ST. PETERSBURG DEPARTMENT OF V.A.STEKLOV INSTITUTE OF MATHEMATICS OF THE RUSSIAN ACADEMY OF SCIENCES

As a manuscript

Pavel Sergeyevich Chuprikov

THEORETICAL AND EMPIRICAL ANALYSIS OF FUNDAMENTAL BOTTLENECKS IN NETWORKING AND DISTRIBUTED COMPUTING

PhD Dissertation

for the purpose of obtaining academic degree
Doctor of Philosophy in Computer Science HSE

Academic supervisors:
Researcher, PhD, Sergey I. Nikolenko
Research Assistant Professor, PhD, Kirill Kogan

St. Petersburg — 2019

Contents

1 Introduction

This section begins with a brief introduction into the current limitations of big data processing in relation to computer networks and distributed data processing, which are the primary topics of analysis in this dissertation, and then states the main goals and objectives of the research.

1.1 Dissertation topic and its relevance

Over the last decades, the volume and variety of data which is necessary to transmit and process has been increasing exponentially. Between 2013 and 2020, the global data volume is predicted to grow exponentially, from 4.4 to 44 zettabytes [1]. The expression "big data" usually refers to datasets whose size is beyond the ability of typical computing platforms to store and analyze. Due to these storage and computational limitations, data chunks should be *distributed* over multiple locations for big data processing. In this case, a network *interconnects* distributed instances of big data applications.

Big data both aggravates classical challenges of networking and presents new difficulties. For example, as one consequence of data growth, traditional cloud computing is increasingly replacing dedicated hardware by dynamically allocated virtual computing resources. They are usually paid based on allocation rather than actual use, which often leads to customers paying more than necessary. Recently, *serverless* architectures have exploded in popularity, and many major players in the cloud computing market introduced serverless solutions: AWS Lambda [2], Google Cloud Functions [3], Azure Functions [4], and others.

Although serverless computing is only a special case of cloud computing and is not able to cover every desired functionality, it remains an important representative case providing cost savings to the end user. In particular, the "function as a service" paradigm brings new challenges to resource allocation. Recent works have only begun to pose these questions for serverless computing, studying it in the context of queueing theory [5] and measuring relevant metrics for already existing solutions [6]. The present thesis takes the next step towards a theory of serverless cloud computing, proposing a comprehensive and novel formalization of the resource allocation model for serverless computing and presenting new resource allocation algorithms supported by both theoretical analysis and a comprehensive evaluation study.

In addition to requiring more flexible resource allocation methods, exascale data amounts and delay-sensitive objectives of big data applications impose significant feasibility constraints on the cloud computing paradigm. One particular limitation is in deploying a network solely as a simple interconnect. Although traditional networks mostly operate on partial information from data streams (packet headers), in some cases network elements are already able to process entire data streams at line-rate (e.g., in case of encryption). Therefore, exposing the structural layout of packet payloads can allow preprocessing of transmitted data before it is actually processed by cloud computing resources. According to [7, 8], the average final output size jobs is 40.3% of the initial data sizes in Google, 8.2% in Yahoo, and 5.4% in Facebook. The present thesis exploits these properties of data streams and aims to reduce processing load on traditional cloud computing resources. We use two major innovative approaches to achieve this goal: (1) intermediate in-network aggregations of data streams and (2) in-network data processing on the data plane of network elements. Implementations of both approaches extend existing network interconnects.

Finally, the challenges also propagate down to the level of an individual network element. The two basic functions of a network element are *packet classification*, i.e., deciding which action to take based on a packet's header, and *buffer management*, i.e., making admission, processing, and transmission decisions for incoming packets. The present thesis advances both.

1.2 Goals and objectives of the research

In this section, we define the specific objectives that we set for the research outlined in the thesis. We define four major areas of the results, proceeding from the level of an individual network element up to the level of large-scale distributed computing: packet classification, buffer management, cost-efficient serverless computing, and intermediate data aggregations.

Buffer management: multiple packet characteristics. When *multiple data streams* meet at a network element, the corresponding data packets are stored in a single network buffer waiting for processing and transmission. Buffers are controlled by *buffer management policies*, which play a major role in optimization of the desired objectives. Advanced economic models bring novel objectives, such as weighted throughput, which are largely not supported by existing policies. In addition, the heterogeneity in processing requirements exacerbated by in-network data processing is also not accounted for. The previous works consider only one of the two packet characteristics: in [9, 10] authors study behavior of a single queue with heterogeneous processing requirements, while [11] addresses a multi-valued case for FIFO queues; in addition, more involved buffering architectures [12] (combined input-output switches) and [13] (crossbar switches) have been considered for multi-valued packets. The interaction of the two characteristics and their impact on weighted throughput optimization has not been studied before. In order to provide theoretical guidance for the optimal choice of buffer management policy in such a setting for a single queue buffering architecture, this dissertation aims to: (1) build a model capturing both packet weights and processing; (2) devise new management policies with worst-case performance guarantees; (3) evaluate their performance on realistic data traces.

Packet classification: inter-device resource sharing and LPM infrastructure reuse. The basic function of *single packet processing* at a network element is *packet classification*. General-purpose data stream processing brings new flexibility requirements and adds more complexity to the policies implemented through the packet classification. Current single-switch approaches consist of software-based solutions and hardware-based solutions. The former include decision tree structures [14, 15], hashing [16], and encoding [17], in general they require modification to the classification algorithm. The latter rely on *ternary content-addressable memory (TCAM)* and optimize its space usage [18, 19] and, in particular, range encoding [20]. On the network-wide level the two existing works [21, 22] present algorithms that duplicate rules and are hard computationally. It is highly desirable to support the added complexity *without* expensive infrastructure upgrade, e.g., in the form of additional TCAMs or complex logic changes. This includes two very specific objectives: (1) first, it should be made possible to implement general ternary bit string classification using traditional *Longest-Prefix-Match (LPM)* representation in an implementation-agnostic way; (2) second, already rising to the network-wide level, an efficient inter-

device resource sharing scheme is necesssary to implement large policies (classifiers) when rule capacity is limited in each network element.

Serverless computing: cost-efficient resource allocation. The first essential step towards cost-efficient serverless computing is to construct a model that would realistically capture the costs and constraints of resource management. These include *allocation cost*, *maintenance cost*, and, most importantly, the *delay* that resource allocation incurs. For the resulting model, it is then necessary to construct efficient algorithmic solutions that would require no assumptions regarding the arrival patterns of requests and would have the performance guarantees suitable for an economic setting. In contrast, existing rule-based solutions [23] require deep knowledge of arrival behavior, while learning-based techniques [24] assume the behavior similar to that seen during the training. Last but not least, it is important to control the latency the proposed solutions add to the request processing, and understand how it affects the revenue.

Data aggregation: planning with both network infrastructure and applications. In cloud computing, the network infrastructure with its constraints and objectives and an application with its specific behavior and its own set of objectives seldom interact in a meaningful way, and there is little information exchange between them. For instance, an application aggregating too much data at a single node may cause performance degradation due to TCP-incast [25] or link overload. To exploit intermediate data aggregations efficiently, one has to find a way for the two layers to cooperate. The most relevant work [26] is tied to a specific network topology; [27] introduces implementation support for intermediate aggregations, and [28] is concerned exclusively with the processing latency and does not take network infrastructure into account. The specific challenge addressed in the present thesis is to find the minimal necessary information required to share from each layer that would allow for unified design principles of aggregation planning.

2 Key results and conclusions

This section summarizes the key contributions of this thesis, addressing specific objectives introduced above. Here we provide a brief list of key results, and then will expand upon each research direction in more detail in Section 4.

2.1 New buffer management settings and policies

Buffer management with two packet characteristics. We analyze the buffer management problem for a single queue that stores packets heterogeneous *both* in their relative values (rewards) and the amount of required processing [29]. Previous works [9] addressed either one or the other of the two characteristics. We prove lower bounds on the competitive ratio (see [30]) for several natural priority-based algorithms, including a constant general lower bound shown for an arbitrary online algorithm. The main result of this part is a $\left(1 + \frac{W+2}{V}\right)$ upper bound for the special case with only two possible values; here W is the maximal amount of required processing, and the two possible values are 1 and V. The proof is based

on an inductive argument with a per-packet alignment between optimal and priority-based algorithms (similar to that in [9]), augmented by special treatment of packets with value 1.

2.2 New packet classification schemes and algorithms

Ternary to LPM transformation algorithms. We have designed two algorithms, `MinGroupPartition` and `MaxCoveragePartition`, for the task of transforming a general ternary bit-string packet classifier into an *equivalent group* of more constrained LPM classifiers [31]. The key part of this result is a novel connection between the ability to construct complex lookup keys and order theory, specifically Dilworth's decomposition theorem [32]. We show that `MinGroupPartition` minimizes the total number of groups, while `MaxCoveragePartition` maximizes the number of rules covered by a given number of LPM groups. These algorithms prove it possible to represent expressive ternary bit-string classification rules using traditional longest-prefix match capabilities of existing network infrastructures.

Complex network-wide packet classification. To improve classification across multiple network elements, we design a simple and space-efficient scheme, called `OneBit`, for distributing data flow (stream) classification rules along a flow's path [33]. `OneBit` relies on just a single bit of packet metadata and, compared to previous work, does not employ any heuristics, is very computationally easy (linear in the number of rules), and allows for a polynomial time decision procedure for the feasibility of multi-flow distribution. Similarly to the ternary-to-LPM transformation algorithms above, it allows to implement very complex policies without any network upgrade.

2.3 New cost-efficient resource allocation for serverless computing

Elastic resource allocation. We have explored the trade-off between delaying user requests and using available resources efficiently for the elastic resource allocation problem [34]. The proposed model incorporates an allocation cost α, a maintenance cost β, $\beta < 1$, and the time allocation takes (normalized to 1). We have shown a simple greedy algorithm `NRAP` to be at most $2 \cdot \left(1 + \frac{\alpha}{1-\alpha-\beta}\right)$-competitive for $\alpha < 1 - \beta$. For $\alpha \geq 1 - \beta$, we present a parameterized algorithm ρ-`AAP` that actively delays requests; we prove that ρ-`AAP` is $\frac{\rho}{1-\rho}\left(\left\lceil \frac{\rho\alpha}{1-\beta} \right\rceil + 1\right)$-competitive (with a small amount of extra buffer space). Compared to previous works [35], which largely relied either on past history or on a deep understanding of request arrivals, the guarantees we provide in this part of the dissertation are worst-case and thus hold for any arrival sequence.

2.4 New data aggregation schemes for compute-aggregate tasks

Compute-aggregate task planning. To optimize the placement of intermediate aggregations during the execution of compute-aggregate tasks, we formally introduce the *compute-aggregate minimization (CAM)* problem [36]. The formulation succinctly captures the characteristics of both the network (topology and transmission costs) and the application (aggregation size function). We argue that these characteristics are sufficient for the construction of an *aggregation plan* that effectively exploits intermediate data aggregations and minimizes the overall transmission cost. In particular, as part of this dissertation,

a hardness result was established for the non-associative case, and the relation between CAM and the minimum Steiner tree was characterized. Moreover, the aggregation plan, as it is defined in the CAM problem, does not restrict when exactly transmissions and aggregations happen; it is conveniently decoupled both from the network and from the application. Compared to previous work exploiting in-network aggregation [26], the treatment we present in this thesis is more general (in particular, it does not restrict network topology) and, apart from specific new results, provides a novel general framework for reasoning about compute-aggregate task planning based on the notion of the aggregation size function.

3 Publications and approbation of the research

Each of the key results presented in the previous section has been published in one of the peer-reviewed research papers listed below.

First-tier publications:

- Chuprikov P., Nikolenko S. I., Davydow A., Kogan K. Priority Queueing for Packets with Two Characteristics* // IEEE/ACM Transactions on Networking. 2018. vol. 26 (1). pp. 342-355. The content is in Appendix A.

