включать *RED* [105] или *CoDel* [123] с различными параметрами, а r может представлять собой, например, энергопотребление сети [203], среднюю задержку пакетов или занятость сети на соответствующем интервале времени. Каждый алгоритм реализован с фиксированным набором параметров; QM ничего не знает об их внутреннем устройстве кроме того, что они останавливаются и могут работать со скоростью сети (line rate).

Обозначим через $\mathcal{T}$ общий период работы QM. Текущий преобладающий подход состоит в том, чтобы использовать фиксированный алгоритм, конфигурированный администратором сети, в течение всего периода $\mathcal{T}$. У этого статического подхода есть свои ограничения. Во-первых, администратор должен иметь

глубокое понимание паттернов трафика, чтобы сделать правильный выбор; даже имея статистику входящего трафика, выбор оптимального алгоритма — это в общем случае очень сложная задача, а предсказание будущих паттернов трафика ещё сложнее.

Ещё важнее то, что паттерны сетевого трафика, скорее всего, будут заметно меняться в течение периода $\mathcal{T}$, и в результате качество работы каждого конкретного алгоритма тоже будет меняться. Например, рис. 2.12 (это реальный результат экспериментов) показывает временной интервал $\mathcal{T}$, разделённый на T_1 и T_2, и каждый из двух рассматриваемых алгоритмов, *CoDel* [123] и *RED* [105], лучше работает на своём интервале. Таким образом, для оптимальных результатов нужно запустить разные алгоритмы на разных подинтервалах T_i; заштрихованная часть графика на рис. 2.12 показывает потери, которые приносит выбор одного алгоритма для всего интервала $\mathcal{T}$.

Главной целью этого раздела является разработка инфраструктуры управления очередями так, чтобы *автоматически* адаптироваться к изменяющимся паттернам трафика и находить алгоритм, который гипотетически лучше всего работает в текущей ситуации. Цель оптимизации — выбрать такую последовательность алгоритмов в течение периода $\mathcal{T}$, которая максимизировала бы общее качество работы $r(\mathcal{T})$. Инфраструктура QM не знает ни то, какой трафик ожидается в течение $\mathcal{T}$, ни качества работы разных алгоритмов управления буфером на этом трафике.

Будем измерять качество работы каждого алгоритма на малых *интервалах измерения* и выбирать алгоритмы, которые будем использовать в течение следущих интервалов. Обозначим вознаграждение алгоритма P, работающего на интервале измерения t, через $r_P(t)$, и разделим $\mathcal{T}$ на *стационарные* подинтервалы $T_1, \ldots, T_q$, на которых наилучший алгоритм предположительно не меняется, т.е. для каждого j один и тот же алгоритм $P_j^* \in \mathcal{P}$ показывает оптимальное ожидаемое вознаграждение на T_k. Например, на рис. 2.12 изображены два подинтервала с оптимальными алгоритмами $P_1^* = \text{CoDeL}$ и $P_2^* = \text{RED}$ соответственно. Оптимальное вознаграждение $r(\mathcal{T})$ можно получить, запуская P_j^* на каждом T_j.

К сожалению, QM не может менять алгоритмы идеально по двум причинам: QM не знает ни то, когда начинается каждый T_j, ни то, какой алгоритм является оптимальным P_j^*. Основная цель этого раздела состоит в том, чтобы пред-

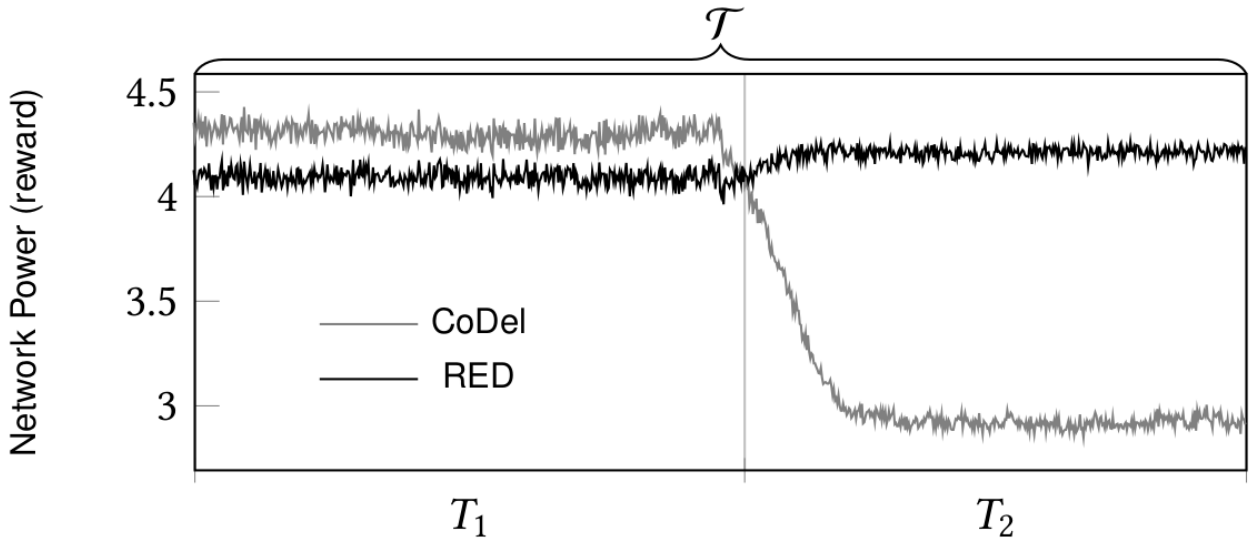


Рис. 2.12: Пример поведения двух алгоритмов; заштрихованные части показывают потери, к которым приводит использование одного и того же алгоритма в течение всего интервала $\mathcal{T}$.

ложить подходы приблизиться к этому желаемому оптимальному поведению, решая задачу *стационарной оптимизации*: выбор и запуск наилучшего алгоритма P^* в течение стационарного интервала T. Мы оставим задачу *детекции нестационарности*, т.е. обнаружения переходов между двумя стационарными интервалами, для дальнейших исследований.

Основная цель стационарной оптимизации заключается в том, чтобы запускать наилучший алгоритм P^* для самой большой возможной части интервала T. Чтобы найти P^*, некоторую часть T придётся потратить на исследование методом проб и ошибок. Есть два основных подхода к такому исследованию:

(1) *оценка, затем использование* (evaluate then exploit): выбрать фиксированный *интервал исследования* Δ в начале интервала T, в течение которого оценивать качество разных алгоритмов, а затем использовать алгоритм, который окажется лучше других, в течение оставшейся части интервала T;

(2) *постоянное исследование* (continuous exploration): сочетать исследование с использованием текущего лучшего алгоритма; этот подход продолжает «платить» цену исследования, но зато может сойтись к лучшему алгоритму с вероятностью 1 и может обрабатывать динамические (меняющиеся со временем) вознаграждения алгоритмов.

Чтобы применить известные алгоритмы с доказанными результатами об их качестве, нужно ввести некоторые предположения относительно свойств про-

исходящего процесса. Рассмотрим несколько случаев, которые приводят к совершенно разным задачам.

Самый простой случай возникает, если мы можем найти наилучший алгоритм, попробовав каждый из алгоритмов только один раз, т.е. если наилучший алгоритм *всегда* лучше любого другого, на любом временном интервале.

Предположение 1 (DISJ(Δ))**.** Для множества $\Delta = t_1 \sqcup \ldots \sqcup t_n$ интервалов измерения и любого $P \in \mathcal{P}$, $P \neq P^*$, верно, что $r_{P^*}^{\min} > r_P^{\max}$, где $r_P^{\min} = \min_i r_P(t_i)$, $r_P^{\max} = \max_i r_P(t_i)$, и P^* — определённый выше наилучший алгоритм.

Один способ ослабить предположение DISJ(Δ) состоит в том, чтобы предположить некоторое (неизвестное) распределение вероятностей на возможном поведении трафика. Предположение стационарности здесь состоит в том, что ожидаемое вознаграждение для каждого алгоритма остаётся постоянным, и вознаграждения на разных интервалах времени являются независимыми случайными величинами.

Предположение 2 (AVGEQ(Δ))**.** Для множества $\Delta = t_1 \sqcup \ldots \sqcup t_n$ интервалов измерения, любого $P \in \mathcal{P}$ и любого $1 \leqslant i < j \leqslant n$, $\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$, и $r(t_i)$ и $r(t_j)$ независимы.

Однако на практике долгосрочные зависимости часто действительно возникают, так что нужно искать постановки задачи с ещё более слабыми предположениями (которые всё же можно проанализировать). В следующей постановке задачи мы предполагаем, что чем чаще мы пробуем тот или иной алгоритм, тем ближе среднее вознаграждение становится к его истинному качеству работы; это строгое ослабление AVGEQ(Δ).

Предположение 3 (LIMIT(Δ))**.** Для множества $\Delta = t_1 \sqcup \ldots \sqcup t_n$ интервалов измерения, любого $P \in \mathcal{P}$ и любой последовательности $1 \leqslant i_1 < \ldots < i_k \leqslant n$ существует предел $\lim_{n,k\to\infty} \frac{1}{k} \sum_j r_P(t_{i_j})$.

Наконец, введём самую общую ситуацию ADV(T), в которой входы могут быть произвольными и даже состязательными.

Предположение 4 (ADV(Δ))**.** Ограничений нет, то есть $r_P(t_i)$ являются произвольными.

Заметим, что на практике эти предположения относятся к свойствам распределения трафика.

Evaluate then exploit. В этой постановке задачи мы предполагаем, что специальный период оценки Δ, являющийся префиксом T, зарезервирован для чистого исследования; в машинном обучении эта задача известна как *идентификация лучшей ручки* (best arm identification). Главное предположение здесь состоит в том, что Δ является репрезентативным для всего интервала T, и вознаграждения алгоритмов не меняются со временем, или что по крайней мере наилучший алгоритм останется наилучшим. Награда, полученная в течение Δ, нерелевантна для целевой функции обучающего алгоритмы и не приводит к потерям в общей цене обучения, так что выбор длины Δ представляет собой поиск фундаментального компромисса между исследованием и использованием, т.е. между точностью определения лучшей ручки и долей времени, когда работают субоптимальные алгоритмы. Предположим, что Δ содержит n интервалов измерения $t_1, t_2, \ldots, t_n$.

Задача 2.8.1 (FindBest). *Для* $\Delta = t_1 \sqcup \ldots \sqcup t_n$ *найти алгоритм* $P \in \mathcal{P}$, *который максимизирует* $\sum_{i=1}^{n} r_P(t_i)$.

Теперь виден естественный компромисс между предположениями и качеством работы: нам хотелось бы максимально ослабить предположения, но более общие методы скорее всего будут работать хуже.

В предположении disj(Δ) нет никакой проблемы: эвристика LocalGreedy идеальна. В предположении avgeq(Δ) есть естественный способ уменьшить дисперсию оценок: усреднить по нескольким измерениям вознаграждения. Прямолинейная стратегия Uniform запускает каждый алгоритм $\lfloor n/|\mathcal{P}| \rfloor$ раз, а затем выбирает алгоритм с наибольшим средним вознаграждением. Известно, что в предположении avgeq(Δ) вероятность ошибки стратегии Uniform убывает экспоненциально по n [71]. Однако интуитивно ясно, что если алгоритм уже много раз плохо себя зарекомендовал, лучше потратить остаток Δ на более точное различение других алгоритмов. В работе [54] эта интуиция была формализована в виде эвристики SuccessiveRejects, которая показывает доказуемо лучшие результаты, чем Uniform, но всё ещё не имеет параметров. В предположении limit(Δ) мы не можем гарантировать, что в конце любого конкретного интервала Δ оптимальный алгоритм будет выбран с высокой вероятностью: всегда возможно, что n просто было недостаточно велико. Однако можно доказать,

Предположение	Ограничения	Метод	Точность
	FindBest		
DISJ(Δ)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	exact
AVGEQ(Δ)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	SR	$1 - e^{O(n)}$
LIMIT(Δ)	$\exists\ \lim_{n,k\to\infty} \frac{1}{k} \sum_{j=1}^{k} r_P(t_{i_j})$	SH	$>$ RR
	MaxReward		
DISJ(T)	$r^{\min}_{P^*} > r^{\max}_{P}$	LG	0
AVGEQ(T)	$\mathbb{E}[r_P(t_i)] = \mathbb{E}[r_P(t_j)]$	UCB1	$O(\log n)$
ADV(T)	—	Exp3.1	$O(\sqrt{n})$

LG = LocalGreedy, SR = SuccessiveRejects,
SH = SuccessiveHalving, RR = RoundRobin

Таблица 2.1: Результаты для FindBest и MaxReward.

что методу SuccessiveHalving [119] требуется меньше измерений, чем методу, пробующему алгоритмы циклически, в предположении LIMIT(Δ). Наконец, в предположении ADV(Δ) ничего не поделать: исследовательская часть интервала не имеет никакого отношения к тому, что произойдёт дальше. Таблица 2.1 показывает краткое резюме известных результатов.

Постоянное исследование. Принимать решения на основе префикса Δ — это плохая идея в ситуации, когда Δ не является репрезентативным для всего периода T. В конце концов, идентификация лучшей ручки — это всего лишь улучшенная версия эвристики Uniform, и у неё те же проблемы: большая вероятность ошибки в предположении AVGEQ(Δ) и недостаточно быстрая сходимость в предположении LIMIT(Δ). Чтобы решить эти проблемы, можно разрешить оценивать любой алгоритм в любой момент, приходя к классическому поиску баланса между *исследованием и использованием* (exploration vs. exploitation), т.е. между сбором новых данных и использованием текущей лучшей гипотезы.

Задача 2.8.2 (MaxReward)**.** *Для множества $T = t_1 \sqcup \ldots \sqcup t_n$ интервалов измерения назначить алгоритм из $\mathcal{P}$ каждому интервалу так, чтобы максимизировать $\mathbb{E}\left[\sum_{i=1}^{n} r(t_i)\right]$.*

В задаче MaxReward баланс между исследованием и использованием достигается выбором алгоритма, а не выбором Δ. Это более гибкий подход: решения можно принимать онлайн, на основе собранных к настоящему моменту данных. Далее мы продолжаем те же предположения с Δ на весь интервал T.

Случай DISJ(T) снова тривиален: эвристика LocalGreedy всё ещё работает идеально. В предположении AVGEQ(T) средние вознаграждения каждого алгоритма предполагаются равными на всех интервалах измерения $t_1, \ldots, t_n$. Качество работы алгоритма всё ещё можно оценить, усредняя по нескольким измерениям вознаграждения. Это стандартная постановка задачи для задач о многоруких бандитах, и классические результаты в этой области применимы и к нашей задаче [192]. Например, алгоритм ϵ-GREEDY (запустить текущий лучший алгоритм с вероятностью $(1-\epsilon)$, а с вероятностью ϵ — случайный алгоритм) даёт ожидаемую цену обучения (regret) $O(\epsilon n)$, что линейно по $n = |T|$.

Лучшие алгоритмы мотивированы идеей «оптимизма при неопределённости»: предполагать лучшее при недостатке точных данных. Алгоритм UCB1 [56] имеет ожидаемую цену обучения $O(\log n)$; в интервале t_i UCB1 выбирает

$$P = \operatorname{argmax}_P \left[\hat{\mu}_{P,i} + \sqrt{2 \log i / n_{P,i}} \right],$$

где $\hat{\mu}_{P,i}$ — это текущая оценка μ_P (выборочное среднее вознаграждение), а $n_{P,i}$ — это число раз, которое алгоритм P был выбран. У UCB есть также вариант UCB-V [55], который использует приоритеты

$$\hat{\mu}_{P,i} + \sqrt{\frac{2V_{i,n_{P,i}} E_{n_{P,i},i}}{n_{P,i}}} + c\frac{3bE_{n_{P,i},i}}{n_{P,i}},$$

где $V_{i,n_{P,i}}$ — выборочная оценка дисперсии для ручки i, $E_{n_{P,i},i}$ — функция исследования, обычно $E_{n_{P,i},i} = \zeta \log i$, а c, ζ — некоторые константы.

Если нет совсем никаких ограничений (предположение ADV(T)), то оптимальное решение MaxReward может использовать несколько разных алгоритмов в течение T, что было не нужно для DISJ(T) или AVGEQ(T). Здесь не может быть общих гарантий качества, но поскольку мы знаем, что T стационарно, мы можем ограничиться сравнением только с решениями в виде одного алгоритма, т.е. с $\max_P \sum_{i=1}^{n} r_P(t_i)$. В отличие от FindBest, теперь можно получить нетривиальные результаты даже в конкурентном случае (по крайней мере в среднем) при помощи стохастического алгоритма. Один возможный алгоритм в этом случае называется Exp3.1 [57], и для него показано, что ожидаемая цена обучения составляет $O(\sqrt{n})$ [57]; она снова сублинейна, то есть потери на каждом интервале стремятся к нулю, хотя и медленнее, чем для алгорит-

ма UCB1 в предположении AVGEQ($\boldsymbol{T}$). Заметим, что обучающиеся алгоритмы должны быть быстрыми, потому что им нужно не замедлять работу сетевых элементов. К счастью, все рассмотренные выше алгоритмы имеют очень низкую вычислительную сложность; например, UCB1 или UCB-V должен только поддерживать приоритетную очередь алгоритмов и хранить несколько чисел для каждого алгоритма; число разных алгоритмов на практике невелико и не зависит от сетевого трафика или нагрузки.

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также более детальное описание предлагаемого подхода, показаны в работе [6].

Глава 3 Алгоритмы классификации пакетов

3.1 Введение в классификацию пакетов и обзор литературы

Классификация пакетов — это ключевая функциональность таких важных сетевых сервисов, как Quality of Service (QoS) и управление доступом; она приобрела ещё большее значение по мере распространения протокола OpenFlow [169], который реализует свои возможности через иерархическое сопоставление кортежей (hierarchical tuple matching) с заданными действиями; широко распространилась также классификация по нескольким полям (multi-field classification). Правило классификации, которое задаёт точные значения для всех полей, может быть представлено конкатенацией всех полей; затем классификатор можно реализовать в памяти с адресацией по содержимому (content-addressable memory, CAM) или при помощи обычной хеш-функции, использовав память, линейную по числу полей. Задача становится сложнее, если поле представлено префиксом (ограничение только на главные биты) или интервалом (ограничение, заключающее значение внутри интервала), поскольку конкатенация перестаёт работать.

Было предложено много разнообразных подходов, реализуемых программно (software-based approaches); см., например, подробный обзор [198]. Оценки сложности, полученные при помощи методов вычислительной геометрии, утверждают, что программный классификатор пакетов с N правилами и $k \geqslant 3$ полями использует либо $O(N^k)$ памяти и $O(\log N)$ времени, или $O(N)$ пространства и $O(\log^{k-1} N)$ времени [110, 171]. Таким образом, программные подходы оказываются либо слишком медленными, либо слишком требовательными к памяти даже с небольшим числом префиксных или интервальных полей: уже для 100 правил и 4 полей память в $O(N^k)$ в реальности составляет около 100MB, а время $O(\log^{k-1} N)$ составляет около 350 доступов к памяти [110].

Тернарная память с адресацией по содержимому (ternary content-addressable memory, TCAM) была введена для того, чтобы преодолеть эти ограничения для сопоставления префиксов [166]. TCAM, которая является стандартом *de facto* для реализации сетевых классификаторов, — это класс быстрой памяти для сопоставления заголовков пакетов с набором сохранённых значений, представленных в виде кортежей из слов VALUE и MASK (маска скрывает биты, значения которых не важны). В отличие от программных решений, TCAM может эффективно представлять классификацию на нескольких полях при помощи префиксов, но раскрытие интервалов может привести к экспоненциальному взрыву; каждое интервальное поле вводит дополнительный мультипликативный фактор.

Более сложные сервисы должны быть более выразительны на уже участвующих в классификации полях или на новых, и многие правила классификации естественным образом представляются интервалами. Поиск более эффективных реализаций классификаторов на интервалах IP или MAC-адресов, портов, длин пакетов и т.д. является крайне актуальной и важной задачей [85]. Современные механизмы классификации пакетов ограничены и в основном рассматривают кортежи из пяти полей, среди которых не более двух могут содержать интервалы, что существенно ограничивает возможности классификатора.

Формализуем постановку задачи. Пакеты классифицируются по их заголовкам согласно правилам классификации, хранящимся в памяти или базе данных. *Заголовок* пакета содержит k полей; i-е поле, $1 \leqslant i \leqslant k$, — это строка из W_i битов. *Классификатор* $\mathcal{K}$ — это упорядоченное множество из N правил $R_1 \ldots R_N$. *Правило* R_j — это упорядоченное множество из k полей и соответствующее действие A_j; *поле* F_i представляется интервалом значений из W_i битов, т.е. каждое правило содержит k интервалов $R = (I_1, \ldots, I_k)$, $I_i = [l_i, u_i]$. Будем предполагать, что последнее правило каждого классификатора $\mathcal{K}$ — это «catch-all» правило $R_N = (*, \ldots, *)$ с действием $A_N = \text{TRANSMIT}$, которое указывает передавать такие пакеты без изменений. Обозначим через $\mathcal{K}^{-F}$ классификатор, полученный из $\mathcal{K}$ удалением подмножества полей F из каждого правила; через $\mathcal{K}^{+F}$ обозначим классификатор, полученный из $\mathcal{K}$ расширением его правил множеством полей F (значения которых определены отдельно для каждого правила), и упростим обозначения до $\mathcal{K}^{-|F|}$ и $\mathcal{K}^{+|F|}$, когда поля ясны из контекста.

Правила применяются последовательно, т.е. на множестве правил опреде-

лён ациклический порядок $\prec$, и если заголовок соответствует и R_x, и R_y для некоторых $x \prec y$, применяется правило R_x. Классификаторы называются семантически эквивалентными, если к каждому возможному входящему пакету они применяют одно и то же правило.

Исследования в области эффективной реализации классификаторов пакетов делятся на две основные категории: алгоритмические решения (обычно программные) и решения, основанные на TCAM. Алгоритмические решения обычно основаны на одном из трёх методов: деревья принятия решений, хеширование или сжатие на основе кодирования. В работах [113, 182, 197] предлагались методы разделения многомерного пространства правил. Возможные подходящие правила находятся посредством спуска по дереву принятия решений. Существуют методы для балансирования разбиений в каждом узле дерева, но часто не удаётся избежать репликации (копирования) правил; см. также схожий подход в [211]. Все эти подходы неизбежно ищут компромисс между временной и пространственной сложностью. Например, алгоритм ABC Сонга и Тёрнера для распределения фильтров позволяет достичь более высокой пропускной способности с болеее низкими требованиями к памяти, а также может менять реализацию, чтобы получить более низкую сложность по времени или по памяти [184]. Работы [94, 122] предлагают решения, использующие хеширование для того, чтобы найти правила, которым подходит этот пакет. Эффективные представления на основе кодирования были предложены, в частности, в работах [148, 174, 176, 180].

В других работах предлагаются эффективные реализации, использующие TCAM. Память TCAM не поддерживает интервалы «из коробки», и интервал зачастую нельзя представить одной записью в TCAM. Две наиболее популярных схемы кодирования — это бинарное кодирование (binary encoding) [188] и кодирование на основе кодов Грея (SRGE encoding) [70]. При бинарном кодировании W-битовый интервал кодируется как объединение непересекающихся поддеревьев в бинарном дереве с 2^W листьями. Каждое поддерево представляется одной префиксной записью в TCAM. В этой схеме максимальное число записей, требущееся для кодирования интервала (увеличение в худшем случае), составляет $2(W-1)$. Например, для $W = 5$ интервал $[16, 23]$ кодируется одной

записью 10***, а интервал $[1, 30]$ кодируется $2W - 2 = 8$ записями:

$$00001,\ 0001*,\ 001**,\ 01***,\ 10***,\ 110**,\ 1110*,\ 11110,$$

которые представляют восемь поддеревьев: $[1]$, $[2, 3]$, $[4, 7]$, $[8, 15]$, $[16, 23]$, $[24, 27]$, $[28, 29]$ и $[30]$ соответственно. Кодирование SRGE может улучшить оценку в худшем случае до $2(W - 2)$, представляя значения кодом Грея. В этой схеме кодирования записи TCAM не обязательно являются префиксами и не обязательно представляют собой непересекающиеся подмножества интервала. Например, для $W = 5$ интервал $[1, 30]$ можно закодировать как

$$*0001,\ *001*,\ *01**,\ *1***.$$

Обе схемы кодируют правило с несколькими интервальными полями числом записей, которое может быть экспоненциально большим от числа полей k; верхняя оценка составляет $(W - 2)^k$ и $(W - 4)^k$ соответственно. Если можно использовать правила, которые негативно кодируют дополнения интервалов (записи типа «deny»), верхняя оценка улучшается до W записей [177,178]. Работы [178,179] предлагают схемы кодирования правил с k интервальными полями со сложностью, линейной от k; однако они применимы только к одному правилу, и чтобы закодировать классификатор с несколькими правилами, придётся менять стандартную архитектуру TCAM.

Некоторые работы пытались сократить число записей TCAM для заданного классификатора при помощи различных эвристик [75, 162]. Такие эвристики могут, в частности, кодировать самые часто встречающиеся интервалы дополнительными битами [76, 156], применять перестановки блоков [201], создавать основанные на деревьях архитектуры [196] или удалять избыточность [155]. Хотя многие из этих эвристик можно улучшить рандомизацией [151], они обычно либо ограничивают число полей в правиле, либо плохо масштабируются, когда число полей увеличивается. Существуют схемы, представляющие обновления правил интервальными TCAM правилами с отметками времени (timestamp-based) [193]. Были предложены и эффективные схемы для классификации и поддержки обновлений, основанные на непересекающихся правилах [69, 185, 200].

С геометрической точки зрения правило с несколькими префиксами пред-

ставляет собой многомерный гиперпараллелепипед в пространстве поиска, классификатор — это множество гиперпараллелепипедов с присвоенными им приоритетам, а задача классификации состоит в том, чтобы найти гиперпараллелепипед с наивысшим приоритетом, который покрывает данную точку. Задача локализации точки состоит в том, чтобы найти область, содержащую данную точку, по заданному множеству непересекающихся гиперпараллелепипедов. Оценки сложности для задачи локализации точки говорят, что программный классификатор пакетов с N правилами и $k \geqslant 3$ полями должен использовать либо память $O(N^k)$ и время $O(\log N)$, либо память $O(N)$ и время $O(\log^{k-1} N)$ [171]. В общем случае гиперпараллелепипеды, соответствующие правилам классификатора, могут пересекаться, что делает классификацию пакетов по меньшей мере столь же сложной задачей, как локализация точки. В этой главе мы используем свойство порядковой независимости, чтобы сократить число полей, участвующих в классификации. Например, уменьшение числа полей в представлениях классификаторов до двух соответствует задаче локализации точки в двумерном пространстве, что приводит к представлениям с логарифмическим временем поиска (lookup time) и (около)линейной памятью [84, 99, 143].

Использование порядковой независимости для улучшения времени доступа ранее исследовалось в [63, 64]. Однако до результатов, представленных в этой главе, не было работ, которые бы рассматривали порядковую независимость именно для сокращения ширины классификаторов и поддержки дополнительной выразительности. Мы не рассматриваем оптимизацию потребления энергии в этой главе, но в случае TCAM памяти, предполагая, что потребление энергии линейно зависит от числа ячеек памяти TCAM (которые участвуют в поиске), сокращение ширины классификатора неявно также приводит к сокращению потребления энергии.

3.2 Использование порядковой независимости для классификации пакетов

В этом разделе мы вводим уровень абстракции, который основывается на понятии *порядковой независимости* (order independence) классификаторов по отношению к содержащимся в них правилам. Мы покажем, что это ключевое свой-

ство для простой и эффективной реализации классификаторов. Мы покажем, что если классификатор порядково независимый, то новые поля классификации, даже интервальные, не увеличивают ни временной, ни пространственной сложности. Мы используем это соображение, чтобы построить эффективные схемы классификации, а также рассмотрим общий случай с порядковой зависимостью, что приведёт к гибридному подходу, в котором порядково независимая часть правил реализована программно с линейной памятью и логарифмическим временем поиска в худшем случае, а остальные правила размещены в TCAM. Более точно говоря, основные результаты этого раздела таковы:

(1) найдены свойства классификатора, которые гарантируют нулевую стоимость добавления новых полей, представленных интервалами или префиксами; это добавление происходит прозрачным образом для схемы реализации;

(2) предложен способ сократить число полей классификации, представленных интервалами или префиксами, что приводит к семантически эквивалентным результатам классификации;

(3) представлен способ свести задачу классификации с несколькими полями к задаче классификации на не более чем двух полях с гарантированной скоростью работы в худшем случае и линейной памятью, что улучшает предыдущие попытки сведения, которые не дают адекватных гарантий в худшем случае для классификаторов с более чем двумя полями [185, 188].

Будем говорить, что два правила R_1 и R_2 в классификаторе *пересекаются*, или *конфликтуют*, если существует по крайней мере один заголовок, который подходит (matches) и R_1, и R_2. Например, для $w = 4$, $R_1 = (1\ 0\ 0\ *)$, $R_2 = (0\ 1\ *\ *)$ и $R_3 = (1\ *\ *\ *)$ правила R_1 и R_3 пересекаются (заголовок (1 0 0 0) подходит им обоим), а правила R_1 и R_2, напротив, *не пересекаются*. R_1 и R_2 называются *порядково независимыми* (order-independent), если соответствующие множества подходящих заголовков не пересекаются, и *порядково зависимыми* (order-dependent) в противном случае. Для таких правил каждый заголовок подходит под хотя бы одно из них. Два правила R_1 и R_3 называются *транзитивно* порядково зависимыми на подмножестве полей $\mathcal{F}$, если существует такое правило R_2, что R_1 и R_2 являются порядково зависимыми, и R_2 и R_3 тоже являются порядково зависимыми.

Классификатор $\mathcal{K}$ называется *порядково независимым* (order-independent), если любые два его правила являются порядково независимыми по фильтрам; иначе $\mathcal{K}$ называется порядково зависимым (order-dependent). Заметим, что порядковая независимость эквивалентна тому факту, что для любого классификатора $\mathcal{K}'$ с теми же правилами, что у $\mathcal{K}$, но отсортированными в другом порядке (кроме последнего «catch-all» правила), любой заголовок пакета p подходит одному и тому же правилу в $\mathcal{K}$ и $\mathcal{K}'$. Это условие порядковой независимости выполняется, когда правила не пересекаются, т.е. для каждой пары правил существует по крайней мере одно поле, в котором соответствующие интервалы (или префиксы) не пересекаются. Например, классификатор $\mathcal{K}$ с двумя правилами, основанный на двух полях $R_1 = ([1,3],[4,5])$ и $R_2 = ([5,6],[4,5])$, является порядково независимым, а $\mathcal{K}'$ с двумя правилами $R_3 = ([1,3],[4,5])$ и $R_4 = ([2,4],[4,5])$ порядково зависимый: R_1, R_2 не пересекаются, поскольку интервалы в их первом поле не пересекаются, а R_3 и R_4 пересекаются: например, $(3,4)$ подходит обоим правилам из $\mathcal{K}'$. Обозначим через $\mathcal{K}(S)$ классификатор, который использует только подмножество полей S в классификации.

Порядковая независимость классификаторов или их частей — это важное и мощное свойство для эффективного представления и использования классификаторов. Классификаторы с сопоставлением по самому длинному префиксу (longest prefix matching) на одном поле являются порядково независимыми по крайней мере среди всех префиксов одной и той же длины; записи форвардинга в OpenFlow [169] обычно сохраняют порядковую независимость для простоты управления; многие классификаторы, представляющие сервисы с приоритетными схемами «best-match» являются порядково независимыми. Порядково зависимые классификаторы, представляющие соглашения об уровне обслуживания (service level agreement, SLA), часто возникают как непреднамеренные побочные продукты объединения нескольких сервисов в одном классификаторе. По нашему опыту, на практике многие классификаторы, представляющие такие сервисы, как QoS или ACL, являются порядково независимыми. Увеличение числа правил в классификаторе уменьшает шансы на то, что он останется порядково независимым *в целом*, но размер его *максимальной порядково независимой части* будет увеличиваться. Добавление новых полей к каждому правилу увеличивает шансы на порядковую независимость. В этом разделе мы вводим первую интуицию того, как работает порядковая независимость и

как она может радикально сократить требующееся для записи классификатора место в TCAM памяти.

Чтобы изучить влияние свойства порядковой независимости, мы проанализировали классификаторы из набора тестов ClassBench, порождённые с реалистичными параметрами [83], и реальные классификаторы, предоставленные Cisco Systems. Данные в таблицах 3.1 и 3.2 показывают, что новые методы эффективных представлений порядково независимых частей классификаторов могут существенно уменьшить размер требующихся классификаторов. Если классификатор не полностью порядково зависим, можно осуществлять поиск в порядково независимой и порядково зависимой частях по отдельности, а затем вернуть правило с наивысшим приоритетом из подходящих. Если найдётся подходящее правило в порядково независимой части, может быть уже не нужно проверять порядково зависимую часть; для этого в порядково независимую часть берутся правила с более высокими приоритетами, чем в других частях.

Пример 3.2.1. Рассмотрим порядково независимый классификатор $\mathcal{K} = (R_1, R_2, R_3)$ с двумя полями по 5 битов каждое и правилами

$$\begin{aligned} R_1 &= ([1,3],[4,31]), \\ R_2 &= ([4,4],[2,30]), \\ R_3 &= ([7,9],[5,21]). \end{aligned}$$

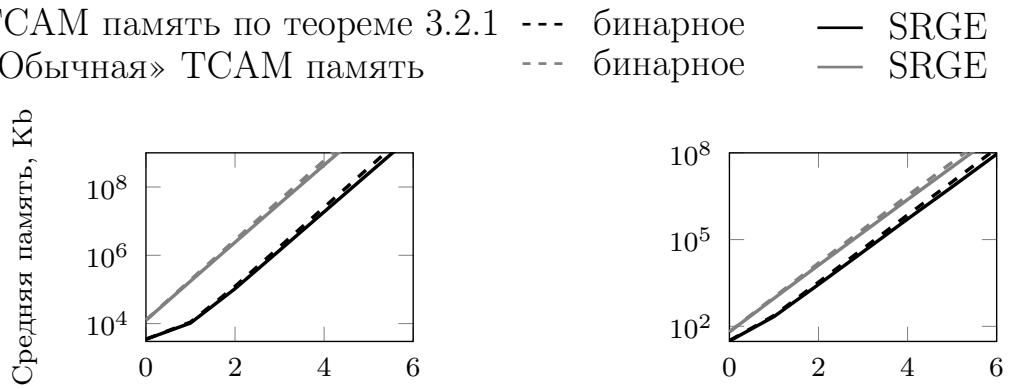
Пусть $\mathcal{K}^{+1}$ — это классификатор, который получается из $\mathcal{K}$ добавлением одного нового поля из 5 битов, что приводит к правилам

$$\begin{aligned} R_1^{+1} &= ([1,3],[4,31],[1,28]), \\ R_2^{+1} &= ([4,4],[2,30],[4,27]), \\ R_3^{+1} &= ([7,9],[5,21],[3,18]). \end{aligned}$$

Поскольку $\mathcal{K}$ порядково независимый, новые поля можно пропустить во время основной использующей TCAM классификации, и получится максимум одно подходящее поле, которое нужно будет верифицировать, чтобы избежать ложных срабатываний (см. рис. 3.2). Бинарное кодирование [188] классификатора $\mathcal{K}^{+1}$ требует $42+28+50 = 120$ записей TCAM, а SRGE кодирование [70] требует $24 + 8 + 32 = 64$ записей. С другой стороны, бинарное и SRGE кодирования $\mathcal{K}$ требуют только $6 + 7 + 10 = 23$ и $6 + 4 + 8 = 18$ записей TCAM соответственно.

	Правила		Исходный $\mathcal{K}$			По теореме 3.2.2		
	Всего	Пор. нез., все поля	Шир., биты	Размер, Kb Бин.	SRGE	OI шир., биты	Размер, Kb Бин.	SRGE
acl1	49870	49779	120	7922	7655	31	1517	1517
acl2	47276	44178	120	11289	11289	82	4189	4189
acl3	49859	47674	120	10771	10571	91	5018	5008
acl4	49556	46670	120	10079	9904	97	5379	5370
acl5	40362	38962	120	6121	6121	63	2950	2950
fw1	47778	43675	120	19454	19438	72	3926	3911
fw2	48885	48826	120	10866	10866	52	2498	2498
fw3	46038	41615	120	15090	15073	84	4161	4145
fw4	45340	42857	120	33500	33368	76	4025	4008
fw5	45723	39962	120	12478	12445	76	3759	3745
ipc1	49840	48294	120	8041	7924	50	2580	2579
ipc2	50000	50000	120	5859	5859	36	1757	1757
cisco1	584	538	120	78	78	52	34	34
cisco2	269	249	120	68	68	21	7	7
cisco3	95	92	120	11	11	30	3	3
cisco4	364	329	120	79	79	38	18	18
cisco5	148	120	120	19	19	17	5	5

Таблица 3.1: Сравнение размеров классификаторов. Слева направо: число правил (общее и в максимальном порядково независимом множестве); размер TCAM с двумя кодированиями (бинарным [188] и SRGE [70]) для классификаторов ClassBench: стандартное представление и сокращённое по теореме 3.2.2.



16-битовые интервальные поля, ClassBench 16-битовые интервальные поля, Cisco

Рис. 3.1: TCAM память для классификаторов ClassBench и CISCO как функция от числа дополнительных 16-битовых интервальных полей.

Теорема 3.2.1 (расширение полей)**.** *Пусть $\mathcal{K}^{+m}$ — классификатор, который получается из порядково независимого классификатора $\mathcal{K}$ добавлением m но-*

	Правила		Исходный $\mathcal{K}^{+2}$			По теореме 3.2.1		
	Всего	Пор. нез., все поля	Шир., биты	Размер, Kb Бин.	SRGE	OI шир., биты	Размер, Kb Бин.	SRGE
acl1	49870	49779	152	1752769	1462501	31	3857	3509
acl2	47276	44178	152	2499525	2159699	82	145925	126505
acl3	49859	47674	152	2391264	2027239	91	177771	152514
acl4	49556	46670	152	2234839	1895078	97	216994	186701
acl5	40362	38962	152	1359256	1172403	63	127818	110663
fw1	47778	43675	152	4303234	3720573	72	194313	164893
fw2	48885	48826	152	2399603	2071400	52	7382	6695
fw3	46038	41615	152	3337763	2873120	84	170688	144596
fw4	45340	42857	152	7438741	6390798	76	195130	165332
fw5	45723	39962	152	2745105	2366939	76	180420	152652
ipc1	49840	48294	152	1789100	1521153	50	52378	45391
ipc2	50000	50000	152	1301839	1123612	36	1757	1757
cisco1	584	538	120	78	78	52	34	34
cisco2	269	249	120	68	68	21	7	7
cisco3	95	92	120	11	11	30	3	3
cisco4	364	329	120	79	79	38	18	18
cisco5	148	120	120	19	19	17	5	5

Таблица 3.2: Сравнение размеров классификаторов. Слева направо: число правил (общее и в максимальном порядково независимом множестве); размер TCAM для тех же классификаторов, что в таблице 3.1, расширенных 2 новыми случайными синтетическими 16-битовыми интервальными полями, с двумя кодированиями (бинарным [188] и SRGE [70]): стандартное представление и сокращённое по теореме 3.2.1.

вых полей произвольной ширины, т.е. правило $R = (I_1, \ldots, I_k)$ *в* $\mathcal{K}$ *заменяется на* $R^{+m} = (I_1, \ldots, I_k, \ldots, I_{k+m})$ *в* $\mathcal{K}^{+m}$*. Тогда* $\mathcal{K}$ *с проверкой на ложные срабатывания для единственного подходящего правила является семантически эквивалентным* $\mathcal{K}^{+m}$*.*

По теореме 3.2.1, введение дополнительных полей, основанных на префиксах или интервалах, к порядково зависимому классификатору влияет только на размер кодирования его порядково зависимой части. Пространственная и временная сложность классификации в порядково независимой части не увеличивается, потому что новые поля в порядково независимой части можно игнорировать, не меняя при этом результата классификации; требуется только дополнительная проверка не более одного подходящего правила, чтобы избежать ложного срабатывания. Ранее новые префиксные или интервальные поля

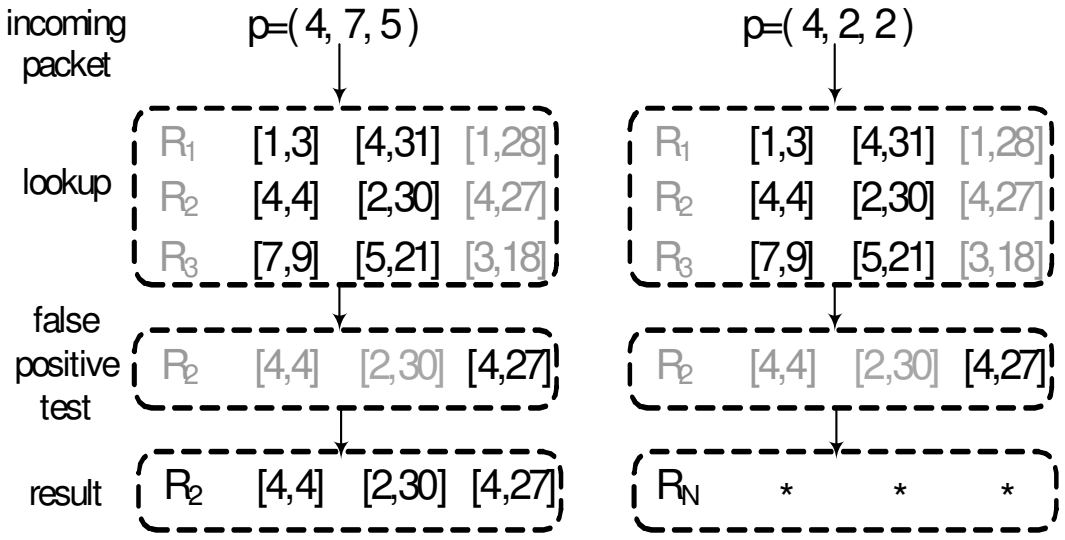


Рис. 3.2: Визуализация процедуры поиска на правилах с подмножеством полей в примере 3.2.1. Поля, проверяемые на том или ином шаге, показаны чёрным; те поля, которые не рассматриваются в данном шаге, — серым. Пакет $(4, 2, 2)$ подходит правилу R_2, но на нём не проходит проверка на ложные срабатывания на добавленном поле R_2^{+1}, и в результате классификатор выдаёт catch-all правило.

существенно увеличивали сложность программных решений (вспомним оценку в $O(\log^{k-1} N)$ времени поиска с линейной памятью). Точно таким же образом в основанных на TCAM решениях каждый интервал конвертируется в префиксы перед конфигурированием, и каждое новое интервальное поле добавляет *мультипликативный* фактор к требуемой TCAM памяти. Это является одной из причиной того, что классификаторы в виде 5-кортежей, в которых интервальными являются только поля порта-источника и порта-адресата (source port, destination port), в настоящее время являются индустриальным стандартом. Поддержка дополнительных полей, которые могли бы поддерживать интервальные правила, могла бы существенно расширить выразительность классификации пакетов (например, интервалы по дате/времени, длине пакетов и т.д.).

Эффект теоремы 3.2.1 показан в таблицах 3.1 и 3.2. Мы добавили два новых интервала на 16 битах каждый к классификаторам из тестового набора и получили существенную экономию памяти. Мы сравниваем требования по памяти, полученные из теоремы 3.2.1, с результатами двух стандартных схем кодирования интервалов: бинарной [188] и SRGE [70]. В дополнение к этому рис. 3.1 показывает, как средняя требующаяся память зависит от числа интервальных полей; в нашем подходе удалось существенно замедлить экспоненциальный рост.