  Contribution of the disseration's author: Theorems 1–3 characterizing the behaviour of priority-queue (PQ) based policies; Theorem 7 establishing a general lower bound; Theorem 11 presenting lower bounds for PQ-based policies in a two-valued case; Theorem 12 showing an upper bound for $\mathtt{PQ}_{v,-w}$ in the two-valued case; and Theorem 17 with lower bounds for PQ-based policies with β-pushout.

- Chuprikov P., Kogan K., Nikolenko S. General Ternary Bit Strings on Commodity Longest-prefix-match Infrastructures* // Proceedings of IEEE ICNP 2017. The content is in Appendix B.

  Contribution of the dissertation's author: Theorem 1 establishing a necessary and sufficient condition for prefix-reorderability; the `PrefixToLPM` algorithm with a proof of correctness in Theorem 2; the `MinGroupPartition` algorithm with a proof of correctness in Theorem 3; Theorem 4 showing a hard example for MinGR; the `MaxCoveragePartition` algorithm with a proof of correctness in Theorem 5; bit-expanding heuristic described in Section V; Observations 3 and 4 linking prefix-reorderability with rule disjointness; implementation of the above algorithms.

- Chuprikov P., Nikolenko S., Kogan K. On Demand Elastic Capacity Planning for Service Autoscaling* // Proceedings of IEEE INFOCOM 2016. The content is in Appendix D.

  Contributions of the dissertation's author: Theorems 1 and 2 stating non-competitiveness in a bufferless setting; A lower bound in a buffered setting with small allocation cost in Theorem 3; the `NRAP` algorithm and its competitiveness in Theorems 4 and 5; the ρ-`AAP` algorithm and its analysis in Theorems 6 and 7; Theorem 8–11 providing latency guarantees for `NRAP` and ρ-`AAP`; Theorems 12 and 13 presenting general lower bounds for a bounded-delay (BD) model; Theorem 14

*The author of the dissertation is the main author of the paper

that upper-bounds NRAP in the BD model; Theorem 15 with a general lower bound for limited resources BD model (LBD); Theorems 16 and 17 providing bounds for NRAP in LBD model;

Second-tier publications:

- Chuprikov P., Davydow A., Kogan K., Nikolenko S. I., Sirotkin A. Formalizing Compute-Aggregate Problems in Cloud Computing // Proceedings of SIROCCO 2018. The content is in Appendix E.

  Contributions of the disseration's author: formalization of a compute-aggregate task planning problem and nonassociative hardness (Section 3 of the paper); Theorems 2, 6 and 7 connecting CAM to the minimum Steiner tree problem (Sections 4.1 and 4.2 of the paper).

Other publications:

- Chuprikov P., Kogan K., Nikolenko S. I., How to implement complex policies on existing network infrastructure* // Proceedings of ACM SOSR 2018. The content is in Appendix C.

  Contributions of the dissertation's author: the OneBit algorithm and its performance guarantees in Theorem 5.2; a network-wide solution presented in Section 6; performance comparison with existing algorithms.

The obtained results are supported in two ways. First, in almost all settings considered in this work we present rigorous proofs of worst-case performance guarantees valid under a realistic model. Second, for practical purposes where worst-case behavior might be rare, we evaluate proposed solutions on synthetic traces. Buffer management policies (Section 4.1) were run under traffic based on CAIDA traces [37]. Classifier transformation algorithms (Section 4.2) have been evaluated on the *Classbench* test suite [38]. Resource allocation algorithms (Section 4.3) were tested using randomly generated network-traffic-like pattern of arrivals [39].

4 Contents

This Section provides an overview of the research projects that led to the results presented in Section 2 and describes how these results achieve the objectives stated in Section 1.2. Again, we begin with the level of an individual network element, presenting our results on buffer management and packet classification, and then proceed upwards to resource allocation for serverless computing and compute-aggregate task planning.

4.1 Processing of multiple data streams

Interconnecting infrastructure should support advanced economic models expressed through new types of objectives. Buffer management policies play a critical role in the optimization of those objectives. Network traffic is highly heterogeneous, and its characteristics have an impact on the process of optimization, but they are often not being taken into account by buffer management policies. Introduction of in-network data processing would bring even more heterogeneity into processing requirements, making the current policies even less efficient. In response to new challenges, the paper titled “Priority Queueing

for Packets with Two Characteristics" (see Appendix A and [29]) studies the effect of two packet characteristics, namely, processing requirements and value, on the optimization of weighted throughput in a single queue. This study leads to the first set of results from Section 2, and their overview is presented in this section.

The considered model assumes a single queue that is able to hold B unit-sized packets and that all arriving packets are unit-sized. Each arriving packet p has two characteristics: processing requirements *(work)* $w(p) \in \{1, \ldots, W\}$, and value $v(p) \in \{1, \ldots, V\}$, we will sometimes denote such p as $(w(p) \mid v(p))$. Both $w(p)$ and $v(p)$ are known at arrival. The time is assumed to be slotted, and each time slot t is subdivided intro three phases: (1) *admission*, when a set of new packets arrives, and the management policy decides which packets to admit to the queue, possibly *pushing out* already admitted ones; (2) *processing*, when a single packet from the queue is scheduled for processing; and (3) *transmission*, when a single *fully processed* packet, i.e., a packet p that has been scheduled for processing at least $w(p)$ times, is chosen for transmission. Note, the results that follow impose no constraints on the transmission order, so the transmission decision is trivial: transmit the only fully processed packet if any. The goal is to find a buffer management algorithm that would maximize the total weighted throughput, i.e., the total value of all transmitted packets.

Unfortunately, due to the online nature of the problem it is impossible to find an algorithm producing an optimal solution for every sequence of requests. The performance guarantees are, thus, provided relative to an unknown optimal solution using competitive analysis [30]. An *online* algorithm ALG is called α-competitive for some $\alpha \geq 1$ iff for *any* finite arrival sequence σ the total weighted throughput is at least $1/\alpha$ times the total weighted throughput of an optimal *offline* algorithm OPT.

Algorithms. The previous works have demonstrated that if either $V = 1$ or $W = 1$, then simple priority-based algorithms that prefer either higher value or lower processing requirement are, in fact, optimal [9]. For that reason, priority-based algorithms are natural candidates for the model described above. The difference is that here the choice of the priority (the admission/processing order) is not obvious. A generalized notion of a priority-queue-based algorithm is defined next.

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

- $\mathtt{PQ}_f$ *is greedy, i.e., it accepts incoming packets as long as there is space in a buffer;*
- $\mathtt{PQ}_f$ *is work-conserving, i.e., it processes a packet as long as its buffer is not empty;*
- $\mathtt{PQ}_f$ *orders and processes packets in its queue in the order of decreasing values of f;*
- $\mathtt{PQ}_f$ *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: $f(p) = \min_{q \in \mathrm{IB}^{\mathtt{PQ}_f}} f(q)$, and $f(p') > f(p)$. Here $\mathrm{IB}^{\mathtt{PQ}_f}$ denotes the set of packets in $\mathtt{PQ}_f$'s buffer on the current timeslot.*

In short, on admission $\mathtt{PQ}_f$ considers new packets and buffered packets together keeping those with higher value of f; for processing $\mathtt{PQ}_f$ chooses a packet p with the highest $f(p)$. There are three promising candidates for the priority function f; in what follows, v and w denote, respectively, value and *current*

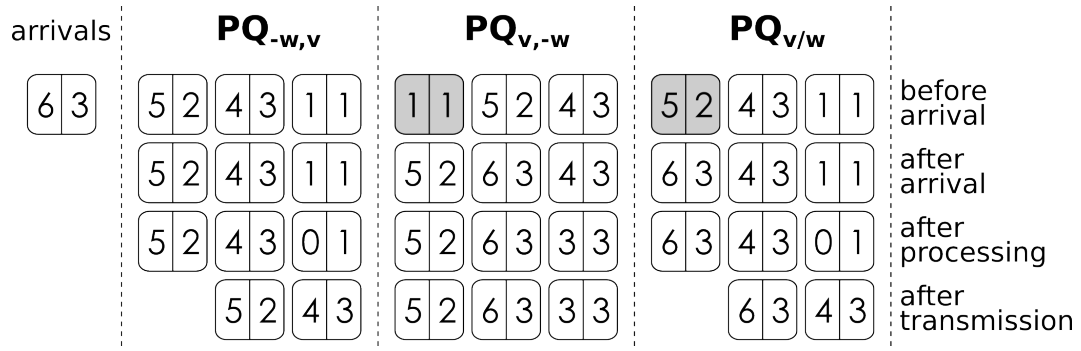


Figure 1: A sample time slot of $\mathsf{PQ}_{-w,v}$, $\mathsf{PQ}_{v,-w}$, and $\mathsf{PQ}_{v/w}$.

processing requirements: (1) $\mathsf{PQ}_{v,-w} = \mathsf{PQ}_{v-w/(W+1)}$ that prefers packets with higher value, ties given to lower processing; (2) $\mathsf{PQ}_{-w,v} = \mathsf{PQ}_{-w+v/(V+1)}$ that prefers packets with lower processing, ties given to higher value; and (3) $\mathsf{PQ}_{v/w}$ that prefers packets with higher value-to-work ratio. The example of their behavior is shown in Figure 1.

Lower Bounds. However promising, all three algorithms turn out to perform badly in terms of competitiveness. The next theorem combines Theorems 1, 2, and 3 from [29] demonstrating that each of the three policies has an approximation ratio linear either in W or V, suggesting that the interplay between the two packet characteristics makes the buffer management problem substantially more difficult.

Theorem. *For a buffer of size B, maximal amount of required processing W, and maximal value V:*

- $\mathsf{PQ}_{-w,v}$ *is exactly V-competitive;*
- $\mathsf{PQ}_{v,-w}$ *is at least* $\left(W \cdot \frac{V-1}{V} - o(1)\right)$*-competitive; and*
- $\mathsf{PQ}_{v/w}$ *is at least* $\min\{V, W\}$*-competitive.*

The bounds presented above hold only for specific algorithms and do not imply that an optimal one does not exist. For the latter kind of result, a *general* lower bound is required, constructing an adversarial input sequence able to "fool" *any* online algorithm. Section V of [29] presents a number of general lower bounds[†] that are polynomial in either V or W. The bounds differ in constraints imposed on an algorithm's implementation: FIFO processing order, fixed buffer size B, or being based on priority (PQ_f). A bound free of constraints is much weaker, but is still able to show that no optimal online algorithm exists.

Theorem. *For a buffer of size B, maximal packet required processing W, and available packet values* 1 *and $V > 1$, every online deterministic algorithm* ALG *is at least* $\left(1 + \frac{V-1}{V^2} - O\left(\frac{1}{W}\right)\right)$*-competitive.*

Two-valued case (an upper bound). The lower bounds demonstrate that in general natural priority-based algorithms are far from optimal and are even far from general lower bounds. The logical step is to add additional restrictions to the problem. One such restriction is to assume that there are only *two possible values*, i.e. either $v(p) = 1$ or $v(p) = V$. It corresponds to a scenario where all packets are divided into "commodity" packets $(w \mid 1)$ and "golden" or high-priority packets $(w \mid V)$.

[†]Not a part of this dissertation.

Theorem. *For a buffer of size B, maximal required processing W, and available packet values* 1 *and V:*

1. $\mathtt{PQ}_{-w,v}$ *is at least V-competitive;*
2. *if $W \geq V$ then* $\mathtt{PQ}_{v/w}$ *is at least V-competitive;*
3. $\mathtt{PQ}_{v,-w}$ *is at least* $\left(\frac{W}{V} + o(1)\right)$*-competitive.*

First, note that if $W < V$ then $\mathtt{PQ}_{v/w}$ operates exactly as $\mathtt{PQ}_{v,-w}$ since any $(w \mid V)$ is better than $(w' \mid 1)$. Second, the bound for $\mathtt{PQ}_{v,-w}$ has become much less strict, which is intuitive: if a processing of a given higher-valued packet was a wrong decision, loss per time slot should not exceed $\frac{V}{W}$ vs $\frac{1}{1}$. In fact, the intuition is (almost) true.

Theorem. *For a buffer of size B, maximal required processing W, and available packet values* 1 *and V* $\mathtt{PQ}_{v,-w}$ *is at most* $\left(1 + \frac{W+2}{V}\right)$*-competitive.*

The proof of the above result is based on an inductive argument similar to that in [9] comparing sorted (according to priority) sequences of packets in $\mathtt{PQ}_{v,-w}$'s and OPT's buffers. The novelty is in the mapping between "commodity" $(w \mid 1)$ packets processed by OPT and processing cycles spent by $\mathtt{PQ}_{v,-w}$. Given the V-competitiveness of $\mathtt{PQ}_{-w,v}$, $\mathtt{PQ}_{v,-w}$ and $\mathtt{PQ}_{-w,v}$ come very close to a $\min\{V, W/V\}$ lower bound[†] on priority-based algorithms (Theorem 6 in [29]).

Beta push-out case. The previous discussion treated a decision to drop a newly arrived packet and a decision to push-out an already admitted one equally. However, there are reasons to prefer the former to the latter (e.g., the latter consumes more resources on a network device). The work [9] introduced a copying cost model to account for the difference; in that model, each admitted packet reduces the objective function by α. Similarly to [9] a modified version of the $\mathtt{PQ}_f$ family of algorithms, $\mathtt{PQ}_f^\beta$ is considered here. $\mathtt{PQ}_f^\beta$ only pushes out a packet if a new packet is β times better ($\beta > 1$).

Theorem. *For a buffer of size B, maximal required processing W, and maximal packet value V:*

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

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

(3) $\mathtt{PQ}_{v,-w}^\beta$ *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.*

Results summary. Table 1 summarizes all the theoretical results presented in [29]. In the simulation study based on CAIDA [37] traces, the throughput of $\mathtt{PQ}_{v/w}$ has mostly remained within 90% of optimal (we used an overapproximation) with $\mathtt{PQ}_{v,-w}$ showing similar performance when there are only two packet values (see Figure 5 in Appendix A).

[†]Not a part of this dissertation.

Processing policy	General case	Two-valued case	
Adversarial general lower bounds			
Any online algorithm	$\frac{\min\{\sqrt[2B]{W}, \sqrt[B]{V}\}-1}{2}$†	$1 + \frac{V-1}{V^2} - O\left(\frac{1}{W}\right)$	
Any priority queue	$\sqrt{W}$†	$\min\{V, W/V\}$†	
Any FIFO online algorithm	$\left(\frac{\sqrt{W}}{B} + 1 - \frac{1}{B}\right)$†	$\frac{V+B-1}{W+B-1}$†	
Lower and upper bounds for specific algorithms			
Processing policy	Lower bound	Lower bound	Upper bound
$\mathsf{PQ}_{-w,v}$, $\mathsf{PQ}^{\beta}_{-w,v}$	V	V	V
$\mathsf{PQ}_{v,-w}$, $\mathsf{PQ}^{\beta}_{v,-w}$	$\frac{W(V-1)}{V} - o(1)$	$\frac{W}{V} + o(1)$	$1 + \frac{W+2}{V}$
$\mathsf{PQ}_{v/w}$, $W \geq V$	V	V	
$\mathsf{PQ}^{\beta}_{v/w}$, $\beta W \geq V$	V	V	
$\mathsf{PQ}_{v/w}$, $W < V$	W	$\frac{W}{V} + o(1)$	$2 + \frac{2}{V}$

Table 1: Results summary for buffer management: lower and upper bounds.

4.2 Packet classification: a basic network function for single packet processing

As we have stated earlier, *packet classification* is one of the core functionalities behind single packet processing, and it directly affects the performance of every network element. The purpose of packet classification is to distinguish among different types *(classes)* of packets in order to perform class-specific handling. This is useful both as a form of data processing, and as a way to differentiate among multiple data streams.

An input to the classification process is a packet *header* H that is represented by a sequence of bits $(h_1, \ldots, h_w)$ over the $\{0, 1\}$ alphabet and the output is an opaque *action* to be applied to the packet. The w above is the *classification width*. The classification behavior is defined by a *packet classifier* $\mathcal{K} = (R_1, \ldots, R_N)_{\prec}$, which is an ordered (prioritized) by $\prec$ set of *rules*. Each rule R_i consists of a *filter* F_i and an *action* A_i. The filter defines constraints on a packet header and is represented by a sequence of ternary bits $(f_1, \ldots, f_w)$ over the $\{0, 1, *\}$ alphabet, $*$ representing "don't care". A header $(h_1, \ldots, h_w)$ *matches* a filter (or a rule containing it) $(f_1, \ldots, f_w)$ iff for all i either $h_i = f_i$ or $f_i = *$. To classify a packet, a *lookup* into a classifier is performed, and an action of a rule whose filter matches a packet's header is returned as a result. If there are two rules R and R' that *intersect*, i.e., there exists a header matching both, the action from the higher priority rule is returned. A toy example of a packet classifier $\mathcal{K}$ is presented in Figure 2(a), higher priority rules are at the top.

The discussion that follows is largely concerned with transforming one classifier $\mathcal{K}$ into another classifier $\mathcal{K}'$. It is absolutely required that the transformation does not change $\mathcal{K}$'s semantics, in other words $\mathcal{K}$ and $\mathcal{K}'$ must be *equivalent*. Formally, two classifiers $\mathcal{K}$ and $\mathcal{K}'$ are equivalent iff for every header the lookup results for $\mathcal{K}$ and $\mathcal{K}'$ are the same.

Section 4.2.1 gives an overview of an approach published in a paper titled "General ternary bit strings on commodity longest-prefix-match infrastructure" (see Appendix B and [31]) corresponding to the second result from Section 2. Section 4.2.2 describes a paper "How to implement complex policies on existing network infrastructure" (see Appendix C and [33]) where the third result from the Section 2 is presented.

$\mathcal{K}$	#1	#2	#3	#4	Action
R_1	0	1	0	0	A_1
R_2	0	$*$	$*$	$*$	A_2
R_3	1	0	1	$*$	A_3
R_4	1	$*$	0	$*$	A_4

(a) Original classifier $\mathcal{K}$

$\mathcal{K}^B$	#1	#2	#3	#4	Action
R_1^B	0	0	1	0	A_1
R_2^B	0	$*$	$*$	$*$	A_2
R_3^B	1	1	0	$*$	A_3
R_4^B	1	0	$*$	$*$	A_4

(b) A B-reordering of $\mathcal{K}$, where $B = (1, 3, 2, 4)$

Figure 2: An example of a packet classifier with four rules and a packet header's width $w = 4$.

4.2.1 Representing general ternary bit strings on LPM infrastructure

A general packet classifier without any further constraints is called a *ternary* classifier. Ternary classifiers are the most powerful, they are found in modern packet processing abstractions [40] and are required for certain applications (e.g., [41]). Unfortunately, software-based implementations of ternary classifiers are inefficient: they require either too much space or time [42]; hardware solutions (TCAMs) being very expensive and power hungry are not present in sufficient capacity on commodity network devices. In contrast, once a classifier satisfies LPM constraints (to be defined shortly), it becomes much easier to represent the classifier efficiently, so the support for LPM classification is widely available. Formally, an *LPM classifier* satisfies the following: (1) all $*$s come after all 0s and 1s (e.g., rules R_2 and R_3 in Figure 2(a)), in other words, the classifier must be *prefix*; (2) given any two intersecting rules the one with a longer prefix has a higher priority (e.g., rules R_2^B and R_3^B in Figure 2(b) violate the property). The mismatch between application requirements and infrastructure capabilities demands a solution able to represent an arbitrary *ternary* classifier $\mathcal{K}$ on an *LPM* infrastructure with classification width at most w_{LPM} (usually, w_{LPM} is either 32 or 128 bits).

An approach presented here is agnostic to any particular LPM implementation. Its high-level overview is the following: first the classification width is reduced to w_{LPM}, then using a novel prefix-reorderability property the result is made prefix, and, finally, the priorities are adjusted to satisfy *LPM* constraints. The main contribution of the paper is the ternary-to-prefix transformation, to be explained first.

Prefix-reorderability. Network elements are capable of constructing complex lookup keys, in particular, they are able to reorder bits of a header. Reordering of header's bits allows for reordering of classifier's bits, which is a kind of transformation potentially able to turn the classifier into a prefix one. Formally, let $B = (b_1, b_2, \ldots, b_k)$, $k \leq w$ and $1 \leq b_i \leq w$, be a sequence of distinct bit indices representing the new bit order. Then for a header $H = (h_1, h_2, \ldots, h_w)$ and a filter $F = (f_1, f_2, \ldots, f_w)$, we define $H^B = (h_{b_1}, h_{b_2}, \ldots, h_{b_k})$ and $F^B = (f_{b_1}, f_{b_2}, \ldots, f_{b_k})$. Finally, for a rule $R = (F, A)$, $R^B = (F^B, A)$. The *B-reordering* of a classifier $\mathcal{K} = (R_1, R_2 \ldots, R_N)$ is a classifier $\mathcal{K}^B = (R_1^B, R_2^B, \ldots, R_N^B)$, where before each lookup an input header H is replaced with H^B. For example, a prefix $(1, 3, 2, 4)$-reordering of a classifier from Figure 2(a) is presented in Figure 2(b). It is not hard to see that if B is a permutation of $(1, \ldots, w)$ then classifiers $\mathcal{K}^B$ and $\mathcal{K}$ are equivalent. The question is whether there exists a permutation B such that $\mathcal{K}^B$ is a prefix classifier, i.e., whether $\mathcal{K}$ is *prefix-reorderable*.

It turns out useful to consider for a rule $R = (F, A)$ a set of bit indices i such that $f_i \neq *$ denoted by $\text{exact}(F)$, e.g., in Figure 2(a) $\text{exact}(R_4) = \{1, 3\}$. A curious observation is that if $\mathcal{K}^B$ is prefix, then for any $R \in \mathcal{K}$ bits from $\text{exact}(R)$ must precede in B bits from $\{1, \ldots, w\} \setminus \text{exact}(R)$; otherwise,

$\mathcal{K}$	#1	#2	#3	#4	Action
R_1	0	0	0	*	A_1
R_2	0	0	1	*	A_2
R_3	*	1	0	0	A_3
R_4	0	0	*	*	A_4
R_5	*	0	1	*	A_5
R_6	*	1	0	*	A_6
R_7	*	0	*	*	A_7

(a) An original classifier $\mathcal{K}$

$\mathcal{K}_1$	#1	#2	#3	#4	Action
R_1	0	0	0	*	A_1
R_2	0	0	1	*	A_2
R_4	0	0	*	*	A_4
R_7	*	0	*	*	A_7

$\mathcal{K}_2$	#1	#2	#3	#4	Action
R_3	*	1	0	0	A_3
R_5	*	0	1	*	A_5
R_6	*	1	0	*	A_6

(b) A prefix-reorderable partition of $\mathcal{K}$

Figure 3: An example of a non-prefix-reorderable classifier

the rule R would not be prefix. The observation imposes constraints on B, which may contradict each other: consider a classifier with $F_1 = (0\ *)$ and $F_2 = (*\ 1)$. Generally, if $\mathcal{K}$ contains two rules R and R' such that neither $\mathrm{exact}(R) \subseteq \mathrm{exact}(R')$ nor $\mathrm{exact}(R') \subseteq \mathrm{exact}(R)$, then $\mathcal{K}$ is *not* prefix reorderable. The first theorem shows that the above condition is sufficient and is easy to check; in what follows $\mathrm{exact}(\mathcal{K}) = \{\mathrm{exact}(R) : R \in \mathcal{K}\}$:

Theorem (chain criterion)**.** *A classifier $\mathcal{K}$ is prefix-reorderable iff for every $E_1, E_2 \in \mathrm{exact}(\mathcal{K})$ either $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ holds, i.e., $\mathrm{exact}(\mathcal{K})$ can be reordered to form a "chain": $E_{i_1} \subseteq \ldots \subseteq E_{i_{|\mathrm{exact}(\mathcal{K})|}}$. The permutation of bit indices can be found in $O(|\mathcal{K}| \cdot w)$ time (if one exists).*

Once a permutation B witnessing $\mathcal{K}$'s prefix-reorderability is found, it only remains to transform the prefix classifier $\mathcal{K}^B$ into an LPM classifier, which can be done with a trie-based algorithm `PrefixToLPM` in $O(|\mathcal{K}| \cdot w)$ time (Theorem 2 in [31]). Note, neither transformation increases the number of rules.