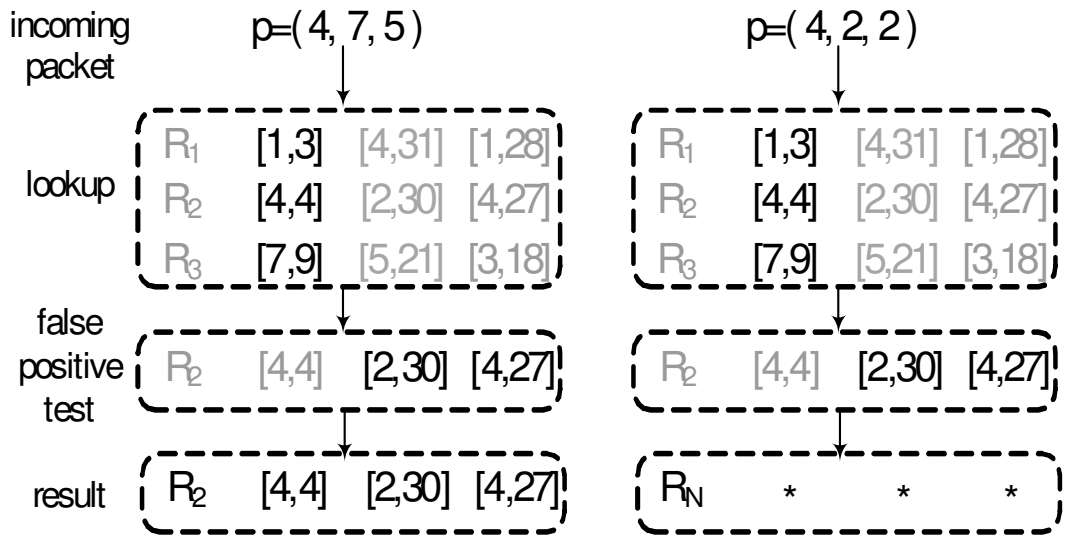


Рис. 3.3: Визуализация процедуры поиска на правилах с подмножеством полей в примере 3.2.2. Поля, проверяемые на том или ином шаге, показаны чёрным; те поля, которые не рассматриваются в данном шаге, — серым. Пакет $(4, 2, 2)$ подходит правилу $R_2^{-\{2,3\}}$, но не проходит проверку на ложные срабатывания, так что классификатор возвращает catch-all правило.

В то время как теорема 3.2.1 показывает эквивалентное представление $\mathcal{K}^{+m}$ после добавления новых полей, теорема 3.2.2 говорит, что мы можем игнорировать подмножество полей, если оставшиеся поля сохраняют порядковую независимость классификатора. Таблица 3.1 показывает эффект теоремы 3.2.2: для большинства тестовых классификаторов при помощи порядковой независимости удалось сразу же получить существенную экономию пространства (в 2–5 раз), причём на самих классификаторах, а не их искусственно расширенных версиях. Этот эффект не зависит от выбранной схемы кодирования интервалов.

Заметим, что таблицы 3.1 и 3.2 не содержат требований по памяти для проверки на ложные срабатывания (на одном подходящем правиле), поскольку она не требует конвертации интервалов в префиксы, и в её реализации можно использовать другой тип памяти (например, SRAM или SDRAM), которая не является такой дорогой, и её не так важно экономить, как TCAM. Более того, память, требующаяся для проверки на ложные срабатывания, является линейной по числу правил (до конвертации интервалов в префиксы) и ширине классификации (в битах).

Пример 3.2.2. Рассмотрим классификатор $\mathcal{K} = (R_1, R_2, R_3)$ с тремя полями

по 5 битов в каждом и правилами

$$\begin{aligned} R_1 &= ([1,3],[4,31],[1,28]), \\ R_2 &= ([4,4],[2,30],[4,27]), \\ R_3 &= ([7,9],[5,21],[3,18]). \end{aligned}$$

В этом случае первое поле необходимо и достаточно для того, чтобы гарантировать порядковую независимость, так что мы получаем, что

$$\mathcal{K}^{-\{2,3\}} = \{R_1^{-\{2,3\}}, R_2^{-\{2,3\}}, R_3^{-\{2,3\}}\} = \{([1,3]),([4,4]),([7,9])\}$$

уже является порядково независимым (см. рис. 3.3). Бинарное кодирование $\mathcal{K}$ требует $42 + 28 + 50 = 120$ записей TCAM, а SRGE кодирование $\mathcal{K}$ требует $24 + 8 + 32 = 64$ записей TCAM. С другой стороны, бинарное кодирование $\mathcal{K}^{-\{2,3\}}$ требует $2+1+1 = 4$ записей TCAM, а SRGE кодирование требует только $2 + 1 + 2 = 5$ записей TCAM.

Заметим, что удаление дополнительных интервальных полей классификации в $\mathcal{K}^{-\{2,3\}}$ может существенно сократить требования к памяти, потому что того же $\mathcal{K}^{-\{2,3\}}$ достаточно, чтобы представить $\mathcal{K}$ из примера 3.2.2. Нужна только одна дополнительная проверка на ложные срабатывания у подходящего правилу $R_i^{-\{2,3\}}$ заголовка. Чтобы ещё более увеличить производительность, удалённые из R_i поля можно проверять параллельно. Отметим, что во многих реализациях обработки пакетов проверки на ложные срабатывания можно выполнять параллельно; например, у платформы Cisco c12000 есть специальный логический механизм, который может проверять соответствие до 32 интервалов параллельно со скоростью сети [79].

Теорема 3.2.2 (сокращение полей)**.** *Пусть $\mathcal{K}^{-m}$ — классификатор, который получается из порядково независимого классификатора $\mathcal{K}$ удалением m полей, т.е. каждое правило $R = (I_1, \ldots, I_k)$ заменяется на $R^{-m} = (I_1, \ldots, I_{k-m})$. Если $\mathcal{K}^{-m}$ является порядково независимым, то $\mathcal{K}^{-m}$ с проверкой на ложные срабатывания для единственного подходящего правила является семантически эквивалентным $\mathcal{K}$.*

По теореме 3.2.2, чтобы распознать ложное срабатывание правила $R_i^{-m} \in \mathcal{K}^{-m}$, требуется проверить только m дополнительных полей правила

R_i. Если найдено ложное срабатывание, можно выдавать catch-all правило. В классификаторе $\mathcal{K}^{-m}$ может найтись не более одного подходящего правила, что радикально сокращает сложность реализации.

Поскольку баланс между временем поиска и требованиями к памяти для классификации по нескольким полям существенно зависит от числа участвующих в классификации полей k, естественной задачей становится поиск путей сокращения числа k. Для программных решений любое дополнительное поле, которое представляется как префикс или интервал, приводит к дополнительному увеличению сложности, т.е. времени поиска или занятой памяти [171]. Если сокращённый классификатор $\mathcal{K}^{-m}$ содержит не более двух полей, можно эффективно реализовать поиск за время, логарифмическое от N, с (около)линейной памятью [84, 99, 143, 188]. Для решений, использующих TCAM, если поле классификатора представляется интервалом, удаление этого поля уменьшает требуемую память TCAM пропорционально числу префиксов, которое нужно, чтобы представить этот интервал. В этом разделе мы представляем задачи оптимизации, которые являются шагами к достижению этой цели. Более того, удаление полей может привести к более короткому формату классификации. Обычно ширина формата памяти TCAM составляет 72, 144 или 288 битов, так что если удастся сократить ширину представления так, что она перейдёт за один из этих барьеров, общая требующаяся память TCAM уменьшится сразу в два раза. На протяжении этого раздела мы представим несколько идей для эвристик сокращения классификаторов, которые используют свойства порядковой независимости, а также введём соответствующие комбинаторные задачи.

Начнём с оптимизации порядково независимого классификатора на основе теоремы 3.2.2 (см. пример 3.2.2).

Задача 3.2.1 (Fields Subset Minimization, FSM)**.** *Найти максимальное подмножество полей $\boldsymbol{M}$ порядково независимого классификатора $\mathcal{K}$, для которого $\mathcal{K}^{-M}$ остаётся порядково независимым; если таких максимальных подмножеств несколько, выбрать $\boldsymbol{M}$ с максимальной общей шириной (чтобы минимизировать ширину правил, остающихся для поиска).*

FSM можно применять только к порядково независимым классификаторам, и решение этой задачи не всегда может сократить число полей классификации для порядково независимых классификаторов. Вполне может оказаться так,

например, что каждое поле буквально одного правила требуется, чтобы сохранить порядковую независимость всего классификатора. Один из способов справиться с этими недостатками — построение *мультигруппового* (multi-group) представления. Подмножества полей классификации может оказаться достаточно, чтобы гарантировать порядковую независимость подмножества правил классификатора. Предположим, что правила классификатора $\mathcal{K}$ могут быть распределены по β группам так, что:

(1) каждое правило принадлежит только одной группе;

(2) правила в каждой группе порядково независимы на подмножестве из k полей классификатора $\mathcal{K}$ (кроме правила «catch-all»);

(3) разные группы могут переиспользовать одни и те же поля, чтобы сохранять порядковую независимость.

По теореме 3.2.2, поиск в группе возвращает не более одного подходящего правила, которое нужно проверить на ложные срабатывания по всем оставшимся полям; алгоритм возвращает подходящее правило, прошедшее эту проверку и обладающее наивысшим приоритетом (из β подходящих правил). Назовём это *мультигрупповым* представлением $\mathcal{K}$. Мультигрупповые представления существуют для любого порядково зависимого классификатора, но каждая группа должна быть порядково независима. Следующая теорема очевидно следует из построения такого представления.

Теорема 3.2.3. *Мультигрупповое представление классификатора семантически эквивалентно исходному классификатору.*

Поиск в группах может проходить параллельно. Если каждая группа в мультигрупповом представлении может быть порядково независимой на не более чем двух полях, мы получаем семантически эквивалентное SW-представление классификатора с линейной памятью и гарантированным в худшем случае логарифмическим временем поиска. Пример 3.2.3 показывает, как правила классификатора $\mathcal{K}$ могут быть разбиты на группы.

Пример 3.2.3. Рассмотрим $\mathcal{K} = \{R_1, R_2, R_3, R_4, R_5\}$ с тремя полями из четырёх

битов в каждом и правилами

$$\begin{aligned} R_1 &= ([5,10],[4,7],[4,5]), \\ R_2 &= ([1,4],[4,7],[4,5]), \\ R_3 &= ([1,9],[1,3],[4,6]), \\ R_4 &= ([1,9],[4,7],[1,3]), \\ R_5 &= ([1,9],[4,7],[5,6]). \end{aligned}$$

Хотя всё множество правил $\{R_1, R_2, R_3, R_4, R_5\}$ не является порядково независимым (например, $R_1 \cap R_5 \neq \emptyset$), правила можно разделить на два независимых подмножества, $\{R_1, R_2, R_3\}$ и $\{R_4, R_5\}$. Первые два поля необходимы для того, чтобы правила в первой группе стали порядково независимыми, а для порядковой независимости во второй группе достаточно третьего поля.

Рис. 3.4 показывает визуализацию процедуры поиска после этой группировки: для входящего пакета $p = (2, 4, 5)$ система запускает параллельный поиск в двух группах; эти два поиска возвращают R_2 и R_5, проверяя только поля $\{1, 2\}$ и $\{3\}$ соответственно. Затем нужно проверить, не было ли это ложным срабатыванием, сопоставляя p с остальными полями уже только для возвращённых на первом шаге правил, а затем выбрать окончательный ответ по приоритету правил. Не более одного правила будет возвращено для проверки на ложные срабатывания из каждой группы. Если ни одно подходящее правило не найдено, или если последняя проверка обнаружит, что это было ложное срабатывание, в сравнение приоритетов будет отправлено catch-all правило, так что каждая группа отправляет для сравнения приоритетов ровно одно правило для любого пакета.

В некоторых порядково независимых классификаторах удаление любого поля сделает их порядково зависимыми, что означает, что FSM не может сократить число полей. Это приводит нас к следующей задаче оптимизации, которую можно применять и к порядково зависимым классификаторам.

Задача 3.2.2 (l-Groups of Rules, l-MGR)**.** *Для данного классификатора $\mathcal{K}$ найти распределение правил $\mathcal{K}$ на минимальное число непересекающихся групп, в котором разные группы правил могут быть основаны на тех же или разных подмножествах из не более чем $l \leqslant k$ полей, и каждая группа является порядково независимой на этих полях.*

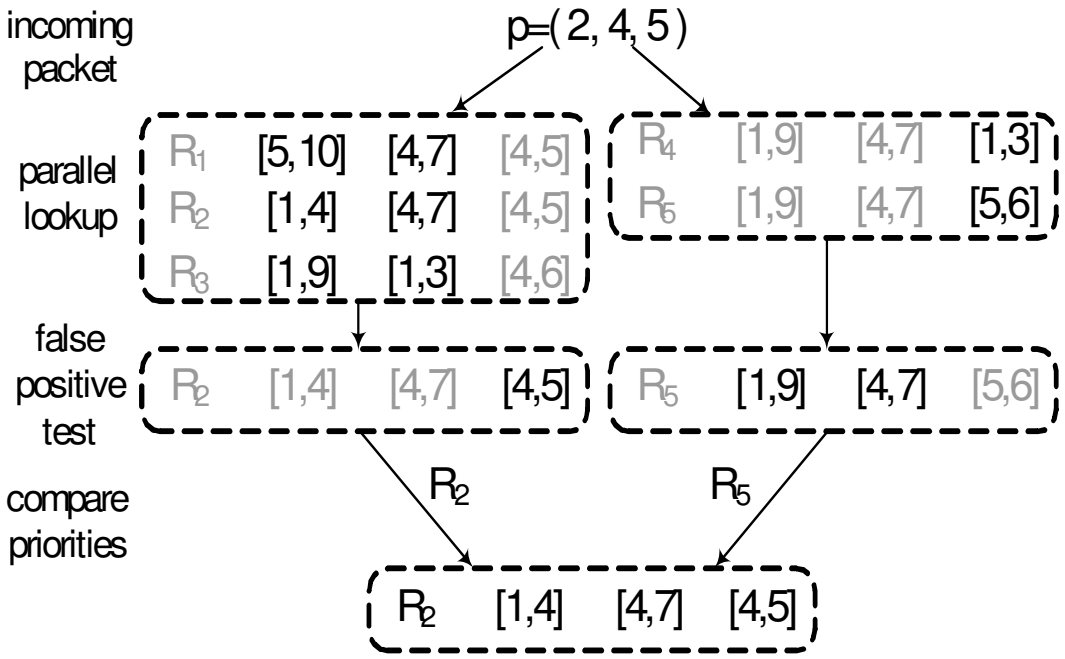


Рис. 3.4: Визуализация процедуры поиска с мультигрупповым представлением правил в примере 3.2.3. Поля, проверяемые на том или ином шаге, показаны чёрным; те поля, которые не рассматриваются в данном шаге, — серым.

Задача 3.2.2 показывает, как вести работу с порядково зависимыми классификаторами. Однако число поисков (lookups), которые могут быть запущены параллельно, является параметром системы, и минимальное число групп, найденное в результате решения Задачи 3.2.2, может оказаться неприемлемым. В этом случае мы можем разложить порядково зависимый классификатор $\mathcal{K}$ на порядково независимую часть $\mathcal{I}$ и порядково зависимую часть $\mathcal{D}$. Можно проводить параллельный поиск на обеих частях и возвращать лучшее из подходящих правил. Чтобы хранить $\mathcal{D}$, можно использовать TCAM или другую общую реализацию. Поскольку на практике подмножество порядково зависимых правил на k полях должно быть гораздо меньше, чем N, таким образом можно сэкономить большую часть TCAM памяти. Задача состоит в том, как найти правила, которые можно выделить в множество I. Один возможный путь — найти максимальное порядково независимое подмножество правил на всех k полях и решить задачу 3.2.1 или 3.2.2 на $\mathcal{I}$. Поиск максимального порядково независимого множества правил на k полях становится особенно интересным в случае, когда мы расширяем уже существующие классификаторы новыми полями (теорема 3.2.1). Другое направление состоит в том, чтобы найти порядково независимую часть, которая уже обладает нужными свойствами.

Задача 3.2.3 (Maximum Rule Coverage, l-MRC)**.** *Для данного классификатора $\mathcal{K}$ и положительного числа $l \leqslant k$ найти максимальное подмножество $\mathcal{I} \subseteq \mathcal{K}$, которое является порядково независимым на не более чем l полях.*

Пример 3.2.4. Рассмотрим классификатор $\mathcal{K} = \{R_1, R_2, R_3\}$ с тремя полями и первые три правила из примера 3.2.3:

$$\begin{aligned} R_1 &= ([5,10],[4,7],[4,5]), \\ R_2 &= ([1,4],[4,7],[4,5]), \\ R_3 &= ([1,9],[1,3],[4,6]). \end{aligned}$$

Если $l = 2$, можно получить максимальное подмножество $\mathcal{K}' = \mathcal{K}$ из трёх правил, так как первые два правила гарантируют, что множество $\mathcal{K}$ из этих трёх правил является порядково независимым. Точно так же, если $l = 1$, и достаточно выбрать только одно поле, чтобы различить между правилами, можно получить порядково независимый набор из двух правил. Рассматривая первое поле, можно получить подмножество $\mathcal{K}' = (R_1, R_2)$. Используя только второе поле, можно получить другие возможные порядково независимые подмножества, например $\mathcal{K}' = (R_1, R_3)$ или $\mathcal{K}' = (R_2, R_3)$. Также можно видеть, что все три правила вместе не являются порядково независимыми на основе никакого одного поля, и соответственно три вышеупомянутых подмножества являются максимальными.

Обобщённый вариант задачи l-MRC сочетает преимущества декомпозиции на I и D и мультигруппового представления.

Задача 3.2.4 ((β, l)-MRC)**.** *Для классификатора $\mathcal{K}$ и двух положительных чисел $l \leqslant k$ и β найти максимальное подмножество правил $\mathcal{I} \subseteq \mathcal{K}$, которые могут быть распределены по не более чем β группам так, что каждая группа порядково независима на не более чем l полях.*

В некоторых случаях может быть разумно поместить в $\mathcal{D}$ несколько больше правил, чем строго необходимо, и тем самым существенно уменьшить число групп; на практике это обычно случается с более общими правилами, которые находятся ближе к концу классификатора.

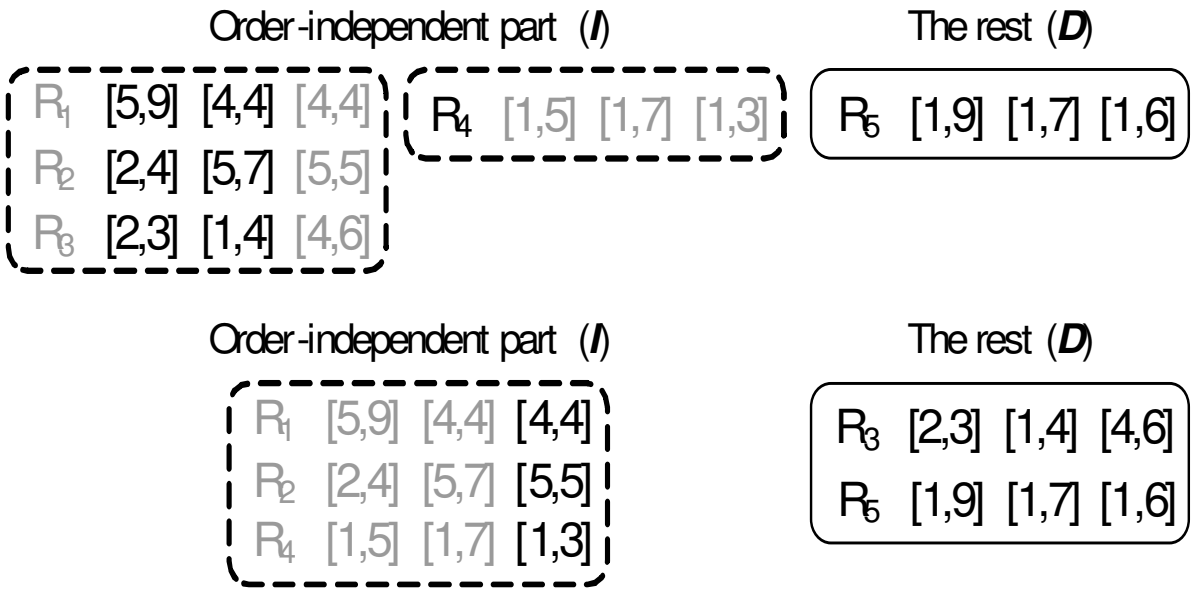


Рис. 3.5: Иллюстрация к примеру 3.2.5. Вверху: мультигрупповое представление $\mathcal{I} = \{R_1, R_2, R_3, R_4\}$ и $\mathcal{D} = \{R_5\}$. Внизу: более компактное представление с $\mathcal{I} = \{R_1, R_2, R_4\}$ и $\mathcal{D} = \{R_3, R_5\}$. Поля, проверяемые на том или ином шаге, показаны чёрным; те поля, которые не рассматриваются в данном шаге, — серым.

Пример 3.2.5. Рассмотрим классификатор $\mathcal{K}$ с тремя полями из пяти битов в каждом и пятью правилами

$$\begin{aligned} R_1 &= ([5,9],[4,4],[4,4]),\\ R_2 &= ([2,4],[5,7],[5,5]),\\ R_3 &= ([2,3],[1,4],[4,6]),\\ R_4 &= ([1,5],[1,7],[1,3]),\\ R_5 &= ([1,9],[1,7],[1,6]). \end{aligned}$$

Максимальное порядково независимое подмножество на всех трёх полях — это $\mathcal{I} = \{R_1, R_2, R_3, R_4\}$, и одно из возможных мультигрупповых представлений $\mathcal{I}$ состоит из двух групп: $\{R_1, R_2, R_3\}$ (на основе первых двух полей) и $\{R_4\}$. Но если мы положим $\mathcal{D} = \{R_3, R_5\}$, остальные правила будут образовывать одну группу, которая является порядково независимой на одном поле, а именно третьем; это проиллюстрировано на рис. 3.5.

В этом разделе мы обсуждаем частный случай, в котором каждое поле представляется префиксом, так что поля могут быть представлены отдельными строками битов. Этот частный случай позволяет нам представлять классификаторы как булевы формулы и применять хорошо известные методы булевой

оптимизации, чтобы сократить ширину классификатора и число правил. Для данного изложения мы конкатенируем все поля и считаем, что правило — это тернарная строка в формате Value–Mask–Action (поскольку ширина полей фиксирована, мы не теряем информацию в результате такой трансформации). В этом разделе мы рассматриваем порядково независимые классификаторы; от классификатора требуется решить, нужно ли применять действие к пакету, т.е. вычислить некоторую булеву функцию.

Правило $s = s_1 s_2 \dots s_k$, $s_j \in \{0, 1, *\}$, может быть представлено как конъюнкция $f_s(x_1, \dots, x_k) = \bigwedge_{s_i=1} x_i \wedge \bigwedge_{s_i=0} \bar{x}_i$. Более того, порядково независимое множество правил $\mathcal{R} = \{R_1, \dots, R_N\}$ с правилами $s_1, \dots, s_k$ может быть представлено как неупорядоченная дизъюнкция отдельных правил, т.е. как формула глубины 2 в дизъюнктивной нормальной форме (ДНФ).

Пример 3.2.6. Множество правил ширины 5

$$\begin{array}{lllll} (0 & 1 & * & 0 & *) \to 1 \\ (* & 1 & 0 & 1 & *) \to 1 \\ (* & 1 & 1 & 1 & 0) \to 1 \\ (* & 1 & 1 & 1 & 1) \to 1 \end{array}$$

является порядково независимым и эквивалентным

$$\begin{aligned} &f(x_1, x_2, x_3, x_4, x_5) = \\ &\quad = (\bar{x}_1 \wedge x_2 \wedge \bar{x}_4) \vee (x_2 \wedge x_4 \wedge \bar{x}_3) \vee (x_2 \wedge x_3 \wedge x_4 \wedge \bar{x}_5) \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_5). \end{aligned}$$

Такие образом, минимизация памяти, требующейся для представления порядково независимых правил, эквивалентна следующей задаче.

Задача 3.2.5 (MinDNF)**.** *Для заданной булевой функции f найти ДНФ минимального размера, эквивалентную функции f.*

Минимизация ДНФ может привести и к сокращению числа правил (слияние клозов в ДНФ), и к сокращению ширины, требующейся для классификации (удаление лишних переменных в клозах).

Пример 3.2.7. Минимизируем множество правил из примера 3.2.6. Мы применяем широко известную эвристику, основанную на правиле резолюции [175],

к клозам в ДНФ-представлении функции f:

$$\begin{aligned}(x_1 \wedge \bar{x}_2 \wedge \bar{x}_5) \vee (x_1 \wedge \bar{x}_2 \wedge x_5) &= (x_1 \wedge \bar{x}_2),\\ (x_1 \wedge \bar{x}_2) \vee (x_1 \wedge \bar{x}_2) &= x_2.\end{aligned}$$

В итоге мы получаем $f(x_1, x_2, x_3, x_4, x_5) = (\bar{x}_1 \wedge x_2) \vee x_2$, и второй клоз здесь может поглотить первый: $f(x_1, x_2, x_3, x_4, x_5) = x_2$. После минимизации соответствующего ДНФ-представления можно снова выразить результат как множество правил, в данном случае как одно правило: $(*1***) \to 1$. Таким образом, в этом примере мы сократили число правил с четырёх до одного и сократили число битов, участвующих в поиске, с четырёх (чтобы применить правила из примера 3.2.6 в первоначальном виде, нужно было бы запросить все биты, кроме четвёртого) до одного (второго) бита.

Задача *MinDNF* широко изучалась в теории сложности. Её бинарная версия (по формуле φ в ДНФ и числу k найти, существует ли эквивалентная ДНФ размера меньше k) вряд ли содержится в классе NP. Она очевидно лежит в классе $\mathrm{NP}^{\mathrm{NP}}$: чтобы решить эту задачу, нужно угадать короткую формулу в ДНФ и затем подтвердить NP-оракулом, решающим задачу выполнимости, что эти две формулы действительно эквивалентны. Из теории сложности известно, что *MinDNF* является $\mathrm{NP}^{\mathrm{NP}}$-полной ($\Sigma_2^P$-полной) задачей, и был доказан ряд результатов о её неприближаемости [142, 195]. На практике классические эвристики для *MinDNF* основаны на картах Карно и алгоритме Квайна–Маккласки.

Другой взгляд на эту задачу возник для функций, заданных таблицами истинности [50]; это значит, что длина входа теперь экспоненциально велика от числа переменных, и алгоритмам также разрешается использовать экспоненциально большую память. В этом случае *MinDNF* можно представить как частный случай задачи покрытия множествами (Set Cover), что приводит к тому, что для поиска минимальных ДНФ можно использовать жадные алгоритмы для задачи покрытия множествами. Известно, что этот алгоритм $O(n)$-приближённый, где n — число переменных в формуле (для функций, заданных таблицами истинности, он становится соответственно $O(\log T)$-приближённым, где T — размер таблицы истинности), и соответствующая нижняя оценка тоже известна [50]. Однако этот подход не может быть использован на практике для оптимизации классификаторов: реальные классификаторы осуществляют поиск по нескольким сотням битов, и составить таблицы истинности явно было

бы невозможно.

На первый взгляд, *MinDNF* — это более общая задача, чем *FSM*, потому что она может сократить и ширину классификатора, и число правил. Однако за счёт дополнительной проверки на ложные срабатывания *FSM* может уменьшить ширину классификатора гораздо сильнее, чем даже оптимальные результаты *MinDNF*.

Пример 3.2.8. Рассмотрим следующий классификатор:

$$\begin{aligned} R_1 &= (\ \texttt{100*},\ \ \texttt{001*}\ ), \\ R_2 &= (\ \texttt{1010},\ \ \texttt{0001}\ ), \\ R_3 &= (\ \texttt{000*},\ \ \texttt{****}\ ), \\ R_4 &= (\ \texttt{001*},\ \ \texttt{****}\ ). \end{aligned}$$

FSM может сократить ширину $\mathcal{K}$ с восьми битов до четырёх (если считать, что поля нельзя делить на части) или даже двух (если можно побитовым образом сокращать часть ширины поля).

С другой стороны, единственная MinDNF эвристика, которая здесь применима, — это правило резолюции, которым можно слить R_3 и R_4, получив

$$\begin{aligned} R_1 &= (\ \texttt{100*},\ \ \texttt{001*}\ ), \\ R_2 &= (\ \texttt{1010},\ \ \texttt{0001}\ ), \\ R'_3 &= (\ \texttt{00**},\ \ \texttt{****}\ ). \end{aligned}$$

У этого классификатора ширина 8, и если даже выбросить биты с одинаковыми значениями (второй бит первого поля), получится ширина 7, то есть экономия памяти получается значительно меньше, чем у FSM.

Таблица 3.3 показывает, как эвристики MinDNF применяются к нашему тестовому множеству классификаторов; мы применили MinDNF к порядково независимому множеству из таблиц 3.1 и 3.2. Число бинарных правил иногда существенно превышает число исходных правил, и MinDNF не даёт существенного сокращения ширины, т.е. лишь малое число битов полностью превращаются в «don't care» биты («звёздочки») даже после всех возможных редукций в стиле MinDNF.

В таблице 3.3 приводятся и «чистая» ширина (для получения которой убираются только биты, которые полностью заполнились «звёздочками»), и ещё бо-

	Исходный классификатор					После редукций MinDNF						OI
	Правила		Правила		Шир.,	Прав.,	Шир.,	Сокр.,	Прав.,	Шир.,	Сокр.,	шир.,
	исх.	OI	бин.	SRGE	биты	бин.	биты	биты	SRGE	биты	биты	биты
acl1	49870	49779	67511	65240	120	67505	90	90	65234	90	90	31
acl2	47276	44178	90772	90772	120	90233	104	104	90230	104	104	82
acl3	49859	47674	85252	83630	120	85226	106	106	83605	106	106	91
acl4	49556	46670	77837	76424	120	77755	106	106	76338	106	106	97
acl5	40362	38962	47514	47514	120	46261	96	94	46249	96	94	63
fw1	47778	43675	159525	159519	120	159461	112	112	159458	112	112	72
fw2	48885	48826	92646	92646	120	92316	88	88	92316	88	88	52
fw3	46038	41615	122495	122477	120	122259	112	112	122244	112	112	84
fw4	45340	42857	278887	277921	120	277799	104	104	276807	104	104	76
fw5	45723	39962	99574	99412	120	99421	112	112	99273	112	112	76
ipc1	49840	48294	66718	65734	120	66715	112	112	65731	112	112	50
ipc2	50000	50000	50000	50000	120	50000	112	112	50000	112	112	36
cisco1	584	538	603	603	120	475	104	86	474	104	86	52
cisco2	269	249	565	565	120	564	104	84	564	104	84	21
cisco3	95	92	92	92	120	92	88	68	92	88	68	30
cisco4	364	329	629	629	120	629	104	84	629	104	84	38
cisco5	148	120	139	139	120	139	104	84	139	104	84	17

Таблица 3.3: Экспериментальные результаты сокращения методом MinDNF в порядково независимых подмножествах классификаторов.

лее сокращённая ширина, для получения которой мы вырезали биты, имеющие одно и то же значение, не обязательно $*$. Это булев аналог нашего предыдущего метода сокращения классификаторов, поскольку мы аналогичным образом можем использовать одну проверку, чтобы сопоставить биты, всегда имеющие одно и то же значение, и результат этой проверки не изменится в зависимости от того, какое правило будет выбрано. Однако даже с этим усилением MinDNF не может существенно сократить ширину классификатора.

Пример FSM показывает, что алгоритмы, которые используют константное число дополнительных проверок на ложные срабатывания, могут существенно сократить требующуюся для оптимального представления булевых выражений память.

Заметим, что минимизировать TCAM в случае порядково зависимых классификаторов можно при помощи следующего эвристического подхода: сократить каждое множество правил, которое производит одно и то же действие, отдельно; в дальнейшем мы обозначим это действие через 1, а все остальные — через 0. Отличие от предыдущего случая здесь в том, что нам нужно закодировать тот факт, что правило применимо, и не применимо ни одно из предыдущих правил, которые производят другое действие. Поскольку мы переиспользуем

одни и те же выдающие ноль правила много раз, самым естественным представлением в данном случае будет булева схема.

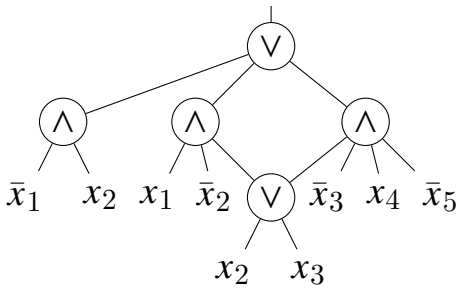
Пример 3.2.9. Рассмотрим следующее упорядоченное множество правил:

$$\begin{array}{ll}
(0\ \ 1\ \ *\ \ *\ \ *) & \to 1 \\
(*\ \ 0\ \ 0\ \ *\ \ *) & \to 0 \\
(*\ \ *\ \ 0\ \ 1\ \ 0) & \to 1 \\
(1\ \ 0\ \ *\ \ *\ \ *) & \to 1 \\
(*\ \ *\ \ *\ \ *\ \ *) & \to 0
\end{array}$$

Оно может быть представлено в виде следующей булевой формулы:

$$f(X) = (\bar{x}_1 \wedge x_2) \vee ((x_2 \vee x_3) \wedge \bar{x}_3 \wedge x_4 \wedge \bar{x}_5) \vee ((x_2 \vee x_3) \wedge x_1 \wedge \bar{x}_2)\,,$$

которая эквивалентна следующей схеме (мы копируем входные переменные для ясности иллюстрации):



Каждое упорядоченное множество правил соответствует схеме глубины 3 с чередованием OR–AND–OR, т.е. схеме из класса AC_3^0. Более того, правила TCAM представляют собой частный случай AC_3^0: все $\wedge$-гейты, соответствующие правилам под некоторым правилом с действием 0, должны быть связаны с соответствующим $\vee$-гейтом первого уровня. Таким образом, мы приходим к задаче *Ordered Min-*AC_3^0: для данной булевой функции f найти реализующую её *упорядоченную* AC_3^0 схему минимального размера, т.е. схему, для которой существует линейный порядок $\prec$ на $\vee$-гейтах первого уровня, для которого, если $\wedge$-гейт второго уровня связан с $\vee$-гейтом g, он также связан с теми $\vee$-гейтами g', для которых $g' \prec g$.

С точки зрения сложности в худшем случае задача Min-AC_d^0 изучалась в работе [50]. Она очевидно сложнее, чем *MinDNF*, хотя нам не известны результаты о неприближаемости решения задачи *Min-*AC_d^0 для малых значений d,

например $d = 3$ (результаты о неприближаемости известны для больших d). Нам также не известны результаты конкретно об *Ordered Min-*AC_3^0, так что для практических приложений мы рекомендуем обобщения эвристических правил для *MinDNF*, которые обсуждались выше. В порядково зависимом случае нет простого сведения со случая нескольких действий до двух действий: правила с разными действиями могут чередоваться. Можно только свести общую задачу минимизации TCAM к оптимизации упорядоченной AC_3^0 схемы с несколькими действиями.

Все упомянутые выше результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы, а также позволили выделить конкретные ситуации, где лучше применять те или иные алгоритмы, что открывает возможности для практического внедрения. Мы проводили эксперименты на 12 классификаторах из Classbench, порождённых с реалистичными параметрами [83], каждый с $\approx$ 50K правилами на 6 полях, а также на 5 настоящих классификаторах, предоставленных Cisco Systems; мы показали, что 90-95% правил можно реализовать с линейной памятью и гарантированным в худшем случае логарифмическим временем, оставляя только 5-10% для реализации в TCAM. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [14, 26, 37, 42].

3.3 Мультигрупповые представления классификаторов пакетов

Результаты, показанные в этом разделе, главным образом мотивированы задачей, крайне релевантной для сетевых приложений: представлением таблиц маршрутизации IPv6 на существующей архитектуре, предназначенной для IPv4. Хотя на первый взгляд кажется, что реализация поиска в таблицах маршрутизации протокола IPv6 должна быть аналогична IPv4 (семантика сопоставления самых длинных префиксов остаётся той же самой), есть целый ряд тонкостей, которые оказываются важны для построения эффективных коммутаторов для протокола IPv6. Ниже мы обсуждаем основные принципы их дизайна и намечаем пути эффективной реализации.

Одна тема, часто встречающаяся среди программных FIB реализаций

IPv6, — это различные формы представления в виде дерева принятия решений, например *префиксных деревьев* [62,92,98,168,183,186]. Обычно узлы дерева соответствуют префиксам, метки на узлах определяют «действия» (например, переслать к данному пункту назначения), и LPM-поиск (longest prefix match) состоит в том, чтобы спуститься по дереву вдоль пути, отмеченного последовательными битами ключа поиска (например адресом пункта назначения). Однако префиксные деревья по природе своей *последовательные*, то есть LPM-поиск состоит из нескольких последовательных поисков, каждый из которых опирается на результаты предыдущих, и общее число шагов ограничено только шириной адресного пространства (128 для IPv6), потому что префиксное дерево может содержать несколько подходящих записей для данного IPv6 адреса, и при LPM-поиске нам нужно найти самый точный из них (самый длинный префикс).

Заметим, что каждый шаг требует отдельного нового случайного доступа к памяти. Есть верхний предел на число последовательных доступов к памяти, возможных для каждого пакета со скоростью сети (line rate). Это приводит к строгим ограничениям на число уровней в дереве, и поэтому большинство представлений FIB в виде префиксных деревьев для протокола IPv4 не могут быть наивным образом расширены до IPv6 с сохранением требующихся времени поиска и памяти. Поскольку адресное пространство разрежено, традиционные формы сокращения числа шагов обхода дерева (например, сжатие уровней, которое было предложено для IPv6 в [168, 186]) неприменимы для IPv6. Поэтому большинство предлагаемых методов концентрируются на том, чтобы эффективно кодировать разреженные многоуровневые поддеревья внутри узлов дерева, чтобы сэкономить время на доступы к RAM [62, 92, 98, 183].

Чтобы справиться с этим ограничением, IPv6 FIB представления могут использовать нетривиальный уровень *параллелизма* (то, сколько параллельных поисков можно делать для каждого пакета со скоростью сети), который является внутренней характеристикой каждого сетевого элемента. Самый прямолинейный вариант параллелизации поиска в IPv6 FIB состоял бы в том, чтобы разбить префиксы на группы так, чтобы каждая группа содержала все префиксы одной и той же длины [152, 199, 213], а затем выполнять поиск по каждой группе префиксов одновременно. В итоге этот алгоритм выдаёт то из подходящих правил, которое принадлежит группе с самой большой длиной префикса. Эта

схема очень привлекательна для реализации, потому что поиск внутри групп сводится к простому сопоставлению точных значений, ключом для которого являются столько начальных битов адреса, сколько нужно для длины префикса группы. Этот поиск легко провести или в CAM, или при помощи хеширования. К сожалению, число разных длин префиксов, а значит и число параллельных тредов поиска, которые нужно запустить, слишком велико в случае протокола IPv6, поскольку практически никакое оборудование (кроме GPU [115]) не имеет необходимого числа независимых ядер.

Таким образом, мы приходим к главному вопросу этого раздела: как мы можем найти альтернативные разложения для IPv6 FIB, которые были бы более эффективны и более подвержены параллелизации, чем разложения по длине префикса? Разложение должно быть таким, что:

(1) поиск внутри группы максимально прост, предпочтительно по точному совпадению;

(2) число групп остаётся разумным, скажем не более пары десятков;

(3) число участвующих в классификации битов в каждой группе существенно меньше, чем исходная ширина классификатора (например, было бы замечательно сократить ширину со 128 до 32 или даже 13 битов).

Последнее требование позволило бы нам представлять IPv6 FIB на существующей сетевой инфраструктуре, предназначенной для IP и MPLS.

В этом разделе мы даём положительный ответ на этот вопрос, введя несколько эвристик, которые служат новым уровнем абстракции и помогают свести поиск в IPv6 FIB к нескольким параллельным запускам поиска в классификаторах на основе длины префиксов гораздо меньшей ширины, вплоть до 32 и в некоторых случаях даже 13 битов. Эти эвристики основаны на понятии *порядковой независимости* для множеств правил классификации.

Как и в предыдущем разделе, введём две операции на классификаторах пакетов, а именно ограничение (ширины) и разбиение. Пусть B — упорядоченное множество индексов битов, $B \subseteq \{0, \ldots, w-1\}$, которое означает подмножество битов из заголовков пакетов. Для заголовка H (под)заголовок, полученный оставлением только битов H с индексами из B, называется *ограничением* (reduction) H на B и обозначается H^B. Аналогично, для правила R или фильтра

F ограничение правила R (*ограничение фильтра F*) на B обозначается через R^B (соответственно F^B) и представляет собой (под)правило (подфильтр), определённый на битовых позициях с индексами из B. Наконец, для классификатора $\mathcal{K} = (R_1, \ldots, R_N)$ будем обозначать через $\mathcal{K}^B$ (под)классификатор, полученный из $\mathcal{K}$ заменой каждого правила R в классификаторе на R^B. Понятия сопоставления, пересечения и порядковой независимости тоже естественным образом переносятся на такие подпоследовательности.

Заметим, что любое сокращение префиксного классификатора снова является префиксным классификатором. Далее заметим, что сокращённое правило подходит большему числу разных пакетов, чем исходное, потому что некоторые биты фильтров, возможно представляющие интерес, игнорируются; поэтому успешное сопоставление в сокращённом классификаторе не означает сразу же успешного сопоставления в исходном классификаторе. Поэтому срабатывание сокращённого классификатора должно быть дополнительно валидировано посредством проверки на ложные срабатывания, кроме тех случаев, когда удаляются только биты «don't care» (как в случае декомпозиций на основе длин префиксов).

Мы также вводим понятие разбиений классификаторов, т.е. разделения правил классификатора на непересекающиеся группы правил. Рассмотрим классификатор $\mathcal{K} = (R_1, \ldots, R_N)$ и подмножество индексного множества $C \subseteq \{1, \ldots, N\}$. Классификатор $\mathcal{K}_C = (R_i : i \in C)$ называется *группой* (*классификации*) $\mathcal{K}$, определённой множеством C. Для любого разбиения $\mathcal{C} = \{C_1, \ldots, C_k\}$ множества $\{1, \ldots, N\}$, где $\bigcup C_i = \{1, \ldots, N\}$ и $C_i \cap C_j = \emptyset$ при $i \neq j$, *разбиение классификатора*, определённое $\mathcal{C}$, даётся ровно k группами классификации на C: $\mathcal{K}_\mathcal{C} = \{\mathcal{K}_{C_1}, \ldots, \mathcal{K}_{C_k}\}$. Заметим, что любое разбиение префиксного классификатора семантически эквивалентно исходному классификатору в предположении, что среди результатов сопоставлений в каждой группе результат будет выбран в соответствии с длиной префикса.

Интуитивно говоря, разложения на основе длины префиксов имеют очень привлекательное структурное свойство: *для каждого входа каждый подклассификатор содержит не более одного подходящего правила.* Благодаря этому свойству представление каждой группы становится проще, потому что все префиксы в группе можно представить как точные значения; это *сильная* порядковая независимость. К сожалению, разложения на основе длины префиксов

могут в худшем случае содержать слишком много групп для IPv6. Один из интересных вопросов, на которые мы ищем ответ в этом разделе, состоит в том, существуют ли у классификаторов другие структурные свойства, которые позволили бы нам рассматривать префиксы с различной длиной внутри одной группы, сохраняя при этом простоту представления. Мы увидим, что порядково независимые разложения являются естественным обобщением разложения на основе длины префиксов в этом контексте: для данного классификатора $\mathcal{K}$ верно, что $\mathcal{K}$ содержит не более одного подходящего правила для любого заголовка, тогда и только тогда, когда $\mathcal{K}$ порядково независимый. Это специальный вид порядковой независимости, который мы назовём *слабой* порядковой независимостью. Всё вышеизложенное побуждает нас ограничиться разложениями классификаторов, которые гарантируют порядковую независимость (слабую или сильную) внутри каждой группы на подмножестве битовых индексов, которая помогает справиться с балансированием времени и памяти в представлениях IPv6 FIBs.

Задача 3.3.1 (Минимальное слабое (сильное) порядково независимое сведение)**.** *Для данного классификатора $\mathcal{K}$ (не обязательно порядково независимого) и константы l распределить правила $\mathcal{K}$ по минимальному числу непересекающихся групп, в котором разные группы префиксов могут быть основаны на подмножествах из не более чем l битовых индексов, и каждая группа является слабо (сильно) порядково независимой на этих битах.*

Поиск подмножества битовых индексов, которое сохраняет порядковую независимость на этих битах для порядково независимого классификатора (как в слабом, так и в сильном смысле), является вычислительно сложной задачей уже для одной группы; мы покажем это, сведя к этому случаю задачу о минимальном тестовом множестве [108]. Для множества объектов X множество $\tau \subseteq 2^X$ называется *тестовым множеством*, если для каждой пары элементов $x, y \in X$ существует тест $t \in \tau$, для которого $x \in t$, но $y \notin t$ (или $y \in t$ и $x \notin t$).

Задача 3.3.2 (Минимальное тестовое множество)**.** *Для тестового множества τ на X найти тестовое множество $Y \subseteq X$ с минимальным $|Y|$.*

Теорема 3.3.1. *Частный случай задачи о минимальном (слабом или сильном) порядково независимом сведении, в котором все правила из $\mathcal{K}$ состоят только из битов 0 и 1, эквивалентен задаче о минимальном тестовом множестве.*

Теорема 3.3.1 показывает, что и слабая, и сильная формы минимального порядково независимого сведения по крайней мере так же сложны, как задача о минимальном тестовом множестве, так что результаты о сложности и неприближаемости последней переносятся и на первую. Таким образом, мы получили, что для префиксного классификатора с N правилами обе задачи порядково независимого сведения неприближаемы с фактором $(1-\epsilon)\ln N$ для любого $\epsilon > 0$, если $NP \nsubseteq DTIME(N^{\log\log N})$ [66].