Next, there will be considered two approaches dealing with non prefix-reorderable classifiers. The first approach relaxes the prefix-reorderability property and takes advantage of the parallelism available at network devices, and the second uses additional rules.

Minimal multi-group representation. Assume that classifier's rules are partitioned into prefix-reorderable groups. Then each group can be transformed into an equivalent LPM classifier using the algorithms discussed above. Since modern network devices are able to perform several lookups at line rate, it is feasible to perform a lookup in each of the constructed classifiers and return the highest priority result as a final answer. For example, the classifier $\mathcal{K}$ in Figure 3(a) is not prefix reorderable, nevertheless, it can be partitioned into $\mathcal{K}_1$ and $\mathcal{K}_2$ as in Figure 3(b). Both $\mathcal{K}_1$ and $\mathcal{K}_2$ are prefix-reorderable; hence, once they are transformed into LPM form, two LPM lookups would be enough to implement $\mathcal{K}$. The number of lookups supported at line rate is limited, thus it is desirable to minimize the number of groups.

Problem (MinGR)**.** *Find a partition of rules of a given classifier $\mathcal{K}$ into a minimal number of disjoint prefix-reorderable groups.*

The crucial step towards a solution for MinGR is to move from partitioning of $\mathcal{K}$ to partitioning of $\mathrm{exact}(\mathcal{K})$. From the chain criterion it is known that $\mathcal{K}' \subseteq \mathcal{K}$ is prefix reorderable iff $\mathrm{exact}(\mathcal{K}')$ is a chain. It follows that given a solution to MinGR it is possible to produce a *chain cover* of $\mathrm{exact}(\mathcal{K})$ of the same size. In the opposite direction, from any chain cover of $\mathrm{exact}(\mathcal{K})$ it is possible to build group-wise

prefix-reorderable partition of $\mathcal{K}$ of the same size by grouping the rules according to exact. The MinGR problem is, thus, equivalent to the problem of finding the smallest chain partition of $\mathrm{exact}(\mathcal{K})$.

Note, the last problem is expressed solely in terms of a *partial order* $\mathrm{exact}(\mathcal{K})$ with relation $\subseteq$, and we can employ existing algorithmic solutions to the chain cover problem running in $O(|\mathrm{exact}(\mathcal{K})|^{5/2})$ time [32]. Adding the time to construct a partial order and the time to recover rule partitions leads to the `MinGroupPartition` algorithm. It is expected in practice (and was confirmed in evaluation) that $|\mathrm{exact}(\mathcal{K})|$ is much less than $|\mathcal{K}|$.

Theorem. *The* `MinGroupPartition` *algorithm finds an optimal solution for the* MinGR *problem in time* $O(|\mathrm{exact}(\mathcal{K})|^{5/2} + |\mathcal{K}|^2 w)$.

The number of groups produced by `MinGroupPartition` and, hence, the number of lookups required, can still be too large to implement at line rate on the LPM infrastructure.

Theorem. *There exists a classifier* $\mathcal{K}$ *with* $|\mathcal{K}| = O(\frac{1}{\sqrt{w}} 2^{w/2})$ *such that an optimal solution for the* MinGR *problem requires exactly* $|\mathcal{K}|$ *groups.*

Even though such a degenerate case as in the previous theorem is unlikely to occur, there may exist some *small* subset of rules that are "bad" for prefix-reorderablity rendering a soultion produced by `MinGroupPartition` infeasible to implement. The remedy to the issue is a "mixed" representation, where some part of the classifier is represented using traditional non-LPM techniques (e.g., in a small TCAM). The optimization objective is to minimize the size of that part given a limit on the number of LPM groups.

Problem (MaxCov)**.** *Given a classifier* $\mathcal{K}$ *and a constant* $\beta > 0$*, partition the largest possible subset of* $\mathcal{K}$*'s rules into at most* β *prefix-reorderable groups.*

To solve the MaxCov problem a `MaxCoveragePartition` algorithm slightly alters the underlying graph from [32] and runs minimum weight matching to account for "non-covered" rules.

Theorem. *The* `MaxCoveragePartition` *algorithm produces an optimal solution for the* MaxCov *in time* $O(|\mathrm{exact}(\mathcal{K})|^2 |\mathcal{K}| + |\mathcal{K}|^2 w)$.

A nice property that both `MaxCoveragePartition` and `MinGroupPartition` share is that non-prefix-reorderable classifiers are handled without increasing the number of rules. Still, sometimes it might be worthwhile to trade some memory for a smaller number of groups or a smaller size of the traditionally-represented part.

Problem ($\mathrm{MaxCov\text{–}m}$)**.** *Given a classifier* $\mathcal{K}$*, a constant* $\beta > 0$*, and a maximal number of rules* M*, find the largest subset* $\mathcal{K}' \subseteq \mathcal{K}$ *and a multigroup classifier* $\mathcal{K}^*$ *equivalent to* $\mathcal{K}'$ *with* $|\mathcal{K}^*| \leq M$ *such that* $\mathcal{K}^*$ *can be split into at most* β *prefix-reorderable groups.*

As the $\mathrm{MaxCov\text{–}m}$ problem does not restrict the ways in which $\mathcal{K}'$ can be modified, it is unlikely that an optimal solution can be found efficiently. A proposed heuristic repeatedly takes a rule R that cannot be fit into any of $\mathcal{K}^*$'s groups, and adds more indices to $\mathrm{exact}(R)$ by expanding non-exact bits in the R's filter, essentially duplicating the rule.

Shrinking the classification width. It still may be the case that the classification width of the input classifier $\mathcal{K}$ may be larger than w_{LPM} supported by the infrastructure. To reduce the width in [18] authors suggest exploiting a *rule-disjointness* property. A classifier $\mathcal{K}$ is rule-disjoint on a set of bit indices B

iff in $\mathcal{K}^B$ no two rules intersect. If $\mathcal{K}$ is rule disjoint on B, then $\mathcal{K}$ is equivalent to $\mathcal{K}^B$ with each action augmented by a false-positive check against the corresponding rule from $\mathcal{K}$. The width is reduced if $|B| < w$. A notable observation is that rule-disjointness and prefix-reorderability do not conflict with each other. Nevertheless, it is unclear which property should be satisfied first for the best performance.

Implementation and Evaluation. The optimizations described above were implemented in a platform-agnostic way using the P4-language infrastructure [40]. They also were evaluated based on synthetic classifiers from the Classbench [38] suite. A heuristic that first splits the rules into rule-disjoint groups and then into LPM groups while doing bounded "don't care" bit expansions performed the best. For instance, we were able to represent on 32-bit-wide LPM infrastructure 80% to 99% of rules for ACL-based classifiers, 38% to 100%—for firewall-based, and almost 100%—for IP-chain-based (see Tables I and II in Appendix B).

4.2.2 Representing complex policies

The previous section approached the representation of complex policies on existing infrastructures from the point of view of a single network element. To satisfy network-wide rule capacity constraints, [21] and [22] suggested a "virtual pipeline" approach to resource sharing. In particular, [22] introduced the splitting of a policy among the switches on the *path of a network flow* as a building block in the network-wide policy representation. Formally, given a classifier $\mathcal{K}$, the *splitting* of $\mathcal{K}$ is defined as a *sequence* of classifiers that is *equivalent* to $\mathcal{K}$. A lookup into a sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ is performed through a sequence of lookups into each of $\mathcal{K}_i$ in order, each time applying an action returned from the ith lookup before proceeding to the $(i+1)$th. The intuition is that if $\mathcal{K}$ represents a policy for a given flow, then $\mathcal{K}_i$ should be stored on the ith switch along the flow's path. An *l-splitting* of a classifier $\mathcal{K}$ is a splitting of $\mathcal{K}$ having l elements. The formal problem is stated below.

Problem (Flow`Split`)**.** *Given a classifier $\mathcal{K}$ and a sequence of switch capacities $c_1, c_2, \ldots, c_l$, find an l-splitting $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ of $\mathcal{K}$ such that $\mathcal{K}_i$ has at most c_i rules.*

The complexity of the Flow`Split` problem comes from intersecting rules, i.e., rules that match the same header. For instance, if a classifier $\mathcal{K}$ in Figure 4(a) is split arbitrarily as in Figure 4(b), then the equivalency may be lost: a header $(1\ 0\ 1\ 0)$ is matched twice: first in R_1 and then in R_2. A possible fix is to add **nop**-rules to $\mathcal{K}_2$ as shown in Figure 4(c).

There are two existing solutions to the issue of intersecting rules. `Palette` [21] expands $*$ bits, so that rules could be split into non-intersecting groups, i.e. no two $\mathcal{K}_i$ and $\mathcal{K}_j$, $i \neq j$, would match the same header. The optimization problem that arises from such approach is computationally hard, forcing `Palette` to resort to heuristics. The other approach, *One Big Switch* (`OBS`) [22], allows rules at different switches to intersect. To construct $\mathcal{K}_i$ `OBS` chooses a multidimensional rectangle r_i and sets $\mathcal{K}_i$ to be a "projection" of all remaining (not covered by $\mathcal{K}_j$, $j < i$) rules on r_i. To preserve the equivalency, in every later switch $\mathcal{K}_j$, $j > i$, a special high priority rule is added, mapping r_i to the **nop** action (similar to Figure 4).

$\mathcal{K}$	#1	#2	#3	#4	Action
R_1	$*$	$*$	1	0	A_1
R_2	1	0	$*$	$*$	A_2
R_3	0	0	$*$	$*$	A_3
R_4	$*$	$*$	1	1	A_4

(a) An original classifier $\mathcal{K}$

$\mathcal{K}_1$	#1	#2	#3	#4	Action
R_1	$*$	$*$	1	0	A_1
R_3	0	0	$*$	$*$	A_3

$\mathcal{K}_2$	#1	#2	#3	#4	Action
R_2	1	0	$*$	$*$	A_2
R_4	$*$	$*$	1	1	A_4

(b) Not a splitting of $\mathcal{K}$

$\mathcal{K}'_2$	#1	#2	#3	#4	Action
R_1	$*$	$*$	1	0	**nop**
R_2	1	0	$*$	$*$	A_2
R_3	0	0	$*$	$*$	**nop**
R_4	$*$	$*$	1	1	A_4

(c) A version of $\mathcal{K}_2$ augmented with **nop**-rules to preserve equivalency

Figure 4: An example of a classifier splitting

A boolean minimization technique. The first technique, introduced in this dissertation, takes the iterative approach of OBS as a basis, but allows $\mathcal{K}_i$ to represent an arbitrary subset of rules, not necessarily a rectangle. The price of that flexibility is a larger number of **nop**-rules introduced to subsequent switches.

The technique relies on *Boolean Minimization* [43] and is called BM. It operates in l steps, the ith step, $1 \leq i \leq l$, takes as input a classifier $\mathcal{K}^{(i-1)}$ ($\mathcal{K}^{(0)} = \mathcal{K}$) and using a heuristics (an algorithm's parameter) selects a subset $\mathcal{K}'_i \subseteq \mathcal{K}^{(i-1)}$. Then, $\mathcal{K}_i$ is set to $\mathcal{K}^{(i-1)}$ with all rules *not* in $\mathcal{K}'_i$ having the **nop** action, and boolean minimization is then run over $\mathcal{K}_i$. If the result fits in c_i, $\mathcal{K}^{(i)}$ is constructed similarly to $\mathcal{K}_i$, except that all rules *in* $\mathcal{K}'_i$ are being mapped to **nop**. The $(i+1)$th step begins next. Figure 4 shows one possible result of BM. The subset-selection heuristic can supply more than one candidate for $\mathcal{K}'_i$.

One-bit of metadata. It can be shown (see Section 4 in [33]) that all three approaches BM, Palette, and OBS are incomparable in general. Still, they all share three common limitations: (1) *memory expansion* due to either **nop** rules (OBS, BM) or rule duplication (Palette); (2) *slow running times* due to hard underlying optimization problems; and (3) *lack of support for dynamic fields*, i.e., the fields that are changed by actions and are also used for classification. A solution avoiding all three limitations and requiring just one bit of metadata is presented next.

Consider a naive approach of putting first (according to priority) c_1 rules of $\mathcal{K}$ into $\mathcal{K}_1$, then next c_2 rules into $\mathcal{K}_2$, and so on until all rules have been assigned. The *only* thing that may go wrong is that a header first matched at some $\mathcal{K}_i$ would be matched again at $\mathcal{K}_j$, $j > i$ since the first match in such a scheme is *always* the right one. A simple fix for repeated matching implemented in a OneBit algorithm introduces a special **matched** bit to a header indicating whether the header has already been matched, the classification is then performed only if **matched** is not set. Every action in each $\mathcal{K}_i$ is modified to set the **matched** bit, and the default action (applied in case of no-match) of $\mathcal{K}_1$ resets **matched**. The following theorem states that OneBit has indeed avoided the limitations of the earlier approaches.

Theorem. *Given a* FlowSplit*'s instance* $(\mathcal{K}, \{c_i\}_i)$ *with* $\sum_i c_i \geq |\mathcal{K}|$*, the* OneBit *algorithm constructs in* $O(|\mathcal{K}|)$ *time an* l*-splitting of* $\mathcal{K}$ *that remains correct in the presence of dynamic fields and requires* $|\mathcal{K}|$ *rules in total.*

Network-wide solution. The OneBit algorithm solves the policy-representation problem only for a single flow. Network-wide, there are multiple flows, each is a subject to its own policy and has its own forwarding path. In [22] the following joint policy placement problem was stated:

Problem (MultiFlowSplit)**.** *Given a set of nodes V, node capacities $c : V \to \mathbb{N}$, and a set of k path/classifier pairs $(P_i, \mathcal{K}_i)$, where P_i is a sequence of vertices $(v_1^i, \ldots, v_{l_i}^i)$, find a capacity allocation $a : V \times [k] \to \mathbb{N}$ such that $\sum_{i=1}^{k} a(v,i) \leq c(v)$, and a solution for each* FlowSplit *problem with $\mathcal{K} = \mathcal{K}_i$ and $c_j = a(v_j^i, i)$ for $j = 1, \ldots, l_i$.*

To solve MultiFlowSplit OBS's authors suggested an iterative heuristic that repeatedly refined per-flow capacity allocation, attempting to solve individual FlowSplit problems on every iteration using OBS technique. There was no bound on the number of iterations since there was no criterion for OBS to succeed in policy splitting. In contrast, the simplicity of a similar criterion for OneBit leads to a one-shot solution to the MultiFlowSplit problem via a reduction from the per-flow capacity allocation problem to the maximum flow problem.

4.3 Elastic processing

Elasticity offered by the cloud allows for greater flexibility and operational savings. To be efficient, the resource capacity should match the offered load as close as possible. Since it is infeasible to adjust the capacity instantaneously, some degree of capacity planning is required. Traditionally, it has been a user's responsibility, but recently, a new "pay-per-use" approach has emerged in the form of *serverless computing* [2, 3, 4], where users only pay for actually processed requests. The work "On demand elastic capacity planning for service auto-scaling" (see Appendix D and [34]) addresses the challenges of building an efficient capacity planning scheme in a serverless setting, it constitutes the fourth set of results from Section 2.

Model description. The time is assumed to be slotted. While a given request may need different types of resources (e.g., memory, processing, or network bandwidth), all are jointly modelled as a single *(virtual) resource unit.* Every request r has an associated value $v(r)$ and is considered processed if it has been allocated exactly one resource unit for exactly one time slot. Once processed, r adds $v(r)$ to the revenue; the scaling is such that $\min_r\{v(r)\} = 1$. Reflecting operational expenses, the allocation of a resource unit costs α, and keeping a resource unit allocated has a maintenance cost β per time slot. Finally each time slot t is divided into three phases: (1) *arrival:* a set A_t of new requests arrives; (2) *processing:* a set P_t of requests is chosen for processing, $|P_t|$ must be less than N_t, the number of allocated resources; (3) *prediction* the resource capacity N_{t+1} is planned for the next time slot. Note, phases (2) and (3) can be handled in parallel, the essential constraint is that the prediction happens *before* the next arrival to account for the allocation delay. The set of requests to choose P_t from depends on the ability to delay request processing, the study of a fundamental trade-off between delaying requests and dropping them altogether is the main topic of [34].

Objective. The objective is to minimize $\sum_t(\sum_{r\in P_t} v(r) - \beta N_t - c_t)$, where c_t is the cost of changing the resource capacity from N_t to N_{t+1}. Hereafter a linear cost model is assumed, i.e., $c_t = \max\{0, N_{t+1} - N_t\}$. In the economic setting, average case guarantees are insufficient, so the competitive worst-case analysis [30] is used instead. A given *online* algorithm ALG is called α-competitive, $\alpha \geq 1$, if for any finite

sequence of requests σ ALG's objective value $\texttt{ALG}(\sigma)$ is at least $\texttt{OPT}(\sigma)/\alpha$, where $\texttt{OPT}(\sigma)$ is an objective value of the *optimal offline* algorithm OPT; ALG is not competitive if no such α exists.

Bufferless setting. If a request cannot be delayed, i.e., $P_t \subseteq A_t$ must hold, then such a setting is called *BufferLess with Heterogeneous values* (BLH). Unfortunately, even if it is profitable to process one request alone ($\alpha < 1 - \beta$), there is no competitive algorithm:

Theorem. *If $\alpha < 1 - \beta$, then in* BLH *any deterministic online allocation policy is non-competitive.*

The above theorem relies on the inability to predict the right amount of resources for the first arrival: there may always arrive more requests. To make the setting more realistic, an upper bound B on the number of allocated resource units is imposed in the *Bounded BufferLess* setting with *Heterogeneous values* (BBLH). Nonetheless, B can be arbitrarily large, so non-competitiveness still holds:

Theorem. *If $\alpha < 1 - \beta$, then in* BBLH *any deterministic online allocation policy is non-competitive.*

Buffered setting. Motivated by the negative results of the bufferless setting, a *buffer* of size B is introduced into the model. Requests from A_t that have not been immediately serviced during t are hold in the buffer (at most B in total) for later processing. There are two variations of the model: *Buffered with Heterogeneous values* (BH) useful for upper bounds (as a more general one) and *Buffered with Uniform values* (BU) for lower bounds. Still, no optimal online algorithm exists.

Theorem. *If $\alpha < 1 - \beta$, then in* BU *every online algorithm is at least $\left(2 - \frac{\alpha}{1-\beta}\right)$-competitive.*

The first strategy that takes advantage of the buffer is called NRAP.

Definition. *Next Round Allocation policy (*NRAP*) operates as follows:*

- on arrival: *requests are accepted as long as there is room in the buffer; if the buffer is full then higher-valued requests always push out lower-valued ones from the buffer;*
- on processing: *either requests left from the previous time slot or those that have pushed them out are chosen for processing;*
- on prediction: *k resource units are predicted iff there are going to be k requests in the buffer by the end of the timeslot (i.e., not counting the requests under processing).*

It follows that NRAP never allocates more resources than necessary, but being to eager it may spend a lot on (de-)allocation. Nevertheless, NRAP has constant competitiveness once $\alpha < 1 - \beta$:

Theorem. *If $\alpha < 1 - \beta$, then in* BH *the* NRAP *algorithm is at most $\left(2 \cdot \frac{1-\beta}{1-\alpha-\beta}\right)$-competitive.*

The $\alpha < 1 - \beta$ condition is essential for NRAP to stay competitive since in general it performs one allocation per processed request. If allocation cost is higher and approaches 1 then NRAP's competitiveness tends to infinity.

Theorem. *If $\alpha < 1 - \beta$, then in* BU *the* NRAP *algorithm is at least $\left(1 + \frac{1-\beta}{1-\alpha-\beta}\right)$-competitive.*

In order to support allocation costs greater than $1 - \beta$, the next capacity planning algorithm, called ρ-AAP, does not allocate a resource unit until there is a sufficient number of pending requests to cover the allocation cost. Each resource unit has its own batch of *pre-assigned* requests.

Definition. *The* Amortizing Allocation Policy ρ-AAP *with parameter* $\rho > 1$ *operates as follows.*

- on arrival: *incoming requests are accepted as long as the remaining buffer capacity plus the number of allocated resource units is more than* $(B - \lceil \frac{\rho\alpha}{1-\beta} \rceil)/(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1) - \lceil \frac{\rho\alpha}{1-\beta} \rceil$ *(positive integer for simplicity); only requests accepted in the current time slot are allowed to be pushed out.*
- on processing: *for each allocated resource unit choose one of* preassigned *requests for processing.*
- on prediction: *while the total value of* non-preassigned *requests in the buffer is at least* $\lceil \frac{\rho\alpha}{1-\beta} \rceil$*:*
  - *predict that an additional resource unit* v' *will be needed for the next time slot;*
  - *preassign to* v' *as few as possible non-preassigned requests with total value at least* $\lceil \frac{\rho\alpha}{1-\beta} \rceil$*;*

  for every allocated resource unit with preassigned requests, predict that this unit will be needed for the next timeslot; deallocate those units that do not.

The following theorem summarizes the competitiveness guarantees of the ρ-AAP policy. For a simpler analysis ρ-AAP is given small extra buffer space compared to OPT.

Theorem. *If* $\alpha \geq (1 - \beta)$*, the* ρ-AAP *algorithm with a buffer of size* $B + \lceil \frac{\rho\alpha}{1-\beta} \rceil$ *has competitive ratio at most* $\frac{\rho}{\rho-1}\left(\left\lceil \frac{\rho\alpha}{1-\beta} \right\rceil + 1\right)$ *in* BH *and at least* $(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$ *in* BU *against* OPT *with a buffer of size* B*.*

Latency analysis. To maintain quality of service it is important to control latency experienced by requests. If a request r arrives at time slot $t_a(r)$ and leaves the system at $t_l(r)$, then $\mathrm{lat}(r) = t_l(r) - t_a(r)$. For an algorithm ALG there are two possible definitions of latency: *worst case latency* $\mathrm{lat}(\mathsf{ALG}) = \sup_\sigma \max_{r\in\sigma} \mathrm{lat}(r)$; and *average case latency* $\mathrm{lat}_{\mathrm{avg}}(\mathsf{ALG}) = \sup_\sigma (\sum_{r\in\sigma} \mathrm{lat}(r)/|\sigma|)$.

While latency of the NRAP algorithm is optimal, ρ-AAP may hold requests indefinitely until enough of them are accumulated. In order to perform latency analysis for ρ-AAP, we assume that at least one request arrives every time slot. We introduce a modified version of ρ-AAP, called ρ-AAPm, that drops all non-preassigned requests if no requests arrive.

Theorem. *In the* BH *model: (1)* $\mathrm{lat}(\mathsf{NRAP}) = 1$*; (2)* $\mathrm{lat}_{\mathrm{avg}}(\mathsf{NRAP}) = 1$*; (3)* $\mathrm{lat}(\rho\text{-}\mathsf{AAPm}) = \left\lceil \frac{\rho\alpha}{1-\beta} \right\rceil$*; and (4)* $\mathrm{lat}_{\mathrm{avg}}(\rho\text{-}\mathsf{AAPm}) \geq \frac{1}{2}\lceil \frac{\rho\alpha}{(1-\beta)} \rceil + \frac{1}{2}$.