Хотя найти точные минимальные сведения вычислительно трудно, мы предлагаем несколько эвристик, которые можно использовать на практике. Все предложенные алгоритмы следуют общей схеме, показанной в алгоритме 3: они последовательно строят группы при помощи некоторой процедуры OneGroup, удаляют новую группу из текущего множества префиксов, а затем снова вызывают процедуру OneGroup, пока не выполнится условие остановки, т.е. либо все префиксы данного классификатора будут уже покрыты, либо число созданных групп достигнет заданного уровня параллелизма β. Другим параметром является бит $exact$, который показывает, пытаемся ли мы строить группы только с точными значениями или группы, в которых допустимы значения $*$.

Основным компонентом наших эвристик является процедура MaxOI, которая для данного множества интервалов $[s_i, f_i]$, $i = 1, \ldots, n$, строит максимальное множество непересекающихся интервалов. Жадный алгоритм 4 (Earliest Deadline First, EDF) для этой задачи является оптимальным и имеет вычислительную сложность $O(N \log N)$ [144].

Начнём с жадных эвристик без максимальной ширины. Алгоритм 1 (т.е. алгоритм 3 с процедурой OneGroup из алгоритма 1) выбирает следующую группу жадным образом при помощи процедуры MaxOI, а затем пытается сократить её ширину (начиная с конца, потому что последние фильтры содержат больше битов $*$, которые скорее хотелось бы сократить). Процедура OI проверяет, является ли её вход порядково независимым (т.е. все ли интервалы на входе не пересекаются).

Главная проблема алгоритма 1 состоит в том, что первая группа обычно очень велика, и в ней порядковая независимость обеспечивается большим подмножеством битов, так что экономия памяти не так значительна. Чтобы справиться с этой проблемой, мы разработали процедуру постобработки Regroup (Алгоритм 2). Она пытается перемещать правила из более ранних в более позд-

Алгоритм 1: OneGroup, жадный, без максимальной ширины.

Data: Классификатор $\mathcal{K}$ ширины k

```
1  begin
2      C ← MaxOI(K)
3      B ← {1, ..., k}
4      for i = k, ..., 1 do
5          if OI(K^C_{B\{i}}) then
6              B ← B \ {i}
7      return B ,C
```

Алгоритм 2: Процедура Regroup.

Data: Классификатор $\mathcal{K}$, группы $\{(B_1, C_1), \ldots, (B_g, C_g)\}$

```
1  begin
2      for i = 1, ..., g − 1 do
3          for F ∈ C_i do
4              for j = g, g − 1, ..., i + 1 do
5                  if OI(K^{C_j ∪ {F}}_{B_j}) then
6                      C_j ← C_j ∪ {F}
7                      C_i ← C_i \ {F}
8      for j = 1, ..., g − 1 do
9          for i ∈ B_i do
10             if OI(K^{C_i}_{B_i \ {j}}) then
11                 B_i ← B_i \ {j}
12     return {(B_1, C_1), ..., (B_g, C_g)}
```

ние группы, не меняя битов, обеспечивающих порядковую независимость в каждой из групп, а затем снова пытается уменьшить ширину каждой группы.

Следующий набор эвристик распределяет префиксы данного классификатора по группам с заданной максимальной шириной l. Первый подход, который мы назовём MinSimilar, состоит в том, чтобы удалить битовые индексы, у которых наиболее похожие значения во всех префиксах. Пусть $\mathbf{0}_i$ (соответственно, $\mathbf{1}_i$) — число префиксов, у которых i-й бит либо 0 (соответственно, 1), либо $*$, и пусть $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$. Алгоритм 5 удаляет битовые индексы с минимальным S_i: он запускает MaxOI на l битах, которые минимизируют S_i. Если нужны точные значения, мы выбираем их из сокращённого классификатора.

Алгоритм 3: MULTIGROUP

Data: Классификатор $\mathcal{K}$, максимальная ширина l, параллелизм β, бит *exact*

```
1 begin
2     i ← 0, T ← K, G ← ∅
3     while T ≠ ∅ и i < β do
4         G_i = K_{C_i}^{B_i} ← OneGroup(T, l, exact)
5         G ← G ∪ {G_i}
6         T ← T \ G_i
7         i ← i + 1
8     return G
```

Алгоритм 4: MAXOI

Data: Множество префиксов $\mathcal{K} = \{F_1, \ldots, F_n\}$

```
1 begin
2     представить K как множество интервалов F_i = [a_i, b_i]
3     упорядочить K по верхней границе b_i, положить k ← 1, S ← ∅
4     for i = 2, ..., n do
5         if a[i] > b[k] then
6             S ← S ∪ {F_i}, k ← i
7     return S
```

Альтернативный подход состоит в том, чтобы выбирать битовые индексы, которые различают как можно больше пар префиксов. Обозначая через D_i число пар префиксов, которые различаются в бите i, мы получаем эвристику MAXPAIR, показанную в алгоритме 6.

В алгоритмах 5 и 6 мы показали два способа достичь сильной порядковой независимости (т.е. отсутствия значений $*$ в фильтрах): либо удалить все префиксы, в которых есть $*$, до запуска MAXOI, либо запустить алгоритм как обычно, а потом расширить значения $*$ с возможным экспоненциальным увеличением классификатора. Это два крайних случая по отношению к нашим двум целям: сокращению ширины и сокращению числа фильтров.

Следующая эвристика представляет собой нечто среднее между этими двумя случаями. Алгоритм 7 показывает эвристику MINSIMILAR$_\delta$, в которой мы удаляем фильтры с не более чем δ разрядами $*$ перед запуском MAXOI; всё остальное происходит в точности как в MINSIMILAR. Алгоритм 8 делает то же для эвристики MAXPAIR.

Алгоритм 5: OneGroup, эвристика MinSimilar.

Data: $\mathcal{K}$, максимальная ширина l, бит *exact*

begin
  for каждого бита $i = 1, \ldots, k$ **do**
    вычислить $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
  выбрать битовые индексы $B = \{b_1, \ldots, b_l\}$ с наименьшим S_i
  $C \leftarrow \mathcal{K}$
  if *exact* **then**
    $C \leftarrow \{F \in C \mid * \notin F^B\}$
  return MaxOI($\mathcal{K}_C^B$)

Алгоритм 6: OneGroup, эвристика MaxPair.

Data: $\mathcal{K}$, максимальная ширина l, бит *exact*

begin
  for каждого бита $i = 1, \ldots, k$ **do**
    вычислить $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
  выбрать битовые индексы $B = \{b_1, \ldots, b_l\}$ с наибольшим D_i
  $C \leftarrow \mathcal{K}$
  if *exact* **then**
    $C \leftarrow \{F \in C \mid * \notin F^B\}$
  return MaxOI($\mathcal{K}_C^B$)

Все упомянутые выше результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [20, 34, 35].

3.4 Сокращение классификаторов пакетов на распределённых платформах

Как уже обсуждалось выше, специалисты признают, что инфраструктура маршрутизации в интернете стоит перед серьёзными проблемами дальнейшего масштабирования [149,163]. На плоскости управления (control plane) эти проблемы пока ещё не стали критическими, но плоскость данных (data plane) уже становится всё менее эффективной из-за быстрого роста размера таблиц пересылки (forwarding tables), которые составляют базы данных пересылки (forwarding

Алгоритм 7: OneGroup, эвристика MinSimilar$_\delta$.

Data: $\mathcal{K}$, l, δ, бит *exact*

begin
- **for** каждого бита $i = 1, \ldots, k$ **do**
  - вычислить $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
- выбрать битовые индексы $B = \{b_1, \ldots, b_l\}$ с наименьшим S_i
- $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leqslant \delta\}$
- `/* *F^B обозначает число разрядов * в F^B */`
- **return** MaxOI($\mathcal{K}_C^B$)

Алгоритм 8: OneGroup, эвристика MaxPair$_\delta$.

Data: $\mathcal{K}$, максимальная ширина l, бит *exact*

begin
- **for** каждого бита $i = 1, \ldots, k$ **do**
  - $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
- выбрать $B = \{b_1, \ldots, b_l\}$ с наибольшим D_i
- $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leqslant \delta\}$
- `/* *F обозначает число разрядов * в F */`
- **return** MaxOI($\mathcal{K}_C^B$)

information base, FIB); это важная открытая проблема в сетевых приложениях [100, 101].

Эффективные представления FIB на плоскости данных становятся ещё важнее с распространением IPv6, потому что большинство методов, предназначенных для эффективного представления IP FIB, плохо масштабируются на IPv6 из-за значительно большей 128-битной ширины адреса [98, 174, 187, 207]. Распределённые системы и подходы, использующие всю сеть (network-wide), привносят дополнительные сложности в представления FIB.

В этом разделе мы рассматриваем задачу представления FIB на распределённых платформах [79, 81], где несколько сетевых карт (line-cards, LC) связаны между собой коммутационной матрицей (см. рис. 3.6b). Каждая входящая (ingress, RX) LC должна отдельно поддерживать FIB таблицу (или её части), чтобы пересылать трафик на правильную исходящую (egress, TX) LC, которая пересылает его дальше через исходящий порт. Фундаментальный вопрос здесь заключается в том, как эффективно реализовать FIB таблицу (рис. 3.6a) на такой распределённой платформе коммутирования.

Существует два основных вида реализаций FIB для таких распределённых

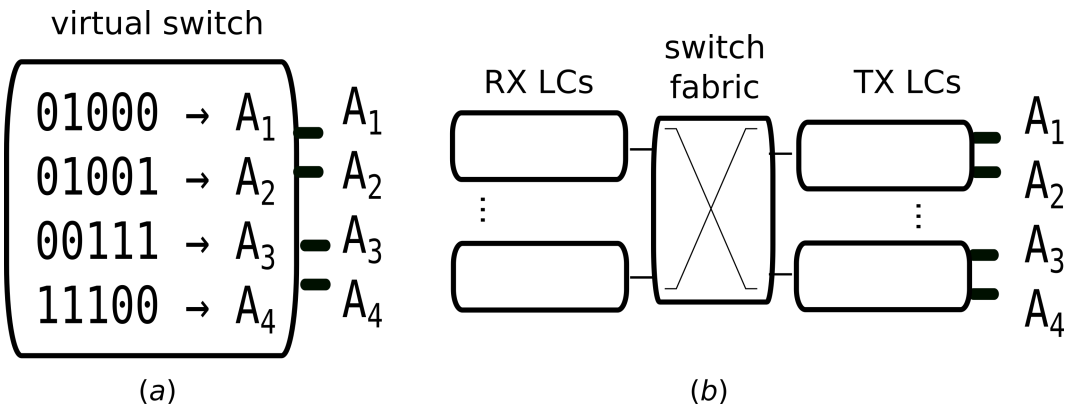


Рис. 3.6: Как представить FIB таблицу (a) на распределённой платформе коммутирования (b).

платформ: *одноэтапная* (one-stage) и *двухэтапная* (two-stage) пересылка. В первом случае указатели на выходные порты находятся уже на входящих LC (рис. 3.7a). Затем исходящая LC инкапсулирует соответствующий трафик указателем, который приходит от входящей LC [79], так что каждой входящей LC нужно хранить информацию о «маршрутизируемых» префиксах всех исходящих LC, в то время как исходящая LC инкапсулирует пакеты на основе указателя на выходной порт, полученного от входящих LC. Важный недостаток этого подхода состоит в том, что поддерживать отображение префиксов на выходные порты для всех исходящих LC на каждой входящей LC — это плохо масштабируемый подход. Но, хотя такие одноэтапные представления не масштабируются, они просты в реализации и не требуют специальной обработки на исходящих LC (пайплайны обработки обычно удваиваются, чтобы изолировать входящий и исходящий трафик); например, различные поколения LC Cisco C12000 реализуют одноэтапную пересылку [79].

В отличие от одноэтапной пересылки, где выходные порты уже вычислены на входящих LC, входящие LC при двухэтапной пересылке должны только выбрать, на какую исходящую LC отправить пакет (см. рис. 3.7b). Далее исходящие LC должны выбрать конкретный выходной порт, запустив дополнительный FIB поиск по всему адресу назначения для каждого пакета. Это позволяет ослабить требования по памяти к входящим LC, и каждая исходящая LC теперь содержит только префиксы, соответствующие её выходным портам. В качестве конкретного примера маршрутизатора можно привести разные поколения LC Cisco CRS-1, которые реализуют двухэтапную пересылку [79].

FIB таблицы можно представлять по-разному. Неформально говоря, два

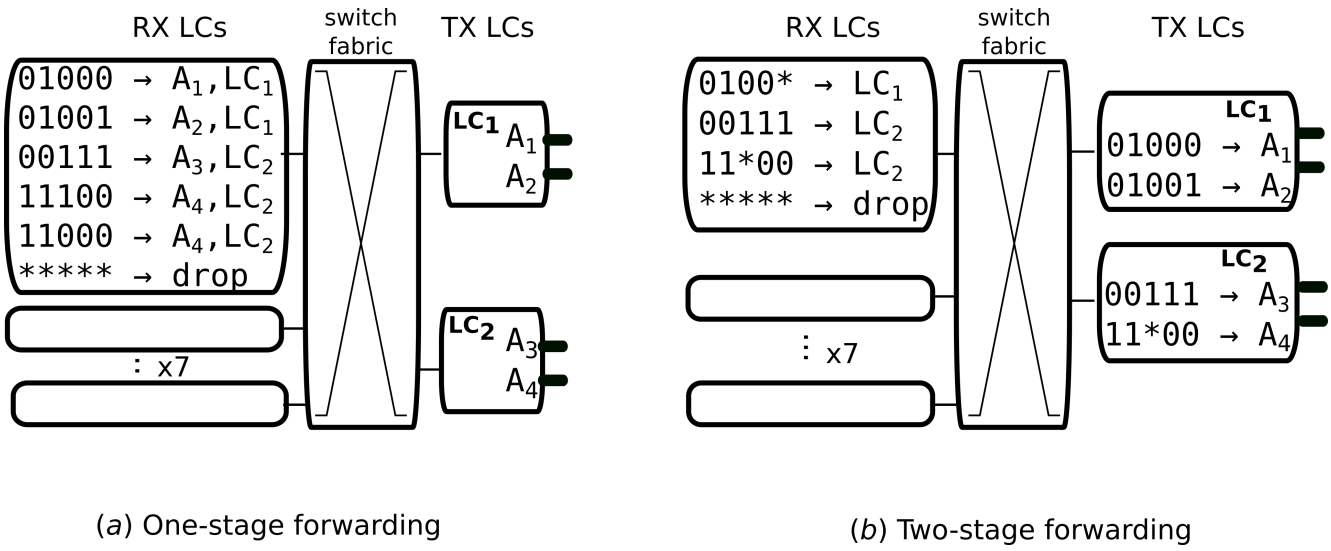


Рис. 3.7: Одно- и двухэтапная пересылка, реализованная на существующих распределённых платформах [80, 81]; поиск делается по всем битам адреса.

представления $\mathcal{K}$ и $\mathcal{K}'$ *эквивалентны*, если для любого поступающего на вход заголовка и $\mathcal{K}$, и $\mathcal{K}'$ выдают одно и то же действие. Различные эквивалентные FIB представления могут оптимизировать разные целевые функции, например требующуюся память или число записей [148, 174].

При одно- и двухэтапной пересылке представления эквивалентны представлениям FIB на виртуальном коммутаторе (см. рис. 3.6a), и они в настоящее время используются при построении реальных распределённых платформ [79, 81]. Заметим, что все биты адресов используются как при одно-, так и при двухэтапной пересылке, что существенно ограничивает эффективность реализаций, особенно на входящих LC.

Основными результатами этого раздела являются новые методы использования структурных свойств FIB для того, чтобы сбалансировать биты, по которым ведётся поиск, между входящими и исходящими LC, минимизируя требующуюся память и время поиска. Мы предлагаем новый *уровень абстракции*, который определяет подмножество индексов битов, которые нужно включить в процесс поиска на входящих и исходящих LC соответственно. В результате классификатор, основанный на этих индексах битов, может быть прозрачным образом представлен другими схемами (см., например, [174, 184]). Наша абстракция, основанная на предлагаемых структурных свойствах, обобщается и далее на случай IPv4, IPv6 или потоковых FIB представлений. Она не требует никаких изменений на плоскости управления всей сетью или в базовой сетевой инфраструктуре: FIB могут задаваться и вычисляться любым существу-

ющим протоколом маршрутизации. В отличие от компактной маршрутизации (compact routing) [149], предлагаемый подход не влияет на решения о маршрутизации, основанные на адресном пространстве, но предлагаемые методы могут достичь существенной экономии памяти без увеличения времени поиска (а на самом деле улучшая его), и они применимы, даже если FIB на входе уже были оптимизированы.

Как побочный продукт мы показываем контринтуитивный результат: IPv6 FIB могут быть реализованы на существующем оборудовании, предназначенном для протокола IPv4 или MPLS (или программной реализации FIB), без увеличения времени поиска или требований к размеру памяти, а также без изменений в самом оборудовании. Мы не ограничиваемся только префиксными классификаторами с приоритетами по длине префикса (longest prefix match, LPM) и рассматриваем общий случай классификаторов. Мы также покажем, как добиться дополнительной экономии с использованием общесетевых (network-wide) представлений.

Начнём с описания модели. Как и в предыдущих разделах, *заголовок пакета* $H = (h_1, \cdots, h_w)$ представляет собой последовательность w битов из пакета и внутренних баз данных сетевого элемента, где каждый бит h_i из H имеет значение ноль или единица, $1 \leqslant i \leqslant w$. Обозначим через W упорядоченное множество из w *индексов* битов в заголовках, то есть $(1, ..., w)$. *Классификатор* $\mathcal{K} = (R_1, ..., R_N)$ — это упорядоченное множество правил, в котором каждое правило R_i состоит из *фильтра* F_i и указателя на соответствующее *действие* A_i. Фильтр F — это упорядоченное множество из w значений, соответствующих битам заголовка. Возможные значения битов — это 0, 1 и $*$ («don't care»). Заголовок H *подходит* (matches) фильтру F, если для каждого бита H соответствующий бит F имеет или то же значение, или $*$. Заголовок H подходит правилу R_i, если он подходит фильтру R_i. На множестве правил определён ациклический порядок $\prec$; если заголовок подходит двум правилам R_i и R_j для $R_i \prec R_j$, применяется действие правила R_i.

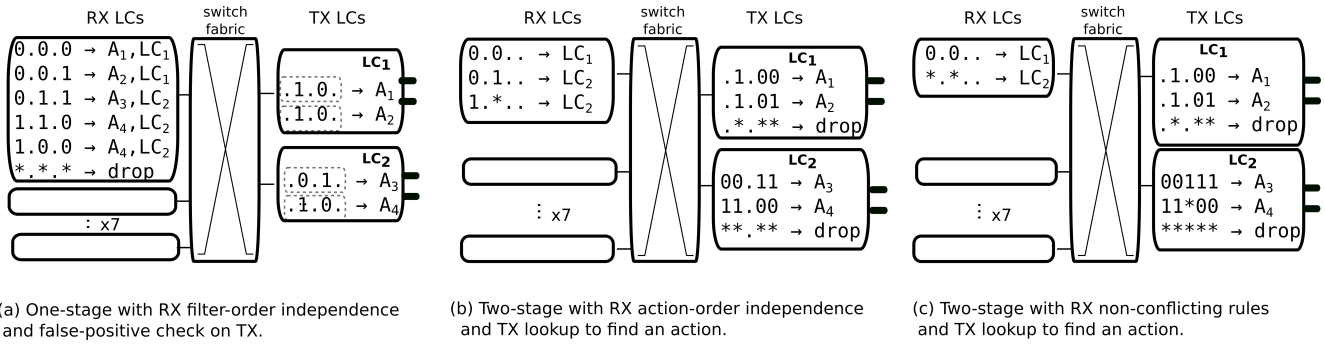
Будем говорить, что два фильтра F и F' одного классификатора *пересекаются*, если существует по крайней мере один заголовок, который удовлетворяет и F, и F'; иначе будем говорить, что F и F' *не пересекаются*. Например, для $w = 4$, $F_1 = (1\ 0\ 0\ *)$, $F_2 = (0\ 1\ *\ *)$ и $F_3 = (1\ *\ *\ *)$, F_1 и F_3 пересекаются (в частности, заголовок (1 0 0 0) подходит обоим фильтрам), а F_1

и F_2 не пересекаются. Классификатор называется *полным*, если ему подходит любой возможный заголовок. Любой неполный классификатор $\mathcal{K}$ можно преобразовать в полный, дописав в конец $\mathcal{K}$ правило по умолчанию («catch-all»), которому подойдут все заголовки, не подошедшие предыдущим правилам.

Пусть B — множество индексов битов, $B \subseteq W$, то есть подмножество битов из заголовков пакетов. Для заголовка H обозначим через H^B (под)заголовок из $|B|$ битов, полученный оставлением только битов из H с индексами из B (в их исходном порядке по W). Например, для $B = \{1, 3, 5, ..., 2\lfloor\frac{w+1}{2}\rfloor - 1\}$ $H^B = (h_1, h_3, h_5, ..., h_{2\lfloor\frac{w+1}{2}\rfloor-1})$. Аналогично, для правила R или фильтра F обозначим через R^B и F^B соответственно (под)правило и (под)фильтр на $|B|$ битах, полученные удалением битов из $W \setminus B$. Наконец, для классификатора $\mathcal{K} = (R_1, ..., R_N)$ обозначим через $\mathcal{K}^B$ (под)классификатор, полученный из $\mathcal{K}$ заменой каждого правила R в классификаторе на R^B. Понятие подходящего заголовка и пересечения фильтров переносятся также на такие подпоследовательности. Аналогично, для заголовка H обозначим через H^{-B} заголовок из $w - |B|$ битов, полученный из H оставлением только битов с индексами из $W \setminus B$. Точно так же определим R^{-B} и F^{-B} как соответственно правило и фильтр, определённые на $w - |B|$ битах удалением условий на биты из B для правила R или его фильтра F. Пусть $\mathcal{K}^{-B}$ — классификатор, полученный заменой каждого правила R в классификаторе $\mathcal{K}$ на R^{-B}. Обозначим множество всех фильтров в $\mathcal{K}$ на основе упорядоченного множества индексов битов $B \subseteq W$ через $\mathcal{F}^B$; $\mathcal{F} = \mathcal{F}^W$. Обозначим множество всех действий в $\mathcal{K}$ через $\mathcal{A}$. Аналогично, обозначим через $\mathcal{F}^i$ множество фильтров на основе битовых индексов B, чьи правила имеют то же действие A_i.

Рассмотрим классификатор $\mathcal{K}$ с функцией классификации $f : \{0,1\}^w \to \mathcal{A}$. Для множества индексов битов $B \subseteq W$ будем говорить, что классификатор $\mathcal{K}^{-B}$ с функцией классификации $g : \{0,1\}^{w-|B|} \to \mathcal{A}$ является *эквивалентным* представлением $\mathcal{K}$, если $f(H) = g(H^{-B})$ для каждого заголовка $H \in \{0,1\}^w$.

Прежде чем формально вводить представления, основанные на предлагаемых структурных свойствах, рассмотрим иллюстративный пример (рис. 3.6a), который служит наглядной мотивацией для наших результатов. Рассмотрим виртуальный коммутатор (рис. 3.6b), представляющий распределённую систему. Основной недостаток одно- и двухэтапной пересылки состоит в том, что все биты адреса должны быть представлены в FIB таблицах. В частности, в



(a) One-stage with RX filter-order independence and false-positive check on TX.

(b) Two-stage with RX action-order independence and TX lookup to find an action.

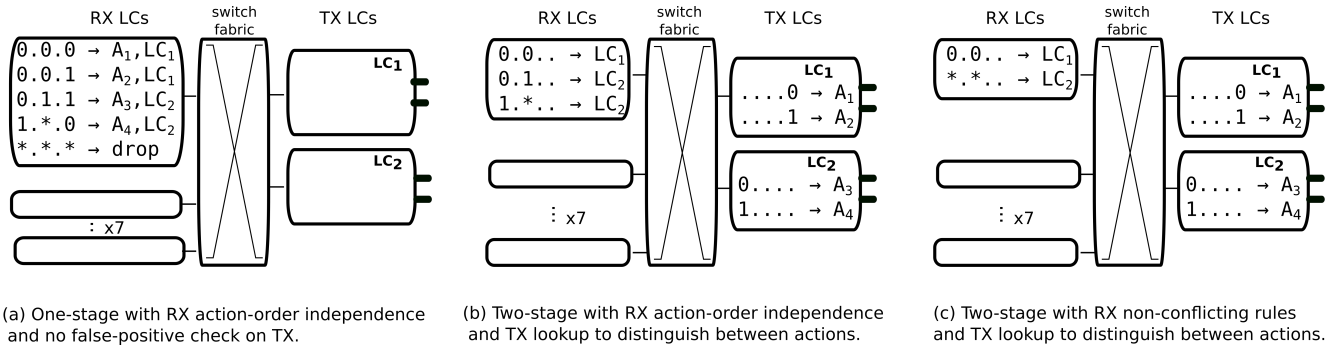
(c) Two-stage with RX non-conflicting rules and TX lookup to find an action.

Рис. 3.8: Эквивалентные FIB представления с сокращёнными битами адресов.

случае одноэтапной пересылки (рис. 3.7a) каждой входящей LC требуется пять 5-битовых записей (не считая «catch-all» правила) без дополнительных FIB таблиц на исходящих картах (рис. 3.7a), так что общий объём памяти, требующийся для восьми карт, составляет $6 \times 5 \times 8 = 240$ битов, и на входящих LC делается один поиск по FIB для каждого пакета.

В случае двухэтапной пересылки (рис. 3.7b в нашем сквозном примере) действия A_1 и A_2 для выходных портов объединяются в одно «действие» LC_1, и действия для выходных портов A_3 и A_4 объединяются в LC_2 на входящих LC. Это позволяет далее сократить FIB записи на входящих LC, объединяя, например, префиксы 01000 и 01001 с одним и тем же действием LC_1 в фильтр 0100∗. Теперь каждая входящая LC содержит 5-битовых записей (не считая «catch-all» правила). Кроме того, каждая исходящая LC_1 и LC_2 требует две 5-битовых записи. Общий объём памяти на всех входящих и исходящих LC в этом примере составляет $4 \times 5 \times 8 + 2 \times 5 \times 2 = 180$ битов.

Заметим, что связующая коммутационная матрица должна поддерживать работу на скорости сети (line rate) для любой комбинации входящей и исходящей LC, и поэтому она не становится «бутылочным горлышком» в данном случае. Кроме того, поиск по FIB на исходящих LC в случае двухэтапной пересылки (или разименование указателя со входящей LC в случае одноэтапной пересылки) обычно представляет собой первый шаг, который выполняют исходящие LC, чтобы определить дальнейшую обработку пакета (см., например, пайплайн-архитектуру обработки пакетов в коммутаторе Cisco C12000 [172]). Пока эти инварианты поддерживаются, мы можем отойти от эквивалентных представлений на уровне LC к эквивалентным представлениям в границах той же распределённой системы, балансируя между вычислением битов представленной таблицы FIB на входящих и исходящих LC. В остальной части этого



(a) One-stage with RX action-order independence and no false-positive check on TX.

(b) Two-stage with RX action-order independence and TX lookup to distinguish between actions.

(c) Two-stage with RX non-conflicting rules and TX lookup to distinguish between actions.

Рис. 3.9: Неэквивалентные FIB представления с сокращёнными битами адресов.

раздела мы формулируем несколько структурных свойств классификаторов и показываем, как их использовать для того, чтобы найти более эффективные представления FIB на распределённых платформах коммутирования.

Когда мы сокращаем ширину классификатора, часто будет получаться классификатор, который не сохраняет семантику (мы не сохраняем *строгую* эквивалентность). Однако удаление того или иного битового индекса из классификатора по сути эквивалентно замене его значения на $*$ во всех фильтрах, так что сокращение ширины не может сократить множество заголовков, покрытых этим фильтром. Значит, для этого ослабления нужно только гарантировать корректное сопоставление заголовков, которые также покрываются исходным классификатором.

Мы вводим новое свойство, *эквивалентность соответствий*, или *матч-эквивалентность* (match equivalence), чтобы улучшить эффективность неполных классификаторов. Это свойство ослабляет функцию классификации, сохраняя только индексы битов, которые сохраняют действие исходного классификатора $\mathcal{K}$ для любого заголовка, который *покрывается* $\mathcal{K}$ (подходит одному из его правил). Формально говоря, неполный классификатор $\mathcal{K}^B$ является *матч-эквивалентным* (ME) классификатору $\mathcal{K}^W$, $B \subseteq W$, если действие для любого подходящего заголовка H^W в $\mathcal{K}^W$ совпадает с действием, которое для соответствующего заголовка H^B выбирает $\mathcal{K}^B$. Если H^W не покрывается $\mathcal{K}^W$, то H^B может либо покрываться, либо не покрываться в $\mathcal{K}^B$. Матч-эквивалентность также можно применить и к полным классификаторам, когда действие «catch-all» правила выбрасывает весь неподходящий трафик и не реализует решения о маршрутизации. В этом случае можно просто использовать свойство матч-эквивалентности на классификаторе $\mathcal{K}$ без «catch-all» правила, а затем добавить его обратно к результату.

Задача 3.4.1 (Минимальный матч-эквивалентный классификатор, MINME)**.** *Для данного классификатора $\mathcal{K}$ найти такой классификатор, матч-эквивалентный $\mathcal{K}$, чьи фильтры занимают минимальную общую память* (*в битах*)*.*

Это определение матч-эквивалентности не даёт напрямую алгоритмов для решения задачи MINME. Далее мы рассматриваем три разных варианта матч-эквивалентности, основанных на разных структурных свойствах.

Сначала рассмотрим семейство матч-эквивалентных классификаторов, которые используют свойство *порядковой независимости по фильтрам* (filter-order independence). Классификаторы, чьи фильтры все попарно не пересекаются (т.е. им не подходит ни один общий заголовок), называются *порядково независимыми по фильтрам*. Интуитивно говоря, представления на основе (только) порядковой независимости по фильтрам не принимают во внимание действия классификатора.

В разделах 3.2 и 3.3 мы уже видели эффект этой характеристики на классификаторы, чьи поля представлены интервалами значений; влияние порядковой независимости по фильтрам на FIB представления там не рассматривалось. Порядковая независимость по фильтрам гарантирует, что подходить данному заголовку будет только один фильтр, и чтобы сохранить это свойство, матч-эквивалентные классификаторы на входящих LC должны сохранять порядковую независимость по фильтрам. Более того, чтобы построить *эквивалентное* решение, нужно проверить на исходящих LC, верно ли, что оставшиеся биты пакета (которые не участвовали в классификации на входящей LC) всё ещё соответствуют принятому пакету (см. рис. 3.8a). Если да, то пересланный пакет посылается через соответствующий выходной порт, а иначе выбрасывается, аналогично тому, как действуют представления на рис. 3.7. Заметим, что в этом случае нет никакой таблицы для классификации на исходящих LC, оставшиеся биты проверяются по ссылке, которая порождается на входящей LC, так что порядковая независимость по фильтрам хорошо подходит для одноэтапной пересылки. Каждый пакет, подходящий входящей FIB (кроме «catch-all» правила) на рис. 3.7a (где показана одноэтапная пересылка) применяет то же действие, что и сжатая входящая FIB таблица на рис. 3.8a. Но теперь пакеты, которые раньше соответствовали только «catch-all» правилу на входящей FIB на рис. 3.7a, могут начать соответствовать сжатой входящей FIB на рис. 3.8a и

будут выброшены только на исходящих LC. Однако это представление всё ещё остаётся эквивалентным, если рассматривать границы всей системы, и позволяет сбалансировать время поиска и требования к размеру памяти между входящими и исходящими LC. В нашем сквозном примере префиксы данного FIB являются порядково независимыми по фильтрам, и FIB на каждой входящей LC может быть сокращён до всего пяти 3-битовых записей (см. рис. 3.8а) (не считая «catch-all» правила), и общая требуемая память в этом представлении составляет $6 \times 3 \times 8 + 2 \times 2 \times 2 = 152$ битов. Важный побочный эффект здесь состоит в том, что такое представление позволяет представить IPv6 таблицы на уже существующем оборудовании для IP или даже MPLS протоколов на входящих LC. Практическая применимость этого подхода была продемонстрирована в экспериментальном исследовании на настоящих IPv6 FIB (см. работы [10, 24]).

Заметим, что порядковая независимость по фильтрам может быть слишком сильным свойством; не обязательно требовать это свойство у правил с одним и тем же действием.

Важный недостаток порядковой независимости по фильтрам состоит в том, что она может помочь сократить число битовых индексов, содержащихся в FIB представлениях на входящих LC, но не может сократить число записей в этих таблицах представлений. Чтобы устранить этот недостаток, мы вводим семейство матч-эквивалентных классификаторов, которые используют свойство *порядковой независимости по действиям* (action-order independence). Эта характеристика основана на наблюдении о том, что результат классификации на самом деле является *действием* (action) правила с наивысшим приоритетом, а не самим фильтром, ассоциированным с этим правилом. Два правила с фильтрами F_1^B и F_2^B называются *порядково независимыми по действиям* (action-order independent), если либо у них одно и то же действие, либо F_1^B и F_2^B не пересекаются; иначе будем говорить, что эти два правила *порядково зависимы по действиям* (action-order dependent). Аналогично, классификатор $\mathcal{K}$ в целом называется порядково независимым по действиям, если каждая пара его правил является порядково независимой по действиям; иначе $\mathcal{K}$ называется порядково зависимым по действиям. Напротив, классификатор $\mathcal{K}'$, который реализует порядковую независимость по действиям на подмножестве $B \subset W$ индексов битов исходного порядково независимого по действиям классификатора $\mathcal{K}$, покрывает те же заголовки, что и $\mathcal{K}$, и в результате $\mathcal{K}'$ является матч-эквивалентным

$\mathcal{K}$ относительно B.

Порядковая независимость по действиям позволяет ещё сильнее сократить число записей для фильтров с одним и тем же действием. В этом случае мы можем сократить как число записей, так и число битовых индексов, участвующих в FIB поиске на входящей LC. Ценой будет немного более сложная ложноположительная проверка и обработка на исходящих LC, для которой, в отличие от случая порядковой независимости по фильтрам, требуется таблица FIB, основанная на подмножестве битовых индексов (см. рис. 3.8b). Заметим, что при использовании двухэтапной пересылки порядковая независимость по действиям позволяет добиться дополнительного сокращения числа записей на входящей LC, потому что число действий сокращается до числа разных исходящих LC. В нашем примере размер памяти для этого вида представления составил $3 \times 2 \times 8 + 3 \times 3 + 3 \times 4 = 69$ битов. Заметим, что в этом конкретном примере на входящих LC не требуется «catch-all» правила, поскольку все комбинации значений уже покрыты.

Поскольку это более общее представление, чем предыдущее, оно может создать ещё более хороший баланс между памятью и временем поиска на входящих и исходящих LC. На практике (см. экспериментальные исследования в работах [10, 24]) достаточно добиться порядковой независимости по действиям на подмножестве индексов битов, чтобы уже найти гораздо более эффективные по памяти представления классификаторов, чем представления, которые используют только порядковую независимость по фильтрам.

При использовании порядковой независимости по действиям реализация матч-эквивалентности имеет огромный потенциал для экономии памяти, и становится очевидно, что нужно более детально понять применимость и выразительность матч-эквивалентности. Для этого мы предлагаем выражать третий тип матч-эквивалентности через понятия *конфликтующих* (conflicting) правил. Два правила R_1^X и R_2^X с разными действиями, $R_1^X \prec R_2^X$, *конфликтуют* относительно битовых индексов $B \subset X \subseteq W$, если существует заголовок H^X, подходящий R_2^X, который не подходит R_1^X, и при этом R_1^B покрывает H^B.

Пример 3.4.1. Рассмотрим следующий классификатор с $|X| = |W| = 3$:

$$\begin{array}{cccccc} & \#1 & \#2 & \#3 & & \\ R_1^X & (* & 1 & 0) & \to & A_1 \\ R_2^X & (1 & 1 & *) & \to & A_2 \end{array}$$

Здесь R_1^X и R_2^X конфликтуют относительно $B = \{1, 2\}$, так как заголовок $H^B = (1\ 1)$ подходит правилу R_1^B, но $H^X = (1\ 1\ 1)$ не подходит R_1^X, в то время как H^B подходит R_1^B, и при этом он подходит R_2^X. С другой стороны, для $B = \{2, 3\}$ правила R_1^X и R_2^X не конфликтуют относительно B.

Классификатор $\mathcal{K}^B$ матч-эквивалентен классификатору $\mathcal{K}^X$ относительно B, $B \subset X$, если никакие два правила $R_1^X \prec R_2^X$ в $\mathcal{K}^X$ с разными действиями не конфликтуют относительно B.

В нашем сквозном примере FIB представление, основанное на неконфликтующих правилах, требует на каждой входящей LC две 2-битные записи (см. рис. 3.8c), и общая требуемая память составляет $2 \times 2 \times 8 + 3 \times 3 + 3 \times 5 = 56$ битов.

Представления, основанные на трёх типах матч-эквивалентности, применимы к FIB таблицам с полями из нескольких полей с общими приоритетами, как, например, в OpenFlow. Оценки сложности, полученные методами вычислительной геометрии, утверждают, что программный классификатор пакетов с N правилами и $k \geqslant 3$ полями использует либо память $O(N^k)$ и время $O(\log N)$, либо память $O(N)$ и время $O(\log^{k-1} N)$ [111, 171]. Поэтому в общем случае новые поля, которые не имеют точных значений во всех правилах, могут привести к существенным дополнительным требованиям либо к размеру памяти, либо ко времени поиска.

Все три типа матч-эквивалентности, которые мы рассматриваем, имеют контринтуитивные следствия. Для эквивалентных представлений на уровне LC увеличение числа битовых индексов в данном классификаторе с фиксированным числом фильтров приведёт к увеличению требований к памяти (например, у протоколов IPv6 или OpenFlow по сравнению с IP). Но это не так для матч-эквивалентности. Более того, матч-эквивалентные FIB таблицы с A действиями, основанные на порядковой независимости по действиям и неконфликтующим правилам, имеют минимальные требования $A \log A$ к размеру памяти, поскольку нам нужен по крайней мере один фильтр для каждого действия, и

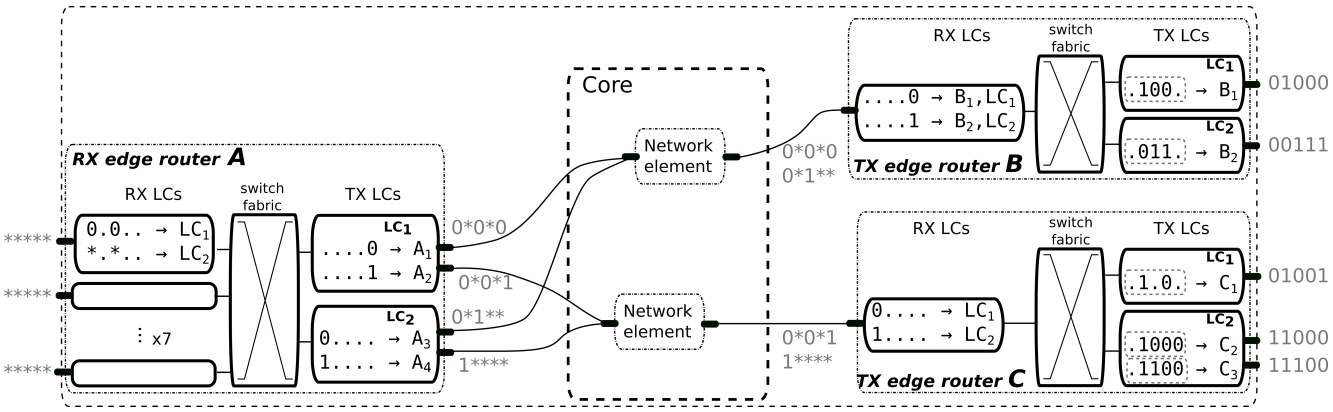


Рис. 3.10: Пример общесетевого представления FIB.

$\log A$ битов требуются, чтобы различить A действий. С использованием матч-эквивалентности у IPv6 или OpenFlow будет больше шансов, чем у IP, чтобы найти индексы битов, различающие между действиями. Это не только существенно сокращает память по сравнению с эквивалентными представлениями на уровне LC, но и позволяет реализовать входящие IPv6 FIB или другие решения (например, OpenFlow) на существующей инфраструктуре IPv4 или даже MPLS с теми же требованиями к размеру памяти и времени поиска, прозрачно для самого оборудования. На рис. 3.8 показаны все предлагаемые представления, которые на системном уровне эквивалентны одно- и двухэтапной пересылке на рис. 3.7, но при этом вносят существенную экономию на входящих и исходящих LC.

До сих пор все предложенные представления работали для заданного (возможно, уже предварительно оптимизированного) FIB и реализовывали эквивалентное представление внутри одного сетевого элемента. В результате, в отличие от компактной маршрутизации, этим схемам не требуется вносить изменения на плоскости управления, и они не влияют на «качество» реализуемого решения о маршрутизации. Далее в этом разделе мы обобщаем предлагаемые FIB представления и обсуждаем понятие *общесетевой* (network-wide) эквивалентности; это может дать ещё большую экономию памяти, но потребует изменений в сигнализации на плоскости управления и возможно на уровне сети. Этот подход может хорошо подойти, например, для программно-конфигурируемых сетей (software-defined networking) с централизованным управлением.

Вернёмся к нашему исходному вопросу: как представлять FIB таблицу на виртуальном коммутаторе, изображённом на рис. 3.6a, в распределённой системе на рис. 3.6b; но теперь будем считать, что распределённая система тоже

виртуальна и может представлять всю сеть. Во многих предыдущих работах вся сеть представлялась как один виртуальный коммутатор; см., например, [49,121]. Очевидно, эквивалентные представления на уровне LC или всей системы применимы во время отображения этого виртуального коммутатора на физические сетевые элементы.

Но мы можем пойти дальше и решить пропустить ложноположительные проверки на исходящих LC граничного маршрутизатора A, в идеале — на исходящих LC соответствующих граничных маршрутизаторов B и C, обращённых к основным интерфейсам (см. рис. 3.10; базовая сеть на рис. 3.10 может работать с IP/IPv6 или MPLS). На рис. 3.9 представлены неэквивалентные решения, позволяющие добиться дополнительной экономии памяти. Как ни удивительно, в случае IPv6 мы можем использовать эти неэквивалентные представления, чтобы уменьшить размер таблиц FIB на входящих и исходящих LC маршрутизатора A, а также, например, представления, основанные на одноэтапной пересылке с независимостью порядка фильтрации на вход маршрутизаторов B и C. Обращаем внимание на то, что в этом случае выходные LC этих маршрутизаторов не нуждаются в выполнении поиска FIB, а только реализуют ложноположительную проверку на основе «оставшихся битов» IPv6-адреса на основе найденного на входящей LC указателя.

Комбинация этих двух представлений позволяет избежать сложных 128-битных представлений на маршрутизаторах A, B и C. Например, семейство LC Engine 3 Cisco C12000 реализует IP FIB как трехуровневое 16-8-8-битное дерево, где каждый уровень требует экспоненциальной памяти [79]. Обычно эти схемы не могут быть напрямую расширены для поддержки IPv6. В результате LC Engine 3 использовали для IPv6 TCAM, который можно было использовать совместно с другими службами, такими как QoS, ACL и т.д. Таким образом, масштабируемость IPv6 была очень ограничена для этого семейства LC. Следующее семейство LC Engine 5 уже имеет четыре модуля TCAM на каждую входящую и исходящую LC в размере 10^6 72-битных записей для поддержки желаемого уровня масштабируемости IPv6, что значительно повышает стоимость и требования к мощности. Общесетевые представления FIB могут значительно снизить эти требования к ресурсам для представлений на граничных маршрутизаторах и позволить развернуть IPv6 в существующей инфраструктуре IP и MPLS. Эти общесетевые представления можно рассматривать как

эквивалентные в границах всей сети, поскольку в худшем случае некоторые пакеты, которые ранее выбрасывались в маршрутизаторе A при использовании эквивалентных решений из рис. 3.7 и рис. 3.8, теперь будут отбрасываться на исходящих LC маршрутизаторов B и C. В отличие от эквивалентных представлений на системном уровне, где коммутационная матрица не является узким местом, здесь мы приходим к принципиальному компромиссу между требованиями к памяти на граничных маршрутизаторах и потенциально «потраченной впустую» пропускной способностью ядра для трафика, который теперь отбрасывается только на выходе граничных маршрутизаторов B и C. Но эти альтернативы всё же стоит рассмотреть. Все предложенные методы, которые используют предложенные структурные свойства, действительны также для общих классификаторов, таких как OpenFlow.

Далее в этом разделе мы предлагаем алгоритмы, которые вычисляют оптимизированные матч-эквивалентные классификаторы для всех трёх рассмотренных выше структурных свойств: порядковой независимости по фильтрам, порядковой независимости по действиям и неконфликтующих правил. Удаление избыточных записей и индексов битов ещё сильнее сократит требующуюся память на входящих и исходящих сетевых картах. Вместо проприетарных алгоритмов мы предлагаем использовать эвристики, основанные на методах булевой минимизации для AC_3^0 схем [50] во всех рассмотренных представлениях FIB (включая традиционные одно- и двухэтапную пересылку). Обозначим эту часть алгоритмов через SE (*strict* equivalence, строгая эквивалентность); SE вычисляет эквивалентное представление с минимальной памятью. К сожалению, некоторые сложные SE алгоритмы имеют вычислительную сложность $O(N^3 \cdot w)$, где N — число правил, а w — их ширина. Мы предложили возможную реализацию SE в разделе 3.2 (OptimizeClassifier). Поэтому в наших экспериментальных исследованиях мы использовали упрощённую версию SE, которая использует только часть известных SE эвристик, но при этом работает за время $O(N^2 \cdot w)$; в предварительных экспериментах на сотнях различных FIB мы не нашли существенных различий между экономией памяти у этой упрощённой и у полной версии SE для всех рассмотренных представлений. Здесь мы используем SE в двух основных случаях:

(1) чтобы удалить избыточные фильтры и битовые индексы;

(2) чтобы проверить потенциальную экономию памяти после удаления какого-

Алгоритм 9: FindBreakers(F,$\mathcal{F}$,B)

```
   Data: F,F,B
1  begin
2      A ← ∅
3      for каждого F′ ∈ F, начиная с F do
           /* два фильтра, которые нарушают структурное свойство,
              могут быть порядково зависимыми по фильтрам или
              действиям, или конфликтовать                        */
4          if F и F′ нарушают структурное свойство относительно
            B then
5              A ← A ∪ {F′}
6  return A
```

либо битового индекса.

Начнём с алгоритма 10 (который использует алгоритм 9), проверяющего, верно ли, что $\mathcal{F}^B$ является MATCH EQUIVALENT (IsME) данному $\mathcal{F}^X$, где B — подмножество индексов битов X. Будем говорить, что два правила *нарушают структурное свойство*, если они порядково зависимы по фильтрам, порядково зависимы по действиям или конфликтуют. Тогда все предложенные алгоритмы могут быть использованы для всех структурных свойств. Заметим, что на самом деле алгоритм 10 проверяет, верно ли, что всевозможные пары правил не нарушают структурное свойство на заданном подмножестве битовых индексов. Алгоритм 10 реализуется при помощи алгоритма 9, и далее в этом разделе мы также будем переиспользовать алгоритмы. В алгоритме 10 мы обозначили

$$\textsc{Succ}(\mathcal{F}, F) = \{F' \in \mathcal{F} \mid F \prec F'\},$$

т.е. множество всех правил $\mathcal{F}$, идущих после F. Все предложенные ниже алгоритмы будут проверять, верно ли, что удаление некоторого бита сохраняет матч-эквивалентность, и будут различаться только тем, как они выбирают следующий бит для удаления.

Алгоритм 11 рассматривает битовые индексы, которые можно удалить, сохраняя матч-эквивалентность. На каждом шаге он удаляет один битовый индекс из всех фильтров; битовый индекс выбирается так, чтобы максимизировать сокращение памяти после его удаления (строки 6-7). Алгоритм продолжается, пока существует битовый индекс, который можно удалить, сохраняя

Алгоритм 10: IsME($\mathcal{F}^X$,B)

```
   Data: F^X, B
1  begin
2      for каждого F_1^X ∈ F^X do
3          if FindBreakers(F_1^X, Succ(F^X, F_1^X), B) ≠ ∅ then
4              return False
5      return True
```

Алгоритм 11: MINME ($\mathcal{K}^X$)

```
   Data: K^X
1  begin
2      B ← X, I^B ← K^X
3      repeat
4          B' ← B
5          for каждого j ∈ B do
6              if IsME(I^{B\{j}}, B \ {j}) then
                   /* удалить j-й битовый индекс, чьё удаление
                      приведёт к наибольшему сокращению размера     */
7                  j* ← arg min_j SE(I^{B\{j}})
8                  B ← B \ {j*}
9      until B = B'
       /* удалить правила, ставшие тривиальными после удаления
          битовых индексов                                          */
10     I^B ← SE(I^B)
11     return B, I^B
```

матч-эквивалентность. В строке 10 SE удаляет правила, которые тривиализуются после сокращения подмножества битовых индексов.

Пример 3.4.2. Рассмотрим следующий классификатор:

#1	#2	#3	#4	
(0	1	0	0)	$\to A_1$
(0	1	1	1)	$\to A_2$
(1	1	1	0)	$\to A_2$

Предложенная эвристика MINME будет рассматривать три бита, которые потенциально могут быть удалены на первом шаге: #1, #2 и #4. Среди этих

битов удаление #2 не приводит к дальнейшему сокращению памяти, в то время как удаление, скажем, #1 приводит к $F_2^{-\{1\}} = (1\ 1\ 1)$ и $F_3^{-\{1\}} = (1\ 1\ 0)$, которые теперь по правилу резолюции можно объединить в $F' = (1\ 1\ *)$; аналогично, удаление #4 приводит к $F' = (*\ 1\ 1)$ на битах 1-3. Таким образом, эта эвристика выбирает либо #1, либо #4, пусть #1, что приводит к следующему матч-эквивалентному классификатору:

$$\begin{array}{cccc} \#2 & \#3 & \#4 & \\ (1 & 0 & 0) & \to A_1 \\ (1 & 1 & *) & \to A_2 \end{array}$$

Теперь уже невозможно ещё сильнее сократить множество правил (у каждого правила теперь своё действие), так что у эвристики не будет дальнейшего предпочтения по поводу того, удалить ли бит #2 или #4; впрочем, удалить можно последовательно оба, и мы получим окончательное представление, основанное только на бите #3.

Теорема 3.4.1. *Для классификатора с N правилами и w-битовыми заголовками вычислительная сложность алгоритма 11 составляет $O(N^2 \cdot w^3) + O(w^2) \cdot T(\mathrm{SE})$, где $T(\mathrm{SE})$ — это вычислительная сложность процедуры* SE.

Можно ещё сильнее улучшить эффективность этого представления и работать с матч-эквивалентными классификаторами, у которых ширину уже нельзя сократить, сохраняя одно из структурных свойств. В частности, мы предлагаем искать матч-эквивалентный классификатор на *подмножестве* правил $\mathcal{I}$ данного классификатора $\mathcal{K}$ и перенести «конфликтующие» правила $\mathcal{I}$ в отдельное множество $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$, так что $\mathcal{I}^B$ матч-эквивалентен $\mathcal{I}$ для некоторого подмножества индексов битов $B \subseteq W$, а $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$ содержит все остальные правила исходного классификатора. Заголовок входящего пакета сопоставляется с обеими группами правил, а затем выбирается правило с наивысшим приоритетом.

Пример 3.4.3. Рассмотрим следующий классификатор:

#1	#2	#3	#4	#5	#6	
(0	1	1	0	0	0)	$\to A_1$
(0	0	0	1	0	0)	$\to A_2$
(1	0	0	0	1	0)	$\to A_3$
(1	1	0	0	0	1)	$\to A_4$
(*	*	0	0	0	0)	$\to A_5$

Правило R_5 не позволяет удалить биты 3-6, потому что оно конфликтует с правилами R_1-R_4 относительно этих битов. Однако если мы положим $\mathcal{D} = \{R_5\}$, оставшийся классификатор $\mathcal{I}^{\{1,2\}}$ становится матч-эквивалентным следующему классификатору на полях $\{1, 2\}$: $(0\,1) \to A_1$, $(0\,0) \to A_2$, $(1\,0) \to A_3$, $(1\,1) \to A_4$.

Заметим, что $\mathcal{D}$ поддерживается традиционным образом, что может быть достаточно дорогостоящим решением. Меняя B, можно найти подходящее представление $\mathcal{I}^B$ и сократить $\mathcal{D}$; применение дополнительного ограничения на максимальное значение $|B|$ может сократить размер $\mathcal{I}^B$. Это дополнительное ограничение может позволить представлять IPv6 классификаторы на существующей IP инфраструктуре. Расщепление исходного классификатора на $\mathcal{I}$ и $\mathcal{D}$ оказывается очень плодотворным для сокращения размера.

Задача 3.4.2 (MINME)**.** *Для данного классификатора $\mathcal{K}$ на множестве битовых индексов W и положительного числа $l \leqslant w$ найти такие подмножество битов B, $B \subseteq W$, $|B| \leqslant l$, и подмножество правил $\mathcal{I} \subseteq \mathcal{K}$, что $\mathcal{I}^B \cup \mathcal{D}$ for $\mathcal{D} = \mathcal{K} \setminus \mathcal{I}$ матч-эквивалентен $\mathcal{K}$ и минимизирует значение $|B| \cdot |\mathcal{I}| + w \cdot |\mathcal{D}|$.*

Алгоритм 12 удаляет не менее чем $w - l$ битов, жадно минимизируя $|\mathcal{D}|$, что эквивалентно минимизации значения $|B| \cdot |\mathcal{I}^B| + w \cdot |\mathcal{D}|$.

Теорема 3.4.2. *Для классификатора с N правилами и w-битовыми заголовками вычислительная сложность алгоритма 12 составляет $O(N^2 \cdot w^3) + O(w^2) \cdot T(\mathrm{SE})$, где $T(\mathrm{SE})$ — вычислительная сложность процедуры* SE.

Можно и далее обобщить идею реализации матч-эквивалентности на подмножестве правил, чтобы ещё улучшить эффективность FIB представления. Мы можем распределить все правила по нескольким непересекающимся группам так, чтобы каждая группа выполняла некоторое структурное свойство на

Алгоритм 12: MINME $(\mathcal{K}^X, l)$

```
   Data: K^X, l
 1 begin
 2     D_0 ← ∅, B ← X, K ← K^X, I ← K, w ← |B|
 3     for i = 1..w do
 4         for каждого j ∈ B do
               /* найти множество D_{i,j} правил, конфликтующих
                  относительно всех битов, кроме j                  */
 5             D_{i,j} ← ∅
 6             for каждого R ∈ I do
 7                 D_{i,j} ← D_{i,j} ∪ FindBreakers(R, Succ(K, R), B \ {j})
           /* выбрать j* с наименьшим числом конфликтуюших правил
              */
 8         j* ← arg min_j |D_{i,j}|
           /* добавить D_ij к D, обновить I_i, B, I и K          */
           /* оптимизировать текущий D_i                         */
 9         D_i ← SE(D_{i-1} ∪ D_{i,j*})
           /* оптимизировать текущий I_i                         */
10         I_i ← SE(K \ D_i)
11         B ← B \ {j}, I ← I^B, K ← K^B
12     i* ← arg min_{i ⩾ w-l} (w|D_i| + (w − i)|I_i|)
13     return i*, D_{i*}, I_{i*}
```

подмножестве индексов битов. Число таких групп может быть определено числом β параллельных и последовательных поисков, которые могут быть выполнены со скоростью сети, что является свойством самого сетевого элемента. Поэтому можно далее обобщить задачу 3.4.2 следующим образом.

Задача 3.4.3 (PARMINME)**.** *Для данного классификатора $\mathcal{K}$ на множестве битовых индексов W и двух положительных чисел $l \leqslant w$ и β распределить правила $\mathcal{K}$ по β непересекающимся группам, где группа $\mathcal{I}_i \subseteq \mathcal{K}$ сохраняет структурное свойство на индексах битов $B_i \subseteq W$, $|B_i| \leqslant l$, и все оставшиеся правила составляют $\mathcal{D} = \mathcal{K} \setminus \cup_{i=1}^{\beta} \mathcal{I}_i$, таким образом, чтобы минимизировать $\sum_{i=1}^{\beta} |B_i| \cdot |\mathcal{I}_i| + w \cdot |\mathcal{D}|$.*

Алгоритм 13 показывает основную идею жадной многогрупповой оптимизации с сохранением матч-эквивалентности; по сути он просто итеративно вызывает алгоритм 12, чтобы построить очередную группу на множестве пока ещё

Алгоритм 13: PARMINME $(\mathcal{K}, l, \beta)$

Data: $\mathcal{K}, l, \beta$

begin
  $\mathcal{D} = \mathcal{K}$, $\mathcal{I}_0 = \emptyset$, $k = 0$
  while $\cup_{s=0}^{k} \mathcal{I}_s \neq \mathcal{K}$ *и* $k \leqslant \beta$ **do**
    $k = k + 1$
    $i_k, \mathcal{D}, \mathcal{I}_k = \text{MINME}(\mathcal{D}, l)$
  return k, $\{(i_s, \mathcal{I}_s)\}_{s=1}^{k}$, $\mathcal{D}$

не распределённых правил без возвращения назад (пока есть неиспользованные группы).

Теорема 3.4.3. *Для классификатора с N правилами и w-битовыми заголовками вычислительная сложность* PARMINME *составляет* $O\left(\beta N^2 \cdot w^3\right) + O(\beta w^2) \cdot T(\text{SE})$, *где* $T(\text{SE})$ — *вычислительная сложность процедуры* SE.

Заметим, что в практической реализации мы можем уменьшить вычислительную сложность предложенного алгоритма двумя разными способами. Во-первых, можно работать на блоках битов. Во-вторых, если ширина классификатора относительно велика (например, в случае IPv6), можно просто выбросить те биты, после отбрасывания которых всё ещё сохраняются нужные структурные свойства, не обязательно добиваясь максимальной экономии памяти каждый раз. Мы применяли эти методы ускорения в наших экспериментальных исследованиях и не нашли существенной разницы между упрощёнными версиями и полными алгоритмами, рассматривавшимися в этом разделе.

Все упомянутые выше результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех приведённых выше теорем, показаны в работах [10, 24].

3.5 Классификаторы, переиспользующие одни и те же классы

В этом разделе мы рассматриваем другие альтернативные способы достичь ещё большей эффективности представления состояния политики классификации на плоскости данных (data plane). Идеи этого раздела основаны на том факте, что похожие «паттерны классификации» (*классы*) переиспользуются в разных политиках; каждый класс состоит из тернарных фильтров, определяющих подмножество подходящих заголовков пакетов. На практике многие вендоры уже поддерживают понятие класса в декларации политики (policy) [82, 173], что позволяет абстрагировать и управлять паттернами классификации более эффективно. Например, Cisco IOS поддерживает до 256 разных QoS политик и до 4096 классов на каждый сетевой элемент [86]. В реальных ситуациях число классов для каждой политики составляет от десятков до сотен, в зависимости от модели приложения [86]. Размер класса зависит от сложности представляемого им паттерна.

В классическом подходе отдельный экземпляр каждого класса аллоцируется для каждого экземпляра политики, содержащей данный класс (см. политики P_1, P_2 и P_3 на рис. 3.11a). Каждый экземпляр класса имеет соответствующее действие, которое назначается соответствующей политикой (например, на рис. 3.11a действие $A_{i,j}$ присвоено экземпляру c_j в политике P_i). Поскольку классы используются в разных политиках, это позволяет рассматривать совмещённые представления политик обслуживания, где каждый класс в идеале появлялся бы только однажды; это могло бы дать существенную экономию в представлениях базовых классификаторов в дорогостоящей памяти, например TCAM. Обычно сложность структурных свойств классификаторов может быть компенсирована дополнительными запусками процедуры поиска (classification lookup), но это общий ресурс для всей процедуры обработки, который нужно экономить. Число классификационных поисков на один пакет является одним из основных факторов, которые ограничивают скорость сети. Например, Cisco C12000 [153] поддерживает не более шести TCAM поисков на каждый пакет со скоростью сети, на все сервисы. Поэтому в этом разделе мы рассматриваем совмещённые представления политик обслуживания, которые не увеличивают число требующихся поисков. Неформально говоря, предлагаемые совмещённые

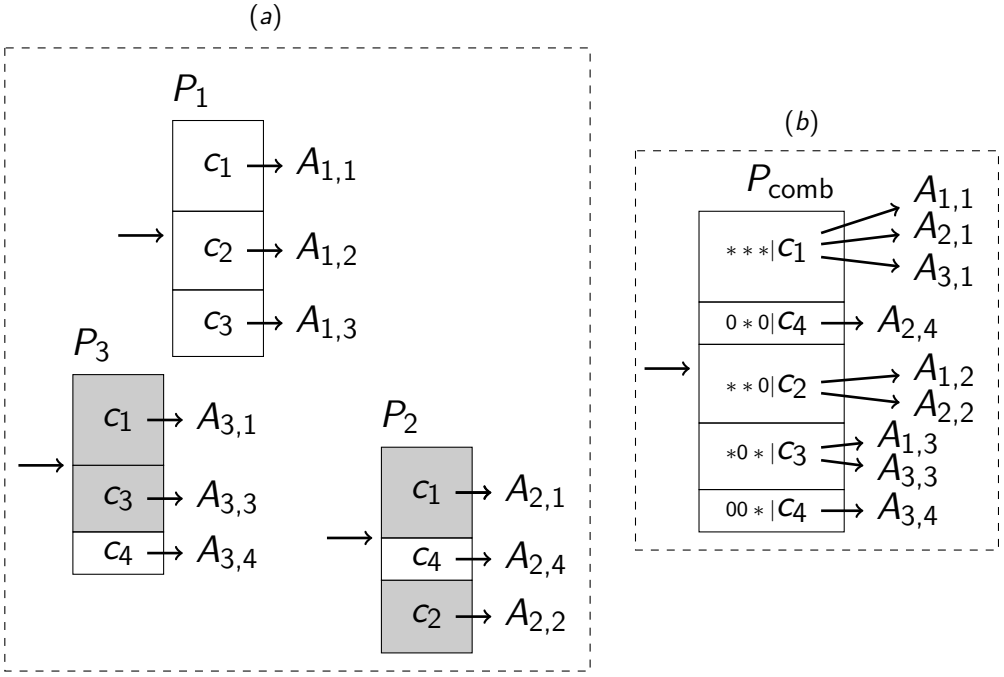


Рис. 3.11: Примеры представлений: (a) отдельные политики P_1, P_2, P_3; (b) представление P_{comb}, которое эмулирует эти политики; экземпляры классов, которые были удалены в P_{comb}, показаны серым.

представления политик обслуживания «эмулируют» поведение этих политик.

Классы представляют собой промежуточный уровень абстракции: *класс* (class) c — это множество фильтров. Обозначим через $w(c)$ число фильтров в классе c. Заголовок H *соответствует* (подходит) классу c, если H подходит хотя бы одному фильтру из c. Два класса (множества фильтров) c и c' *не пересекаются*, что мы будем обозначать через $c \perp c'$, если ни один заголовок не подходит одновременно c и c' (все фильтры c и c' попарно не пересекаются); иначе будем говорить, что они *пересекаются*. Чтобы определить *политику* (policy) P на заданном множестве классов C, нужно выбрать множество классов $C_P \subseteq C$, которые будут принадлежать P, задать последовательность $\mathbb{S}(P)$, в которой каждый класс из C_P встречается ровно один раз, и назначить каждому классу из C_P соответствующее действие. Для каждого входящего заголовка будет возвращено действие первого подходящего класса из $\mathbb{S}(P)$. Если входящий заголовок не подходит ни одному классу из $\mathbb{S}(P)$, то возвращается действие по умолчанию (*default* action). Поскольку классы могут пересекаться, политика определяется не множеством классов, а их последовательностью. Изначально классы были введены для того, чтобы выделить общие паттерны классификации [82, 173], которые могли бы существенно упростить управление политика-

ми. Поэтому вполне вероятно, что один и тот же паттерн классификации будет использоваться сразу в нескольких политиках и не будет переопределяться в каждой из них.

Две политики P_1 и P_2 называются *эквивалентными*, если для любого заголовка они обе выдают одно и то же действие. Заметим, что на одном и том же множестве классов C_P разные последовательности могут определять несколько разных, но эквивалентных политик, потому что некоторые классы в C_P могут попарно не пересекаться. Например, на рис. 3.12a показана политика P, определённая последовательностью $\mathbb{S}(P) = c_3, c_2, c_4, c_1$, которая эквивалентна политике P', определённой последовательностью $\mathbb{S}(P') = c_3, c_2, c_1, c_4$, поскольку c_1 и c_4 не пересекаются.

Чтобы определить политики, эквивалентные P на C_P, мы вводим следующий частичный порядок $\prec_P$ классов в C_P. Будем говорить, что $c_i \prec_P c_j$, если выполнено хотя бы одно из следующих условий:

- c_i пересекается с c_j, c_i встречается перед c_j в $\mathbb{S}(P)$, и действия, соответствующие c_i и c_j в P, различаются (условие непересечения);
- существует такой класс $c_k \in P$, что $c_i \prec_P c_k$ и $c_k \prec_P c_j$ (транзитивность частичного порядка).

Например, рис. 3.12a вводит политику P, чей соответствующий частичный порядок $\prec_P$ проиллюстрирован на рис. 3.12b. На всех рисунках, изображающих частичные порядки, мы опускаем стрелки, показывающие, что $c_i \prec_P c_j$ (c_i предшествует c_j в $\prec_P$), которые следуют по транзитивности.

Заметим, что политика P' эквивалентна политике P, если $\prec_{P'}$ совпадает с $\prec_P$, и каждый класс в P' имеет то же действие, что в P (действия по умолчанию в P и P' тоже должны совпадать).

Неформально говоря, если представить политику P как граф G, чьи вершины соответствуют классам P, а рёбра соответствуют ограничениям частичного порядка в $\prec_P$, то любой топологический порядок на вершинах G образует последовательность классов, которая будет эквивалентна P.

Обозначим через $\mathcal{P} = \{P_1, P_2, \ldots P_{|\mathcal{P}|}\}$ множество из $|\mathcal{P}|$ политик над одним и тем же множеством классов C; через $|C|$ обозначим число классов в C. Заметим, что для входящего заголовка H соответствующая политика P достаётся из внутренних структур данных коммутатора, где и производится классификация

a

		#1	#2	#3	#4	
c_3	F_1	0	1	*	*	A_1
	F_2	*	1	*	0	A_1
c_2	F_3	*	*	0	0	A_2
c_1	F_4	1	0	*	*	A_3
c_4	F_5	0	0	*	*	A_4

b

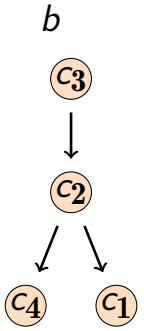


Рис. 3.12: Пример частичного порядка: (a) определение политики $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) частичный порядок $\prec_P$: $c_3 \perp c_1, c_3 \perp c_4$, and $c_1 \perp c_4$.

H; в этом случае будем говорить, что H приходит *в контексте* политики P. *Совмещённая политика* (combined policy) P_{comb}, представляющая группу политик $\mathcal{P}' \subseteq \mathcal{P}$, *эмулирует* $\mathcal{P}'$, если для любого заголовка H и любой политики $P_i \in \mathcal{P}'$ поиск по H в P_i и поиск по H в P_{comb} в контексте P_i дают одно и то же действие. Неформально говоря, это значит, что P_{comb} эмулирует поведение любой политики из $\mathcal{P}'$.

В целом, *представление* $\mathcal{P}_{comb}$, реализующее классификацию в $\mathcal{P}$, может состоять более чем из одной совмещённой политики. Из-за ограничения на один поиск политика из $\mathcal{P}$ должна быть представлена только одной совмещённой политикой в $\mathcal{P}_{comb}$. Поэтому, чтобы построить $\mathcal{P}_{comb}$, политики $\mathcal{P}$ должны быть распределены на несколько непересекающихся групп $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$, где каждая группа $\mathcal{P}^i$ представляется отдельной совмещённой политикой P^i_{comb}. Обозначим через m число групп. Для каждого входящего заголовка в контексте политики $P \in \mathcal{P}$ поиск производится в P^i_{comb}, соответствующей группе $\mathcal{P}^i$, в которую входит P. Будем говорить, что $\mathcal{P}_{comb}$ *эмулирует* $\mathcal{P}$, если каждая $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ эмулирует соответствующую группу политик $\mathcal{P}^i$.

Введём понятие *совместного графа* (joint graph) G^{jnt} для множества политик $\mathcal{P}$ над классами $\mathcal{C}$; это направленный граф $G^{\mathsf{jnt}}(\mathcal{P}) = (\mathcal{C}, E^{\mathsf{jnt}})$, где E^{jnt} содержит ребро от c_i к c_j для $c_i, c_j \in \mathcal{C}$ тогда и только тогда, когда $c_i \prec c_j$ в хотя бы одной политике из $\mathcal{P}$ (пример показан на рис. 3.13b).

Для данного множества политик $\mathcal{P}$ существует идеальная P_{comb}, если соответствующий граф G^{jnt} является ациклическим (этот факт был доказан в [3, 19], но не выносится на защиту). Это подразумевает, что существует алгоритм, который строит идеальное представление, если оно существует. Временная слож-

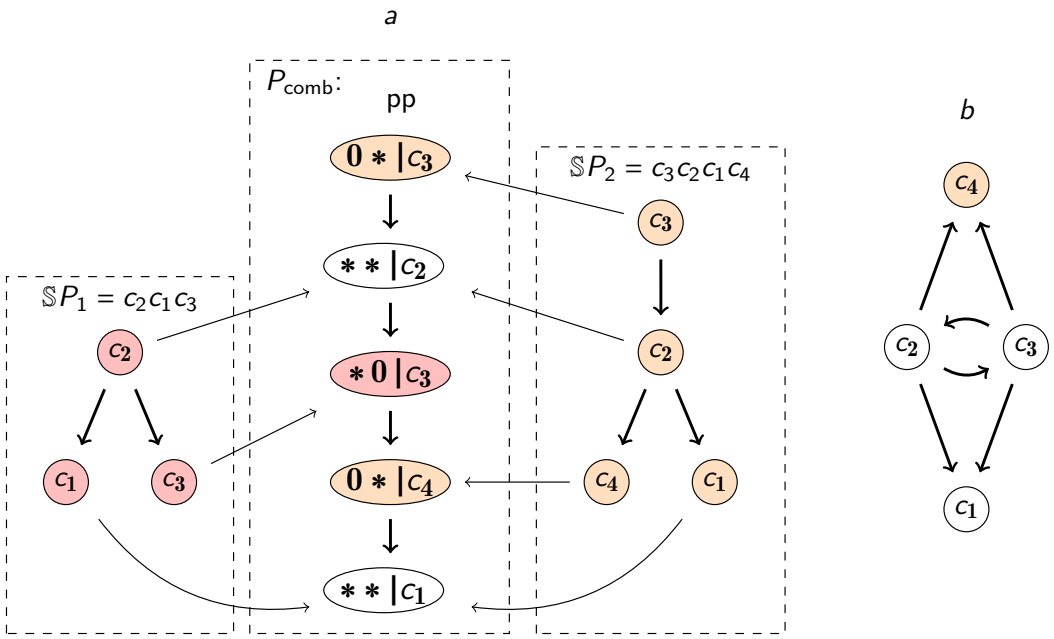


Рис. 3.13: Примеры представлений: (a) $\mathcal{P} = \{P_1, P_2\}$ и P_{comb} с дублирующимся классом c_3; (b) $G^{\text{jnt}}(\mathcal{P})$.

ность этого алгоритма составляет $T_{G^{\text{jnt}}}(\mathcal{P}) = O(\sum_{P \in \mathcal{P}} \left(|P|^2 \ + \ |P| \cdot D(P)\right))$, что требуется для построения G^{jnt}, где $D(P)$ — число пересекающихся пар классов из P, которым соответствуют разные действия.

В случае, когда идеальное представление не может быть построено, приходится дублировать некоторые экземпляры классов, чтобы обработать полученный в политиках частичный порядок. Формально говоря, последовательность $\mathbb{S}$, определяющая порядок экземпляров класса в P_{comb}, является *совместимой* с политикой P_i, если существует подпоследовательность $\mathbb{S}'$ последовательности $\mathbb{S}$, которая состоит из одного экземпляра каждого класса в P_i, и для любых двух классов c_j, c_t из P_i, если $c_j \prec_{P_i} c_t$, то c_j встречается перед c_t в $\mathbb{S}'$. Только экземпляры классов из этой подпоследовательности участвуют в классификации по политике P_i, т.е. в соответствующей P_{comb} только для них i-й бит префикса политики установлен в $*$, а для всех остальных экземпляров i-й бит префикса политики установлен в ноль. Префиксы политик здесь точно такие же, как в случае идеальных политик.

На рис. 3.13a показаны две политики, P_1 и P_2. Поскольку $c_2 \prec_{P_1} c_3$ и $c_3 \prec_{P_2} c_2$, не существует идеальной P_{comb}, удовлетворяющей частичным порядкам классов и в P_1, и в P_2. Рис. 3.13b показывает соответствующий совместный граф, и он действительно содержит цикл. Чтобы выполнить частичные порядки P_1 и P_2 одновременно, можно добавить ещё один экземпляр c_3 к P_{comb} с соот-

ветствующими битами в префиксе политики. В частности, последовательность классов в P_{comb}, показанная на рис. 3.13а, — это $\mathbb{S} = c_3c_2c_3c_4c_1$; её подпоследовательность $c_2c_3c_1$ совместима с частичным порядком $\prec_{P_1}$, а другая подпоследовательность $c_3c_2c_4c_1$ совместима с частичным порядком $\prec_{P_2}$. Теперь P_{comb} содержит два экземпляра c_3: первый используется при классификации в P_2, и его префикс политики — это $0*$; второй экземпляр используется при классификации в P_1, и его префикс политики — это $*0$. В этом случае P_{comb} не является идеальной, но всё ещё эмулирует P_1 и P_2.

Очевидно, дублирование классов должна принимать во внимание число фильтров в этих классах. Обозначим через $\mathrm{W}^+(\mathbb{S})$ общий оверхед (overhead), происходящий из дублированных экземпляров классов в полученной последовательности классов $\mathbb{S}$ и измеренный в фильтрах, т.е. $\mathrm{W}^+(\mathbb{S})$ представляет собой разницу между общим числом фильтров во всех экземплярах классов из $\mathbb{S}$ и общим числом фильтров в исходных классах без дублирования.

Задача 3.5.1 (Policy Sequence Packing, PSP)**.** *Для данного множества политик $\mathcal{P}$ найти последовательность классов $\mathbb{S}$, совместимую со всеми политиками из $\mathcal{P}$, которая минимизирует* $\mathrm{W}^+(\mathbb{S})$.

Теорема 3.5.1. *PSP является NP-трудной даже для двух политик, $|\mathcal{P}| = 2$.*

Обозначим через $|\mathbb{S}|$ число экземпляров классов в $\mathbb{S}$. Если частичные порядки всех политик из $\mathcal{P}$ являются линейными, задача PSP представляет собой взвешенную версию классической задачи о кратчайшей общей надпоследовательности (Shortest Common Supersequence, SCS) [106]. Для данного множества строк SCS находит строку с минимальной общей длиной, которая содержит все эти строки как подпоследовательности. Алгоритм, предложенный в [106], находит оптимальное решение взвешенной версии SCS за время $O(|\mathcal{C}|^{|\mathcal{P}|})$, так что тот же алгоритм может найти оптимальное решение задачи PSP, но только если число политик относительно мало, и частичные порядки всех политик из $\mathcal{P}$ являются линейными.

Наши алгоритмы для PSP будут использовать алгоритмы для взвешенного варианта задачи о разрезающем циклы множестве вершин (Weighted Feedback Vertex Set, WFVS), которая является NP-полной [108]. *Разрезающее циклы множество вершин* (feedback vertex set) — это такое множество вершин в направленном графе $G = (V, E)$ с весами на вершинах, что после их удаления

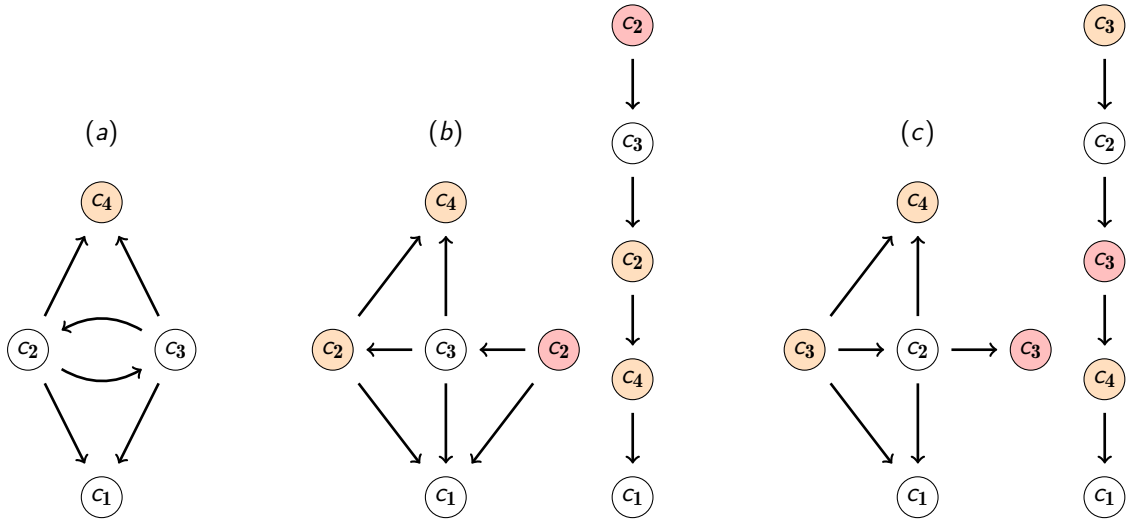


Рис. 3.14: (a) G^{jnt}; (b)-(c) два решения в зависимости от значений $w(c_3)$ и $w(c_2)$.

образуется ациклический граф, и задача WFVS состоит в том, чтобы найти разрезающее циклы множество вершин минимального общего веса. Например, работа [104] предлагает алгоритм для WFVS с аппроксимационным фактором $O(\log|V|\log\log|V|)$, но есть и другие алгоритмы [93]. В дальнейшем обозначим через $\alpha(G)$ аппроксимационный фактор алгоритма для задачи $WFVS$ на графе G. Заметим, что WFVS не сложнее PSP (это доказано в теореме 3.5.1). Поэтому предлагаемые далее алгоритмы используют решение WFVS как свою составную часть.

Основная причина для дублирования классов — это циклы в совместном графе. Алгоритм AO строит G^{jnt} и преобразует его в ациклический граф G^*, топологический порядок на котором даёт корректную последовательность классов $\mathbb{S}$ для P_{comb}.

AO находит разрезающее циклы множество вершин V^{wfvs} в G^{jnt} минимального общего веса, где вес вершины равен числу фильтров в соответствующем классе. Обозначим через $W(V)$ общий вес вершин в V. Подграф, индуцированный на вершинах, которые не входят в V^{wfvs}, является ациклическим, поэтому соответствующие классы появятся в $\mathbb{S}$ только один раз. А для каждого класса $c \in V^{\mathsf{wfvs}}$ последовательность $\mathbb{S}$ будет содержать отдельный экземпляр c для каждой политики, содержащей c.

Чтобы преобразовать G^{jnt} в ациклический граф G^*, алгоритм AO сначала удаляет все классы, содержащиеся в V^{wfvs} (строка 3 алгоритма 14). Затем для каждого класса $c \in V^{\mathsf{wfvs}}$ и каждой политики P_i, содержащей c, вершина $\tilde{c}_i$

Алгоритм 14: AO($\mathcal{P}$)

```
1  begin
2      построить граф G^jnt(P_1, ..., P_l);
3      V^wfvs ← WFVS(G^jnt) с весами вершин w(c) = |c|;
4      инициализировать G* как подграф G^jnt, индуцированный V \ V^wfvs;
5      for каждого c ∈ V^wfvs do
6          for каждой P_i, содержащей c do
7              добавить к G* экземпляр c̃_i класса c;
8      for каждой P_i do
9          for каждого c ≺_{P_i} c', для которого c или c' лежат в V^wfvs do
10             добавить ребро (c̃_i, c̃'_i) к G*;
               /* здесь c̃_i = c, если c ∉ V^wfvs                    */
11     выбрать топологический порядок 𝕊 на вершинах G*;
12     return 𝕊
```

добавляется в граф G^* (строки 5-6); другие вершины из G^* будут связаны с $\tilde{c}_i$ рёбрами, индуцированными частичным порядком на $\prec_{P_i}$ (строки 8-9). Топологический порядок на вершинах G^* (строка 10) образует корректное решение для задачи PSP (см. Теорему 3.5.2). Поскольку G^* можно построить за время не более $T_{G^{\mathsf{jnt}}}(\mathcal{P})$, время работы AO составляет $T_{\mathrm{FVS}}(G^{\mathsf{jnt}}) + T_{G^{\mathsf{jnt}}}(\mathcal{P})$, где $T_{\mathrm{FVS}}(G)$ — время работы алгоритма для задачи WFVS.

Теорема 3.5.2. AO *корректно решает задачу PSP.*

Гораздо более подробный анализ классификаторов, переиспользующих одни и те же классы, в том числе подробное экспериментальное исследование и многие другие результаты на эту тему (которые не выносятся на защиту) содержатся в работах [7, 17].

3.6 Приближённая классификация пакетов

Точные вычисления часто требуют слишком много ресурсов. *Приближённые вычисления* предоставляют потенциально неточные результаты, но помогают обойти ограничения по ресурсам [165]. В этом разделе мы обобщаем классическую *задачу классификации пакетов* (точный случай), где каждое правило состоит из тернарной битовой строки и действия; поскольку правила могут пе-

ресекаться, у них есть приоритеты, и первое *подходящее* правило возвращается как результат классификации для каждого входящего пакета. Существует много исследований, которые используют различные методы оптимизации, чтобы найти *семантически эквивалентные* классификаторы пакетов, для каждого заголовка одно и то же действие выдаётся и в исходном, и в оптимизированном классификаторе [53, 96, 154]; наши исследования на эту тему были предметом предыдущих разделов этой главы. В этом разделе мы рассматриваем случай, когда семантически эквивалентные классификаторы не могут достичь требуемых целей с точки зрения оптимизации, и вводим приближённые представления классификаторов пакетов, которые позволяют «мультиплексировать» несколько действий. Этот дополнительный уровень гибкости позволяет сократить требования к ресурсам, сохраняя требуемый уровень точности.

Эффективность приближённых представлений зависит от двух основных факторов: чёткого понимания допустимых результатов классификации для каждого правила и структурных свойств классификаторов. Первый фактор даёт дополнительную гибкость, которая может улучшить эффективность классификаторов, даже если они несократимы традиционными методами; второй фактор в основном добавляет ограничения. Эффективность приближённых представлений определяется сочетанием этих двух факторов.

Понятие *точных* (exact) и *приближённых* (approximate) классификаторов является центральным для данного раздела. Будем говорить, что классификатор $\mathcal{K}$ является точным, если для каждого правила $R_i \in \mathcal{K}$ соответствующее $\mathcal{A}_i$ состоит из одного элемента; в противном случае $\mathcal{K}$ является *приближённым*. Точные классификаторы — это традиционный способ представлять классификаторы пакетов. Приближённые классификаторы являются обобщением, в котором $\mathcal{A}_i$ представляет собой набор разных допустимых вариантов результата классификации.

Пример 3.6.1. В следующем классификаторе с $w = 4$ заголовок $H = (0\ 1\ 1\ 0)$ подходит R_1, R_4, $R_\perp$, так что H классифицируется правилом R_1:

$\mathcal{K}$	#1	#2	#3	#4	Точный	Приближённый
R_1	0	1	1	0	A_1	A_1, A_4
R_2	1	$*$	0	$*$	A_2	A_2
R_3	1	0	1	$*$	A_3	A_3
R_4	0	$*$	1	$*$	A_4	A_4
R_5	$*$	0	0	$*$	A_5	A_5
$R_\perp$	$*$	$*$	$*$	$*$	D	D

В примере 3.6.1 фильтры классификатора вместе с действиями, показанными в колонках «Точный» и «Приближённый», образуют соответственно точные и приближённые классификаторы. Например, если классификаторы представляют собой таблицы пересылки (forwarding tables), множество действий $\mathcal{A}_1 = \{A_1, A_4\}$ означает, что пакет может быть передан либо через A_1, либо через A_4. Заметим, что в этом случае $\mathcal{A}_1 = \{A_1, A_4\}$ не предоставляет возможностей для балансирования нагрузки, но даёт дополнительные возможности для оптимизации классификаторов. Будем говорить, что классификатор $\mathcal{K}'$ *приближённо эквивалентен* (approximately equivalent) классификатору $\mathcal{K}$, если для каждого заголовка H классифицирующее множество действий $\mathcal{A}' \in \mathcal{K}'$ является подмножеством классифицирующего множества действий $\mathcal{A} \in \mathcal{K}$. В примере 3.6.1 классификатор $\mathcal{K}_e$ является приближённо эквивалентным $\mathcal{K}_a$.

Оптимизационный процесс состоит из множества *базовых операторов* (которые мы ввели выше) и *эвристики*, выбирающей последовательность применяемых операций (*последовательность операций*, *operation sequence*). В этом разделе мы изучаем свойства последовательностей операций, которые позволяют строить эффективные эвристики.

Пусть $\mathcal{S}(\mathcal{K})$ — классификатор, полученный после применения $\mathcal{S}$ на классификаторе $\mathcal{K}$. По определению, каждая применяемая операция удаляет ровно одно правило, так что у $\mathcal{S}(\mathcal{K})$ будет $|\mathcal{K}| - |\mathcal{S}|$ правил, где $|\mathcal{S}|$ — число операции в $\mathcal{S}$ (*длина* $\mathcal{S}$). Поэтому рассматриваемые эвристики могут перейти от минимизации числа правил в оптимизированном классификаторе к построению максимально длинной последовательности операций. Последовательность операций $\mathcal{S}$ *расширяема* (extensible), если $\mathcal{S}$ может быть дополнена ещё одной операцией; в противном случае $\mathcal{S}$ *нерасширяема* (inextensible). Самая длинная последовательность операций всегда нерасширяема, но нерасширяемые после-

довательности могут иметь и не максимальную длину.

Пример 3.6.2. Следующий пример показывает, что разные последовательности операций даже в точном случае могут привести к разным результатам.

$\mathcal{K}$	#1	#2	#3	#4	#5	Действие
R_1	1	1	1	$*$	$*$	A_1
R_2	1	0	$*$	$*$	0	A_1
R_3	1	1	0	$*$	0	A_1
R_4	1	1	1	$*$	0	A_1
$R_\perp$	$*$	$*$	$*$	$*$	$*$	D

Здесь можно применить либо последовательность операций $\mathcal{S}_1 = \mathcal{F}(R_4)$, либо $\mathcal{S}_2 = \mathcal{R}(R_3, R_4)\mathcal{R}(R_2, R)$, где R — правило, полученное в результате $\mathcal{R}(R_3, R_4)$. И $\mathcal{S}_1$, и $\mathcal{S}_2$ нерасширяемы, но только одна из них имеет максимальную длину.

Пример 3.6.2 показывает, что комбинаторный поиск самой длинной последовательности операций — это сложная задача даже в точном случае. Сначала, чтобы сократить сложность оптимизационного процесса, рассмотрим только операции, которые применимы уже на исходном классификаторе $\mathcal{K}$. Будем говорить, что последовательность операций $\mathcal{S}$ называется *начальной*, если $\mathcal{S}$ содержит только операции, применимые на $\mathcal{K}$. Например, в примере 3.6.2 $\mathcal{S}_2 = \mathcal{R}(R_3, R_4)\mathcal{R}(R_2, R)$ не является начальной последовательностью, поскольку вторая резолюция в $\mathcal{S}_2$ опирается на правило, полученное из первой резолюции в $\mathcal{S}_2$. Заметим, что все операции в начальной последовательности $\mathcal{S}$ принимают на вход разные правила; мы называем такие операции *неконфликтующими* (например, $\mathcal{R}(R_3, R_5)$ не может быть в одной начальной последовательности с любым поглощением, применённым к R_3 или R_5). Самая длинная начальная последовательность даёт нижнюю оценку на размер самой длинной последовательности операций. Далее мы рассмотрим задачу поиска максимальной начальной последовательности в точном и приближённом случае; этот анализ показывает комбинаторную сложность минимизации приближённых классификаторов и конструирует строительные блоки для эвристик по поиску максимальных последовательностей операций.