Bounded Delay Model. Instead of bounding latency *a posteriori*, latency can be incorporated into the model. In the *Bounded Delay* (BD) model each request r comes with a deadline $d(r) \geq 1$, s.t., if r arrives on $t_a(r)$ it is *automatically* dropped after $t_a(r) + d(r)$. The number of resources and the buffer size are unbounded. There is still an implicit bound on the buffer size: *maximal arrival rate* $\times$ *maximal deadline.* The BD model does not allow for competitiveness even under a weaker assumption than BU.

Theorem. *If* $\alpha > 1 - \beta$*, then in* BD *any deterministic online policy is not competitive.*

The NRAP policy given an unbounded buffer never pushes out requests and, hence, always processes the request from the previous time slot, allowing to achieve constant competitiveness once $\alpha < 1 - \beta$:

Theorem. *If* $\alpha < 1 - \beta$*, then in* BD *any deterministic online policy is at least* $\left(1 + \frac{\alpha}{2(1-\alpha-\beta)}\right)$*-competitive, and* NRAP *is at most* $\left(1 + \frac{\alpha}{1-\beta-\alpha}\right)$*-competitive*

The last setting called *Limited buffer Bounded Delay* (LBD) imposes a fixed bound B on the number of allocated resources. Compared to BD the LBD model only slightly affects competitiveness.

Theorem. *If $\alpha < 1 - \beta$, then in* LBD *any deterministic online policy is at least $(2 + \frac{\alpha}{1-\beta-\alpha})$-competitive, while* NRAP *is at most $3(1 + \frac{\alpha+\beta}{1-\beta-\alpha})$ and at least 3 competitive.*

The experimental evaluation (see Section VIII in Appendix D) using synthetically generated traces has shown that if the allocation cost α remains small and the buffer size B is large, the proposed algorithms outperform learning-based straw-man approaches. As the allocation cost grows ρ-AAP looses noticably due to being over-cautious and underutilizing available resources.

4.4 Optimizing Compute-Aggregate Task Planning

An important class of big data processing applications are compute-aggregate systems where several data chunks must be aggregated at a given location. These systems are being optimized for latency and cost-efficiency, but rarely does the optimization take network constraints into account leading to TCP-incast [25] issues and link overload. The full joint optimization problem contains many moving parts, for which reason we introduce two *independent* phases; (1) finding the cheapest possible aggregation plan; and (2) performing actual aggregations and data transmissions while optimizing desired objectives. In a paper titled "Formalizing Compute-Aggregate Problems in Cloud Computing" the focus is on the first phase and, in particular, on the unified properties of compute-aggregate tasks that lead to efficient aggregation plans, which is the last result from Section 2

The network is modelled as an undirected graph $G = (V, E)$, where V is a set of computing nodes, and E is a set of connecting links. The compute aggregate task is represented by a target vertex $t \in V$ where the final result aggregated, and a set of initial data chunks $C = \{ \boxed{x_1}, \ldots, \boxed{x_n} \}$, where each chunk x has a location denoted by $\mathrm{v}(\boxed{x}) \in V$ and a size denoted by $\mathrm{size}(\boxed{x}) \in \mathbb{R}_{\geq 0}$. To combine a multitude of possible optimization objectives (e.g., latency, throughput, avoidance of congestion) a single per-link cost function $c : E \leftarrow \mathbb{R}_{\geq 0}$ is used: to transmit $\boxed{x}$ through $e \in E$ one must pay $c(e) \cdot \mathrm{size}(\boxed{x})$.

Move to root is suboptimal. The processing scheme that currently prevails is to send every chunk to the root t and perform all aggregations there. It may be suboptimal or infeasible for three reasons: (1) insufficient memory capacity for low-latency in RAM applications; (2) data is sent to the root simultaneously causing TCP-incast [25]; (3) suboptimal transmission cost (as defined earlier). It is possible to improve on the first two issues by sending data chunks to t one by one and immediately aggregating as they arrive. To gain more flexibility in the aggregation order operations should be *associative*: $\mathbf{aggr}(\boxed{x}, \mathbf{aggr}(\boxed{y}, \boxed{z})) = \mathbf{aggr}(\mathbf{aggr}(\boxed{x}, \boxed{y}), \boxed{z})$, and *commutative*: $\mathbf{aggr}(\boxed{x}, \boxed{y}) = \mathbf{aggr}(\boxed{y}, \boxed{x})$.

Intermediate Aggregations. The principle of *moving computation to data* is widely used in big data processing to reduce network traffic. In this work we further extend that principle to *moving aggregation to data,* which allows data aggregation to happen at *intermediate nodes*. The extension is formalized as follows: an *aggregation plan P* is a sequence of operations $(o_1, o_2, \ldots, o_m)$, where each o_i is either $\mathbf{move}(\boxed{x}, v)$ that moves a chunk $\boxed{x}$ to a vertex v; or $\mathbf{aggr}(\boxed{x}, \boxed{y})$ that merges two chunks $\boxed{x}$ and $\boxed{y}$

located at the same vertex, which results in a new chunk $\boxed{xy}$. After P has been entirely processed, there must remain a single chunk $\boxed{z}$, s.t., $\mathrm{v}(\boxed{z}) = t$.

Every aggregation plan P has an associated transmission cost $\mathrm{cost}(P)$ that is a sum of costs of all P's operations: $\mathrm{cost}(\textbf{move}(\boxed{x}, v)) = \mathrm{size}(\boxed{x}) \cdot d(\mathrm{v}(\boxed{x}), v)$, where $d(u, v)$ is the cheapest $(u \rightsquigarrow v)$-path according to c, and $\mathrm{cost}(\textbf{aggr}(\boxed{x}, \boxed{y})) = 0$ since no data transmission is taking place.

Compared with the move to root strategy, intermediate aggregations: (1) reduce incoming traffic of individual nodes, improving TCP-incast behavior; (2) reduce aggregated data at individual nodes, improving memory efficiency; and (3) reduce overall amount of transferred data and, as a result, the transmission cost. Also, note that an aggregation plan is fully decoupled both from the network transport responsible for data transmission and from the application responsible for aggregation.

Aggregation size function. To state the problem formally, it remains to define $\mathrm{size}(\boxed{xy})$ for any $\textbf{aggr}(\boxed{x}, \boxed{y})$ operation. Knowing the exact value is infeasible during the planning phase: the value may depend on $\boxed{x}$'s and $\boxed{y}$'s content, and it may require to actually perform the aggregation. As a trade-off between accuracy and feasibility each application is asked to provide a size-based approximation in the form of an *aggregation size function* $\mu : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, so that the following equality holds (on execution it holds only approximately) $\mathrm{size}(\boxed{xy}) = \mu(\mathrm{size}(\boxed{x}), \mathrm{size}(\boxed{y}))$. The μ function must satisfy the same commutativity and associativity constraints as **aggr**. These are some examples of μ: $\mu(a, b) = \mathrm{const}$ for finding the top k elements; $\mu(a, b) = \min(a, b)$ 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.

Problem (CAM — compute-aggregate minimization)**.** *Given a connected undirected graph $G = (V, E)$, a cost function c, a target vertex t, a set of initial data chunks C, and an aggregation size function μ, the* $\mathrm{CAM}[\mu]$ *problem is to find an aggregation plan P such that* $\mathrm{cost}(P)$ *is minimized.*

There are two important special cases of CAM: CCAM and TCAM. In the first, for security reasons it is prohibited to aggregate chunks at intermediate nodes. As a result, the set of vertices where initial chunks are located becomes equal to V. In the second case we restrict the topology G to a tree.

Hardness and connection to the minimum Steiner tree problem. Interestingly, without the associativity constraint on μ, there does not exist a polynomial time algorithm with a constant approximation factor unless $\mathrm{P} = \mathrm{NP}$. The proof of the following theorem is based on a reduction from the knapsack problem.

Theorem. *Unless* $\mathrm{P} = \mathrm{NP}$*, there is no polynomial time approximation algorithm for* CAM *without associativity constraint on μ even if G is restricted to two vertices.*

To understand the connection between CAM and the *minimum Steiner tree problem (*MStT*)* it is instructive to consider a special case where all initial chunks have equal sizes and $\mu(x, x) = x$. With such a choice of μ it is always beneficial to merge chunks; hence, the aggregation would happen strictly along the tree. Note, the cost of the aggregation plan would be equal to the total cost of the tree multiplied by x, and we see that the CAM problem becomes equivalent to MStT. In a general case we would need to pay a multiplicative factor, equal to the ratio between maximum and minimum chunk size among all

possible chunks. Using commutativity and associativity we can express both extrema as follows: let $W_C[\mu] = \max_{C' \subseteq C}\{\mu(C')\}$ and $w_c[\mu] = \min_{C' \subseteq C}\{\mu(C')\}$.

Theorem. *If there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial algorithm that solves* $\mathrm{CAM}[\mu]$ *with approximation factor* $\alpha\frac{W_C[\mu]}{w_C[\mu]}$.

There is a simple 2-approximation to the MStT based on the minimum spanning tree of the G's distance closure and a much more involved algorithm leading to $\alpha = \ln 4 + \epsilon \leq 1.39$ [44]. The two special cases, namely CCAM and TCAM, when viewed from MStT perspective become the minimum spanning tree and the unique Steiner tree, respectively. Both can be found in polynomial time, and hence in the above theorem has $\alpha = 1$ for CCAM and TCAM.

Corollary. *There exist polynomial algorithms that solve* $\mathrm{CCAM}[\mu]$ *and* $\mathrm{TCAM}[\mu]$ *on a set of chunks C with approximation factor* $\frac{W_C[\mu]}{w_C[\mu]}$.

Furthermore, if we know that $\min\{a,b\} \leq \mu(a,b) \leq \max\{a,b\}$, then $W_C[\mu] = W_C = \max_{c \in C}\{\mathrm{size}(c)\}$ and $w_C[\mu] = w_C = \min_{c \in C}\{\mathrm{size}(c)\}$. As a result, it is possible to obtain a multiplicative factor which is independent of the exact choice of μ:

Theorem. *If* $\min\{a,b\} \leq \mu(a,b) \leq \max\{a,b\}$ *for all a, b and there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial algorithm that solves* $\mathrm{CAM}[\mu]$ *with approximation factor* $\alpha\frac{W_C}{w_C}$.

Corollary. *If* $\min\{a,b\} \leq \mu(a,b) \leq \max\{a,b\}$ *then there exist polynomial algorithms that solves* $\mathrm{CCAM}[\mu]$ *and* $\mathrm{TCAM}[\mu]$ *with approximation factor* $\frac{W_C}{w_C}$.

Finally, when $\mu(a,b) \geq a + b$, then it never makes sense to merge chunks and we are able to treat all the chunks independently reducing the CAM to the problem of finding shortest paths from t.

Theorem. *If $\mu(a,b) \geq a + b$ for all a, b then there exists a polynomial optimal algorithm for* $\mathrm{CAM}[\mu]$*,* $\mathrm{CCAM}[\mu]$*, and* $\mathrm{TCAM}[\mu]$*; for* $\mathrm{TCAM}[\mu]$ *the running time is $O(|C| + |G|)$.*

5 Conclusion

This dissertation addresses fundamental scalability constraints of modern computing infrastructures, following the specific goals and objectives introduced in Section 1.2.

First, we have made important progress in the field of in-network data processing. To support new objectives and data stream characteristics during processing of multiple data streams, we performed worst-case performance analysis for several natural priority-based buffer management policies (Section 4.1). This analysis has for the first time jointly considered two characteristics, value and processing requirements. This has allowed us to make an optimal choice of the management policy in such a setting. In particular, we have shown that prioritizing according to value-to-work ratio is not always the best option, contrary to natural intuition.

Second, it has been proven possible to represent complex classification policies on existing network infrastructure through: (1) novel ternary-to-LPM classifier transformation algorithms (Section 4.2.1), and (2) an efficient but simple scheme for network capacity sharing (Section 4.2.2). As a result, we have introduced and justified techniques that achieve the expressiveness required for in-network data

processing without excessive costs.