Будем говорить, что последовательность операций $\mathcal{S}$ на $\mathcal{K}$ упорядочена согласно *приоритету правил*, если все операции в $\mathcal{S}$ упорядочены согласно при-

оритетам их базовых правил. Обозначим через $\mathbf{O}$ множество операций, применимых на полученном на вход классификаторе $\mathcal{K}$. Даже неконфликтующие операции в $\mathbf{O}$ могут неявно зависеть друг от друга; к примеру, резолюция может создать новое правило, которое будет пересекаться с другим правилом при обратном поглощении. Сделаем ещё одно дополнительное предположение: если для правила R_i применимы и $\mathcal{F}(R_i)$, и $\mathcal{B}(R_i)$ в исходном классификаторе, то $\mathbf{O}$ содержит только $\mathcal{F}(R_i)$, поскольку не имеет смысла применять обратное поглощение вместо прямого.

Работы [1, 16] содержат подробный анализ последовательностей операций, который не выносится на защиту. Этот анализ показывает, что хотя поиск самой лучшей последовательности операций всегда вычислительно сложен, в точном случае можно относительно легко построить максимальную начальную последовательность. Далее мы обращаемся к приближённому случаю.

В приближённом случае $\mathbf{O}$ может увеличиться по сравнению с точным случаем, поскольку обобщённые операции применимы чаще. Но в отличие от точного, в приближённом случае не всякое подмножество $\mathbf{O}'$ множества $\mathbf{O}$ без конфликтующих операций имеет соответствующую начальную последовательность, состоящую из всех операций из $\mathbf{O}'$ (это один из ключевых результатов работ [1, 19]), и длина максимальной начальной последовательности может быть меньше, чем размер самого большого подмножества $\mathbf{O}$ без конфликтующих операций.

Пример 3.6.3. Неконфликтующие операции из $\mathbf{O}$ не всегда применимы вместе.

$\mathcal{K}$	#1	#2	#3	#4	$\mathcal{A}$
R_1	0	1	1	$*$	A_1
R_2	1	1	$*$	1	A_1, A_2
R_3	$*$	1	1	$*$	A_1, A_2
R_4	0	1	$*$	1	A_2
$R_\perp$	$*$	$*$	$*$	$*$	D

Например, в данном случае можно применить правило резолюции $\mathcal{R}(R_2, R_4)$ или обратное поглощение $\mathcal{B}(R_1)$, но нельзя применить оба, поскольку иначе $\mathcal{A}_2$ стало бы пустым.

Сложность построения самой длинной начальных последовательности возникает из-за обратных поглощений. Для каждого подмножества $\mathbf{O}'$ множества

$\mathbf{O}$, в котором нет конфликтующих операций и обратных поглощений, существует начальная последовательность, состоящая из всех операций из $\mathbf{O}'$ (это один из ключевых результатов работ [1, 19]). Но если в $\mathbf{O}$ содержатся обратные поглощения, то поиск самой длинной начальной последовательности становится вычислительно сложной задачей (её NP-трудность была показана в [1, 19]).

До сих пор мы рассматривали общий случай, в котором каждый фильтр — это тернарная битовая строка. Поскольку минимизация заданного классификатора является вычислительно сложной задачей, мы определяем процесс оптимизации как построение максимальной последовательности из заданного множества операций. Есть, однако, важный частный случай LPM классификаторов (longest prefix match, классификаторы с приоритетами по длине префикса), где каждый фильтр содержит $*$ только после всех 0 и 1. Каждый такой фильтр может быть представлен как префикс над бинарным алфавитом, в котором позиция первого символа $*$ определяет длину префикса. Правила в LPM классификаторе упорядочены согласно их длинам префиксов, причём правило с наивысшим приоритетом имеет самый длинный префикс. Будем предполагать, что LPM классификатор не содержит двух правил с одинаковыми префиксами. В дальнейшем обозначим через $\mathbf{A}$ множество всех действий в данном классификаторе $\mathcal{K}$.

Поскольку фильтры в LPM классификаторах являются префиксами, построение минимального LPM классификатора может быть сделано за полиномиальное время. В точном случае, в работах [97, 191] был введён алгоритм eLP, который вычисляет минимальный LPM классификатор за время $O(|\mathcal{K}| \cdot w \cdot |\mathbf{A}|)$, где w — ширина классификатора в битах. Нам удалось показать, как обобщить eLP на приближённый случай с сохранением оптимальности; обобщённая версия носит название aLP. Ещё одним результатом здесь стало улучшение вычислительной сложности eLP до $O(N \cdot w)$. И алгоритм eLP для точного случая, и алгоритм aLP, который расширяет его на приближённый случай, подробно показаны и проанализированы в работах [1, 19], где также проведён анализ сложности этих алгоритмов.

Теорема 3.6.1. *Временная сложность aLP составляет* $O(|\mathcal{K}| \cdot w \cdot l_{\max})$*, где* $l_{\max}$ *— размер самого большого набора действий в исходном классификаторе* $\mathcal{K}$*.*

Теорема 3.6.2. *Поглощения не могут быть применены к taLPM классификатору $\mathcal{K}_a$ или классификатору $\mathcal{S}(\mathcal{K}_a)$, где $\mathcal{S}$ — последовательность операций, состоящая из резолюций.*

Гораздо более подробный анализ приближённых классификаторов, в том числе подробное экспериментальное исследование и многие другие результаты на эту тему (которые не выносятся на защиту) содержатся в работах [1, 16].

Глава 4 Алгоритмы на плоскости управления: упрощение структуры сети, облачные вычисления и интеллектуальные энергосистемы

4.1 Алгоритмы для упрощения структуры сети

Постоянно возрастающая сложность управления сетью была предметом многих обсуждений и исследований. В целом существуют два основных фактора, которые влияют на сложность сетевых операций: размер и структура управляемого состояния и частота его изменений и обновлений. Предыдущие работы [95,146] предлагают подход, в котором плоскость управления построена так, чтобы использовать контроллеры разных вендоров, применяя реализованные в разных контроллерах сервисы к одному и тому же трафику за счёт использования стандартизованного южного API (southbound API). Это позволяет сетевым операторам запускать сервисы, реализованные на разнородных контроллерах, и система построена так, чтобы работать прозрачно для контроллеров и не требовать дополнительной стандартизации северного API (northbound API). Поэтому важным направлением исследований, которое должно упростить управление сетями, состоит в том, чтобы разработать представления исходной сети более простой виртуальной сетью, которая всё ещё сохраняет релевантные свойства исходной сети.

Уже предпринимались попытки виртуализировать конкретные топологии сетей, оптимизируя при этом различные целевые функции [49,121], но в настоящее время нет способа систематически квантифицировать «эффективность» реализованных представлений сети с точки зрения сложности операций. Если удастся построить более простое представление, которое будет сохранять возможности сети по пропускной способности (bandwidth) и маршрутизации

(routing), то сервисы смогут работать прозрачно и независимо от физической сетевой инфраструктуры. Более того, понимание сетевых ограничений позволяет лучше планировать ресурсы в случае, если требуются дополнительные инвестиции. В этом разделе мы представляем первый шаг в этом направлении.

Первая задача, которая будет рассматриваться в этом разделе, — *планирование загрузки* (capacity planning), под которым подразумевается минимизация общей пропускной способности связей в сети при сохранении её исходной топологии. Поскольку разные сосуществующие приложения могут использовать методы распределения полосы пропускания (bandwidth allocation), оптимизирующие разные целевые функции (например, можно искать кратчайший путь, или самый дешёвый, или балансировать трафик между несколькими путями), мы не предполагаем известным ничего о методах распределения полосы пропускания и маршрутизации для соединения источников и пунктов назначения. Получающийся набор пропускных способностей связей должен допускать любое распределение полосы пропускания и любую маршрутизацию, которую допускал исходный набор; мы называем это свойство *эквивалентностью по полосе пропускания* (bandwidth equivalence).

На рис. 4.1 показан простой пример; все три изображённые там сети *эквивалентны по полосе пропускания*: любой набор маршрутов из s_1 и s_2 к d, который удовлетворяет ограничениям на пропускную способность, показанным на рёбрах одного из графов, также удовлетворяет им для двух других. Однако у этих графов очень разные общие пропускные способности, и результатом планирования загрузки для всех трёх сетей на рис. 4.1 должна быть сеть справа, у которой наименьшая общая пропускная способность и которую нельзя сократить далее. Полезные приложения планирования загрузки включают различные WAN-оптимизации для связанных георасределённых датацентров, где WAN-связи являются очень дорогими [116, 118].

Вторая задача идёт ещё дальше: начиная с некоторой топологии сети, мы разрешаем сжимать рёбра и объединять узлы исходной сети, упрощая её структуру при сохранении возможностей для маршрутизации. Мы сначала определим понятие *эквивалентности по маршрутизации* (routing equivalence) между двумя сетями, которое позволяет отображать совместные наборы машрутов, далее называемые *распределениями полосы пропускания* (bandwidth allocations), между сетями с сохранением полосы пропускания. Упрощённую сеть можно

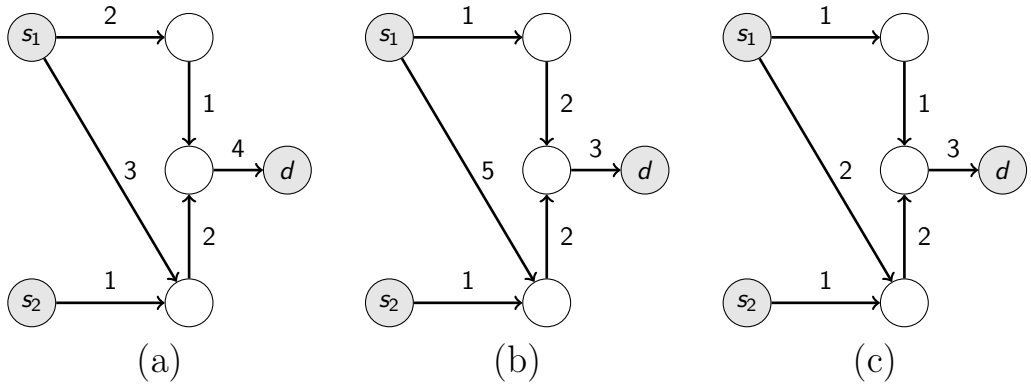
использовать, чтобы найти распределения полосы пропускания, и результат можно будет отобразить обратно на исходную сеть. Поэтому мы вводим преобразования сетей, который упрощают сеть, сохраняя эквивалентность по маршрутизации. Например, рис. 4.2a показывает упрощённый граф G', являющийся эквивалентным по маршрутизации более сложному графу G; рис. 4.2b показывает распределение полосы пропускания от s к d на G' и как оно отображается на G. Тогда можно будет строить простые маршруты на G' и отображать их обратно на G.

К сожалению, эквивалентность по маршрутизации — это сильное свойство, иногда даже слишком сильное. Мы рассмотрим дополнительные преобразования, которые его ослабляют и допускают дальнейшие упрощения: даже если нет биективного отображения между распределениями полосы пропускания, иногда можно гарантировать, что любое распределение полосы пропускания от источников к приёмникам в одной сети также допустимо и в другой. Более того, мы покажем вариант известного парадокса Браесса в этом контексте: удаление ребра из сети может не только не оказать негативное влияние на её пропускную способность, но и позволить сильно уменьшить её размер. В этом разделе мы рассмотрим баланс между дополнительными упрощениями топологии сети и дополнительной пропускной способностью, которая потребуется, чтобы достичь этих упрощений.

В качестве сети мы рассматриваем взвешенный направленный граф $G = (V, E, w, S, D)$ без петель, где V — это множество вершин, E — множество рёбер между ними, а $w : E \to \mathbb{R}^+$ — это весовая функция, которая показывает доступные пропускные способности на рёбрах. Кроме того, некоторые вершины V отмечены как *источники* (sources) $S \subset V$ и *приёмники* (destinations) $D \subset V$. Мы предполагаем, что у вершин-источников нет входящих рёбер, а у вершин-приёмников нет исходящих рёбер (если они есть, мы можем просто заменить внутреннюю вершину-источник s внутренней вершиной v_s и новым источником s', соединённой с этой внутренней вершиной v_s ребром с бесконечной пропускной способностью; аналогично можно поступить и с вершиной-приёмником).

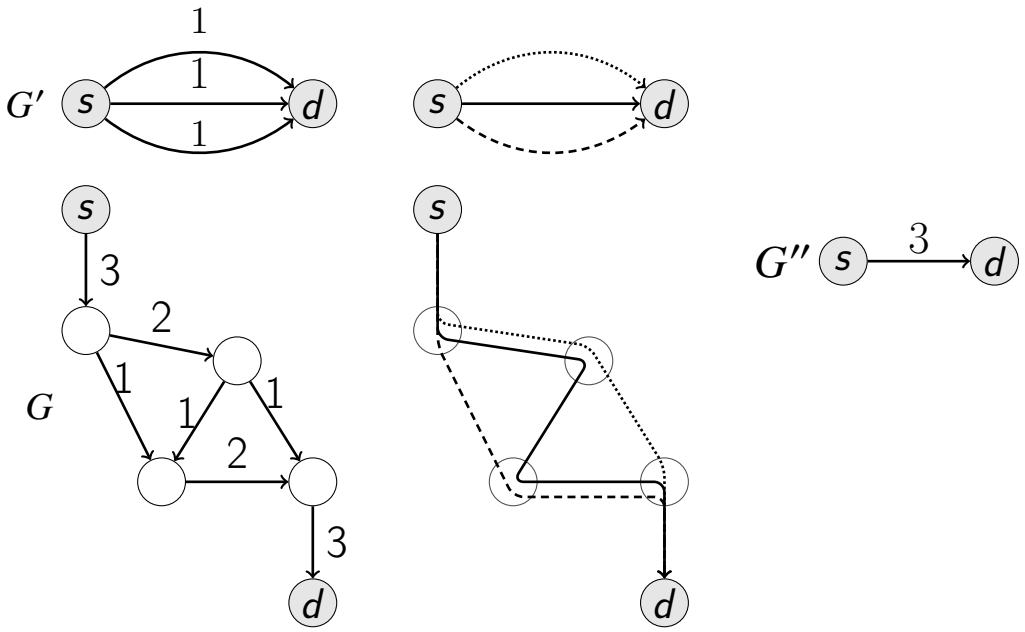
Начнём с понятия распределения полосы пропускания; интуиция здесь в том, чтобы определить множество маршрутов вместе с конкретной полосой пропускания, распределённой на каждый из них.

Определение 4.1.1. *Распределение полосы пропускания* (bandwidth allocation)



(a) (b) (c)

Рис. 4.1: Три сети, эквивалентные по полосе пропускания.



(a) (b) (c)

Рис. 4.2: Упрощения топологии сети: (a) два графа, эквивалентных по маршрутизации, (b) отображение распределения полосы пропускания на G' с тремя маршрутами веса 1 на граф G, (c) любой вес маршрута от s к d в графе G'' также возможен в G' и G.

$A = (P, f)$ на сети $G = (V, E, w, S, D)$ — это множество рёберно простых путей $P = \{p_1, \ldots, p_k\}$ и функция $f : P \to \mathbb{R}^+$, для которых:

(1) для любого $1 \leqslant i \leqslant k$ путь $p_i = (v_1, \ldots, v_{l_i})$ начинается в источнике $v_1 \in S$ и заканчивается в приёмнике $v_{l_i} \in D$;

(2) $f(p_i) \geqslant 0$ для любого i (неотрицательность);

(3) для любого $e \in E$, $\sum_{p \in P : e \in p} f(p) \leqslant w(e)$ (ограничения на пропускную способность).

Заметим, что пути в распределении полосы пропускания могут содержать циклы (но не могут проходить дважды по одному и тому же ребру). Понятие распределения полосы пропускания похоже по сути на разложение потока: мы разбиваем поток на множество взвешенных путей. Разница в том, что в разложении потока циклы не разрешены, и все веса путей положительны.

Определение 4.1.2. Две взвешенные сети $G = (V, E, w, S, D)$ и $G' = (V, E, w', S, D)$ с одной и той же структурой графа, но, возможно, разными весовыми функциями w и w' называются *эквивалентными по пропускной способности* (bandwidth equivalent), что обозначается как $G \simeq G'$, если каждое распределение полосы пропускания A на сети G также является распределением полосы пропускания на сети G', и наоборот.

Определение 4.1.3. Сеть $G^* = (V, E, w^*, S, D)$ является *сетью минимальной пропускной способности* (minimal bandwidth network), если для каждой сети $G = (V, E, w, S, D)$, для которой $G \simeq G^*$, верно, что $w^*(e) \leqslant w(e)$ для любого $e \in E$.

Заметим, что это определение использует конкретную целевую функцию: минимизировать общую пропускную способность всех рёбер $\sum_{e \in E} w(e)$.

Первая задача, которую мы будем рассматривать, — найти сеть минимальной пропускной способности G^* в классе сетей, эквивалентных по пропускной способности данной сети G. Заметим, что если сеть минимальной пропускной способности, эквивалентная данной G, существует и единственна, она очевидно минимизирует общую пропускную способность всех рёбер $\sum_{e \in E} w^*(e)$. Однако не очевидно, что она вообще существует.

К счастью, удаётся построить конструктивное определение, то есть явный алгоритм для построения сети минимальной пропускной способности. Для данной сети G, алгоритм, который находит G^* *грубой силой*, работает следующим образом:

- для каждого подграфа $G' \subseteq G$ с индуцированными пропускными способностями рёбер вычислить максимальный поток $f_{G'}$ от источников S к приёмникам D;
- для каждого ребра $e \in E$ положить его пропускную способность в G^* равной максимуму из всех значений потока, т.е. $w^*(e) = \max_{G' \subseteq G} f_{G'}(e)$.

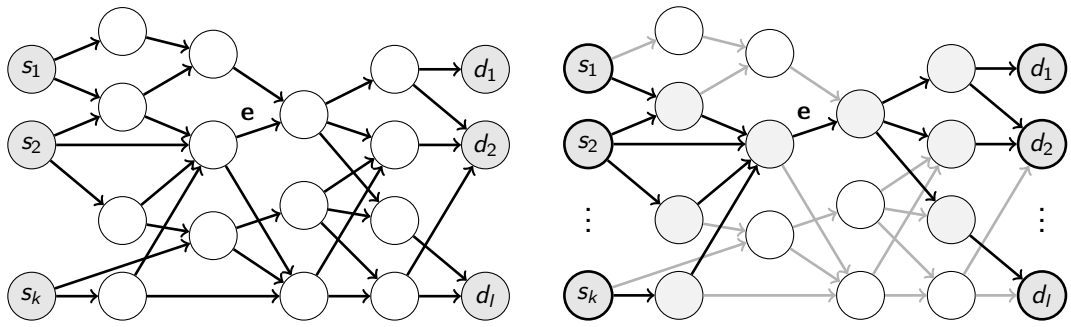


Рис. 4.3: The subgraph G_e used in the DAG-OPT algorithm.

Теорема 4.1.1. *Для каждой сети G алгоритм грубой силы вычисляет единственную сеть минимальной пропускной способности G^*.*

Алгоритм грубой силы очевидно экспоненциален, потому что ему нужно перечислить все подграфы. Далее мы покажем алгоритм, который находит сеть минимальной пропускной способности за полиномиальное время, если в графе нет циклов; поскольку мы ограничиваем вход алгоритма направленными ациклическими графами (directed acyclic graphs, DAG), алгоритм получил название *DAG-OPT*. Чтобы найти сеть минимальной пропускной способности, эквивалентную DAG $G = (V, E, w, S, D)$, DAG-OPT делает следующее:

(1) отсортируем вершины G в топологическом порядке: $u \leqslant v$, если в G есть путь от u до v (здесь и используется то, что G — это DAG);

(2) для каждого ребра $e = (u, v)$:

   (i) рассмотрим подграф G_e графа G, индуцированный подмножеством вершин V_e, которое содержит вершины, сравнимые с u в топологическом порядке:

$$V_e = \{v' \in V \mid v' \leqslant u\} \cup \{v' \in V \mid v' \geqslant v\};$$

   (ii) найдём максимальный поток от источников к приёмникам $f^*_{G_e}$ на G_e;

   (iii) положим пропускную способность $w^*(e)$ равной потоку через e с этом максимальном потоке, $w^*(e) = f^*_{G_e}(e)$.

Алгоритм DAG-OPT проиллюстрирован на рис. 4.3. По сути, чтобы найти оптимальную пропускную способность ребра e, мы удаляем все вершины и

рёбра, которые не связаны с путями через e, а затем решаем задачу о максимальном потоке на получившемся подграфе, что можно сделать за время $O(|V||E|)$ [170].

Теорема 4.1.2. *Алгоритм DAG-OPT выдаёт сеть минимальной пропускной способности G^*, эквивалентную заданной сети (DAG) G, за время $O(|V||E|^2)$.*

Далее мы представляем локальную эвристику, которая хоть и не находит сеть минимальной пропускной способности в общем случае, но даёт хорошие результаты на практике. Мы также покажем, что эта эвристика является оптимальной на графах, чьи ненаправленные версии представляют собой лес, и в отличие от DAG-OPT её можно применить к графам с циклами. Более того, этот алгоритм, который мы будем называть *WPP*, имеет вычислительную сложность $O(|E| + |V| \log |V|)$, в отличие от $O(|V||E|^2)$ у DAG-OPT.

Рассмотрим граф $G = (V, E, w, S, D)$; цель состоит в том, чтобы сократить его общую пропускную способность $\sum_{e \in E} w(e)$. Обозначим общую пропускную способность входящих и исходящих рёбер вершины v через I_v и O_v соответственно; обозначим также $I_e = I_u$ и $O_e = O_v$ для ребра $e = (u, v) \in E$.

Алгоритм основан на локальных изменениях пропускных способностей, которые мы называем *пропагацией весов* (weight propagation, WP). WP основан на простом наблюдении: ребро $e \in E$ не может иметь большую полосу пропускания, чем I_e или O_e. В этом разделе мы предполагаем, что каждый источник s имеет $I_s = \infty$ и каждый приёмник d имеет $O_d = \infty$; вспомним, что у источников нет входящих рёбер, а у приёмников — исходящих.

Чтобы применить WP к ребру $e \in E$, нужно положить $w(e) \leftarrow \min\{w(e), I_e, O_e\}$; это даёт новый граф, который является эквивалентными по пропускной способности графу G. Если $w(e) > \min\{I_e, O_e\}$, то применяя WP к e мы уменьшаем её, а значит, и общую пропускную способность. Алгоритм WPP применяет WP на рёбрах, пока ещё остаются рёбра e, для которых $w(e) > \min\{I_e, O_e\}$.

Чтобы получить оценку сложности, нам нужно оценить число применений WP, что можно сделать правильным выбором рёбер, на которых применять WP в каждый конкретный момент. Метод WPP представлен в Алгоритме 15; он обрабатывает вершины G, выбирая на каждой итерации необработанную вершину v с наименьшим $\min\{I_v, O_v\}$, а затем применяет WP на всех рёбрах, инцидентных v.

Алгоритм 15: Алгоритм WPP.

Data: сеть $G = (V, E, w, S, D)$

begin
  $N \leftarrow V$
  while $N \neq \emptyset$ **do**
    выбрать $v \in N : \min\{I_v, O_v\} = \min_{u \in N}\{I_u, O_u\}$
    Применить WP на всех рёбрах e, инцидентным v
    $N \leftarrow N \setminus \{v\}$

Теорема 4.1.3. *Обработав граф $G = (V, E, w, S, D)$, алгоритм WPP выдаёт граф $G' = (V, E, w', S, D)$, для которого $G' \simeq G$ и для всех $e \in E$ верно, что $w'(e) \leqslant w(e)$ и $w'(e) \leqslant \min\{I'_e, O'_e\}$, за время $O(|E| + |V| \log |V|)$.*

WPP не гарантирует, что получит сеть минимальной пропускной способности в общем случае. Однако WPP достигает отличных результатов на практике и, более того, в важном частном случае леса (когда топология сети представляет собой набор деревьев) WPP является оптимальным.

Теорема 4.1.4. *Пусть $G = (V, E, w, S, D)$ — это граф, чья ненаправленная версия является лесом. Тогда алгоритм WPP, применённый к G, выдаёт сеть минимальной пропускной способности G^*, эквивалентную G.*

Мы показали, как сократить общую пропускную способность сети, сохранив её исходную топологию. Это важно для целого ряда задач (например, для планирования загрузки), но в случае упрощения или виртуализации сети цель другая: упростить управление сетью. Идея преобразований топологии сети, которые мы вводим ниже, состоит в том, чтобы для сети G найти более простую сеть G', которая может выполнить в точности те же запросы на полосу пропускания, что и G. Поскольку G' проще, допустимость запросов на полосу пропускания будет проще проверять. Такие преобразования можно использовать, чтобы упростить управление сетью, представляя сложную сеть как гораздо более простую структуру, которую легче конфигурировать.

Формально говоря, две сети $G = (V, E, w, S, D)$ и $G' = (V', E', w', S, D)$ с одними и теми же множествами источников и приёмников называются *эквивалентными по маршрутизации* (routing equivalent), если существует биективное отображение $g : \mathrm{Path}_G(S, D) \to \mathrm{Path}_{G'}(S, D)$ между путями от источников к приёмникам на G и G' ($\mathrm{Path}_G(A, B)$ обозначает здесь множество путей в графе

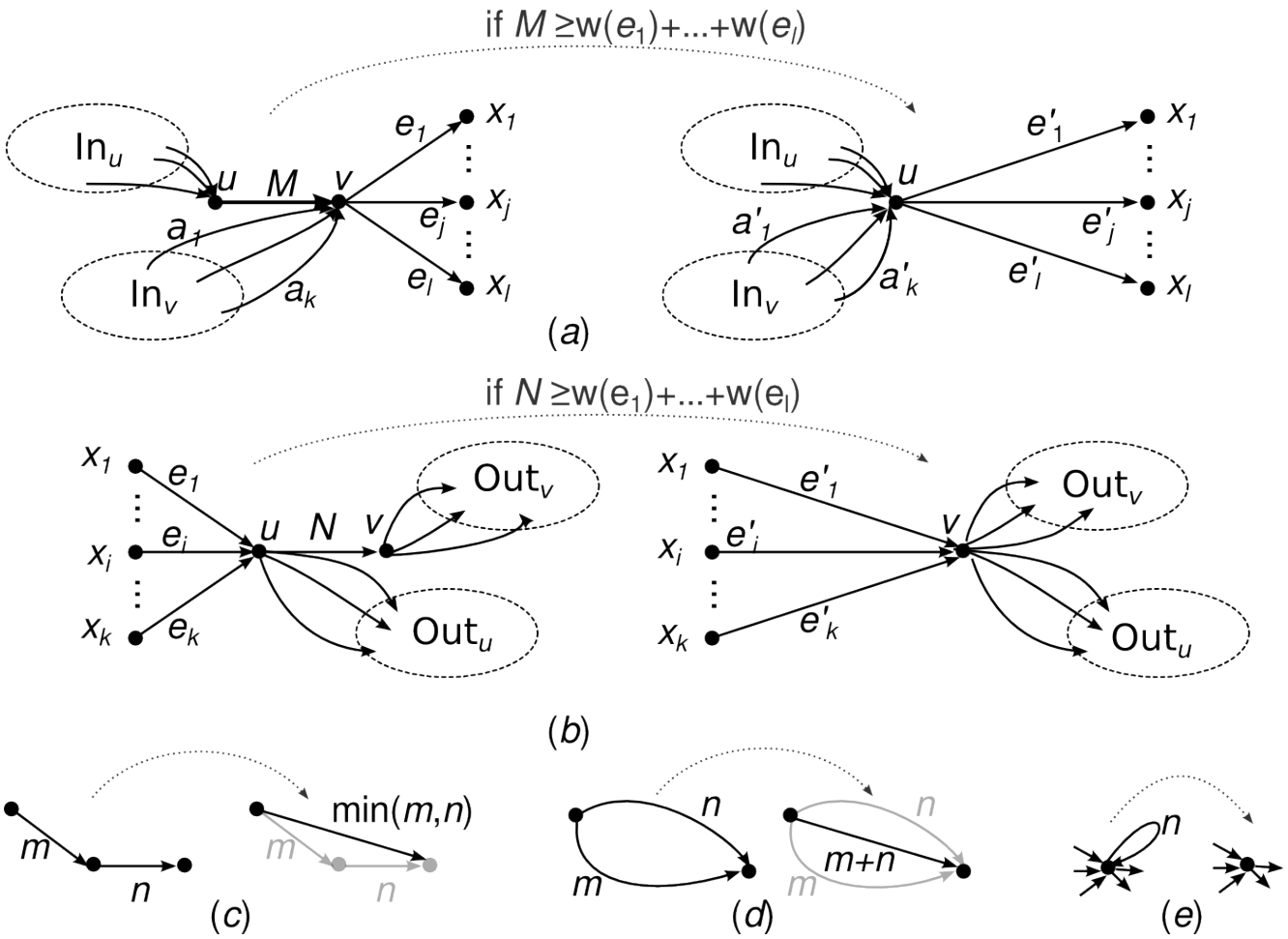


Рис. 4.4: Преобразования топологии сети: (a) стягивание некритического входящего ребра; (b) стягивание некритического исходящего ребра; (c) слияние параллельных рёбер; (d) удаление петель; (e) путь длины 2.

$G = (V, E)$, которые начинаются в $A \subseteq V$ и заканчиваются в $B \subseteq V$), которое сохраняет распределение полосы пропускания, то есть:

(1) для каждого распределения полосы пропускания $A = (\{p_1, \ldots, p_k\}, f)$ на сети G, $A' = (\{g(p_1), \ldots, g(p_k)\}, f')$, где $f'(g(p)) = f(p)$, является распределением полосы пропускания на G';

(2) и наоборот, для каждого распределения полосы пропускания $A' = (\{p'_1, \ldots, p'_k\}, f')$ на сети G', $A = (\{g^{-1}(p'_1), \ldots, g^{-1}(p'_k)\}, f)$, где $f(p) = f'(g(p))$, является распределением полосы пропускания on G.

Наша цель — предложить способы преобразовать сеть G в более простую сеть G', которая была бы эквивалентна по маршрутизации. Процесс трансформации также должен предоставлять функцию g^{-1}, которая отображает любое распределение полосы пропускания с более простой сети G' обратно на G. Пред-

лагаемые преобразования проиллюстрированы на рис. 4.4. Их можно подытожить в следующих двух эвристиках.

1. *Стягивание некритического входящего ребра.* Самая общая форма этой эвристики показана на рис. 4.4a. Рассмотрим ребро $e = (u, v)$, для которого $u \notin S$, $v \notin D$ и $u \neq v$. Если у e пропускная способность $w(e) = M$, и его исходящие рёбра имеют общую пропускную способность $O_v = \sum_{e_j \in Out_v} w(e_j) \leqslant M$, то e не является критическим (не является «бутылочным горлышком») ни для какого распределения полосы пропускания, и его можно сжать в одну вершину u. Это значит, что вершина v и ребро e исчезают, каждое ребро $e_j = (v, x_j) \in Out_v$ заменяется новым ребром $e'_j = (u, x_j)$ с пропускной способностью $w(e'_j) = w(e_j) = n_j$ и каждое ребро $a_i = (y, v) \in In_v$ заменяется новым ребром $a'_i = (y, u)$ с пропускной способностью $w(a'_i) = w(a_i)$. Функция g отображает путь $p = (\ldots, e', e, e_j, \ldots) \in \mathrm{Path}_G(S, D)$ в $g(p) = (\ldots, e', e'_j, \ldots)$. Заметим, что g биективна, поскольку путь $p' = (\ldots, e', e'_j, \ldots) \in \mathrm{Path}_{G'}(S, D)$ отображается либо в $g^{-1}(p') = (\ldots, e', e, e_j, \ldots)$ (если $e' \in In(u)$), либо в $g^{-1}(p') = (\ldots, a_i, e_j, \ldots)$ (если $e' = a'_i$ является одним из новых рёбер, которые заменили In_v). Рис. 4.4a иллюстрирует этот случай, а рис. 4.4c показывает важный частный случай $l = 1$.

2. *Стягивание некритического исходящего ребра.* Эта эвристика симметрична предыдущей, но относительно общей пропускной способности рёбер, входящих в u; она показана на рис. 4.4b. Если $I_u = \sum_{j \in In_u} w(e_j) \leqslant N$, ребро $e = (u, v)$ стягивается в одну вершину v. Функция отображения g строится аналогично предыдущему случаю. Как и выше, рис. 4.4c показывает частный случай этой эвристики для $k = 1$.

Заметим, что преобразования 1 и 2 уменьшают одновременно число рёбер в сети, число вершин в сети и общую пропускную способность всех рёбер.

Теорема 4.1.5. *Сеть $G' = (V', E', w', S, D)$, полученная применением преобразований 1 и 2 к сети $G = (V, E, w, S, D)$, является эквивалентной по маршрутизации сети G.*

Предложенные выше преобразования 1 и 2 производят эквивалентные по маршрутизации более простые графы. Далее мы вводим ослабление свойства эквивалентности по маршрутизации — преобразование, сохраняющее полосу пропускания, — которое позволяет получить ещё более простые графы. Неформально говоря, свойство, гарантированное сохранением полосы пропускания,

состоит в том, что любое допустимое распределение полосы пропускания на полученном графе G' также может быть реализовано и на графе G. Разница в том, что теперь эта реализация не задаётся напрямую преобразованием g, поскольку оно больше не обязано быть биективным. Однако это ослабленное свойство позволяет использовать новые преобразования графов, которые в большинстве случаев дают гораздо более простой граф G'.

Для двух сетей $G = (V, E, w, S, D)$ и $G' = (V', E', w', S, D)$ с одними и теми же множествами источников и приёмников *преобразование, сохраняющее полосу пропускания* (bandwidth-preserving routing transformation), — это отображение $g : \mathrm{Path}_G(S, D) \to \mathrm{Path}_{G'}(S, D)$ между путями от источников к приёмникам на G и G', которое удовлетворяет следующим свойствам:

(1) для каждого распределения полосы пропускания $A = (\{p_1, \ldots, p_k\}, f)$ на сети G $A' = (\{g(p_1), \ldots, g(p_k)\}, f')$, где $f'(g(p)) = \sum_{q \in g^{-1}(g(p))} f(q)$, является распределением полосы пропускания on G';

(2) для каждого распределения полосы пропускания $A' = (\{p'_1, \ldots, p'_k\}, f')$ на сети G' существует функция $f : \cup_{i=1}^{k} g^{-1}(p'_i) \to \mathbb{R}$, для которой $f'(p') = \sum_{p \in g^{-1}(p')} f(p)$ и $A = (\cup_{i=1}^{k} g^{-1}(p'_i), f)$ является распределением полосы пропускания on G.

Добавим следующие два преобразования.

3. *Слияние параллельных рёбер.* Если в сети есть два параллельных ребра с пропускными способностями m и n из вершины u к вершине v, можно заменить (слить) их одним новым ребром от u к v с общей пропускной способностью $m+n$. Это преобразование проиллюстрировано на рис. 4.4d.

4. *Удаление петель.* После некоторых преобразований в графе могут появиться петли; их можно просто удалить (рис. 4.4e).

Рис. 4.5b-g показывает, как последовательные применения вышеописанных преобразований могут радикально упростить граф.

Теорема 4.1.6. *Преобразования 3 и 4 являются преобразованиями, сохраняющим полосу пропускания.*

Для частного случая, в котором исходный граф G представляет собой лес, легко видеть, что граф G', полученный любым преобразованием g, сохраняющим полосу пропускания, на самом деле является эквивалентным по маршрутизации.

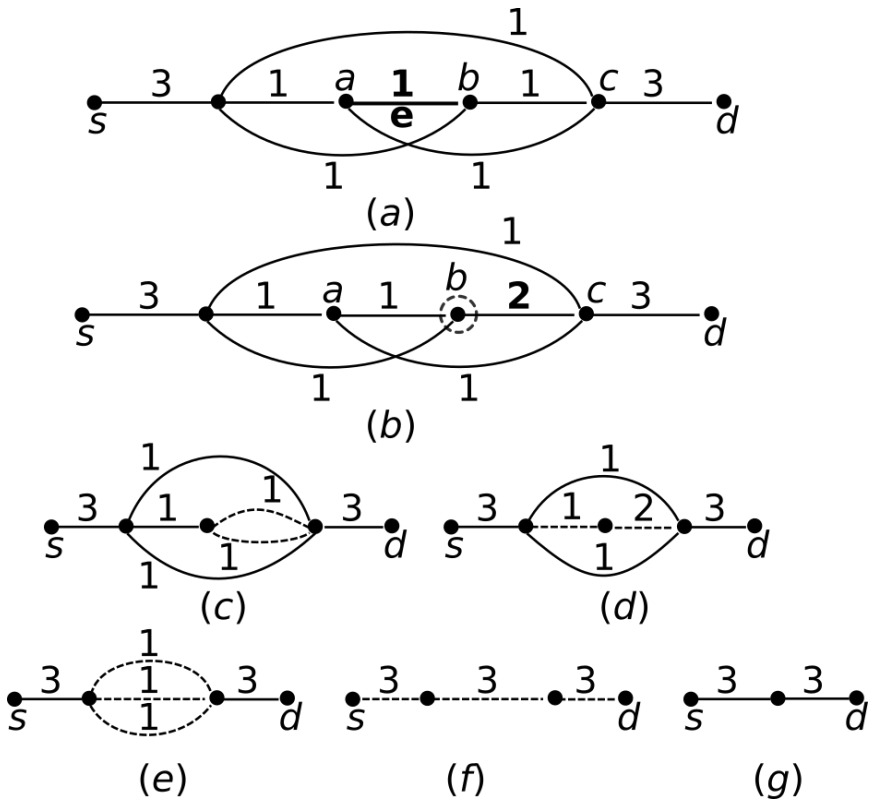


Рис. 4.5: Пример парадокса Браесса и соотношение между общей пропускной способностью и оптимизацией топологии: (a) исходный граф; (b) граф с увеличенной пропускной способностью одного ребра; преобразование 2 применено к ребру (b, c); (c-g) дальнейшие упрощения (пунктирные линии) ведут к полному коллапсу исходного графа.

Теорема 4.1.7. *Преобразование, сохраняющее полосу пропускания, применённое к лесу* $G = (V, E, w, S, D)$*, порождает сеть* $G' = (V', E', w', S, D)$*, которая эквивалентна по маршрутизации сети* G*.*

Рис. 4.5a показывает парадоксальную сеть: она позволяет выделить полосу пропускания 3 от s до d, и в ней нельзя применить ни одно из вышеописанных преобразований для упрощения сети. Однако если удалить одно ребро, а именно центральное ребро $e = (a, b)$ пропускной способности 1, можно запустить цепочку преобразований, которые приведут к тривиальному графу на рис. 4.5g. Заметим, что полоса пропускания, которую можно выделить от s к d в сети, не уменьшится от удаления ребра e на рис. 4.5a, так что это преобразование сохраняет полосу пропускания. Этот феномен можно рассматривать как пример хорошо известного парадокса Браесса [190], и его можно исследовать более детально; например, есть ли эффективный алгоритм, который находит рёбра, которые затрудняют упрощение сети без каких-либо преимуществ?

Начнём опять с сети, которую нельзя упростить преобразованиями, пока-

занными выше. Рассмотрим соотношение между упрощением топологии сети и дополнительной пропускной способностью, которая требуется, чтобы этого упрощения достичь. В качестве примера рассмотрим снова рис. 4.5: исходный граф на рис. 4.5a уже нельзя упростить, но если мы увеличим пропускную способность только одного ребра $e' = (b, c)$ на 1 (рис. 4.5b), граф снова коллапсирует до рис. 4.5g.

Можно естественным образом определить задачи оптимизации, связанные с этим соотношением. Общая задача состоит в том, чтобы найти наилучшее возможное распределение дополнительной пропускной способности с точки зрения упрощения полученного графа. Самый простой возможный граф — это *звезда*, в которой одна внутренняя вершина соединена со всеми источниками входящими рёбрами и со всеми приёмниками исходящими рёбрами (на рис. 4.5g изображён граф-звезда). Задачу можно сформулировать по-разному:

(a) для заданного бюджета пропускной способности C найти такое распределение этого бюджета по рёбрам данного графа, чтобы полученный граф можно было упростить как можно сильнее (с точки зрения числа рёбер, числа вершин или и того, и другого) при помощи сохраняющих полосу пропускания преобразований;

(b) найти минимальную общую пропускную способность, которую можно добавить к данному графу так, чтобы при помощи сохраняющих полосу пропускания преобразований его можно было свести к графу-звезде.

И (a), и (b) представляют собой сложные задачи оптимизации. Для задачи (a) мы предлагаем локальную эвристику, которая позволяет найти разумное значение этого соотношения. Применяя введённые выше преобразования, мы приходим к графу, у которого каждое ребро является критическим (и потому преобразования 1 и 2 нельзя применить). Мы выбираем наилучшее ребро e, чью пропускную способность будем увеличивать, следующим жадным алгоритмом:

(1) для каждого ребра $e \in E$ в графе G:

  (i) найти такую минимальную пропускную способность w_e, что если мы увеличим пропускную способность ребра e до $w(e) + w_e$, e перестанет быть критическим;

(ii) повторять преобразования 1–4 в графе, полученном из G установкой $w(e) \leftarrow w(e) + w_e$, приходя к упрощённому графу G_e, эквивалентному по маршрутизации;

(iii) изменить сложность упрощённого графа $C(G_e)$;

(2) выбрать e, для которого $\frac{C(G)-C(G_e)}{w_e}$ максимально.

В этом алгоритме можно использовать разные графовые меры сложности $C(G)$ в зависимости от целей, например число рёбер, число вершин, сумму этих двух чисел и так далее. Чтобы решить задачу (b), этот процесс нужно повторять, пока не будет получен граф-звезда.

В целом, в этом разделе мы показали, как можно сократить сеть до гораздо более простой сети, на которой можно будет принимать разумные решения по управлению сетью и маршрутизации. Это, однако, лишь первый шаг на пути к более амбициозной цели: представить исходную сеть как маленький виртуальный коммутатор, где можно будет принимать более богатый набор решений (например, планирование), которые затем отображать в операции на физической сети.

Теоретические результаты этого раздела были проверены подробным экспериментальным исследованием на настоящих сетевых топологиях. Наши эксперименты подтверждают, что предлагаемые преобразования применимы на практике. В частности, добавление 10-30% дополнительной пропускной способности к реальным сетям позволяет упростить их до топологии звезды (соединения источников и приёмников через один коммутатор), где все решения о маршрутизации и распределении полосы пропускания можно отобразить обратно на исходную сеть. Это важный шаг к улучшению управления виртуализированной инфраструктурой сети. Результаты этих экспериментальных исследований, а также доказательства всех вышеприведённых теорем, показаны в работе [18].

4.2 Формализация и таксономия задач типа compute-aggregate для облачных вычислений

Современные датацентры хранят данные в разных связанных друг с другом местах. Современные приложения больших данных являются крайне распределёнными, и запросы должны удовлетворять разным и зачастую конфликтую-

щим целям: минимизировать время отклика, добиться ценовой эффективности и многое другое [47,74,150]. Задачи типа *compute-aggregate*, где несколько *блоков данных* (data chunks) должны быть агрегированы в стоке сети, представляют собой важный класс приложений больших данных, реализованных в современных датацентрах. Обычно приложения не могут контролировать, как сетевой транспорт работает с данными. Оптимизация времени отклика должна учитывать свойства сетевых транспортов для того, чтобы избежать, например, incast-проблемы (incast problem) [78, 210], и оптимизация времени отклика для сразу нескольких задач типа compute-aggregate может перегрузить «самые быстрые» (и потому обычно более дорогие) связи. Мы считаем, что нужно более детальное управление, чтобы достичь требуемые целей прозрачно для приложений.

В этом разделе мы рассматриваем постановку задачи в предположении, что каждая задача типа compute-aggregate должна удовлетворять бюджетным ограничениям, поскольку разные пользователи облака могут инвестировать разные экономические ресурсы, чтобы вычислять свои агрегации. Чтобы не перегружать «самые быстрые» связи, они также могут иметь разную цену отправки данных по той или иной связи. Задача теперь делится на две полностью разные фазы:

(1) найти «самый дешёвый» план для данного распределения данных по сети, функции агрегации (которая вычисляет размер, получающийся после агрегации двух разных блоков данных) и цены отправки единицы данных по той или иной связи, а затем

(2) перераспределить агрегации, вычисленные на первой фазе, оптимизируя при этом нужные целевые функции.

В этой постановке задачи мы можем решить первую фазу отдельно, независимо от свойств транспортных протоколов, и уже на второй фазе отдельно решать такие проблемы, как incast. Это естественное обобщение традиционных транспортов, которые созданы для эффективной реализации агрегаций.