Third, for *efficient serverless computing* we have studied the trade-off between delaying service requests and revenue maximization in the serverless setting, using a novel formalization of serverless economics (Section 4.3). As part of this study, we have developed and studied two novel capacity planning algorithms, providing worst-case (assumption-free) performance guarantees and latency analysis for each. One major benefit of these contributions lies in cost efficiency improvements for the most elastic (i.e., flexible) computing paradigm.

Fourth, to exploit *intermediate data aggregations* we have developed a two-phase approach (Section 4.4). The first phase optimizes for budget constraints based only on the information necessary to construct an efficient aggregation plan. Desired objectives (e.g., latency or throughput) are captured indirectly and optimized by the network during the second phase. By separating the two phases, we make network transport less reactive, reducing its task to scheduling of aggregations according to the constructed aggregation plan. Intermediate aggregations used in that plan bring an additional advantage that many aggregation issues (e.g., TCP-incast or memory capacity constraints) should become much less pronounced.

In total, this thesis represents significant progress towards the direction of exascale big data processing in a networked environment. This progress exposes a great potential of the underlying interconnecting infrastructure that has not seen much attention before. Taken together, our results allow to significantly enhance big data processing in computer networks without a radical redesign of underlying infrastructures.

References

[1] Hajirahimova Makrufa Sh., Aliyeva Aybeniz S. About Big Data Measurement Methodologies and Indicators // Intl. J. Modern Education and Computer Science. — 2017. — Vol. 9, no. 10. — P. 1–9.

[2] Amazon. AWS Lambda. — 2017. — https://aws.amazon.com/lambda/.

[3] Google. Cloud Functions. — 2017. — https://cloud.google.com/functions/.

[4] Microsoft. Azure Functions. — 2017. — https://azure.microsoft.com/en-us/services/functions/.

[5] Reserved, on demand or serverless: Model-based simulations for cloud budget planning / E. F. Boza [et al.] // 2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM). — 2017. — Oct. — P. 1–6.

[6] Lloyd Wes [et al.]. Serverless Computing: An Investigation of Factors Influencing Microservice Performance // IC2E. — 2018. — P. 159–169.

[7] Dean Jeffrey, Ghemawat Sanjay. MapReduce: simplified data processing on large clusters // Commun. ACM. — 2008. — Vol. 51, no. 1. — P. 107–113.

[8] The Case for Evaluating MapReduce Performance Using Workload Suites / Yanpei Chen [et al.] // MASCOTS. — 2011. — P. 390–399.

[9] Providing Performance Guarantees in Multipass Network Processors / Isaac Keslassy [et al.] // IEEE/ACM Trans. Netw. — 2012. — Vol. 20, no. 6. — P. 1895–1909.

[10] Online Scheduling FIFO Policies with Admission and Push-Out / Kirill Kogan [et al.] // Theory of Computing Systems. — 2016. — Feb. — Vol. 58, no. 2. — P. 322–344. — URL: https://doi.org/10.1007/s00224-015-9626-4.

[11] Andelman Nir, Mansour Yishay, Zhu An. Competitive Queueing Policies for QoS Switches // Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. — SODA '03. — Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2003. — P. 761–770. — URL: http://dl.acm.org/citation.cfm?id=644108.644235.

[12] Kesselman Alex, Kogan Kirill, Segal Michael. Improved Competitive Performance Bounds for CIOQ Switches // Algorithmica. — 2012. — Jun. — Vol. 63, no. 1. — P. 411–424. — URL: https://doi.org/10.1007/s00453-011-9539-9.

[13] Kesselman Alex, Kogan Kirill, Segal Michael. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing // Distributed Computing. — 2010. — Nov. — Vol. 23, no. 3. — P. 163–175. — URL: https://doi.org/10.1007/s00446-010-0114-4.

[14] Gupta P., McKeown N. Classifying packets with hierarchical intelligent cuttings // IEEE Micro. — 2000. — Jan. — Vol. 20, no. 1. — P. 34–41.

[15] Vamanan Balajee, Voskuilen Gwendolyn, Vijaykumar T. N. EffiCuts: Optimizing Packet Classification for Memory and Throughput // SIGCOMM Comput. Commun. Rev. — 2010. — Aug.. — Vol. 40, no. 4. — P. 207–218. — URL: http://doi.acm.org/10.1145/1851275.1851208.

[16] Fast packet classification using bloom filters / S. Dharmapurikar [et al.] // 2006 Symposium on Architecture For Networking And Communications Systems. — 2006. — Dec. — P. 61–70.

[17] Compressing Forwarding Tables for Datacenter Scalability / O. Rottenstreich [et al.] // IEEE Journal on Selected Areas in Communications. — 2014. — January. — Vol. 32, no. 1. — P. 138–151.

[18] SAX-PAC (Scalable And eXpressive PAcket Classification) / Kirill Kogan [et al.] // Proceedings of the 2014 ACM Conference on SIGCOMM. — SIGCOMM '14. — New York, NY, USA : ACM, 2014. — P. 15–26.

[19] Space and speed tradeoffs in TCAM hierarchical packet classification / Alexander Kesselman [et al.] // Journal of Computer and System Sciences. — 2013. — Vol. 79, no. 1. — P. 111 – 121. — URL: http://www.sciencedirect.com/science/article/pii/S0022000012001237.

[20] Bremler-Barr A., Hendler D. Space-Efficient TCAM-Based Classification Using Gray Coding // IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications. — 2007. — May. — P. 1388–1396.

[21] Kanizo Y., Hay D., Keslassy I. Palette: Distributing tables in software-defined networks // 2013 Proceedings IEEE INFOCOM. — 2013. — April. — P. 545–549.

[22] Optimizing the "One Big Switch" Abstraction in Software-defined Networks / Nanxi Kang [et al.] // Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies. — CoNEXT '13. — New York, NY, USA : ACM, 2013. — P. 13–24.

[23] Amazon AutoScaling. — http://aws.amazon.com/autoscaling/.

[24] Han R., et al. Lightweight Resource Scaling for Cloud Applications // CCGrid. — 2012. — P. 644–651.

[25] Understanding TCP Incast Throughput Collapse in Datacenter Networks / Yanpei Chen [et al.] // Proceedings of the 1st ACM Workshop on Research on Enterprise Networking. — WREN '09. — New York, NY, USA : ACM, 2009. — P. 73–82.

[26] Camdoop: Exploiting In-network Aggregation for Big Data Applications / Paolo Costa [et al.] // Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). — San Jose, CA : USENIX, 2012. — P. 29–42.

[27] Map-reduce-merge: Simplified Relational Data Processing on Large Clusters / Hung-chih Yang [et al.] // Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. — SIGMOD '07. — New York, NY, USA : ACM, 2007. — P. 1029–1040. — URL: http://doi.acm.org/10.1145/1247480.1247602.

[28] Optimal communication structures for big data aggregation / W. Culhane [et al.] // 2015 IEEE Conference on Computer Communications (INFOCOM). — 2015. — April. — P. 1643–1651.

[29] Priority Queueing for Packets With Two Characteristics / P. Chuprikov [et al.] // IEEE/ACM Transactions on Networking. — 2018. — Feb. — Vol. 26, no. 1. — P. 342–355.

[30] Borodin Allan, El-Yaniv Ran. Online Computation and Competitive Analysis. — New York, NY, USA : Cambridge University Press, 1998. — ISBN: 0-521-56392-5.

[31] Chuprikov P., Kogan K., Nikolenko S. General ternary bit strings on commodity longest-prefix-match infrastructures // 2017 IEEE 25th International Conference on Network Protocols (ICNP). — 2017. — Oct. — P. 1–10.

[32] Fulkerson D. R. Note on Dilworths decomposition theorem for partially ordered sets // Proc. Amer. Math. Soc. — 1956.

[33] Chuprikov Pavel, Kogan Kirill, Nikolenko Sergey. How to Implement Complex Policies on Existing Network Infrastructure // Proceedings of the Symposium on SDN Research. — SOSR '18. — New York, NY, USA : ACM, 2018. — P. 9:1–9:7. — URL: http://doi.acm.org/10.1145/3185467.3185477.

[34] Chuprikov P., Nikolenko S., Kogan K. On demand elastic capacity planning for service auto-scaling // IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. — 2016. — April. — P. 1–9.

[35] Lorido-Botran Tania, Miguel-Alonso Jose, Lozano Jose A. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments // Journal of Grid Computing. — 2014. — Dec. — Vol. 12, no. 4. — P. 559–592.

[36] Formalizing Compute-Aggregate Problems in Cloud Computing / Pavel Chuprikov [et al.] // Structural Information and Communication Complexity / Ed. by Zvi Lotker, Boaz Patt-Shamir. — Cham : Springer International Publishing, 2018. — P. 377–391.

[37] for Internet Data Analysis CAIDA The Cooperative Association. — [Online] http://www.caida.org/.

[38] Taylor David E., Turner Jonathan S. ClassBench: A Packet Classification Benchmark // IEEE/ACM Trans. Netw. — 2007. — Jun.. — Vol. 15, no. 3. — P. 499–511.

[39] Zukerman M., Neame T., Addie R. Internet traffic modeling and future technology implications // INFOCOM. — Vol. 1. — 2003. — March. — P. 587–596 vol.1.

[40] P4: Programming Protocol-independent Packet Processors / Pat Bosshart [et al.] // SIGCOMM Comput. Commun. Rev. — 2014. — Jul.. — Vol. 44, no. 3. — P. 87–95.

[41] Fast Regular Expression Matching Using Small TCAM / Chad R. Meiners [et al.] // IEEE/ACM Trans. Netw. — 2014. — Vol. 22, no. 1. — P. 94–109.

[42] Gupta P., McKeown N. Packet classification on multiple fields // SIGCOMM. — 1999. — P. 147–160.

[43] Minimizing Disjunctive Normal Form Formulas and AC^0 Circuits Given a Truth Table / Eric Allender [et al.] // SIAM J. Comput. — 2008. — Vol. 38, no. 1. — P. 63–84.

[44] An Improved LP-based Approximation for Steiner Tree / Jaroslaw Byrka [et al.] // Proceedings of the Forty-second ACM Symposium on Theory of Computing. — STOC '10. — New York, NY, USA : ACM, 2010. — P. 583–592. — URL: http://doi.acm.org/10.1145/1806689.1806769.

Appendices

A Paper “Priority queueing for packets with two characteristics”

Authors. Pavel Chuprikov, Sergey Nikolenko, Alex Davydow, and Kirill Kogan

Abstract. Modern network elements are increasingly required to deal with heterogeneous traffic. Recent works consider processing policies for buffers that hold packets with different processing requirements (number of processing cycles needed before a packet can be transmitted out) but uniform value, aiming to maximize the throughput, i.e., the number of transmitted packets. Other developments deal with packets of varying value but uniform processing requirement (each packet requires one processing cycle); the objective here is to maximize the total transmitted value. In this paper, we consider a more general problem, combining packets with both nonuniform processing and nonuniform values in the same queue. We study the properties of various processing orders in this setting. We show that in the general case, natural processing policies have poor performance guarantees, with linear lower bounds on their competitive ratio. Moreover, we show several adversarial lower bounds for every priority queue and even for every online policy. On the positive side, in the special case when only two different values are allowed, 1 and V , we present a policy that achieves competitive ratio $\left(1 + \frac{W+2}{V}\right)$, where W is the maximal number of required processing cycles. We also consider copying costs during admission.