Первая фаза представляет особый интерес, поскольку она может представлять разные экономические постановки задачи (например, энергоэффективность) во время агрегации; эта фаза также может привести к более эффективному использованию сетевой инфраструктуры, поскольку цена пересылки единицы данных через связи может различаться для разных задач типа compute-

aggregate. Поэтому наша основная цель в этом разделе — найти универсальные свойства задач типа compute-aggregate, которые позволяют разработать единые принципы дизайна «идеальных» агрегаций на первой фазе. Использование свойств функций агрегации для принятия решений требует новых результатов на уровне модели и может привести к более эффективным агрегациям. Место для увеличения эффективности точно есть: средний размер результата задачи оценивался как 40.3% от начального размера данных Google [91], 8.2% в Yahoo и 5.4% в Facebook [77].

Наша основная цель — научиться использовать сеть наилучшим возможным образом для данной задачи типа compute-aggregate. Это задача со многими переменными. В этом разделе мы оставляем большинство из них для уровня сетевого транспорта (например, на этом уровне будет выбираться, как распределить передачу данных во времени) и концентрируемся на *плане агрегации* (aggregation plan), который определяет порядок агрегации блоков данных. План агрегации полностью отделён от реализации транспорта и будет формализован ниже.

Будем моделировать сеть как ненаправленный связный граф $G = (V, E)$, где V — это множество вычислительных узлов, соединённых связями (рёбрами) E. Заметим, что поскольку мы работаем на уровне приложений, мы можем использовать любую виртуальную топологию вместо G, которая будет содержать только информацию, важную для конкретной задачи типа compute-aggregate. Задача представляется как множество начальных блоков данных $C = \{\boxed{x_0}, \boxed{x_1}, \ldots, \boxed{x_k}\}$, где каждый блок $\boxed{x_i}$ характеризуется своим размещением в сети $\mathrm{v}(\boxed{x_i})$ и размером $\mathrm{size}(\boxed{x_i})$. Поскольку многие задачи типа compute-aggregate требуют, чтобы результат был полностью доступен на конкретном узле (например, чтобы обеспечить отклик с низкой задержкой), мы предполагаем, что есть выделенная корневая вершина $t \in V$, где все блоки данных должны быть в итоге агрегированы.

Самая сложная часть формализации — определить, что значит «наилучшим образом»: целевые функции могут зависеть от конкретного приложения и могут включать в себя задержку отклика, пропускную способность или более сложную цель, например избегание перегрузок. Будем моделировать все эти цели единым параметром на каждом ребре, *стоимостью* (cost), который представляет собой достаточно гибкий способ свободно объединять разные целевые

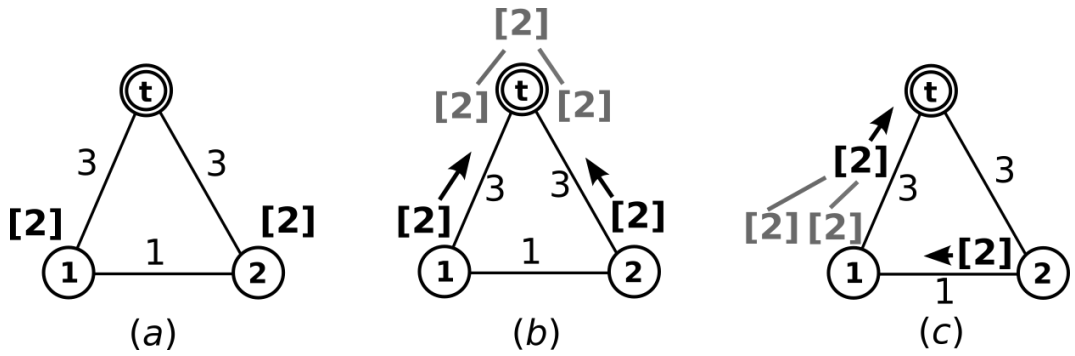


Рис. 4.6: Пример задачи типа compute-aggregate с тремя вершинами t (target), u и v: (a) задача; (b) план типа «переместить в корень» с ценой 12; (c) оптимальный план агрегации с ценой 8. Цены пересылки показаны на каждом ребре рядом с его серединой (например, $c(u, v) = 1$).

функции и поддерживать чёткую цель для задачи оптимизации. Формально говоря, функция стоимости $c : E \to \mathbb{R}_+$ на графе топологии сети G отображает каждое ребро e в цену пересылки одной единицы данных $c(e)$; чтобы передать $\boxed{x}$ через e, нужно заплатить цену $c(e) \cdot \text{size}(\boxed{x})$. Если нужна балансировка нагрузки, её можно достичь выбором маленьких значений c для разных подмножеств рёбер для разных задач.

Простой пример задачи типа compute-aggregate показан на рис. 4.6a. Цены пересылки показаны на рёбрах, в квадратных скобках показаны блоки данных, а корневая вершина обозначена через t.

Чтобы определить план агрегации простым, но достаточно выразительным способом, важно понимать влияние, которое он может оказывать на остальную часть системы (например, на транспортном слое или в вычислительной инфраструктуре). Начнём с простейшей формы плана агрегации, который будем называть «переместить в корень» («move to root»): переместить все блоки данных в корневой узел t (для простоты будем говорить, что план агрегации *перемещает* или *агрегирует*; на практике перенос и агрегация данных реализуются на транспортном и прикладном уровне соответственно). План «переместить в корень» может быть субоптимальным по отношению к цене пересылки. Предположим, что в примере на рис. 4.6 функция агрегации выбирает наилучший блок данных, так что размер результата агрегации не превышает максимума из размеров исходных блоков. Тогда план «переместить в корень» имеет общую цену 12 (рис. 4.6b: два блока размера 2 каждый двигаются по рёбрам цены 3), а на рис. 4.6c один блок двигается к вершине 1 за цену 2, затем блоки агрегируются, и блок размера 2 двигается к t с общей ценой 8.

Но есть и другие проблемы, кроме цены пересылки. Наивная реализация плана «переместить в корень», который двигает все блоки данных в корень и затем агрегирует их, приводит к тому, что транспортному слою нужно перенаправить к t очень большой объём трафика, что может переполнить входящих буферы в этом узле и увеличить задержку вследствие проблемы TCP-инкаста. Более того, в приложении с низкой задержкой, которое агрегирует данные в RAM [72,209], может закончиться доступное место, если все блоки данных нужно будет хранить в t.

Эти проблемы можно решить при помощи промежуточных агрегаций. Блоки данных можно посылать к t последовательно в некотором порядке, а затем немедленно агрегировать прибывающие блоки. Недавние исследования [77, 91] показывают, что окончательный результат задачи типа compute-aggregate часто представляет собой лишь малую часть (обычно меньше половины) общего размера начальных блоков данных; например, в задачах подсчёта результат агрегации — это обычно всего лишь несколько чисел. Поэтому если хранить в памяти промежуточный блок вместо всего множества начальных блоков данных, можно существенно сократить требования к памяти.

В общем случае не каждый порядок можно использовать для промежуточных агрегаций, потому что окончательный результат агрегации может зависеть от этого порядка (например, при конкатенации строковых полей), а план агрегации не должен влиять на результат [205]. К счастью, большинство функций агрегации не зависят от порядка агрегации, то есть являются *ассоциативными*:

$$\mathbf{aggr}(\boxed{x}, \mathbf{aggr}(\boxed{y}, \boxed{z})) = \mathbf{aggr}(\mathbf{aggr}(\boxed{x}, \boxed{y}), \boxed{z}),$$

и *коммутативными*:

$$\mathbf{aggr}(\boxed{x}, \boxed{y}) = \mathbf{aggr}(\boxed{y}, \boxed{x}).$$

Ниже мы предполагаем, что агрегации ассоциативны и коммутативны; такие системы, как *MapReduce*, уже предполагают это для большинства *reduce*-функций и разрешают агрегации промежуточных блоков данных функциями-комбайнерами [91].

Базовый принцип здесь состоит в *оптимизации локальности данных*, который также лежит в основе фреймворка *Hadoop* [202]; это значит, что мы

пытаемся *передвигать вычисления к данным* и в результате экономить на передаче данных. Мы расширяем эту стратегию и пытаемся *переместить агрегации к данным*, разрешая плану агрегации использовать промежуточные узлы. Формально говоря, план агрегации — это последовательность P операций $(o_0, o_1, \ldots, o_m)$, где каждая o_i — это одна из следующих операций:

- либо **move**($\boxed{x}$, v), то есть передвинуть блок данных x в вершину v,
- либо **aggr**($\boxed{x}$, $\boxed{y}$), то есть слить $\boxed{x}$ и $\boxed{y}$, расположенные в одной вершине; в результате в этой вершине появляется новый блок данных $\boxed{xy}$.

Когда все операции применены, в результате должен получиться один блок данных $\boxed{z}$ в корневой вершине: $\mathrm{v}(\boxed{z}) = t$. Например, рис. 4.6b и рис. 4.6c показывают план агрегации для задачи, показанной на рис. 4.6a. Планы агрегации полностью отделены от транспортного слоя, они производят инструкции и ограничения, которые транспортный слой должен будет выполнить.

При таком определении плана агрегации легко назначить ему цену пересылки и поставить задачу минимизации. Во-первых, у каждой операции o_i есть цена, которую мы обозначим как $\mathrm{cost}(o_i)$ и которая определяется следующим образом:

- $\mathrm{cost}(\textbf{aggr}(\boxed{x}, \boxed{y})) = 0$ (данные не передаются);
- $\mathrm{cost}(\textbf{move}(\boxed{x}, v)) = \mathrm{size}(\boxed{x}) \cdot d(\mathrm{v}(\boxed{x}), v)$, где $d(u, v)$ — это общая цена самого дешёвого пути из u к v (если таких путей несколько, можно использовать любой).

Общая цена плана агрегации P, $\mathrm{cost}(P)$, — это сумма цен всех операций в P.

Чтобы формально определить задачу оптимизации для агрегации данных, нужно знать следующее: каким будет для двух блоков данных $\boxed{x}$ и $\boxed{y}$ размер результата их агрегации $\boxed{xy}$? Это напрямую влияет на цену плана агрегации, и разные размеры результата агрегации могут привести к очень разным решениям. Например, если на рис. 4.6 мы бы предположили, что задача состоит, скажем, в сортировке данных, где размер агрегированного блока является суммой размеров входов, цена первого плана осталась бы равной 12, но план на рис. 4.6c теперь стоил бы 14 и стал бы субоптимальным.

К сожалению, размер результата агрегации зависит от конкретного приложения, и в большинстве случаев точное его значение зависит от содержания

данных в $\boxed{x}$ и $\boxed{y}$; более того, чтобы определить его значение, часто нужно сначала выполнить собственно агрегацию (например, число пар ключ–значение в задаче подсчёта слов нельзя предсказать точно, если, собственно, не провести сам подсчёт). Это, конечно, не реализуемо на практике, потому что план агрегации должен быть составлен (и его стоимость должна быть оценена) до того, как приложение начинает выполнять сами агрегации и до того, как транспортный слой начнёт передавать какие-либо даные.

Поэтому мы требуем, чтобы каждое приложение предоставляло свою *функцию агрегации* (aggregation function) $\mu : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$, которая предоставляла бы оценку этого размера, используя только размеры входов, так что для целей оптимизации $\text{size}(\boxed{xy}) = \mu(\text{size}(\boxed{x}), \text{size}(\boxed{y}))$. Мы не ожидаем, что эти функции будут давать идеально точный результат, но они должны предоставлять верный порядок величины, чтобы оптимальное решение оказалось хорошим и на практике. Поскольку **aggr** предполагается ассоциативной и коммутативной, каждая функция агрегации μ тоже должна обладать этими свойствами. Вот несколько примеров μ для практически важных задач:

- $\mu(a, b) = \text{const}$ для задачи поиска лучших k элементов в данных по отношению к некоторому критерию;
- $\mu(a, b) = \min(a, b)$ или $\mu(a, b) = \max(a, b)$ для выбора наилучшего блока данных;
- $\mu(a, b) = a + b$ для конкатенации или сортировки;
- $\max(a, b) \leqslant \mu(a, b) \leqslant a+b$ для объединения множеств (например, для задачи подсчёта слов).

Рис. 4.7 показывает, как выбор μ может повлиять на оптимальный план агрегации. На рис. 4.7a показаны блоки размера 1 в вершине 1, размера 4 в вершине 2 и размера 6 в вершине 3, и цель состоит в том, чтобы агрегировать их в вершине 0. Для $\mu(a, b) = a+b$ оптимальный план состоит в том, чтобы передвинуть каждый блок в корень по отдельности (рис. 4.7b). Для $\mu(a, b) = \max(a, b)$ будет дешевле сначала передвинуть блок размера 4 вдоль ребра $2 \to 3$ и там объединить блоки, а результат размера 6 переместить в корень (рис. 4.7c). Наконец, для $\mu(a, b) = \min(a, b)$ оптимальный план состоит в том, чтобы пройти по всему графу самым маленьким блоком, сливая его с более крупными по

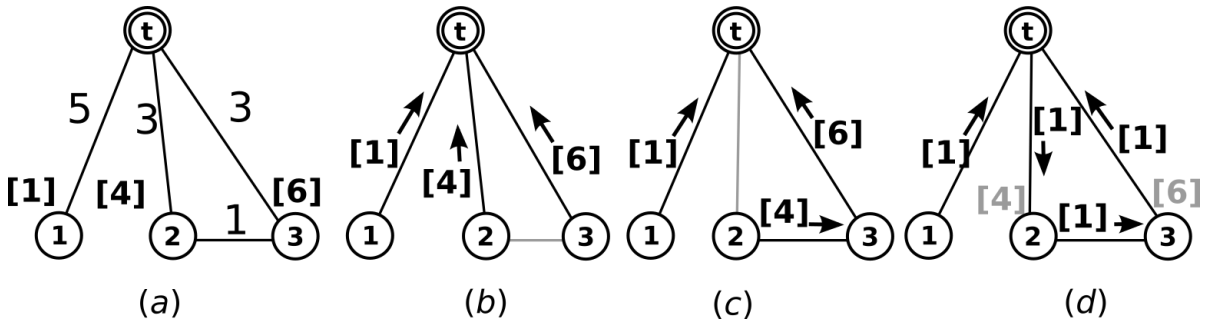


Рис. 4.7: Разные μ приводят к разным планам агрегации: (a) пример задачи; (b) оптимальный план для $\mu(a,b) = a + b$; (c) оптимальный план для $\mu(a,b) = \max(a,b)$; (d) оптимальный план для $\mu(a,b) = \min(a,b)$.

дороге (рис. 4.7d). Таким образом, даже в этом простом примере план агрегации может радикально измениться в зависимости от μ. В итоге мы приходим к следующей задаче оптимизации.

Задача 4.2.1 (CAM — compute-aggregate minimization)**.** *Для заданных ненаправленного связного графа $G = (V, E)$, функции стоимости c, корневой вершины t, множества начальных блоков данных C и функции агрегации μ, задача* CAM[μ] *состоит в том, чтобы найти план агрегации P, для которого минимизируется* cost(P).

Интересно, что если функция агрегации не обладает нужными свойствами, но при этом есть ограничения на структуру графа, в худшем случае всё равно мало что можно сделать.

Теорема 4.2.1. *Если $P \neq NP$, то не существует полиномиального алгоритма с константным аппроксимационным фактором для задачи* CAM[μ] *без условия ассоциативности на μ, даже если ограничить G двумя вершинами.*

Существуют разные типы приложений больших данных, сильно различающиеся топологии сетей в датацентрах и бесконечно разнообразные распределения данных, и все они в совокупности определяют ограничения для задачи типа compute-aggregate. Вряд ли возможно искать оптимальные алгоритмы отдельно для каждого возможного варианта таких ограничений, так что требуется обобщённый метод для построения плана агрегации. В этом разделе мы представляем такой метод и показываем ряд гарантий в худшем случае для различных частных случаев.

Неформально говоря, более сильные ограничения могут приводить к лучшим решениям, как с точки зрения *стоимости плана агрегации*, которая является нашей основной целью, так и с точки зрения *производительности* (времени работы). Например, если граф сети — это дерево, то может быть возможно построить план агрегации за линейное время. Аналогично, иногда более хороший алгоритм (с точки зрения получаемой стоимости) можно найти при дополнительных ограничениях на функцию агрегации. Поэтому возможное решение может зависеть от *топологии сети*, *функции агрегации* и *начального распределения блоков*. Распределение блоков меняется для каждого конкретного примера и вряд ли может дать полезную информацию, поскольку его логично ожидать приблизительно равномерным (большие системы хранения данных стараются достичь равного распределения загрузки). Хотя есть популярные топологии сетей, например *hypercube*, *fat-tree* или *jellyfish*, у них есть много вариантов и исключений.

Мы в основном будем рассматривать функции агрегации, а для структуры сети ограничимся двумя важными частными случаями. Во-первых, дерево — это топология, которая и широко распространена, и имеет большой потенциал для более эффективных алгоритмических решений; будем называть вариант задачи CAM, в котором $\boldsymbol{G}$ является деревом, TCAM (tree CAM). Во-вторых, иногда разумно ограничить агрегацию только теми узлами, которые изначально содержат блоки данных. Это может случиться либо по причинам безопасности, либо потому, что на промежуточных узлах нужно выделять дополнительные ресурсы, что сильно увеличит задержку, а узлы с начальными блоками обычно уже имеют выделенные вычислительные ресурсы с фазы предобработки. Если по этим или другим причинам такое ограничение нужно ввести, то граф сети $\boldsymbol{G}$ можно свести к полному графу на вершинах, содержащих блоки данных, и мы называем этот частный случай CCAM (complete CAM).

Мы свели теоретические результаты, представленные в работах [4, 21, 30], в таблице 4.1. Ячейки таблицы показывают аппроксимационные факторы для разных случаев функции агрегации и $\boldsymbol{\mu}$ и разных постановок задачи (CAM, TCAM и CCAM) вместе со ссылками на конкретные теоремы и следствия, в которых эти результаты доказаны ниже. Строки таблицы 4.1 соответствуют разным частным случаям $\boldsymbol{\mu}$, от медленно до быстро растущих функций, а столбцы соответствуют трём разным постановкам задачи. В таблице $\boldsymbol{\alpha}$ обозна-

	CAM	TCAM	CCAM
Произвольная μ (общий случай)	$\alpha\frac{W_C[\mu]}{w_C[\mu]}$ *[4, теорема 3]*	$\frac{W_C[\mu]}{w_C[\mu]}$ *теорема 4.2.4*	$\frac{W_C[\mu]}{w_C[\mu]}$ *теорема 4.2.4*
$\mu(a,b) \leqslant \min(a,b)$	∞ *теорема 4.2.5*	∞ *теорема 4.2.5*	∞ *теорема 4.2.5*
$\mu(a,b) = \min(a,b)$	2α *[4, теорема 13]*	1 *теорема 4.2.6*	2 *[4, Cor. 14]*
$\min(a,b) \leqslant \mu(a,b) \leqslant \max(a,b)$	$\alpha\frac{W_C}{w_C}$ *[4, теорема 9]*	$\frac{W_C}{w_C}$ *[4, теорема 10]*	$\frac{W_C}{w_C}$ *[4, теорема 10]*
$\mu(a,b) = \max(a,b)$	4α *теорема 4.2.7*	1 *[4, теорема 12]*	4α *теорема 4.2.7*
$\max(a,b) \leqslant \mu(a,b) \leqslant a+b$	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$ *[4, теорема 11]*	1 *[4, теорема 12]*	$2N\sqrt{\alpha V\frac{c_{\max}}{c_{\min}}}$ *[4, теорема 11]*
$\mu(a,b) \geqslant a+b$	1 *[4, теорема 8]*	1 *[4, теорема 8]*	1 *[4, теорема 8]*

Таблица 4.1: Таксономия полученных теоретических результатов относительно функции агрегации μ и конкретного вида графа сети (CAM, TCAM, CCAM). Числа показывают аппроксимационные факторы полиномиальных алгоритмов, представленных в соответствующих теоремах, а α обозначает аппроксимационный фактор полиномиального алгоритма, решающего задачу MStT.

чает аппроксимационный фактор для полиномиального алгоритма для задачи о минимальном дереве Штейнера (MStT), который вызывается как процедура в наших алгоритмах. Аппроксимационный фактор 1 означает, что для этого случая существует полиномиальный оптимальный алгоритм, а ∞ означает, что не существует полиномиального алгоритма с константным аппроксимационным фактором, если P≠NP. Для визуализации, чтобы упростить восприятие двумерной структуры наших результатов, мы также представляем их в трёх рисунках, соответствующих трём рассматриваемым нами случаям топологии сети: рис. 4.8 показывает результаты для CAM, рис. 4.9 показывает аппроксимационные факторы для TCAM, а рис. 4.10 — для CCAM. На каждом рисунке горизонтальная ось соответствует тому, как быстро растёт μ, а каждый прямоугольный блок соответствует результату в виде аппроксимационного фактора

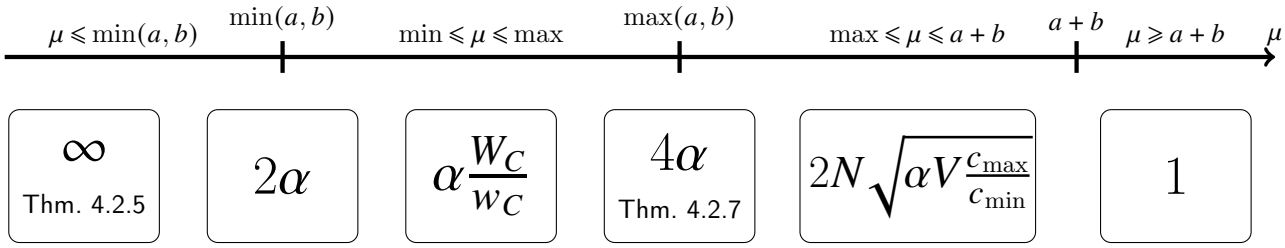


Рис. 4.8: Визуальное представление наших теоретических результатов для задачи CAM.

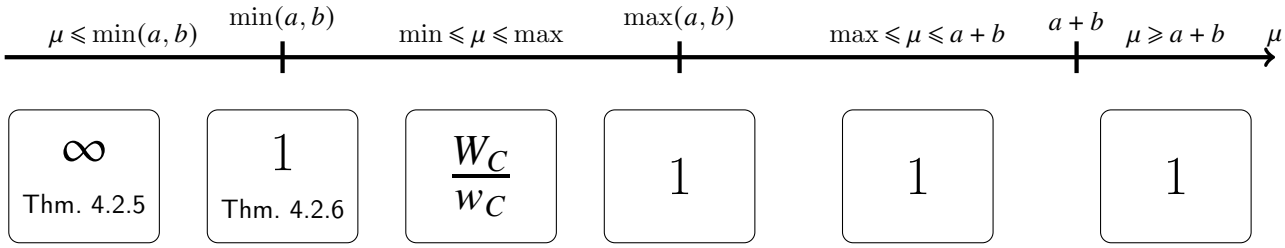


Рис. 4.9: Визуальное представление наших теоретических результатов для задачи TCAM.

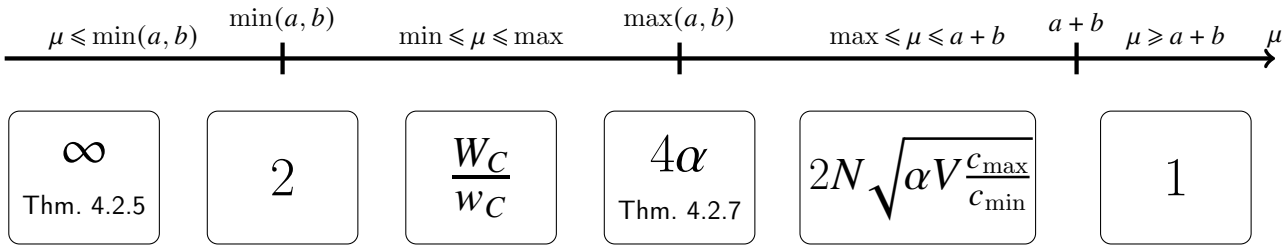


Рис. 4.10: Визуальное представление наших теоретических результатов для задачи CCAM.

для соответствующей задачи и ссылки на конкретную теорему, которая его доказывает для соответствующего алгоритма.

Сначала не будем предполагать никаких ограничений на поведение функции агрегации μ. Чтобы получить разумную стратегию в этом общем случае, мы начнём с рассмотрения более простого случая, в котором все блоки имеют размер x, и результат слияния двух блоков имеет тот же размер, что один из них, т.е. $\mu(x, x) = x$. В этом случае, когда пути двух блоков пересекаются, всегда выгодно объединить их на этом пересечении. Поэтому оптимальный план агрегации всегда следует некоторому подграфу-дереву в G, и вес этого дерева, умноженный на x, составляет цену плана агрегации, поскольку μ не меняет веса. Таким образом, задача сводится к поиску дерева минимального веса, которое соединяет заданное множество вершин, то есть к хорошо изученной задаче о минимальном дереве Штейнера [120], MStT, для которой известно

много разных алгоритмов с константным аппроксимационным фактором. Это сведение и соответствующие результаты не выносятся на защиту; они показаны в работах [4, 21, 30].

Отметим два важных частных случая.

Теорема 4.2.2. *Если каждая вершина в G содержит блок данных, то* MStT *можно решить точно за полиномиальное время.*

Теорема 4.2.3. *Если G является деревом, то* MStT *можно решить точно за полиномиальное время.*

Теоремы 4.2.2 и 4.2.3 по сути говорят, что в этих частных случаях существуют полиномиальные 1-приближённые алгоритмы для MStT.

Теорема 4.2.4. *Существуют полиномиальные алгоритмы, которые решают* CCAM$[\mu]$ *и* TCAM$[\mu]$ *на наборе блоков данных C с аппроксимационным фактором* $\frac{W_C[\mu]}{w_C[\mu]}$.

Однако для многих μ, в том числе важных (например, для объединения множеств), теорема 4.2.4 даёт только довольно слабые приближения; в частности, хотелось бы, чтобы аппроксимационный фактор не зависел от размеров блоков данных и конкретных значений μ, поскольку на практике $\frac{W_C}{w_C}$ может быть очень большим. К сожалению, это не удаётся сделать даже для ограниченного класса функций μ, которые уменьшают свои входы, то есть для класса функций, результат которых меньше, чем min.

Теорема 4.2.5. *Существует функция агрегации μ, для которой $\forall a, b$ $\mu(a, b) \leqslant \min(a, b)$, и нет полиномиального алгоритма с константным аппроксимационным фактором для* CCAM$[\mu]$ *или* TCAM$[\mu]$*, если $P \neq NP$.*

В зависимости от конкретного приложения значение μ может лежать в некотором интервале. Например, если **aggr** представляет собой *объединение множеств*, то $\mu(x, y) \in [\max\{x, y\}, x + y]$, а если **aggr** представляет *внешнее объединение* (outer join), то $\mu(x, y)$ скорее всего всегда будет больше, чем $x + y$. Мы показываем таксономию алгоритмов для различных μ; цель этого раздела состоит в том, чтобы найти разложение общего случая на подинтервалы, которые дают более сильные гарантии в худшем случае. Эти результаты подытожены в таблице 4.1, не все выносятся на защиту и показаны в работах [4, 21, 30].

Мы представляем несколько результатов, которые касаются конкретных функций агрегации, важных на практике.

Теорема 4.2.6. *Существует оптимальный алгоритм для* TCAM[min]*, работающий за полиномиальное время.*

Теорема 4.2.7. *Если существует полиномиальный α-приближённый алгоритм для задачи* MStT*, то существует полиномиальный 4α-приближённый алгоритм для задачи* CAM[max]*, который мы называем* `RECH_MStTMax`*.*

Теорема 4.2.8. *Если существует $a > 0$, для которого $\mu(a, a) = a$, то задача* CAM[μ] *NP-трудна, и если P≠NP, то для её решения нет менее чем $\frac{19}{18}$-приближённых полиномиальных алгоритмов даже в случае, когда веса всех рёбер одинаковы.*

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех вышеприведённых теорем, показаны в работах [4, 21, 23, 30].

4.3 Эффективное распределение запросов в многосвязных микросетях

В этом разделе мы рассматриваем совсем другое применение конкурентного анализа, которое, однако, использует методы, подобные методам Главы 2. Мы рассматриваем эффективное удовлетворение запросов на выделение электроэнергии в контексте многосвязных микросетей, где запрос может быть удовлетворён разными ресурсами генерации энергии: локальными, близлежащими или из региональной сети. Это определяет *вектор балансировки загрузки* (load balancing vector) для каждого запроса, который показывает, какие генераторы подключены и доступны для удовлетворения этого запроса. В самом простом случае вектор балансировки загрузки каждого запроса состоит из одного генератора, так что расписание каждого генератора можно составлять независимо. В этой постановке задача сводится к хорошо известной двумерной задаче об

упаковке полосы (strip packing problem), которая много изучалась в литературе [158]. В нашей модели каждый запрос может иметь вектор балансировки загрузки с несколькими входами, что усложняет задачу.

Запросы могут быть *эластичными* и *неэластичными* [125]; запрос называется неэластичным, если его выполнение должно начаться сразу же после его прибытия. Запрос называется *непрерывным*, если его нельзя прервать до окончания обработки после того, как она началась, и *прерывающимся* в другом случае. Когда генерирующей мощности недостаточно, чтобы удовлетворить все запросы, некоторые непрерывные запросы приходится задерживать, что приводит к необходимости коммутации электричества. Мы будем рассматривать планирование эластичных непрерывных запросов, поскольку:

(1) они представляют собой важный класс запросов,

(2) если увеличить разрешение планирования, неэластичные запросы могут стать эластичными, и

(3) прерывающиеся запросы можно использовать, чтобы дополнить расписание, переиспользуя ресурсы, не занятые непрерывными запросами.

Концепция «пакетированной» электроэнергии не нова [43,181]. Естественное представление запроса уровнем мощности и продолжительностью изоморфно пакету данных, представленному длиной (продолжительностью) и требованиями к вычислениям на процессоре (уровнем мощности). В некоторых упрощающих предположениях абстрактная задача исполнения ограниченных по времени нагрузок (которым требуется определённая мощность в течение определённого времени) некоторым множеством ресурсов генерации энергии с распределённым переключением похожа на распределение пакетов определённой длины, прибывающих на множество исходящих портов, через перестраиваемый оптический коммутатор [129, 204]. Каждый пакет соответствует запросу, каждый входящий порт — ресурсу генерации, а каждый исходящий порт — нагрузке; коммутаторы электричества, работающие с пакетами электроэнергии, уже были реализованы на практике [204].

Когда центральная энергосистема (central grid) имеет ограниченную мощность, оптимальное переиспользование инфраструктуры становится более важным, чем экономия цены для отдельных потребителей. Методы ценообразования, которые пытаются передвинуть пик генерации, — это лишь инструмент для

достижения более эффективного использования инфраструктуры центральной энергосистемы [73, 164]. Для планирования с ограничениями по стоимости входящие порты с более высокой ценой могут быть исключены из вектора балансировки загрузки, и планирование с ограничениями по стоимости для такой архитектуры микросети можно свести к оптимизации периода обработки или другой целевой функции на данном наборе запросов с модифицированными векторами балансировки загрузки. Более того, минимизация цены подразумевает, что пользователь ждёт самого дешёвого источника, что может быть неприемлемо даже для непрерывных эластичных запросов. Поэтому мы рассматриваем минимизацию задержки при исполнении набора запросов на ресурсы. Мы предполагаем, что есть не пренебрежимо малый штраф, который мы называем *ценой конфигурирования* (configuration overhead), в размере V временных слотов за то, чтобы изменить конфигурацию в коммутаторе; отметим, что значение V может влиять на решения в планировании. Мы также рассматриваем дополнительную цель: минимизацию общего числа конфигураций. Среди прочего, это уменьшает износ оборудования для переключения, который напрямую зависит от числа операций по переключению. Мы расширяем предыдущие работы об удовлетворении запросов в перестраиваемых оптических коммутаторах [129], чтобы найти нижние и верхние оценки на минимальное число перестраиваний, которые нужны, чтобы удовлетворить набор запросов, и показываем соотношение между целевыми функциями.

Мы моделируем набор многосвязных микросетей при помощи системы коммутирования $(I, \mathcal{D})$, которая состоит из набора входов I (генераторов) с пропускными способностями портов c_i (их номинальная мощность) и набора запросов (demands) $\mathcal{D}$, которые нужно запланировать; запрос d характеризуется его длиной $l(d)$ (сколько он продлится), шириной $w(d)$ (уровнем мощности) и *вектором балансировки загрузки* (load balancing vector) $v(d)$, который содержит набор входящих портов, доступных для обработки d (набор генераторов, которые могут удовлетворить этот запрос). Будем предполагать, что $(I, \mathcal{D})$ — это направленный ациклический двудольный граф. Планирование происходит партиями (batches). Пока обслуживается предыдущая партия запросов, завершается фаза подготовки для следующей. Будем предполагать, что запросы можно отложить по крайней мере до конца фазы подготовки. Фаза подготовки делится на предсказание запросов, когда предсказываются подзапросы для следую-

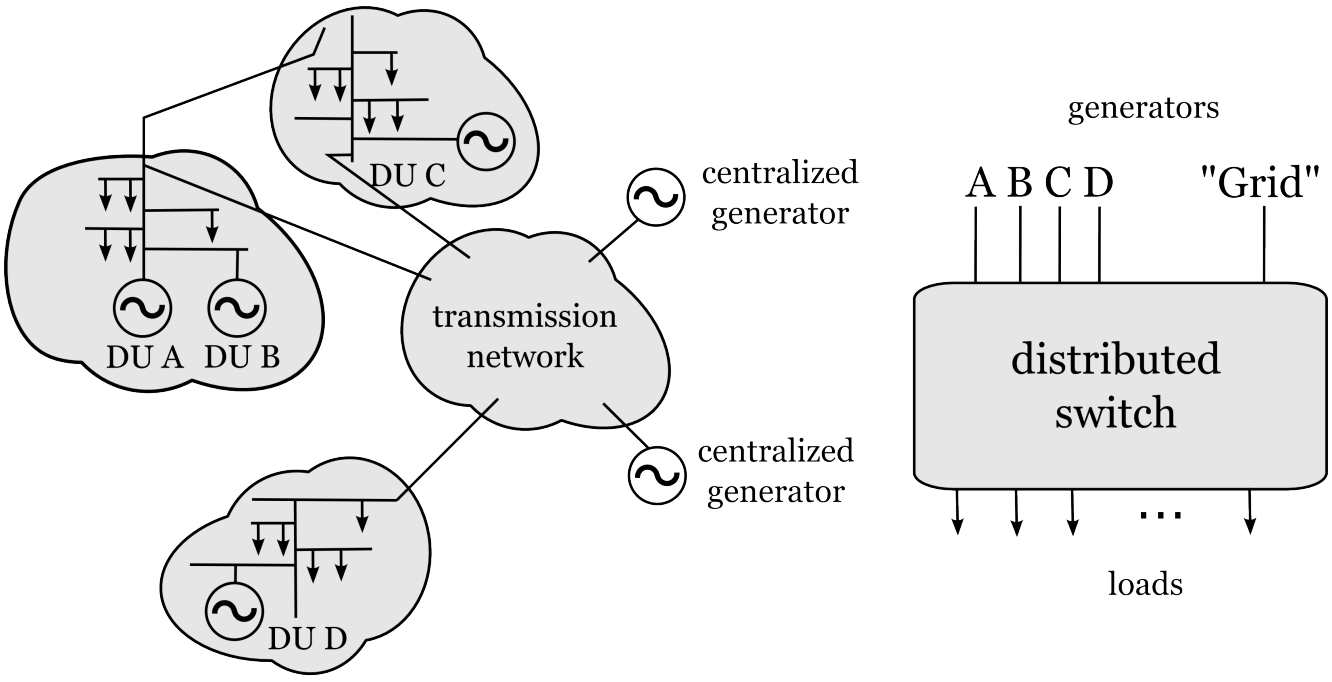


Рис. 4.11: Физическая инфраструктура (a) представлена логически в (b). Заметим, что «Grid» представляет собой все централизованные генераторы, доступные через сеть передачи энергии, «distributed switch» означает набор коммутаторов в микросетях, который определяет соответствие между нагрузками и генераторами.

щей партии, аукцион, в котором каждый запрос получает вектор балансировки загрузки, и планирование, когда оптимизируется желаемая целевая функция. Когда завершается планирование очередной партии, система реализует полученное расписание.

Время дискретно; обозначим через L и l соответственно самую большую и самую маленькую длину запроса во временных слотах. Если запрос d назначен входу i во время t, d использует $w(d)$ мощности порта i в течение интервала времени $[t, t + l(d) - 1]$. *Расписание* (schedule) P — это последовательность *конфигураций*, частичных отображений запросов на входы, которые должны удовлетворять ограничениям, накладываемым мощностями входящих портов (capacities) и векторами балансировки загрузки. *Длина* конфигурации C определяется самым длинным запросом, назначенным к реализации в течение C; конфигурацию можно представить вектором, каждый элемент которого — это множество запросов, назначенных соответствующему входящему порту. Далее мы будем использовать обозначение $\in$ не очень формально; например, будем говорить, что запрос принадлежит конфигурации (которая формально пред-

ставляет собой набор пар «запрос–порт»). Обозначим через $\mathcal{D}_i^t$ набор запросов, в чьих векторах балансировки загрузки разрешён первый (доступный всем) и i-й порт t; через n_i^t обозначим число запросов в $\mathcal{D}_i^t$ во время t. Мы также обозначим через $k(\mathcal{D}_i^t) = \left(\sum_{j=1}^{n_i^t} w_2(j)\right)/c_i$ общую «нормализованную нагрузку» для каждого порта, то есть сумму всех ширин $w_i(j)$ запросов из $\mathcal{D}_i^t$, разделённую на пропускную способность i-го порта c_i. Будем опускать верхний индекс t, когда время ясно из контекста. Без потери общности мы предполагаем, что в начале $k(\mathcal{D}_2) \geqslant k(\mathcal{D}_3) \geqslant \ldots \geqslant k(\mathcal{D}_I)$.

Чтобы сравнить эффективность разных стратегий, будем использовать конкурентный анализ в худшем случае. Алгоритм A имеет *аппроксимационный фактор* α (является *α-приближённым*) относительно некоторой (минимизируемой) целевой функции, если для каждого входа $(\mathcal{D}, I)$ алгоритм A выдаёт расписание со значением целевой функции не более чем в α раз больше оптимального.

Сконцентрируемся на практически интересном случае, в котором каждый запрос может быть удовлетворён двумя генераторами, и один из генераторов, соответствующий центральной энергосистеме, является общим для всех запросов. Эта ситуация возникает естественным образом, когда локальные сети распределения ресурсов, каждая из которых покрывает свой регион, дополняются центральной энергосистемой. В этом случае задача сводится к наиболее эффективному переиспользованию центральной энергосистемы (которая тоже имеет ограниченную мощность).

Будем рассматривать простые алгоритмы, которые можно эффективно реализовать и которые хорошо масштабируются. Общая схема таких алгоритмов показана в алгоритме 16. Для данного набора запросов $\mathcal{D}$ и множества входящих портов I с мощностями c_i, $i \in I$, жадная стратегия планирования заполняет каждую конфигурацию, жадным образом выбирая следующий запрос для обработки. Когда порты (генераторы), доступные для существующих запросов, заканчиваются, текущая конфигурация финализируется и начинается новая.

Основой алгоритма 16 является процедура ChoosePortDemand; она получает на вход текущее состояние (оставшиеся запросы и свободные мощности) и выдаёт пару вход–запрос (i, d) для следующего назначения. Разные алгоритмы, которые мы рассматриваем в этом разделе, различаются своими процедурами ChoosePortDemand. Очевидный общий алгоритм для этого слу-

Алгоритм 16: GreedySchedulingPolicy($\mathcal{D}$, I)

```
1 begin
2     D := 𝒟, C := ∅
3     while D ≠ ∅ do
4         Начать новую конфигурацию C := ∅, I′ := I
5         while есть доступные порты и запросы do
6             (i, d) := ChoosePortDemand(D, I′)
7             C := C ∪ {(i, d)}, c′_i := c′_i − w(d), D := D \ {d}
8         C := C ∪ {C}, D := D \ {d | d ∈ C}
9     return C
```

Алгоритм 17: SG

```
1 Function ChoosePortDemand{𝒟_i}_i, I:
2     for i := 2 to I do
3         if c_i > w(d) для некоторого d ∈ 𝒟_i then
4             return (i, ChooseDemand(𝒟_i, c_i));
5     return (1, ChooseFirst({𝒟_i}_i, I)).
```

чая — это SG, что означает «Shared Greedy» (Алгоритм 17): заполнить мощности каждого порта, кроме первого, а затем выбрать запросы для первого порта. Разные алгоритмы могут различаться в том, как они выбирают запрос для одного порта (процедура ChooseDemand) и в том, как они выбирают запрос, отсылаемый на первый порт для дополнительной обработки (процедура ChooseFirst). Здесь нужно поддерживать баланс между минимизацией числа конфигураций и минимизацией их общей длины (продолжительности). Для этого мы определяем два алгоритма из семейства SG: SLD (Shared Longest Demand, алгоритм 18) и SLP (Shared Longest Port, алгоритм 19). Алгоритм SLD выбирает самый длинный доступный запрос для текущей конфигурации; не важно, какой из них выбирать для процедуры ChooseDemand, а процедура ChooseFirst в SLD разрешает ничьи эвристикой выбора наибольшего порта (выбирая максимальный $k(\mathcal{D}_i)$). SLP, с другой стороны, выбирает процедурой ChooseFirst запрос от порта с максимальной удельной нагрузкой $k(\mathcal{D}_i)$; для разрешения ничьих и для процедуры ChooseDemand он берёт самый длинный запрос. Заметим, что во всех случаях максимизация и минимизация происходит по запросам, которые могут поместиться в доступный набор входов, т.е. ограничение $w(d) < c_i$ всегда присутствует, но опускается для краткости.

Поведение стратегии планирования определяется четырьмя важными пара-

Алгоритм 18: SLD

```
1 Function ChooseDemand(𝒟_i, c_i):
2     return arg max {l(d) | d ∈ 𝒟_i}.
3 Function ChooseFirst(𝒟 = {𝒟_i}_i, I):
4     return arg max_{d∈D} l(d);
```

Алгоритм 19: SLP

```
1 Function ChooseDemand(𝒟_i, c_i):
2     return arg max {l(d) | d ∈ 𝒟_i}
3 Function ChooseFirst(𝒟 = {𝒟_i}_i, I):
4     return d ∈ arg max_{𝒟_j} k(𝒟_j)
```

метрами:

(1) мощностью входящих портов,

(2) длиной запросов,

(3) шириной (мощностями) запросов,

(4) «удельной нагрузкой»;

все они также влияют на размер расписания. Длина запроса не влияет на число конфигураций, но может существенно влиять на общую продолжительность обработки, особенно с ростом L/l. Влияние входящей мощности оказывается нетривиальным даже для единичных мощностей запросов; число конфигураций связано с эффективным использованием входящей мощности.

Базовая задача здесь NP-сложна: даже оптимальное планирование с одним портом уже включает в себя задачу о рюкзаке. В этом разделе мы исследуем влияние каждого параметра на эффективность политик планирования. Во-первых, отметим соотношение между двумя целевыми функциями.

Теорема 4.3.1. *Для входа с максимальной длиной запроса L и минимальной длиной запроса l:*

(1) алгоритм SG *с аппроксимационным фактором $\leqslant \alpha$ относительно конфигураций имеет аппроксимационный фактор $\leqslant \frac{\alpha L h}{L+(l-1)h}$ относительно длины, где h — оптимальное число конфигураций;*

(2) алгоритм SG *с аппроксимационным фактором* $\leqslant \alpha$ *относительно длины имеет аппроксимационный фактор* $\leqslant \frac{L}{t}\left(1 + \frac{\alpha t - L}{l}\right)$ *относительно конфигураций, где* t *— оптимальная длина.*

Начнём со случая с единичными мощностями: у всех входящих портов $c_i = 1$, а у всех запросов $w(d) = 1$; в этом случае $k(\mathcal{D}_i) = |\mathcal{D}_i| = n_i$. Теперь i-й входящий порт полностью использован, если $\mathcal{D}_i$ непусто в конце конфигурации, и алгоритмы различаются в том, как они используют общий порт. Интуиция здесь в том, что назначение запросов на $\mathcal{D}_i$ с самыми длинными n_i должно оптимизировать число конфигураций, и мы действительно покажем, что у SLP число конфигураций будет оптимальным даже в более общей постановке задачи. Более того, оказывается, что мы можем доказать верхнюю оценку на аппроксимационный фактор *любого* алгоритма, использующего общую схему алгоритма SG.

Теорема 4.3.2. *Для системы с одним общим портом и единичными пропускными способностями любая стратегия планирования* ALG *из семейства* SG *имеет аппроксимационный фактор относительно конфигураций не более* $\frac{3}{2}$.

Мы увидим, что для некоторых алгоритмов эта оценка точна. Из теорем 4.3.2 и 4.3.1 также следует общая верхняя оценка $\frac{3Lh}{2(L+(l-1)h)} \leqslant \frac{3L}{2l}$ на аппроксимационный фактор относительно длины. Поскольку оптимальный алгоритм OPT также попадает в семейство алгоритмов SG, не получится показать нетривиальных общих нижних оценок; поэтому нижние оценки мы показываем для конкретных алгоритмов. Здесь важен баланс между числом конфигураций (оптимальное у SLP, хуже у SLD) и их общей длиной (лучше у SLD, хуже у SLP).

Теорема 4.3.3. *Для системы с единичными мощностями и* I *портами,* SLD *имеет аппроксимационный фактор не менее* $\left(\frac{3}{2} - 2^{-(I-1)}\right)$ *относительно конфигураций и длины.*

Заметим, что аппроксимационный фактор стремится к $\frac{3}{2}$ при $I \to \infty$, так что оценка из теоремы 4.3.2 является точной для SLD.

Далее обратимся к случаю различающихся мощностей входящих портов (но всё ещё единичных мощностей запросов). Первый интересный вопрос здесь в том, удастся ли нарушить верхнюю оценку для SG, показанную в теореме 4.3.2,

в случае разных входящих мощностей. Для этого введём дополнительные обозначения. Предположим, что OPT планирует все запросы на протяжении h конфигураций. Разделим все входящие порты, кроме общего, на две группы: $\mathcal{I}_1$ и $\mathcal{I}_2$. $\mathcal{I}_1$ содержит все входящие порты, для которых $k(\mathcal{D}_i) > h$, $2 \leqslant i \leqslant I$; $\mathcal{I}_2$ содержит все остальные порты (кроме первого). Мы покажем, что качество работы SG зависит от соотношения между мощностью общего порта и мощностями $\mathcal{I}_1$ и $\mathcal{I}_2$. Как частный случай SG рассмотрим SLD. Более того, по крайней мере для случая, когда $c_1 > c(\mathcal{I}_1)$, $c_1 \leqslant c(\mathcal{I}_2)$ и разрешены неединичные ограничения на мощность, верхнюю оценку $\frac{3}{2}$ для единичных ограничений на мощность можно нарушить. Интуиция здесь в том, что SG может по ошибке переиспользовать первый порт для запросов из $\mathcal{D}(\mathcal{I}_2)$, и в итоге не сможет переиспользовать первый порт для запросов из $\mathcal{D}(\mathcal{I}_1)$ достаточное число раз.

Теорема 4.3.4. *Для системы с мощностями $c_1 = 2c(\mathcal{I}_1)$ и $c(\mathcal{I}_2) = 2c_1$ число конфигураций в решении* SLD *не менее чем в $\frac{5}{3}$ раз превышает оптимальное в худшем случае.*

Результат в теореме 4.3.4 показывает, что обобщение результатов с единичными ограничениями на мощность нетривиально даже для единичных мощностей запросов. Следующая теорема даёт верхнюю оценку на число конфигураций алгоритма SG в случае неединичных ограничений на мощность. Здесь мы всё ещё рассматриваем запросы единичной мощности.

Теорема 4.3.5. *Число конфигураций в расписании* SG *не более чем в два раза превышает это число в оптимальном расписании, если выполняется одно из следующих условий:*

(1) $c_1 \leqslant c(\mathcal{I}_1)$*;*

(2) $c_1 > c(\mathcal{I}_1)$ *и* $c_1 > c(\mathcal{I}_2)$.

Мы показали, что алгоритм 19 (SLP), который приоритезирует характеристику $k(\mathcal{D}_i)$, является оптимальным.

Теорема 4.3.6. SLP *выдаёт оптимальное число конфигураций для случая различающихся входных мощностей и единичных мощностей запросов.*

SLD является частным случаем SG, так что из теорем 4.3.5 и 4.3.4 вытекают следующие следствия.

Следствие 4.3.1. *Для системы с различными входными мощностями расписание* SLD *не более чем в два раза длиннее оптимального расписания, если выполнено одно из следующих условий:*

(1) $c_1 \leqslant c(\mathcal{I}_1)$*;*

(2) $c_1 > c(\mathcal{I}_1)$ *и* $c_1 > c(\mathcal{I}_2)$*.*

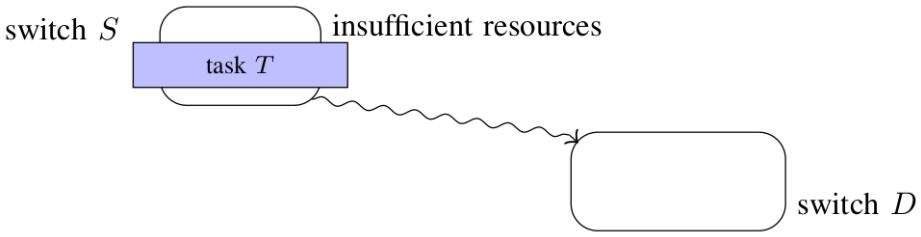
Следующее следствие вытекает очевидным образом, поскольку SLP является оптимальным; доказательство более сильной оценки остаётся интересным открытым вопросом.

Следствие 4.3.2. *При различных входящих мощностях расписание алгоритма* SLP *не более чем в* $\frac{L}{l}$ *раз длиннее, чем расписание алгоритма* OPT*.*

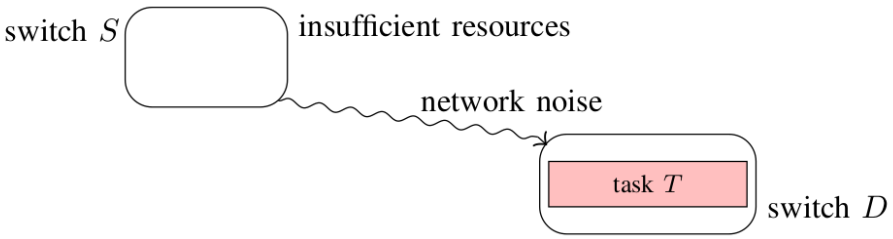
Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех вышеприведённых теорем, показаны в работах [40, 41].

4.4 Устойчивый к шуму подсчёт потоков трафика

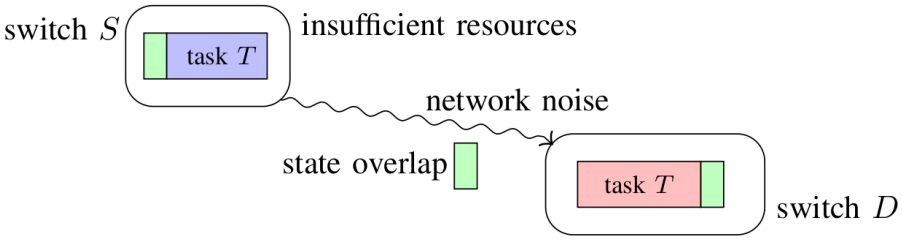
Даже простой подсчёт размеров всех потоков на сетевом элементе может перегрузить недостаточные ресурсы на плоскости данных, когда число потоков становится слишком большим. Рис. 4.12a показывает пример, в котором задача $\boldsymbol{T}$ перегружает ресурсы, доступные на коммутаторе $\boldsymbol{S}$. Существует большое число работ, где предлагаются приближённые алгоритмы, которые жертвуют точностью вычисления размеров потоков, чтобы сократить память, требующуюся на одном сетевом элементе. Такие приближённые решения включают в себя оценки (estimators) [90, 117, 189, 194] и скетчи (sketches): CM [87], CU [103], Pyramid Sketch [208], UnivMon [157], Elastic Sketch [206] и другие. Основанное на реальных сетевых следах исследование систем CEDAR [194], SAC [189] и DISCO [117] показало, что их ошибка может превышать 12% для 8-битных оценок размеров потоков [194]. Средняя относительная ошибка Elastic Sketch, одного из самых продвинутых современных методов, может достигать 4 даже при использовании 0.2 MB для представления 110 тысяч потоков, т.е. около



(a) Выбранный коммутатор S, у которого недостаточно ресурсов.



(b) Перемещение задачи на коммутатор D и сетевой шум.



(c) Устойчивое распределённое исполнение задачи на S и D (RoDiC).

Рис. 4.12: Исполнение задачи мониторинга с состоянием.

15 битов на поток, как показывает рис. 9a в работе [206]. Такой точности может оказаться недостаточно.

Если переместить исполнение задачи мониторинга с назначенного перегруженного элемента на другой элемент, расположенный на пути потоков, точность результата может пострадать из-за *сетевого шума* (network noise) в форме переупорядочивания и/или потери пакетов. Как показано на рис. 4.12b, сетевой шум может не позволить коммутатору D выполнить задачу T столь же точно, как на коммутаторе S, потому что сеть может потерять или переупорядочить пакеты, пока они доставляются от S к D. Эта неточность может возникнуть независимо от того, связаны ли два сетевых элемента надёжным или ненадёжным транспортным протоколом. Поэтому выполнение задачи мониторинга должно каким-то образом включать и исходный сетевой элемент, а также должно быть устойчивым к сетевому шуму.

В этом разделе мы представляем RoDiC (Robust Distributed Computation), общий метод, который корректно исполняет задачу мониторинга на нескольких

сетевых элементах, несмотря на сетевой шум на пути между элементами. Для задач с состоянием (stateful tasks) предлагаемый метод не только поддерживает состояние в нескольких сетевых элементах, но и устойчиво к шуму передаёт части состояния между этими элементами. RoDiC поддерживает мониторинг метрик потоков без добавления каких бы то ни было управляющих пакетов. Пакеты данных каждого потока, который нужно мониторить, получают несколько (например, три) контрольных бита, в которых передаётся состояние потока. Это свойство делает RoDiC применимым к однонаправленным соединениям.

RoDiC — это общий метод, который можно реализовывать по-разному для разных метрик. Однако он опирается на два основных общих принципа дизайна:

(1) *группировка пакетов:* RoDiC делит каждый поток на группы последовательных пакетов и распределяет состояние так, чтобы последующий сетевой элемент работал с метриками мониторинга на более грубом уровне групп пакетов;

(2) *пересечение состояний:* RoDiC поддерживает распределённое состояние на нескольких сетевых элементах в виде пересекающихся блоков; биты синхронизации в пакетах данных передают пересекающуюся часть, чтобы распределённое состояние поддерживалось непротиворечивым несмотря на сетевой шум.

Подход RoDiC и его принцип пересекающихся состояний проиллюстрированы на рис. 4.12c, где показано устойчивое к шуму распределённое выполнение задачи T на коммутаторах S и D. Блоки состояния на S и D пересекаются. S передаёт пересечение состояний на D в рамках пакетов данных.

Поток f проходит по сети через последовательность коммутаторов (сетевых элементов). На коммутаторе-источнике S поток состоит из $|f|$ пакетов на сетевом уровне, пронумерованных последовательно от 0 до $|f| - 1$. Поскольку мы рассматриваем каждый поток на сетевой уровне, исходная передача и повторная передача блока данных транспортным или прикладным слоем появляются в потоке как разные пакеты. Некоторые из последующих коммутаторов помогают коммутатору S в распределённом вычислении метрики потока. Хотя сеть может переупорядочить или потерять некоторые пакеты, цель состоит в том, чтобы разработать распределённый алгоритм, который корректно вычисляет нужную метрику для всех потоков, несмотря на такой сетевой шум.

Алгоритм 20: Вычисление размера потока на двух коммутаторах в модели последовательных потерь.

```
1  begin
2      Function Initialization:
3          c_1 ← 0; c_2 ← 0
4      Function Source (j):
5          h[j] ← ⌊c_1 / 2^{n−t}⌋; c_1 ← (c_1 + 1) mod 2^n
6      Function Destination (j):
7          diff ← Overshoot(h[j]);
8          if 1 ⩽ diff ⩽ 2^{t−1} then
9              c_2 ← c_2 + diff
10     Function Termination:
11         c_2 ← c_2 + Overshoot(⌊c_1 / 2^{n−t}⌋);
12         |f| ← c_2 · 2^{n−t} + c_1 mod 2^{n−t}
13     Function Overshoot (g):
14         return (g − c_2 mod 2^t + 2^t) mod 2^t
```

Каждый участвующий коммутатор s_m использует n_m битов, чтобы поддерживать его блок c_m распределённого состояния. Коммутатор s_m добавляет не более t_m *битов синхронизации* к каждому пакету данных, чтобы передавать управляющую информацию следующему коммутатору. Будем обозначать n_1 и t_1 как просто n и t соответственно.

Сначала рассмотрим модель *последовательных потерь* (consecutive-loss model), которая ограничивает сетевой шум посредством двух параметров. *Параметр потерь* L — это верхний предел на потери *последовательных* пакетов, т.е. коммутатор D получает не менее одного пакета из интервала от i до $i + L$. *Параметр переупорядочивания* R показывает максимальное расстояние переупорядочивания пакетов, т.е. коммутатор D может получить пакет i раньше пакета j, только если $i \leqslant j + R$. Мы ограничиваем пределы возможной устойчивости в модели последовательных потерь следующим образом

Теорема 4.4.1. *Детерминированный алгоритм RoDiC не может гарантировать корректное распределённое вычисление размера потока на двух коммутаторах, если* $L \geqslant 1$ *пакет и* $R \geqslant 2^{n-1} + 1$ *пакетов.*

Хотя теорема 4.4.1 и другие результаты (см. работы [2, 15, 29]) показывают пределы применимости для устойчивого у шуму распределённого подсчёта раз-

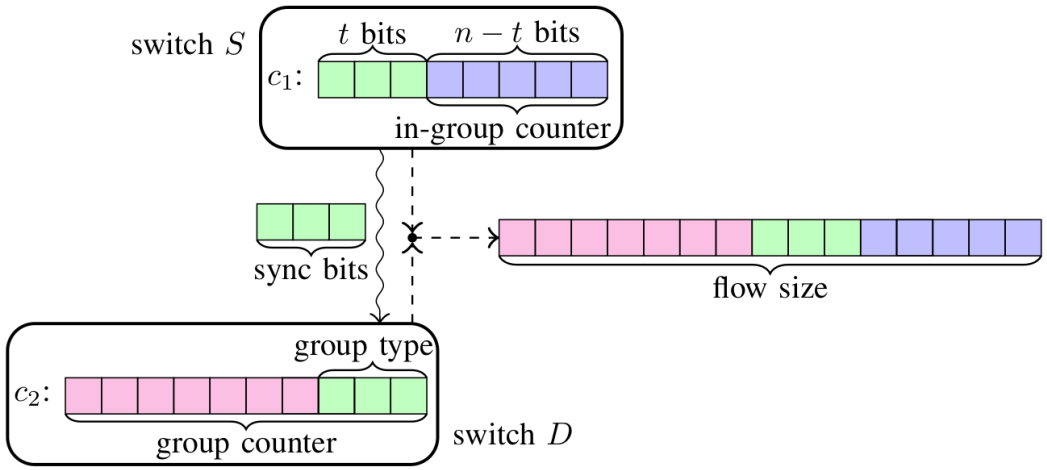


Рис. 4.13: Применение принципов RoDiC группировки пакетов и пересечения состояния к задаче вычисления размеров потоков.

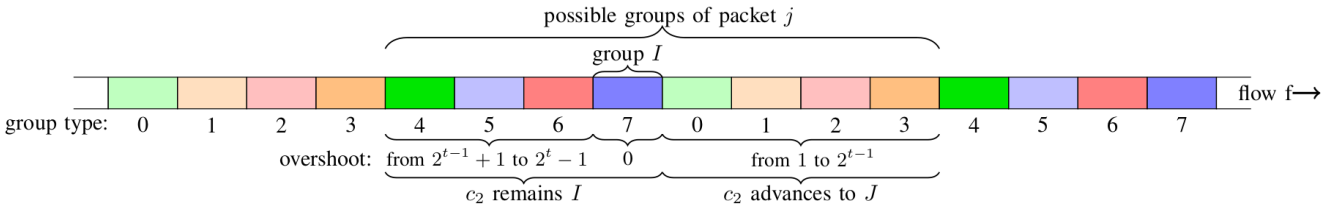
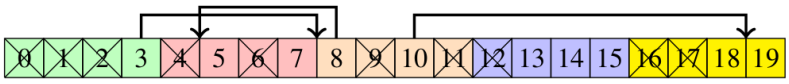


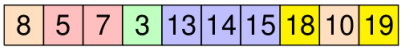
Рис. 4.14: Работа процедуры DESTINATION в алгоритме 20 для $t = 3$, $c_2 = I$, и когда пакет j прибывает на D из группы J.

мера потоков, мы можем построить алгоритм RoDiC, который приближается к этим пределам. Алгоритм 20 следует обоим принципам группировки пакетов и пересечения состояний, как показано на рис. 4.13. Здесь t старших битов c_1 и t младших битов c_2 образуют t-битовое пересечение между состояниями счётчиков на коммутаторах S и D. Эти t битов указывают тип группы и нумеруют типы групп последовательно от 0 до $2^t - 1$. Размер группы составляет $n - t$ битов. Коммутатор S использует свои $n - t$ младших битов c_1, чтобы считать пакеты внутри группы. Счётчик c_2, включая его t младших битов, задающих тип группы, поддерживает наибольший номер группы среди всех групп, которые доставили пакет в коммутатор D.

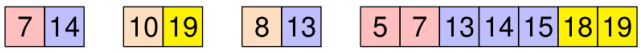
В алгоритме 20 S копирует тип группы из t старших битов c_1 в биты синхронизации $h[j]$ каждого пакета j, а затем увеличивает c_1 на единицу, чтобы учесть этот пакет. Получив пакет j, D использует арифметику по модулю 2^t, чтобы подсчитать, на сколько групп тип группы в $h[j]$ превышает тип группы в t младших битах c_2. Например, для $t = 3$ битов есть восемь типов групп, пронумерованных от 0 до 7, и тип 4 превышает тип 7 на 5 групп. Если сетево-



(a) Поток на коммутаторе S и сетевой шум.



(b) Последовательность потоков, приходящая на коммутатор D.



(c) Четыре примера $\mathrm{span}(2, 2)$.

Рис. 4.15: Примеры интервалов для потока из 20 пакетов.

го шума нет, превышение составляет либо 0, либо 1 группу. С сетевым шумом превышение может достичь $2^t - 1$ групп. Когда превышение составляет 0 или заключено между $2^{t-1} + 1$ и $2^t - 1$ группами, D игнорирует пакет j, поскольку алгоритм 20 предполагает, что c_2 уже учёл группу пакета j. Когда превышение находится между 1 и 2^{t-1} групп, алгоритм предполагает, что пакет j прибыл из более поздней, ещё не учтённой группы, и D увеличивает c_2 на размер превышения. Например, если t составляет 3 бита, тип группы в $h[j]$ равен 1, и $c_2 = 7$, превышение составляет 2 группы, и D увеличивает c_2 до 9. Когда поток заканчивается, алгоритм 20 подсчитывает, на сколько групп тип группы в c_1 превышает тип группы в c_2. Затем алгоритм увеличивает c_2 на значение превышения и конкатенирует все биты c_2 и $n - t$ битов c_1, чтобы получить размер потока.

Теорема 4.4.2. *Алгоритм 20 всегда вычисляет размер потока на двух коммутаторах корректно, если*

$$R \leqslant 2^{n-1} - 2^{n-t} \text{ пакетов и } L + R \leqslant 2^{n-1} - 1 \text{ пакетов.}$$

Модель последовательных потерь представляет максимально допустимый сетевой шум в терминах его локального влияния на поток. Далее рассмотрим альтернативное представление, связанное с глобальными свойствами последовательности пакетов, полученных коммутатором D из потока. Мы всё ещё используем ограничение на переупорядочивание в R пакетов, но новая модель не накладывает явных ограничений на последовательную потерю пакетов, а вместо этого характеризует полученную последовательность пакетов через понятие

Алгоритм 21: Вычисление размера потока на двух коммутаторах в интервальной модели.

```
1 Function Destination (j):
2     diff ← Overshoot(h[j])
3     if 1 ⩽ diff ⩽ γ then c_2 ← c_2 + diff
```

интервала (*span*). Это понятие определяется относительно разбиения потока f на коммутаторе S на группы из 2^k последовательных пакетов; $\mathrm{span}(\gamma, k)$ означает подпоследовательность пакетов на S, которые прибывают на D в том же порядке, пропускают не более чем γ полных групп в начале потока и пропускают не более чем $\gamma - 1$ полных последовательных групп после любого пакета в этой подпоследовательности. Рис. 4.15 показывает четыре примера $\mathrm{span}(2, 2)$ для потока, состоящего из 20 пакетов. *Интервальная модель* (span-based model) ограничивает сетевой шум через параметр переупорядочивания R и существование таких интервалов.

Интервал $\mathrm{span}(\gamma, k)$ также является интервалом $\mathrm{span}(\gamma + 1, k)$. Поэтому невозможность гарантированной корректности подсчёта размера потоков несмотря на существование $\mathrm{span}(\gamma, k)$ подразумевает такую невозможность и для интервалов с большими значениями γ. Она также подразумевает, что для того чтобы получить гарантированно корректный подсчёт, нужно, чтобы существовал интервал с меньшим значением γ. Без переупорядочивания пакетов интервальная модель максимального сетевого шума подразумевает следующие пределы возможного.

Оказывается, что существование $\mathrm{span}(\gamma, k)$ недостаточно для того, чтобы детерминированный алгоритм RoDiC мог гарантировать корректный распределённый подсчёт размера потока, если $R \geqslant 2^n - \gamma \cdot 2^k + 1$ пакетов (этот результат был доказан в [2, 15]).

Параметр γ управляет балансом между существованием интервалов и ограничением на переупорядочивание. Увеличение t ослабляет пределы возможного и по γ, и по R. Для $\gamma = 2^{t-1}$ групп, ограничение на переупорядочивание в интервальной модели составляет $R \geqslant 2^{n-1} + 1$ пакетов, т.е. оно такое же, как в теореме 4.4.1 для модели последовательных потерь. Интервальная модель не просто обобщает модель последовательных потерь посредством параметра γ, но и более эффективно характеризует максимально допустимый сетевой шум.

Далее мы представляем и анализируем алгоритм 21, который достигает пределов возможного, установленных выше. Алгоритм 21 использует $k = n - t$ и отличается от алгоритма 20 только в изменённой процедуре DESTINATION, которая увеличивает c_2, если превышение составляет от 1 до γ групп.

Теорема 4.4.3. *Когда γ составляет не более $2^t - 1$ групп, алгоритм 21 всегда вычисляет размер потока на двух коммутаторах корректно, если существует* $\mathrm{span}(\gamma, n - t)$, *и $R \leqslant 2^n - (\gamma + 1) \cdot 2^{n-t}$ пакетов.*

Завершим этот раздел количественной иллюстрацией устойчивости алгоритма 21 к сетевому шуму, гарантированной теоремой 4.4.3. Для $n = 8$ битов, $t = 2$ битов и $\gamma = 1$ группы алгоритм 21 гарантированно корректно вычисляет размер потока, если существует $\mathrm{span}(1, 6)$ с переупорядочиванием на не более чем 128 пакетов. Если t увеличить до 3 битов, корректное вычисление гарантировано при существовании $\mathrm{span}(1, 5)$, и переупорядочивать можно до 192 пакетов. Когда n и t остаются на уровне 8 и 3 битов соответственно, а γ увеличивается до 6 групп, гарантированно корректное вычисление получается при существовании $\mathrm{span}(6, 5)$ и переупорядочивании не более чем на 32 пакета.

Все показанные выше теоретические результаты были проверены на практике при помощи обширных экспериментальных исследований, результаты которых подтвердили теоретические выводы. Результаты этих экспериментальных исследований, а также доказательства всех вышеприведённых теорем, показаны в работах [2, 15, 29].

Глава 5 Заключение

5.1 Основные результаты

В этом разделе мы кратко перечисляем **основные результаты** настоящей диссертации, которые выносятся на защиту. Все эти результаты были представлены и подробно обсуждены в главах 2, 3 и 4.

5.1.1 Алгоритмы управления буфером и их анализ

Алгоритмы управления буфером с различными требованиями к обработке для одной очереди. Для пакетов, требующих различное время обработки (разное число процессорных циклов), был введён и исследован ряд онлайн-алгоритмов в архитектуре с одной очередью. Доказана нижняя оценка k на конкурентность алгоритма NPO (non-push-out FIFO), $2(1-1/B)$ для $k \geqslant B$, $\frac{2k}{k+1}$ для $k < B$ и $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$ в случае больших k для PO (push-out FIFO). Был введён и исследован новый класс *ленивых* онлайн-алгоритмов, который позволяет доказывать верхние оценки на конкурентность, доказана несравнимость PO и LPO в худшем случае, верхняя оценка $\max\{1, \ln k\} + 2 + o(1)$ и нижняя оценка $\lfloor \log_B k \rfloor + 1 - O(\frac{1}{B})$ для LPO (lazy push-out) и верхняя оценка $2 + \frac{1}{B}$ для LPO в частном случае двух значений требуемого времени обработки. Более того, эти идеи были обобщены: введено семейство алгоритмов Semi-FIFO, которое включает семейство ленивых алгоритмов; доказана общая верхняя оценка $\frac{1}{B} \log_{B/(B-1)} k + 3$ для любого ленивого алгоритма, соответствующая нижняя оценка $(1 + \frac{1}{B} \log_{B/(B-1)} k)$ для LRFIFO, LRevPQ и RFIFO, верхняя оценка 2 и нижняя оценка $2 - \frac{1}{B} \left\lceil \frac{B}{k} \right\rceil$ для LPQ (lazy priority queue), общая нижняя оценка $\left(1 + \frac{\min\{k,B\}-1}{B}\right)$ для любого жадного Semi-FIFO алгоритма и несколько результатов о несравнимости в худшем случае. Эти результаты были обобщены на случай β-выталкивания: доказана верхняя оценка $\left(3 + \frac{1}{B} \log_{\frac{\beta B}{\beta B - 1}} k\right) \frac{1-\alpha}{1-\alpha \log_\beta k}$ и нижняя оценка $\frac{1-\alpha}{1-\alpha \log_\beta k}$ для цены копирования $0 < \alpha < \frac{1}{\log_\beta k}$ для любого лени-

вого алгоритма с β-выталкиванием LA_β. Эти результаты были опубликованы в работах [12, 28, 32].

Алгоритмы управления буфером для нескольких разделённых очередей. Была рассмотрена система с несколькими очередями, где каждая очередь принимает пакеты с некоторым заданным требованием к обработке, и общий размер всех очередей в буфере тот же, что размер буфера в архитектуре с одной очередью. Для этой системы был введён и изучен ряд онлайн-алгоритмов, в частности алгоритмы Longest-Queue-First (LQF), Shortest-Queue-First (SQF), Minimal-Queue-First (MQF), MaxQF, Packet-Round-Robin (PRR), Cycle-Round-Robin (CRR). В качестве нижних оценок было доказано, что LQF не менее чем $\frac{m}{2}$-конкурентен для $m = \min\{k, B\}$, SQF не менее чем k-конкурентен, MQF не менее чем $\left(1 + \frac{k-1}{2k}\right)$-конкурентен, MaxQF не менее чем k-конкурентен, PRR не менее чем $\frac{3k(k+2)}{4k+16}$-конкурентен, и CRR не менее чем $\frac{k}{H(k)}$-конкурентен, где $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$. Главный позитивный результат — это верхняя оценка: MQF не более чем 2-конкурентен. Для частного случая двух очередей было доказано, что конкурентность MQF составляет не менее чем $\left(1 + \frac{1+\left\lfloor \frac{aB-1}{b} \right\rfloor}{B+\left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor+1\right)\right\rceil}\right)$ и не более чем $\left(1 + \frac{1+\left\lfloor \frac{aB-1}{b} \right\rfloor}{B+\left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor+1\right)\right\rceil}\right)$. Был доказан ряд результатов о несравнимости в худшем случае, изучены возможности взаимной эмуляции между архитектурами с одной и несколькими очередями, где были найдены возможности и ограничения для эмуляции архитектур с несколькими очередями при помощи одной очереди. Эти результаты были опубликованы в работах [5, 22, 39].

Алгоритмы управления буфером с различными требованиями к обработке и несколькими выходными портами. В этой части рассматривается коммутатор с общей памятью, где один буфер разделяется между всеми типами трафика, но в отличие от предыдущей модели с одной очередью каждое ядро обрабатывает только один тип трафика. Введены и проанализированы несколько онлайн-алгоритмов: Non-Push-Out-Harmonic-Static-Threshold (NHST), Non-Push-Out-Equal-Static-Threshold (NEST), Non-Push-Out-Harmonic-Dynamic-Threshold (NHDT), Longest-Queue-Drop (LQD), Biggest-Packet-Drop (BPD) и Longest-Work-Drop (LWD). Сначала рассмотрен случай, в котором у каждой очереди есть одно фиксированное значение требуемого времени обработки, которое одинаково для всех пакетов в этой очереди.

Для алгоритмов без выталкивания показано, что NHST является $(kZ + o(kZ))$-конкурентным, NEST является $(n + o(n))$-конкурентным, а NHDT не менее чем $\left(\frac{1}{2}\sqrt{k \ln k} - o(\sqrt{k \ln k})\right)$-конкурентен, если B асимптотически больше, чем k. Для алгоритмов с выталкиванием показано, что LQD не менее чем $(\sqrt{k} - o(\sqrt{k}))$-конкурентен, BPD не менее чем $(\ln k + \gamma)$-конкурентен для $B \geqslant \frac{k(k+1)}{2}$, LWD не менее чем $\left(\frac{4}{3} - \frac{6}{B}\right)$-конкурентен для $k \geqslant 6$. Здесь основной результат — это верхняя оценка, показывающая, что LWD не более чем 2-конкурентен. Далее рассмотрена модель, разрешающая принимать в одну и ту же очередь пакеты с разнородными требованиями к обработке. Это добавляет алгоритм с выталкиванием Biggest Average Drop (BAD). В качестве нижних оценок доказано, что BPD и BAD оба не менее чем $(n+1)/2$-конкурентны, LQD не менее чем $(n/2 - o(n))$-конкурентен для $k \geqslant n(n-1)$ и $B \geqslant k + n$, и LWD с порядком обработки FIFO не менее чем $(\log_{B/n} k)(1 - n/B) + 1$-конкурентен. Здесь основной результат — это верхняя оценка, показывающая, что LWD не более чем $1 + \frac{B}{T+B}$-конкурентен, где T — минимальное число пакетов, передаваемых между любыми двумя последовательными итерациями. В-третьих, рассмотрено другое обобщение, в котором каждый пакет единичного размера с требуемым временем обработки 1 имеет два параметра: метку исходящего порта и ценность; целью является максимизация общей переданной ценности. Здесь вводятся алгоритмы Minimal-Value-Drop (MVD) и Maximal-Ratio-Drop (MRD). Показано, что MVD не менее чем $\frac{m-1}{2}$-конкурентен для $m = \min\{k, B\}$, MRD не менее чем $\frac{4}{3}$-конкурентен, если ценность каждого пакета совпадает с номером его исходящего порта, и не менее чем V-конкурентен в общем случае, если $n \geqslant B - V^2 + 1$. Эти результаты были опубликованы в работах [9, 11, 27, 31].

Алгоритмы управления буфером с различными требованиями к обработке и переменным размером. Здесь рассматривается общая модель для задачи, в которой сетевой элемент должен управлять принятием и обработкой пакетов в одной очереди ограниченного размера, а входящий трафик состоит из пакетов, каждый из которых характеризуется размером (size) (например, в байтах) и требующимся временем обработки (в циклах процессора). В этой модели рассмотрены несколько разных приоритетных очередей: Shortest Remaining Processing Time (SRPT), Longest-Packet (LP) и Most-Effective-Packet (MEP), а также жадные алгоритмы с выталкиванием (PO) и без (NPO). Пока-

зано, что NPO не менее чем kL-конкурентен и не более чем $kL\frac{B}{B-L}$-конкурентен для SRPT-приоритетов, а конкурентность PO составляет не менее L для SRPT-приоритетов и $B \geqslant 2L$. Основными результатами здесь тоже являются верхние оценки: во-первых, для $B > 2L$ PO не более чем $\frac{4L-2}{L_a}$-конкурентен для SRPT-приоритетов, где L_a — средняя длина пакетов, переданных PO. Для достаточно больших буферов PO не более чем $\frac{(2L-1)(N+1)}{NL_a}$-конкурентен для SRPT-приоритетов, где L_a — средняя длина пакетов, переданных PO, и $N = \lceil \frac{B-2L+1}{2L-1} \rceil$, PO не более чем $(k+3)$-конкурентен для LP-приоритетов и достаточно больших буферов, и PO is $\Omega(\log k)$-конкурентен для MEP-приоритетов. Эти результаты были опубликованы в работах [13, 38].

Алгоритмы управления буфером для пакетов с несколькими характеристиками. В этом разделе снова рассматривается коммутатор с одной очередью, где буфер размера B делится между всеми типами трафика. Разница в том, что в этом разделе изучается влияние *двух* характеристик, ценности и требующегося времени обработки, на взвешенную пропускную способность. Введено общее определение приоритетной очереди PQ_f, которая упорядочивает и обрабатывает пакеты в соответствии с функцией f, и рассмотрены несколько естественных вариантов таких очередей: $\mathrm{PQ}_{-w,v} = \mathrm{PQ}_{-w+v/(V+1)}$ упорядочивает пакеты по возрастанию их требующегося времени обработки, выбирая в случае равенства пакеты более высокой ценности; $\mathrm{PQ}_{v,-w} = \mathrm{PQ}_{v-w/(W+1)}$ упорядочивает пакеты по убыванию их ценности, выбирая в случае равенства пакеты с меньшим требующимся временем обработки; $\mathrm{PQ}_{v/w}$ упорядочивает пакеты по убыванию их отношения ценности и работы, т.е. эта очередь приоритезирует пакеты, которые дают наибольшую ценность на один временной слот обработки. Показано несколько общих нижних оценок: для $B = 1$ и $V > \sqrt{W}$ любой онлайн-алгоритм ALG не менее чем $\sqrt{W}$-конкурентен; в случае двух ценностей, или если $V \leqslant \sqrt{W}$, любой онлайн-алгоритм ALG не менее чем $\min\{V, W/V\}$-конкурентен для $B = 1$; для произвольного B и $V > \sqrt{W}$ любой онлайн-алгоритм ALG, который сохраняет FIFO порядок обработки и передачи пакетов, не менее чем $\left(\frac{\sqrt{W}}{B} + 1 - \frac{1}{B}\right)$-конкурентен; в случае двух ценностей, или если $V \leqslant \sqrt{W}$, любой онлайн-алгоритм ALG с FIFO порядком обработки и передачи пакетов не менее чем $\frac{V+B-1}{W+B-1}$-конкурентен. Показаны также общие нижние оценки для приоритетных очередей: для $V > \sqrt{W}$ и любой функции приоритета f с

$f(w,v) \in \mathbb{R}$ алгоритм PQ_f не менее чем $\sqrt{W}$-конкурентен; в случае двух ценностей, или если $V \leqslant \sqrt{W}$,алгоритм PQ_f не менее чем $\min\{V, W/V\}$-конкурентен. Доказан ряд нижних оценок для конкретных алгоритмов: $\mathrm{PQ}^{\beta}_{-w,v}$ не менее чем V-конкурентен как в случае произвольных ценностей пакетов, так и в случае двух ценностей; $\mathrm{PQ}^{\beta}_{v/w}$ не менее чем $\min\{V, W\}$-конкурентен в случае произвольных ценностей пакетов; в случае двух ценностей, если $\beta W \geqslant V$, то $\mathrm{PQ}^{\beta}_{v/w}$ не менее чем V-конкурентен; $\mathrm{PQ}^{\beta}_{v,-w}$ не менее чем $\left(\frac{(V-1)}{V}W - o(1)\right)$-конкурентен в случае произвольных ценностей пакетов и не менее чем $\left(\frac{W}{V} + o(1)\right)$-конкурентен в случае двух ценностей. Эти результаты были опубликованы в работах [8, 25].

Алгоритмы управления буфером с ограниченной задержкой. В этой части рассматривается коммутатор с одной очередью, где буфер делится между всеми типами трафика, и предполагается, что каждый прибывающий пакет p снабжён тремя характеристиками: (1) число требующихся циклов обработки (*работа*, work) $w(p) \in \{1, \ldots, W\}$, (2) *ценность* (value) пакета $v(p) \in \{1, \ldots, V\}$, (3) *допуск* (slack) $s(p) \in \{1, \ldots, S\}$, который определяет, через какое время после прибытия пакет должен быть передан, прежде чем он будет потерян и ничего не добавит в целевую функцию (взвешенную пропускную способность). Мы снова рассматриваем главным образом приоритетные очереди, в частности следующие: $\mathrm{PQ}_{-w} = \mathrm{PQ}_{-w+v/(V+1)}$ упорядочивает пакеты по возрастанию их требующегося времени обработки, разрешая «ничьи» по ценности; $\mathrm{PQ}_v = \mathrm{PQ}_{v-w/(W+1)}$ упорядочивает пакеты по убыванию их ценности, разрешая «ничьи» по требующемуся времени обработки; $\mathrm{PQ}_{v/w}$ упорядочивает пакеты по убыванию их отношения ценности к работе, т.е. она приоритезирует пакеты, который дают наивысшую ценность на один цикл обработки; разрешать «ничьи» можно или по работе, или по ценности, и все приведённые ниже результаты сохраняются в обоих случаях, так что мы не различаем эти варианты в обозначениях; PQ_{v/w_0} упорядочивает пакеты по убыванию их отношения ценности к исходной работе, т.е. в отличие от $\mathrm{PQ}_{v/w}$ приоритет пакета не увеличивается по мере того, как он постепенно обрабатывается и его значение w уменьшается; $\mathrm{PQ}_{v/s}$ и PQ_{v/s_0} упорядочивают пакеты по убыванию их отношения ценности к допуску и ценности к исходному допуску соответственно. Показана общая нижняя оценка: конкурентность любого детерминированного онлайн-алгоритма A не может быть ограничена сверху константой; более того, конкурентность любого детер-

минированного онлайн-алгоритма не может быть ограничена сверху константой, даже если существует такая фиксированная константа c, что для любого пакета $s(p) \geqslant w(p)+c$. Показан ряд нижних оценок: если w, s и v — допустимые значения требующегося времени обработки, допуска и ценности соответственно, и s всегда превышает cw для некоторой константы c, то конкурентность алгоритмов PQ_v, $\mathrm{PQ}_{v/w}$, PQ_{v/w_0}, $\mathrm{PQ}_{s/w}$, PQ_{s/w_0} составляет не менее чем

$$\frac{1-(\frac{w-2}{w})^{\frac{1}{w}\left\lfloor \log_{\frac{w}{w-2}} v\right\rfloor}}{1-\frac{w-2}{w}}\frac{1}{c},$$

что не является константой по v и w. Для мультипликативного ограничения на допуск также показана общая нижняя оценка: если w, s и v — допустимые значения требующегося времени обработки, допуска и ценности соответственно, и s всегда превышает cw для некоторой константы c, то если $w > c$, конкурентность любого онлайн-алгоритма без выталкивания составляет не менее V. Другой результат касается алгоритмов с α-выталкиванием: алгоритм $\mathrm{PQ}_{v/w}$ с α-выталкиванием имеет нижнюю оценку на конкурентность $\Omega(w)$ для произвольно больших допусков. Эти результаты были опубликованы в работах [3, 19].

Выбор алгоритмов управления буфером методами машинного обучения. Были построены и изучены принципы построения самомодифицирующегося модуля постановки в очередь (queueing module, QM), который является одним из основных компонентов любого сетевого элемента, и выбор между возможными алгоритмами управления буфером был формализован как задача о многоруких бандитах. Введены несколько предположений, которые различают важные практические случаи: $\textsc{disj}(\Delta)$, $\textsc{avgeq}(\Delta)$, $\textsc{limit}(\Delta)$ и $\textsc{adv}(\Delta)$. Предложены алгоритмы на основе задачи о многоруких бандитах, изучена их эффективность при различных предположениях. Эти результаты были опубликованы в работе [6].

5.1.2 Алгоритмы классификации пакетов

Использование порядковой независимости для классификации пакетов. В этой части были найдены свойства классификатора, которые гарантируют нулевую стоимость добавления новых полей, представленных интер-

валами или префиксами; это добавление происходит прозрачным образом для схемы реализации, предложен способ сократить число полей классификации, представленных интервалами или префиксами, что приводит к семантически эквивалентным результатам классификации, а также представлен способ свести задачу классификации с несколькими полями к задаче классификации на не более чем двух полях с гарантированной скоростью работы в худшем случае и линейной памятью. Введено понятие порядковой независимости для фильтров, правил классификации и классификаторов. Показано, что $\mathcal{K}$ с ложноположительной проверкой одного подходящего правила является семантически эквивалентным представлением $\mathcal{K}^{+m}$, где $\mathcal{K}^{+m}$ — это классификатор, который получается из $\mathcal{K}$ добавлением m новых полей произвольной ширины, и если $\mathcal{K}^{-m}$ является порядково независимым, то $\mathcal{K}^{-m}$ с ложноположительной проверкой одного подходящего правила является семантически эквивалентным представлением $\mathcal{K}$, где $\mathcal{K}^{-m}$ — это классификатор, который получается из порядково независимого классификатора $\mathcal{K}$ удалением m полей, т.е. каждое правило $R = (I_1, \ldots, I_k)$ заменено на $R^{-m} = (I_1, \ldots, I_{k-m})$. На основе этих результатов мы ввели задачу Fields Subset Minimization (FSM): найти максимальное подмножество полей M порядково независимого классификатора $\mathcal{K}$, для которого $\mathcal{K}^{-M}$ остаётся порядково независимым; если таких максимальных подмножеств несколько, выбрать M с максимальной общей шириной (чтобы минимизировать ширину правил, остающихся для поиска). Введён ряд эвристических алгоритмов для решения задачи FSM, которые приводят к существенному сокращению размера классификаторов, часто на несколько порядков величины. Также было введено *мультигрупповое* (multi-group) представление, которое распределяет правила классификатора $\mathcal{K}$ по β группам так, что: (1) каждое правило принадлежит только одной группе; (2) правила в каждой группе порядково независимы на подмножестве из k полей классификатора $\mathcal{K}$ (кроме правила «catch-all»); (3) разные группы могут переиспользовать одни и те же поля, чтобы сохранять порядковую независимость. Это приводит к задачам оптимизации l-Groups of Rules (l-MGR), Maximum Rules Coverage (l-MRC) и (β, l)-MRC, решение которых даёт ещё более сильное сокращение размеров классификаторов и позволяет параллелизовать поиск в группах. Результаты этих алгоритмов сравниваются с булевой минимизацией на основе задачи *MinDNF*. Эти результаты были опубликованы в работах [14, 26, 42].

Мультигрупповые представления классификаторов пакетов. В этой части введённые выше мультигрупповые представления приводят к разработке альтернативных разложений для IPv6 FIB-таблиц, более эффективных и позволяющих более сильную параллелизацию, чем стандартное разложение по длинам префиксов. Вводятся две операции на классификаторах пакетов, а именно ограничение (ширины) и разбиение. Введена задача *минимального слабого (сильного) порядково независимого сведения*: для данного классификатора $\mathcal{K}$ (не обязательно порядково независимого) и константы l распределить правила $\mathcal{K}$ по минимальному числу непересекающихся групп, в котором разные группы префиксов могут быть основаны на подмножествах из не более чем l битовых индексов, и каждая группа является слабо (сильно) порядково независимой на этих битах. Показана связь между этой задачей и задачей о *минимальном тестовом множестве*: для тестового множества τ на X найти тестовое множество $Y \subseteq X$ с минимальным $|Y|$. Показано, что частный случай задачи минимального (слабого или сильного) порядково независимого сведения, в котором все правила из $\mathcal{K}$ состоят только из битов 0 и 1, эквивалентен задаче о минимальном тестовом множестве. Предложены несколько эвристик, которые можно использовать на практике для поиска минимальных сведений на основе процедуры OneGroup, которая может быть реализована по-разному, например методом Earliest Deadline First (EDF), выбирающем следующую группу жадно с MaxOI, используя введённую нами эвристику MaxPair или её обобщение, эвристику MinSimilar$_\delta$. Все эти эвристические алгоритмы приводят к большому сокращению размера классификаторов, и одним из главных практических применений этой части является (крайне актуальная для сетевых приложений) задача представления таблиц маршрутизации IPv6 на существующей инфраструктуре, предназначенной для IPv4. Эти результаты были опубликованы в работах [20, 34, 35].

Сокращение классификаторов пакетов на распределённых платформах. В этой части рассматривается задача представления FIB на распределённых платформах, где несколько сетевых карт (line-cards, LC) связаны между собой коммутационной матрицей. Каждая входящая (ingress, RX) LC должна отдельно поддерживать FIB таблицу (или её части), чтобы пересылать трафик на правильную исходящую (egress, TX) LC, которая пересылает его дальше

через исходящий порт. Фундаментальный вопрос здесь заключается в том, как эффективно реализовать FIB таблицу на такой распределённой платформе коммутирования. Изучено понятие эквивалентных представлений: для классификатора $\mathcal{K}$ с функцией классификации $f : \{0,1\}^w \to \mathcal{A}$ и набором битовых индексов $B \subseteq W$ классификатор $\mathcal{K}^{-B}$ с функцией классификации $g : \{0,1\}^{w-|B|} \to \mathcal{A}$ является *эквивалентным* представлением $\mathcal{K}$, если $f(H) = g(H^{-B})$ для любого заголовка $H \in \{0,1\}^w$. Введено новое свойство, *матч-эквивалентность*, предназначенное для улучшения эффективности неполных классификаторов. Это свойство ослабляет функцию классификации, сохраняя только индексы битов, которые сохраняют действие исходного классификатора $\mathcal{K}$ для любого заголовка, который покрывается $\mathcal{K}$ (подходит одному из его правил). Формально говоря, неполный классификатор $\mathcal{K}^B$ является *матч-эквивалентным* (ME) классификатору $\mathcal{K}^W$, $B \subseteq W$, если действие для любого подходящего заголовка H^W в $\mathcal{K}^W$ совпадает с действием, которое для соответствующего заголовка H^B выбирает $\mathcal{K}^B$. Если H^W не покрывается $\mathcal{K}^W$, то H^B может либо покрываться, либо не покрываться в $\mathcal{K}^B$. Сформулирована задача поиска минимального матч-эквивалентного классификатора: для данного классификатора $\mathcal{K}$ найти классификатор, матч-эквивалентный классификатору $\mathcal{K}$, чьи фильтры занимают минимальную общую память (в битах). Рассмотрены разные варианты матч-эквивалентности на основе трёх структурных свойств: порядковой независимости по фильтрам (относящемся к классификаторам, чьи фильтры все попарно не пересекаются), порядковой независимости по действиям (когда фильтры с одним и тем же действием могут пересекаться) и конфликтующих правил (когда никакие два правила с разными действиями не конфликтуют на заданном подмножестве битов). Введены эффективные эвристические алгоритмы для решения задачи о матч-эквивалентности для всех трёх свойств и показано, что они существенно сокращают размеры классификаторов в распределённых архитектурах. Изучена временная сложность этих эвристик: показано, что для классификатора с N правилами и w-битовыми заголовками вычислительная сложность MINME составляет $O(N^2 \cdot w^3) + O(w^2) \cdot T(\mathrm{SE})$, а вычислительная сложность PARMINME составляет $O\left(\beta N^2 \cdot w^3\right) + O(\beta w^2) \cdot T(\mathrm{SE})$, где $T(\mathrm{SE})$ — вычислительная сложность процедуры SE, которая вычисляет семантически эквивалентное представление с минимальной памятью. Эти результаты были опубликованы в работах [10, 24].

Классификаторы, переиспользующие одни и те же классы. В этом разделе рассматриваются другие альтернативные способы достичь ещё большей эффективности представления состояния политики классификации на плоскости данных (data plane), основанные на том факте, что похожие «паттерны классификации» (*классы*) переиспользуются в разных политиках; каждый класс состоит из тернарных фильтров, определяющих подмножество подходящих заголовков пакетов. Введён частичный порядок $\prec_P$ на классах в $\mathcal{C}_P$ на основе того, пересекаются ли они, и их положения в политике (классификаторе). Введена задача Policy Sequence Packing (PSP): для данного множества политик $\mathcal{P}$ найти последовательность классов $\mathbb{S}$, совместимую со всеми политиками из $\mathcal{P}$, которая минимизирует $\mathrm{W}^{+}(\mathbb{S})$. Показано, что PSP является NP-трудной даже для двух политик, $|\mathcal{P}| = 2$. Введены алгоритмы для PSP на основе взвешенного варианта задачи о разрезающем циклы множестве вершин (Weighted Feedback Vertex Set, WFVS), в том числе алгоритм AO, и показано, что AO корректно решает задачу PSP. Эти результаты были опубликованы в работах [3, 19].

Приближённая классификация пакетов. В этом разделе обобщается классическая *задача классификации пакетов* (точный случай), где каждое правило состоит из тернарной битовой строки и действия; поскольку правила могут пересекаться, у них есть приоритеты, и первое *подходящее* правило возвращается как результат классификации для каждого входящего пакета. Вводится *приближённое* представление классификаторов пакетов, позволяющее «мультиплексировать» несколько разных действий. Этот дополнительный уровень гибкости позволяет сократить требования к ресурсам, сохраняя требуемый уровень точности. Классификатор $\mathcal{K}$ является точным, если для каждого правила $R_i \in \mathcal{K}$ соответствующее множество действий $\mathcal{A}_i$ состоит из одного элемента; в противном случае $\mathcal{K}$ является *приближённым*. Введено понятие *оптимизационного процесса*, состоящего из набора *базовых операция* и *эвристики*, выбирающей последовательность применяемых операций (operation sequence). В приближённом случае $\mathbf{O}$ (множество операций, применимых к поданному на вход классификатору $\mathcal{K}$) может увеличиться по сравнению с точным случаем, так как обобщённые операции применимы чаще. В этом случае вводится алгоритм eLP для точного случая и алгоритм aLP, который расширяет его на приближённый случай. Изучена сложность этих алгоритмов: показано, что вре-

менная сложность aLP составляет $O(|\mathcal{K}| \cdot w \cdot l_{\max})$, где $l_{\max}$ — размер самого большого набора действий в исходном классификаторе $\mathcal{K}$, и что поглощения не могут быть применены к maLPM классификатору $\mathcal{K}_a$ или классификатору $\mathcal{S}(\mathcal{K}_a)$, где $\mathcal{S}$ — последовательность операций, состоящая из резолюций. Эти результаты были опубликованы в работах [1, 19].

5.1.3 Алгоритмы на плоскости управления

Алгоритмы для упрощения структуры сети. Здесь рассмотрено важное направление исследований, целью которого является упрощение управления сетью: разработка представлений исходной сети посредством более простой виртуальной сети, которые всё ещё сохраняли бы желаемые свойства исходной сети. Введено понятие *распределение полосы пропускания* как набора маршрутов вместе с конкретными полосами пропускания, аллоцированными вдоль этих маршрутов. На его основе введены понятия *эквивалентности по полосе пропускания* между двумя сетями, при которой полученный набор пропускных способностей рёбер должен допускать любое распределение полосы пропускания и маршрутизации, которое допускала исходная сеть, а также *эквивалентности по маршрутизации*, которая позволяет отображать совместные множества маршрутов (которые и называются *распределениями полосы пропускания*) между сетями с сохранением пропускных способностей. Показано, что для каждой сети G алгоритм грубой силы вычисляет единственную сеть минимальной пропускной способности G^*. Введён алгоритм DAG-OPT и показано, что он выдаёт сеть минимальной пропускной способности G^*, эквивалентную заданной сети (DAG) G, за время $O(|V||E|^2)$. Также введён более эффективный эвристический алгоритм WPP, основанный на локальных изменениях пропускных способностей, называющихся *пропагацией весов* (WP). Показано, что после обработки графа $G = (V, E, w, S, D)$ алгоритм WPP выдаёт граф $G' = (V, E, w', S, D)$, для которого $G' \simeq G$ и для всех $e \in E$ верно, что $w'(e) \leqslant w(e)$ и $w'(e) \leqslant \min\{I'_e, O'_e\}$, за время $O(|E| + |V| \log |V|)$. Более того, если $G = (V, E, w, S, D)$ — это граф, чья ненаправленная версия является лесом, то алгоритм WPP, применённый к G, выдаёт сеть минимальной пропускной способности G^*, эквивалентную G. Введён ряд преобразований топологии сети, сохраняющих эквивалентность по маршрутизации: стягивание некритического входящего ребра, стягивание некритического входящего ребра, слияние парал-

лельных рёбер, удаление петель. Показано, что сеть $G' = (V', E', w', S, D)$, полученная применением сохраняющих пропускную способность преобразований к сети $G = (V, E, w, S, D)$, является эквивалентной по маршрутизации сети G. Введён жадный алгоритм для применения этих преобразований, который показал отличные результаты на практике для сокращения топологий сети. Кроме того, был построен и изучен частный случай парадокса Браесса в этой постановке. Эти результаты были опубликованы в работе [18].

Формализация и таксономия задач типа compute-aggregate для облачных вычислений. В этой части рассматривается задача оптимизации выполнения задач типа compute-aggregate в распределённых вычислениях в предположении, что каждая задача типа compute-aggregate должна удовлетворять некоторым бюджетным ограничениям, поскольку разные пользователи облака могут инвестировать разные экономические ресурсы для вычисления своих агрегаций. Чтобы не перегружать самые быстрые сетевые связи, пересылка данных по разным рёбрам графа сети может также иметь разную цену. Задача теперь делится на две полностью разные фазы: (1) найти «самый дешёвый» план для данного распределения данных по сети, *функции агрегации* (которая вычисляет размер, получающийся после агрегации двух разных блоков данных) и цены отправки единицы данных по той или иной связи, а затем (2) перераспределить агрегации, вычисленные на первой фазе, оптимизируя при этом нужные целевые функции. Для первой фазы вводятся понятия *плана агрегации* и *функции агрегации* μ, которая является ключевой для полученной в итоге таксономии. Введена и изучена задача CAM (compute-aggregate minimization): для данного ненаправленного связного графа $G = (V, E)$, функции стоимости c, корневой вершины t, множества начальных блоков данных C и функции агрегации μ задача CAM$[\mu]$ состоит в том, чтобы найти план агрегации P, для которого минимизируется $\mathrm{cost}(P)$. Также рассматриваются два важных частных случая: TCAM (для деревьев) и CCAM (для полных графов). Показано, что если $P \neq NP$, то не существует полиномиального алгоритма с константным аппроксимационным фактором для задачи CAM без условия ассоциативности на μ, даже если ограничить G двумя вершинами. Показано, что эти задачи часто сводятся к поиску дерева минимального веса, соединяющего заданное множество вершин, то есть к задаче поиска минимального

дерева Штейнера MStT. Показано, что если каждая вершина в G содержит блок данных, то MStT можно решить точно за полиномиальное время, и если G является деревом, то MStT можно решить точно за полиномиальное время. Показано, что существуют полиномиальные алгоритмы, которые решают CCAM[μ] и TCAM[μ] на наборе блоков данных C с аппроксимационным фактором $\frac{W_C[\mu]}{w_C[\mu]}$. Показано, что существует функция агрегации μ, для которой $\forall a, b$ $\mu(a,b) \leqslant \min(a,b)$, и нет полиномиального алгоритма с константным аппроксимационным фактором для CCAM[μ] или TCAM[μ], если P≠NP. Показано, что существует оптимальный алгоритм для TCAM[min], работающий за полиномиальное время, и если существует полиномиальный α-приближённый алгоритм для задачи MStT, то существует полиномиальный 4α-приближённый алгоритм для задачи CAM[max], который мы называем `RECH_MStTMax`. Показано, что если существует $a > 0$, для которого $\mu(a,a) = a$, то задача CAM[μ] NP-трудна, и если P≠NP, то для её решения нет менее чем $\frac{19}{18}$-приближённых полиномиальных алгоритмов даже в случае, когда веса всех рёбер одинаковы. Все эти и другие результаты вместе приводят к таксономии аппроксимационных факторов для разных μ и разных вариантов задачи CAM. Эти результаты были опубликованы в работах [4, 21, 23, 30].

Эффективное распределение запросов в многосвязных микросетях. В этом разделе мы рассматриваем эффективное удовлетворение запросов в контексте многосвязных микросетей, где запрос может быть выполнен при помощи разных ресурсов генерации энергии: локальными, близлежащими или из региональной сети. Определён *вектор балансировки загрузки* (load balancing vector) для каждого запроса, который показывает, какие генераторы подключены и доступны для удовлетворения каждого запроса. Мы моделируем набор многосвязных микросетей при помощи системы коммутирования $(I, \mathcal{D})$, которая состоит из набора входов I (генераторов) с пропускными способностями портов c_i (их номинальная мощность) и набора запросов (demands) $\mathcal{D}$, которые нужно запланировать; запрос d характеризуется его длиной $l(d)$ (сколько он продлится), шириной $w(d)$ (уровнем мощности) и *вектором балансировки загрузки* (load balancing vector) $v(d)$, который содержит набор входящих портов, доступных для обработки d (набор генераторов, которые могут удовлетворить этот запрос). Будем предполагать, что $(I, \mathcal{D})$ — это направленный ацикличе-

ский двудольный граф. Введена процедура ChoosePortDemand, которая получает получает на вход текущее состояние (оставшиеся запросы и свободные мощности) и выдаёт пару вход–запрос (i, d) для следующего назначения; разные алгоритмы различаются своими процедурами ChoosePortDemand. Введено семейство алгоритмов Shared Greedy (SG), в том числе алгоритмы Shared Longest Demand (SLD) и Shared Longest Port (SLP). Доказан ряд нижних и верхних оценок на аппроксимационные факторы этих алгоритмов, В частности, показано, что для входа с максимальной длиной запроса L и минимальной длиной запроса l: (1) алгоритм из семейства SG с аппроксимационным фактором $\leqslant \alpha$ относительно конфигураций имеет аппроксимационный фактор $\leqslant \frac{\alpha L h}{L+(l-1)h}$ относительно длины, где h — оптимальное число конфигураций; (2) алгоритм из семейства SG с аппроксимационным фактором $\leqslant \alpha$ относительно длины имеет аппроксимационный фактор $\leqslant \frac{L}{t}\left(1+\frac{\alpha t-L}{l}\right)$ относительно конфигураций, где t — оптимальная длина. Показано, что для системы с одним общим портом и единичными пропускными способностями любая стратегия планирования ALG из семейства SG имеет аппроксимационный фактор относительно конфигураций не более $\frac{3}{2}$, а для системы с единичными мощностями и I портами, SLD имеет аппроксимационный фактор не менее $\left(\frac{3}{2}-2^{-(I-1)}\right)$ относительно конфигураций и длины. Показано, что для системы с мощностями $c_1 = 2c(\mathcal{I}_1)$ и $c(\mathcal{I}_2) = 2c_1$ число конфигураций в решении SLD не менее чем в $\frac{5}{3}$ раз превышает оптимальное, а число конфигураций в расписании SG не более чем в два раза превышает это число в оптимальном расписании, если выполняется одно из следующих условий: (1) $c_1 \leqslant c(\mathcal{I}_1)$; (2) $c_1 > c(\mathcal{I}_1)$ и $c_1 > c(\mathcal{I}_2)$. Показано, что SLP выдаёт оптимальное число конфигураций для случая различающихся входных мощностей и единичных мощностей запросов. Эти результаты были опубликованы в работах [40, 41].

Устойчивый к шуму подсчёт потоков трафика. В этой части мы рассматриваем задачу подсчёта размеров всех потоков на сетевом элементе, что может перегрузить недостаточные ресурсы на плоскости данных, когда число потоков становится слишком большим. Распределение этой задачи мониторинга может привести к неточному подсчёту из-за *сетевого шума* в форме переупорядочивания и/или потери пакетов. Мы вводим RoDiC (Robust Distributed Computation), общий метод, который корректно исполняет задачу мониторинга

на нескольких сетевых элементах, несмотря на сетевой шум на пути между элементами. Для задач с состоянием (stateful tasks) предлагаемый метод не только поддерживает состояние в нескольких сетевых элементах, но и устойчиво к шуму передаёт части состояния между этими элементами. Показаны пределы применимости: доказано, что детерминированный алгоритм RoDiC не может гарантировать корректное распределённое вычисление размера потока на двух коммутаторах, если $L \geqslant 1$ пакет и $R \geqslant 2^{n-1} + 1$ пакетов. С другой стороны, показано, что предложенный алгоритм всегда вычисляет размер потока на двух коммутаторах корректно, если $R \leqslant 2^{n-1} - 2^{n-t}$ пакетов и $L + R \leqslant 2^{n-1} - 1$ пакетов. Мы вводим понятие интервала $\mathrm{span}(\gamma, k)$, подпоследовательности пакетов на S, которые прибывают на D в том же порядке, пропускают не более чем γ полных групп в начале потока и пропускают не более чем $\gamma - 1$ полных последовательных групп после любого пакета в этой подпоследовательности. Мы представляем и анализируем алгоритм для вычисления размеров потоков на двух коммутаторах в интервальной модели и доказываем, что когда γ составляет не более $2^t - 1$ групп, предложенный алгоритм всегда вычисляет размер потока на двух коммутаторах корректно, если существует $\mathrm{span}(\gamma, n - t)$, и $R \leqslant 2^n - (\gamma + 1) \cdot 2^{n-t}$ пакетов. Эти результаты были опубликованы в работах [2, 15, 29].

5.2 Открытые проблемы для дальнейших исследований

Завершим диссертацию кратким обзором открытых задач, которые непосредственно относятся к теме диссертации, представляют как теоретический, так и практический интерес для сетевых приложений и могут быть решены в дальнейших исследованиях.

Один из важных ещё не решённых вопросов состоит в том, как формализовать *время отклика* (latency) в виде конечной целевой функции. Первый вариант состоит в том, чтобы не оптимизировать время отклика, а вводить ограничения на него, установив «допуск» (максимальную задержку) для каждого пакета. На уровне коммутатора такой подход называется моделью с *ограниченной задержкой* (bounded delay model); она изучалась теоретически в работе [134]. Это позволяет оптимизировать другие целевые функции, например пропускную способность, и вводить несколько характеристик пакетов (ценность, требования к времени обработки и т.д.). Важный недостаток этой модели состоит в том, что

она плохо подходит для принятия глобальных решений по управлению сетью. В частности, неясно, как разделить допуск данного потока между сетевыми элементами и пакетами в потоке. Отметим, что формализация на основе допусков, представленная в настоящей диссертации и работе [3, 19], работает только с отдельными пакетами, а работа на уровне потоков требует других целевых функций.

Эту проблему можно решить жадной оптимизацией времени отклика на каждом коммутаторе. Недавно предложенные сетевые транспорты *pFabric* и *pHost* оптимизируют так называемое *время окончания потока* (flow completion time, FCT) [49, 107]. При этом подходе всё ещё есть некоторая неясность с тем, как представлять FCT в конечной целевой функции и какая дополнительная информация нужна на уровне потоков. FCT определяется в [49] как разница между временем e_f, когда поток полностью обработан в приёмнике последнего пакета, и временем b_f, когда первый пакет прибывает на источник. Если нет мультиплексирования нескольких потоков (т.е. i-й пакет потока f прибывает на источник в момент времени $b_f + i$), и размеры потоков (в пакетах) известны алгоритму *a priori*, то можно ввести алгоритм с приоритетами SRPT (shortest remaining processing time), который оптимизирует среднее FCT, представленное как $e_f - b_f$, для каждого потока f, и *pFabric* основан на этом алгоритме [49]. Однако аналитических результатов об FCT в реалистичных постановках задач пока получить не удаётся; работа о *pFabric* [49] была вдохновлена работой [65], которая делает два основных предположения: что потоки *атомарны*, т.е. поток непрерывен и доступен сразу целиком, и что размеры всех потоков известны заранее. Оба этих предположения часто не выполняются на практике: в сети из коммутаторов задержки на предыдущем сетевом элементе могут привести к прерываниям потоков, прибывающих далее, и размер потока также часто неизвестен в тот момент, когда прибывает его первый пакет. Важной и интересной открытой проблемой является разработка теоретических моделей времени отклика в управлении буфером.

Другое интересное направление дальнейших исследований — автоматизация разработки стратегий управления по данным целевым функциям и характеристикам трафика, что может позволить автоматически адаптировать эти стратегии к поведению сети при помощи методов машинного обучения. Более того, было бы интересно определить уровень виртуализации поверх разнород-

ных вычислительных ресурсов, который распределял бы вычисления по данным характеристикам сервисов и целевой функции. В настоящей диссертации были сделаны первые шаги в этом направлении в работах [5, 22, 36], но это направление ещё далеко от своего завершения.

Литература

[1] V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Packet Classifiers With Controlled Accuracy. IEEE Transactions on Networking, vol. 29, no. 3, 2021, pp. 1141–1154.

[2] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. IEEE Transactions on Networking, vol. 29, no. 1, 2021, pp. 275–288.

[3] A. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Competitive buffer management for packets with latency constraints. Computer Networks, vol. 189, no. 22, Article ID 107942, 2021.

[4] P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalization and taxonomy of compute-aggregate problems for cloud computing applications. Computer Networks, vol. 189, no. 22, Article ID 107915, 2021.

[5] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. Towards Software-Defined Buffer Management. IEEE Transactions on Networking, vol. 28, no. 5, 2020, pp. 2337–2349.

[6] P. Chuprikov, S.I. Nikolenko, K. Kogan. Towards Declarative Self-Adapting Buffer Management. ACM SIGCOMM Computer Communication Review, vol. 50, no. 3, 2020, pp. 30–37.

[7] V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. IEEE Transactions on Networking, vol. 28, no. 3, 2020, pp. 1088–1101.

[8] P. Chuprikov, S.I. Nikolenko, K. Kogan, A. Davydow. Priority Queueing for Packets with Two Characteristics. IEEE Transactions on Networking, vol. 26, no. 1, 2018, pp. 342–355.

[9] P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Admission Control in Shared Memory Switches. Journal of Scheduling, vol. 21, no. 5, 2018, pp. 533–543.

[10] K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. Efficient FIB Representations on Distributed Platforms. IEEE Transactions on Networking, vol. 25, no. 6, 2017, pp. 3309–3322.

[11] P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Heterogeneous Packet Processing in Shared Memory Buffers. Journal of Parallel and Distributed Computing, vol. 99, 2017, pp. 1–12.

[12] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. The Impact of Processing Order on Performance: A Taxonomy of Semi-FIFO Policies. Journal of Computer and System Sciences, vol. 88, 2017, pp. 220–235.

[13] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub, M. Segal. Large profits or fast gains: A Dilemma in Maximizing Throughput with Applications to Network Processors. Journal of Network and Computer Applications, vol. 74, 2016, pp. 31–43.

[14] K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. Exploiting Order Independence for Scalable and Expressive Packet Classification. IEEE Transactions on Networking, vol. 24, no. 2, 2016, pp. 1251–1264.

[15] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Robust Distributed Monitoring of Traffic Flows. 27th IEEE International Conference on Network Protocols (ICNP 2019), 2019, pp. 1–11.

[16] V. Demianiuk, K. Kogan, S.I. Nikolenko. Approximate Classifiers with Controlled Accuracy. 38th IEEE International Conference on Computer Communications (INFOCOM 2019), 2019, pp. 2044–2052.

[17] V. Demianiuk, S.I. Nikolenko, P. Chuprikov, K. Kogan. New Alternatives to Optimize Policy Classifiers. 26th IEEE International Conference on Network Protocols (ICNP 2018), 2018, pp. 121–131

[18] S.I. Nikolenko, K. Kogan, A. Fernàndez Anta. Network Simplification Preserving Bandwidth and Routing Capabilities. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

[19] A.P. Davydow, P. Chuprikov, S.I. Nikolenko, K. Kogan. Throughput Optimization with Latency Constraints. Proc. 2017 IEEE International Conference on Computer Communications (INFOCOM 2017), 2017.

[20] P. Chuprikov, K. Kogan, S.I. Nikolenko. General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[21] P. Chuprikov, A.P. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Planning in Compute-Aggregate Problems as Optimization Problems on Graphs. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[22] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, A.V. Sirotkin, P. Eugster. A Programmable Buffer Management Platform. Proc. 25th IEEE International Conference on Network Protocols (ICNP 2017), 2017.

[23] P. Chuprikov, S.I. Nikolenko, K. Kogan. On Demand Elastic Capacity Planning for Service Auto-scaling. Proceedings of 2016 IEEE International Conference on Computer Communications (INFOCOM 2016), 2016.

[24] K. Kogan, S.I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich. FIB Efficiency in Distributed Platforms. Proc. 24th IEEE International Conference on Network Protocols (ICNP 2016), 2016.

[25] P. Chuprikov, K. Kogan, S.I. Nikolenko. Priority Queueing with Multiple Packet Characteristics. Proceedings of IEEE INFOCOM 2015, pp. 1418–1426.

[26] K. Kogan, S.I. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). ACM SIGCOMM 2014, ACM Press, 2014, pp. 15–26.

[27] P. Eugster, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Shared-Memory Buffer Management for Heterogeneous Packet Processing. 34th International Conference on Distributed Computing Systems (ICDCS 2014).

[28] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Online Scheduling FIFO Policies with Admission and Push-Out. Theory of Computing Systems, vol. 58, no. 2, 2016, pp. 322-344.

[29] V. Demianiuk, S. Gorinsky, S.I. Nikolenko, K. Kogan. Distributed Counting along Lossy Paths Without Feedback. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), Lecture Notes in Computer Science, vol. 11685, 2018, pp. 30–33 (CORE rank B).

[30] P. Chuprikov, A. Davydow, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Formalizing Compute-Aggregate Problems in Cloud Computing. 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018), 2018, pp. 377–391 (CORE rank B).

[31] P. Eugster, A. Kesselman, K. Kogan, S.I. Nikolenko, A.V. Sirotkin. Essential Traffic Parameters for Shared Memory Switch Performance. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015), Lecture Notes in Computer Science, vol. 9439, Springer, 2015, pp. 61–75. (CORE rank B).

[32] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. A Taxonomy of Semi-FIFO Policies. Proc. 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), 2012, pp. 295–304. (CORE rank B).

[33] K. Kogan, S.I. Nikolenko, V. Demianiuk, P. Chuprikov, A.P. Davydow. Personal Insights on Three Research Directions in Networked Systems. Proc. 10th International Conference on Communication Systems and Networks (COMSNETS 2018), IEEE, 2018.

[34] P. Chuprikov, K. Kogan, S.I. Nikolenko. How to implement complex policies on existing network infrastructure. Proc. 4th Symposium on SDN Research (SOSR 2018), ACM, 2018, pp. 1–8.

[35] S.I. Nikolenko, K. Kogan, G. Retvari, E.R. Berczi-Kovacs, A. Shalimov. How to Represent IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes. Proc. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): GI 2016: 9th IEEE Global Internet Symposium (IEEE GI 2016), 2016.

[36] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S.I. Nikolenko, P. Eugster. BASEL (Buffering Architecture SpEcification Language). Proc. 12th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2016), 2016, pp. 69–74.

[37] K. Kogan, S.I. Nikolenko, P. Eugster, E. Ruan. Strategies for Mitigating TCAM Space Bottlenecks. 22nd IEEE Annual Symposium on High-Performance Interconnects (HOTI 2014), IEEE, 2014, pp. 25–32.

[38] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, G. Scalosub. Balancing Work and Size with Bounded Buffers. Proc. 6th International Conference on Communication Systems and Networks (COMSNETS 2014), 2014, pp. 1–8.

[39] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin. Multi-Queued Network Processors for Packets with Heterogeneous Processing Requirements. Proc. 5th International Conference on Communication Systems and Networks (COMSNETS 2013), 2013, pp. 1–10.

[40] K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient Demand Assignment in Multi-Connected Microgrids with a Shared Central Grid. Proc. 3rd IFIP Conference on Sustainable Internet and ICT for Sustainability 2013 (SustainIT 2013), 2013.

[41] K. Kogan, S.I. Nikolenko, S. Keshav, A. Lòpez-Ortiz. Efficient demand assignment in multi-connected microgrids. Proc. 4th Energy-Efficient Computing and Networking (e-Energy 2013), pp. 277–278.

[42] K. Kogan, S.I. Nikolenko, W. Culhane, P. Eugster, E. Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013), pp. 153–154.

[43] R. Abe, H. Taoka, and D. McQuilkin. Digital grid: Communicative electrical grids of the future. *IEEE Trans. Smart Grid*, 2(2):399–410, 2011.

[44] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Transactions on Algorithms*, 5(1), 2008.

[45] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Trans. on Algorithms*, 5(1), 2008.

[46] W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén. Competitive queue policies for differentiated services. In *INFOCOM*, pages 431–440, 2000.

[47] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX*, pages 281–296, 2010.

[48] S. Albers and M. Schmidt. On the performance of greedy algorithms in packet buffering. *SIAM Journal on Computing*, 35(2):278–304, 2005.

[49] M. Alizadeh and et al. pfabric: minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446, 2013.

[50] E. Allender, L. Hellerstein, P. McCabe, T. Pitassi, and M. E. Saks. Minimizing DNF formulas and $\boldsymbol{AC}^0_d$ circuits given a truth table. In *Conference on Computational Complexity*, 2006.

[51] E. Allender, L. Hellerstein, P. McCabe, T. Pitassi, and M. E. Saks. Minimizing disjunctive normal form formulas and $\boldsymbol{AC}^0$ circuits given a truth table. *SIAM J. Comput.*, 38(1):63–84, 2008.

[52] N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies for qos switches. In *SODA*, pages 761–770, 2003.

[53] D. Applegate, G. Călinescu, D. S. Johnson, H. J. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *ACM-SIAM SODA*, 2007.

[54] J. Audibert and S. Bubeck. Best arm identification in multi-armed bandits. In *COLT*, 2010.

[55] J. Audibert, R. Munos, and C. Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876 – 1902, 2009. Algorithmic Learning Theory.

[56] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[57] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[58] Y. Azar and A. Litichevskey. Maximizing throughput in multi-queue switches. *Algorithmica*, 45(1):69–90, 2006.

[59] Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. *Algorithmica*, 43(1-2):81–96, 2005.

[60] Y. Azar and Y. Richter. An improved algorithm for CIOQ switches. *ACM Transactions on Algorithms*, 2(2):282–295, 2006.

[61] V. Bajaj. Ten interesting things about India power, 2012.

[62] M. Bando and J. H. Chao. Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps. In *IEEE INFOCOM*, pages 821–829, 2010.

[63] T. Banerjee-Mishra, S. Sahni, and G. S. Seetharaman. Pc-duos+: A tcam architecture for packet classifiers. *IEEE Trans. Computers*, 63(6):1527–1540, 2014.

[64] T. Banerjee-Mishra, S. Sahni, and G. S. Seetharaman. Pc-trio: A power efficient tcam architecture for packet classifiers. *IEEE Trans. Computers*, 64(4):1104–1118, 2015.

[65] A. Bar-Noy, M. M. Halldórsson, G. Kortsarz, R. Salman, and H. Shachnai. Sum multicoloring of graphs. *J. Algorithms*, 37(2):422–450, 2000.

[66] P. Berman, B. DasGupta, and M. Kao. Tight approximability results for test set problems in bioinformatics. *JCSS*, 71:145–162, 2005.

[67] S. Biswas. India struggles to deliver enough power, 2012.

[68] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[69] A. Bremler-Barr, D. Hay, and D. Hendler. Layered interval codes for TCAM-based classification. In *IEEE Conference on Computer Communications*, 2009.

[70] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using Gray coding. *IEEE Trans. Computers*, 61(1):18–30, 2012.

[71] S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in multi-armed bandits problems. In *Algorithmic Learning Theory*, pages 23–37, Berlin, Heidelberg, 2009.

[72] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[73] T. Carpenter, S. Singla, P. Azimzadeh, and S. Keshav. The impact of electricity pricing schemes on storage adoption in Ontario. In *e-Energy*, page 18, 2012.

[74] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[75] Y. Chang, C. Lee, and C. Su. Multi-field range encoding for packet classification in TCAM. In *IEEE Infocom Mini-Conference*, 2011.

[76] H. Che, Z. Wang, K. Zheng, and B. Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Trans. Computers*, 57(7):902–915, 2008.

[77] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz. The case for evaluating MapReduce performance using workload suites. In *MASCOTS*, pages 390–399, 2011.

[78] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN*, pages 73–82, 2009.

[79] Cisco 12000 series internet router architecture: Line card design. http://www.cisco.com/c/en/us/support/docs/routers/12000-series-routers/47242-arch12000-lcdesign.html.

[80] CISCO 12000 Series routers. http://www.cisco.com/en/US/products/hw/routers/ps167/index.html.

[81] Cisco CRS forwarding Processor Cards. http://www.cisco.com/c/en/us/products/collateral/routers/carrierroutingsystem/datasheetc78730790.pdf.

[82] Class of service configuration guide. https://www.juniper.net/documentation/en_US/junos11.1/information-products/topic-collections/security/software-all/ class-of-service/index.html.

[83] ClassBench: A packet classification benchmark. http://www.arl.wustl.edu/classbench/.

[84] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7(2):202–220, 1986.

[85] Configuring IP ACLs. http://www.cisco.com/en/US/docs/switches/datacenter/sw/4_1/nx-os/security/configuration/guide/sec_ipacls.pdf.

[86] Configuring modular QoS packet classification. http://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-2/qos/configuration/guide/qc42clas.html.

[87] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[88] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. LOOM: Optimal aggregation overlays for in-memory big data processing. In *HotCloud*, Philadelphia, PA, 2014. USENIX Association.

[89] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. Optimal communication structures for big data aggregation. In *INFOCOM*, pages 1643–1651, April 2015.

[90] A. Cvetkovski. An algorithm for approximate counting using limited memory resources. In *SIGMETRICS*, pages 181–190, 2007.

[91] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[92] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[93] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Inf. Process. Lett.*, 86(3):129–136, 2003.

[94] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood. Fast packet classification using bloom filters. In *ACM/IEEE ANCS*, 2006.

[95] A. Dixit, K. Kogan, and P. Eugster. Composing heterogeneous SDN controllers with flowbricks. In *ICNP*, pages 287–292, 2014.

[96] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *SIGMETRICS*, pages 311–322, 2006.

[97] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing optimal IP routing tables. In *INFOCOM*, pages 88–97, 1999.

[98] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *Computer Communication Review*, 34(2):97–122, 2004.

[99] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[100] A. Elmokashfi and A. Dhamdhere. Revisiting BGP churn growth. *CCR*, 44(1):5–12, 2014.

[101] A. Elmokashfi, A. Kvalbein, and C. Dovrolis. BGP churn evolution: A perspective from the core. *Trans. Networking*, 20(2):571–584, 2012.

[102] M. Englert and M. Westermann. Lower and upper bounds on FIFO buffer management in QoS switches. *Algorithmica*, 53(4):523–548, 2009.

[103] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, Aug. 2003.

[104] G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, Feb 1998.

[105] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[106] D. E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artif. Intell.*, 57(2-3):143–181, 1992.

[107] P. Gao and et al. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, pages 1–12, 2015.

[108] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to np-completeness, 1979.

[109] M. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.

[110] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 147–160, New York, NY, USA, 1999. ACM.

[111] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[112] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro*, 20(1):34–41, 2000.

[113] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[114] E. L. Hahne, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. In *SPAA*, pages 53–58, 2001.

[115] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[116] C. Hong and et al. Achieving high utilization with software-driven WAN. In *SIGCOMM*, pages 15–26, 2013.

[117] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, Y. Cheng, and H. Wu. Discount counting for fast flow statistics on flow size and flow volume. *IEEE/ACM Trans. Netw.*, 22(3):970–981, June 2014.

[118] S. Jain and et al. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, pages 3–14, 2013.

[119] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *AISTATS*, volume 51, pages 240–248, Cadiz, Spain, 09–11 May 2016.

[120] C. Kaklamanis, M. Chlebík, and J. Chlebíková. Algorithmic aspects of global computing the steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207 – 214, 2008.

[121] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*, pages 13–24, 2013.

[122] Y. Kanizo, D. Hay, and I. Keslassy. Optimal fast hashing. In *IEEE Infocom*, 2009.

[123] N. Kathleen and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[124] J. Kawahara, K. Kobayashi, and T. Maeda. Tight analysis of priority queuing policy for egress traffic. *CoRR*, abs/1207.5959, 2012.

[125] S. Keshav and C. Rosenberg. On load elasticity. In *Proc. IEEE COMSOC MMTC E-Letter*, To Appear.

[126] I. Keslassy and et al. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[127] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. In *INFOCOM*, pages 3191–3199, 2011.

[128] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[129] A. Kesselman and K. Kogan. Nonpreemptive scheduling of optical switches. *IEEE Transactions on Communications*, 55(6):1212–1219, 2007.

[130] A. Kesselman, K. Kogan, and M. Segal. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.

[131] A. Kesselman, K. Kogan, and M. Segal. Improved competitive performance bounds for CIOQ switches. *Algorithmica*, 63(1-2):411–424, 2012.

[132] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *STOC*, pages 520–529, 2001.

[133] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. *SIAM J. Comput.*, 33(3):563–583, 2004.

[134] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[135] A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. *J. Algorithms*, 46(1):79–95, 2003.

[136] A. Kesselman and Y. Mansour. Harmonic buffer management policy for shared memory switches. *Theor. Comput. Sci.*, 324(2-3):161–182, 2004.

[137] A. Kesselman and Y. Mansour. Harmonic buffer management policy for shared memory switches. *Theor. Comput. Sci.*, 324(2-3):161–182, 2004.

[138] A. Kesselman, Y. Mansour, and R. van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43(1-2):63–80, 2005.

[139] A. Kesselman and A. Rosén. Scheduling policies for CIOQ switches. *Journal of Algorithms*, 60(1):60–83, 2006.

[140] A. Kesselman and A. Rosén. Controlling CIOQ switches with priority queuing and in multistage interconnection networks. *Journal of Interconnection Networks*, 9(1/2):53–72, 2008.

[141] S. Khot and R. Saket. Hardness of minimizing and learning DNF expressions. In *FOCS*, pages 231–240, 2008.

[142] S. Khot and R. Saket. Hardness of minimizing and learning DNF expressions. In *IEEE FOCS*, 2008.

[143] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[144] J. M. Kleinberg and É. Tardos. *Algorithm design*. Add.-Wesley, 2006.

[145] K. Kobayashi, S. Miyazaki, and Y. Okabe. Competitive buffer management for multi-queue switches in QoS networks using packet buffering algorithms. In *Proceedings of the* 21st *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 328–336, 2009.

[146] K. Kogan, A. Dixit, and P. Eugster. Serial composition of heterogeneous control planes. In *ONS*, 2014.

[147] K. Kogan, A. Lòpez-Ortiz, S.I. Nikolenko, A.V. Sirotkin, and D. Tugaryov. FIFO Queueing Policies for Packets with Heterogeneous Processing. In *MedAlg 2012*, LNCS vol. 7659, pages 248–260, 2012.

[148] A. Korösi, J. Tapolcai, B. Mihalka, G. Mészáros, and G. Rétvári. Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously. In *IEEE ICNP*, 2014.

[149] D. V. Krioukov, K. C. Claffy, K. R. Fall, and A. Brady. On compact routing for the internet. *Comput. Commun. Rev.*, 37(3):41–52, 2007.

[150] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[151] N. Lesh and M. Mitzenmacher. Bubblesearch: A simple heuristic for improving priority-based greedy algorithms. *Inf. Process. Lett.*, 97(4):161–169, 2006.

[152] K. Lim, K. Park, and H. Lim. Binary search on levels using a bloom filter for ipv6 address lookup. In *ACM/IEEE ANCS*, pages 185–186, 2009.

[153] Cisco XR 12000 series gigabit ethernet line cards. https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html.

[154] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 18(2):490–500, 2010.

[155] A. X. Liu, C. R. Meiners, and Y. Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *IEEE Infocom*, 2008.

[156] H. Liu. Efficient mapping of range classifier into Ternary-CAM. In *IEEE Hot Interconnects*, 2002.

[157] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[158] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[159] Z. Lotker and B. Patt-Shamir. Nearly optimal FIFO buffer management for two packet classes. *Computer Networks*, 42(4):481–492, 2003.

[160] B. Malakooti. *Operations and Production Systems with Multiple Objectives*. John Wiley & Sons, 2013.

[161] Y. Mansour, B. Patt-Shamir, and O. Lapid. Optimal smoothing schedules for real-time streams (extended abstract). In *PODC*, pages 21–29, 2000.

[162] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 20(2):488–500, 2012.

[163] D. Meyer, L. Zhang, and K. Fall. Report from the IAB workshop on routing and addressing. http://www.ietf.org/internet-drafts/draft-iab-raws-report-02.txt, 2007.

[164] A. K. Mishra, D. E. Irwin, P. J. Shenoy, J. Kurose, and T. Zhu. SmartCharge: cutting the electricity bill in smart homes with energy storage. In *e-Energy*, page 29, 2012.

[165] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, 2016.

[166] Netlogic Microsystems. Content addressable memory. http://www.netlogicmicro.com.

[167] S.I. Nikolenko, K. Kogan. Single and Multiple Buffer Processing. *Encyclopaedia of Algorithms*, Springer, 2014, pp. 1–9.

[168] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE JSAC*, 17(6):1083 –1092, 1999.

[169] OpenFlow 1.3 specification, 2012. http://www.openflow.org/wk/index.php/OpenFlow_1_3_proposal.

[170] J. B. Orlin. Max flows in o(nm) time, or better. In *STOC*, pages 765–774, 2013.

[171] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21(3):629–656, 1996.

[172] Pipelined packet switching and queuing architecture. US7643486B2, 2010.

[173] QoS: Modular QoS command-line interface configuration guide, cisco ios. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_mqc/configuration/ 15-mt/qos-mqc-15-mt-book/qos-mqc.html.

[174] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing ip forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM*, 2013.

[175] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[176] O. Rottenstreich, A. Berman, Y. Cassuto, and I. Keslassy. Compression for fixed-width memories. In *IEEE ISIT*, 2013.

[177] O. Rottenstreich and I. Keslassy. On the code length of TCAM coding schemes. In *IEEE ISIT*, 2010.

[178] O. Rottenstreich and I. Keslassy. Worst-case TCAM rule expansion. In *IEEE Infocom Mini-Conference*, 2010.

[179] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. On finding an optimal TCAM encoding scheme for packet classification. In *IEEE Infocom*, 2013.

[180] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim. Compressing forwarding tables for datacenter scalability. *IEEE Journal on Selected Areas in Communications*, 32(1):138–151, 2014.

[181] H. Saitoh and J. Toyoda. A new concept of electric power network for the effective transportation of small power of dispersed generation plants. *IEEE J. Trans. PE*, 115(6):568–575, 1995.

[182] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.

[183] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster IP route lookup. In *IEEE ICNP*, pages 358–367, 2005.

[184] H. Song and J. S. Turner. ABC: Adaptive binary cuttings for multidimensional packet classification. *IEEE/ACM Trans. Netw.*, 21(1):98–109, 2013.

[185] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[186] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perf. Eval. Rev.*, 26(1):1–10, 1998.

[187] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *SIGMETRICS*, 1998.

[188] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM*, 1998.

[189] R. Stanojevic. Small active counters. In *INFOCOM*, pages 2153–2161, 2007.

[190] R. Steinberg and W. I. Zangwill. The prevalence of braess' paradox. *Transportation Science*, 17(3):301–318, 1983.

[191] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[192] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, first edition, 1998.

[193] O. R. T. Mizrahi and Y. Moses. TimeFlip: scheduling network updates with timestamp-based TCAM ranges. In *IEEE INFOCOM*, 2015.

[194] E. Tsidon, I. Hanniel, and I. Keslassy. Estimators also need shared values to grow together. In A. G. Greenberg and K. Sohraby, editors, *INFOCOM*, pages 1889–1897. IEEE, 2012.

[195] C. Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4):597–611, 2001.

[196] B. Vamanan and T. N. Vijaykumar. TreeCAM: decoupling updates and lookups in packet classification. In *ACM CoNEXT*, 2011.

[197] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, 2010.

[198] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann, 2005.

[199] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *ACM SIGCOMM*, 1997.

[200] Z. Wang, H. Che, M. Kumar, and S. K. Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Trans. Computers*, 53(12):1602–1614, 2004.

[201] R. Wei, Y. Xu, and H. J. Chao. Block permutations in Boolean space to minimize TCAM for packet classification. In *INFOCOM*, pages 2561–2565, 2012.

[202] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[203] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *SIGCOMM*, pages 123–134, New York, NY, USA, 2013.

[204] World's first digital grid router controls electricity packets, boosts grid access for renewables in Japan, 2013.

[205] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou. Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *Companion Proc. of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 44–53, New York, NY, USA, 2014. ACM.

[206] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM 2018*, Forthcoming.

[207] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. In *SIGCOMM*, 2014.

[208] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, Aug. 2017.

[209] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[210] Y. Zhang and N. Ansari. On architecture design, congestion notification, TCP incast and power consumption in data centers. *IEEE Communications Surveys and Tutorials*, 15(1):39–64, 2013.

[211] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the aggregatability of router forwarding tables. In *IEEE INFOCOM*, 2010.

[212] A. Zhu. Analysis of queueing policies in QoS switches. *J. Algorithms*, 53(2):137–168, 2004.

[213] G. Zhu, S. Yu, and J. Dai. Binary search on prefix covered levels for IP address lookup. In *WiCOM*, pages 3859–3862, 2009.