Priority Queueing for Packets with Two Characteristics

Pavel Chuprikov, Sergey I. Nikolenko, Alex Davydow, Kirill Kogan

***Abstract*—Modern network elements are increasingly required to deal with heterogeneous traffic. Recent works consider processing policies for buffers that hold packets with different processing requirement (number of processing cycles needed before a packet can be transmitted out) but uniform value, aiming to maximize the throughput, i.e., the number of transmitted packets. Other developments deal with packets of varying value but uniform processing requirement (each packet requires one processing cycle); the objective here is to maximize the total transmitted value. In this work, we consider a more general problem, combining packets with both nonuniform processing and nonuniform values in the same queue. We study the properties of various processing orders in this setting. We show that in the general case natural processing policies have poor performance guarantees, with linear lower bounds on their competitive ratio. Moreover, we show several adversarial lower bounds for every priority queue and even for every online policy. On the positive side, in the special case when only two different values are allowed, 1 and V, we present a policy that achieves competitive ratio $\left(1 + \frac{W+2}{V}\right)$, where W is the maximal number of required processing cycles. We also consider copying costs during admission.**

I. Introduction

Modern networks require implementation of advanced economic models that can be represented by desired objectives, network topology, buffering architecture, and its management policy. The current Internet architecture is mostly built for fairness, while consideration of other objectives such as network utilization, throughput, profit and others is required [27], [32]. For a given network topology and buffering architecture, design of management policies that optimize a desired objective is extremely important; a management policy of a single network element includes admission control and scheduling policies. Admission control is one of the critical elements of management policy. Most admission control policies are based on a simple characteristic such as buffer occupancy, whereas traffic has additional important characteristics such as processing requirements or value that are either not taken into account at all or a separate queue is allocated per traffic type. Incorporation of new characteristics (e.g., required processing per packet) in admission decisions and implementation of additional objectives beyond fairness lead to new challenges in design and implementation of traditional network elements.

In this work, we consider a single-queue switch where a buffer of size B is shared among all types of traffic. We do not assume any specific traffic distribution but rather analyze our switching policies against adversarial traffic using competitive analysis [6], [35], which provides a uniform throughput guarantee for online algorithms under all possible traffic patterns. An online algorithm ALG is α-*competitive* for some $\alpha \geq 1$ if for any arrival sequence σ the total value transmitted by ALG is at least $1/\alpha$ times the total value transmitted in an optimal solution obtained by an offline clairvoyant algorithm (denoted OPT). If an online algorithm is not α-competitive for any constant α independent of the input, it is said to be *non-competitive*. Note that a *lower bound* on the competitive ratio can be proven with a specific hard example while an *upper bound* represents a general statement that should hold over all possible inputs. In practice, the choices of processing order, implementation of push-out mechanisms etc. are likely to be made at design time. From this point of view, our study of worst-case behaviour aims to provide a robust estimate on the settings that can handle all possible loads.

The purpose of this work is to study the impact of both packet values and required processing on weighted throughput; to the best of our knowledge, this is the first attempt to study such impact. The paper is organized as follows. In Section II, we formally introduce the model we will use in this work, a model with both required processing and values. In Section III, we survey previous work in related buffer processing algorithms. In Section IV, we introduce several algorithms based on priority queueing that appear promising for this setting; these algorithms differ in the way how they order packets: by required processing, by value, or by a ratio of these numbers (i.e., by value per one processing cycle). In Section IV we begin with a negative result: we show that all of these algorithms have at least linear competitive ratio in the general case. Moreover, in Section V we proceed to show a general lower bound for *any* online algorithm proven in an adversarial fashion; this is an important new result for this model as previously considered special cases (uniform values with heterogeneous processing and uniform processing with variable values) allowed for optimal online policies. However, in Section VI we introduce an important special case when there are only two different possible values, i.e., packets may have different required processing but their value is limited to 1 and V. The maximal number of required processing cycles is W. In our main result, we present a policy based on a priority queue that achieves competitive ratio $\left(1 + \frac{W+2}{V}\right)$.

A previous version of this work has appeared at INFOCOM 2015. Compared to the conference version, we have added several novel results including general lower bounds shown in Section V, in particular, a general adversarial lower bound in Theorem 10, significantly extended the discussion parts of the paper, added Figure 4 and published software used for experiments.

P. Chuprikov is with the Steklov Mathematical Institute at St. Petersburg and IMDEA Networks Institute, Madrid, Spain; e-mail: pschuprikov@gmail.com. S.I. Nikolenko is with the Steklov Mathematical Institute at St. Petersburg; e-mail: sergey@logic.pdmi.ras.ru. A. Davydow is with the Steklov Mathematical Institute at St. Petersburg; e-mail: adavydow@gmail.com. K. Kogan is with IMDEA Networks Institute; e-mail: kirill.kogan@imdea.org.

Note that while it may appear suspicious to compare packet values with required processing, in fact we are comparing ratios of the most valuable (resp., heaviest) packet to the least valuable (resp., lightest) packet because the minimal required processing and minimal value are always set to 1. In Section VII, we consider the β-push-out case, which takes copying cost into account by introducing additional penalties for push-out (a detailed explanation of β-push-out is given in Section VII). Section VIII presents simulation results where the proposed algorithms are evaluated with synthesized traces, and Section IX concludes the paper.

II. Model Description

We use a model similar to the one introduced in [1], [20] and subsequently used in [14]–[16], [23]–[25]. Consider a single queue that is able to hold B unit-sized packets and that handles the arrival of a sequence of packets, each of which is unit-sized. A new part of the problem setting in this work is to combine two different characteristics of a packet. 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 $v(p) \in \{1, \ldots, V\}$.

These numbers are known for every arriving packet; for a motivation of why required processing may be available see [36], and values are usually defined externally. 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., [5], [37], while for $V = 1$ it becomes a single queue of unit-valued packets with different required processing, as considered, e.g., in [22]–[24]. We will denote a packet with required processing w and value v by $(w \mid v)$, and a sequence of n packets with the same parameters w and v by $n \times (w \mid v)$.

The queue performs three main tasks, namely:

(1) *buffer management*, i.e., admission control of newly arrived packets;
(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. Even though a fully processed packet is eligible for transmission, in some settings it can be deliberately delayed, e.g., if FIFO transmission order is required [24], [25]. We consider transmission order constraints only once in Theorem 5 and assume that the packet is transmitted as soon as it is fully processed.

We assume discrete slotted time, where each time slot consists of three phases (see Fig. 1 for an illustration):

(i) *arrival*: new packets arrive, and admission control decides if a packet should be dropped or, possibly, an already admitted packet should be pushed out;
(ii) *processing*: one packet is selected for processing by the scheduling unit;
(iii) *transmission*: at most one fully processed packet is selected for transmission and leaves the queue.

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 its value to the objective function only upon being successfully transmitted; note that only one packet may be transmitted per time slot. The goal is to devise buffer management algorithms that maximize the overall throughput, i.e., the total value of all packets transmitted out of the queue.

For an algorithm ALG and time slot t, we denote by $\mathrm{IB}^{\mathrm{ALG}}(t)$ 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. 1); if the timeslot is clear from the context we write simply $\mathrm{IB}^{\mathrm{ALG}}$. For every time slot t and every packet p currently stored in the queue, its number of *residual processing cycles*, denoted $w_t(p)$, is defined to be the number of processing cycles it requires before it can be successfully transmitted, and its *value*, denoted $v(p)$, is the number it contributes to the objective function upon transmission.

Three fundamental properties are often used in online algorithms. First, a policy is called *greedy* if it always accepts packets in the queue whenever it has free space. Greedy algorithms are usually amenable to efficient implementation and transmit everything if there is no congestion. Second, a policy is called *work-conserving* if it is always processing as long as it has packets with nonzero required processing in the buffer. Third, a policy is called *push-out* if it is allowed to drop packets that already reside in its queue; note that it does not make sense for a push-out policy to be non-greedy in the basic setting, but in the setting of Section VII where admitted packets incur nonzero copying cost this may not be the case. In what follows, we will assume that all push-out policies are greedy and all policies are work-conserving.

III. Related Work

Rich literature has been devoted to special cases of our model where one characteristic is assumed to be uniform. In particular, admission control policies for the case of single-queued buffers where packets with uniform processing and varying intrinsic value arrive have been thoroughly studied. In the case of two values (1 and V) and First-In-First-Out (FIFO) processing order, the works [5], [37] present a deterministic non-push-out policy with competitive ratio $(2 - \frac{1}{V})$, i.e., bounded by a constant. For the more general case, when packet values vary between 1 and V, the works [5], [37] prove that the competitive ratio cannot be better than $\Theta(\log V)$. In [4], this upper bound was improved to $2 + \ln V + O(\ln^2 V/B)$. In the push-out case with two packet values, the greedy policy was shown in [17] to be at least 1.282 and at most 2-competitive. Later, the upper bound on the greedy policy was improved to 1.894 [19]; this work also considers the β-push-out case and proves that the greedy policy is at least 1.544-competitive.

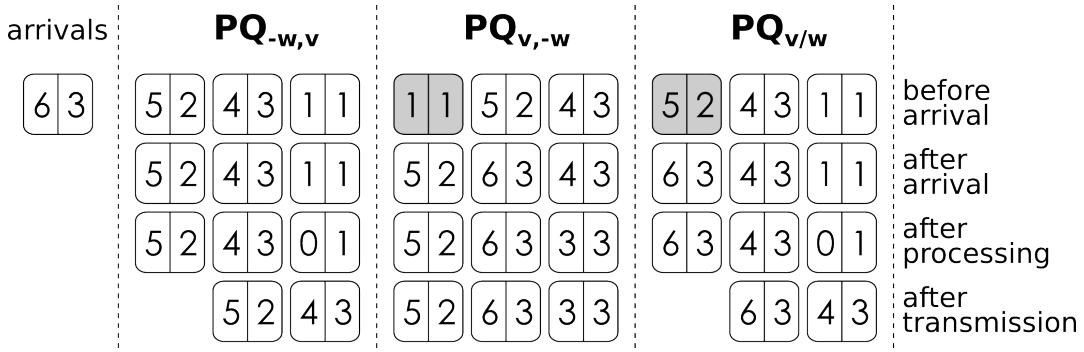


Fig. 1. A sample time slot of $\mathrm{PQ}_{-w,v}$, $\mathrm{PQ}_{v,-w}$, and $\mathrm{PQ}_{v/w}$.

Policies with memory have been considered in [3], [8], [26], [28].

Recently, packets with required processing but with uniform packet values in various settings have been considered in [9], [14]–[16], [23]–[25]. These works also follow the paradigm of competitive analysis, and their main results usually constitute good processing policies that have constant or logarithmic upper bounds on the competitive ratio. For a buffer with one queue of packets with uniform value, priority queue that orders packets according to their required processing is known to be optimal [14]. Our current work can be viewed as part of a larger research effort concentrated on studying competitive algorithms for management of bounded buffers. Initiated in [2], [17], [29], this line of research has received tremendous attention over the past decade. A survey by Goldwasser [11] provides an excellent overview of this field. Pruhs [33] provides a comprehensive overview of a related field of competitive online scheduling for server systems; however, scheduling for server systems usually concentrates on average response time and does not allow jobs to be dropped, while we focus mostly on throughput and allow push-out.

To control increasing queueing delays introduced by packet buffers, the bounded-delay model with varying intrinsic value was introduced by Kesselman et al. [18]. In that model, each packet is associated with a *slack* value s, which denotes a hard deadline when a packet should be processed. The greedy algorithm that always processes a packet with the earliest deadline is known to be 2-competitive [13], [18], and the best known competitive ratio is $2\sqrt(2) - 2 \approx 1.828$, as shown by Englert and Westermann [7]. A recent experimental study [34] evaluated the performance of different algorithms under a compatible deadline model. Note that a maximal slack value implicitly bounds a buffer size even if the buffer is theoretically unlimited. For this reason, the bounded-delay model appears to be more attractive for competitive analysis than the model, where a buffer is bounded explicitly. In this work, we not only consider an explicitly bounded buffer but also take into account required processing, which has a huge impact on the performance as both our theoretical results and simulation study will show.

Another very interesting class of results in competitive analysis are adversarial lower bounds that hold over all algorithms. Such bounds, when they can be proven, indicate that one cannot hope to get an optimal online algorithm, and a clairvoyant offline algorithm will always be able to outperform it. One well-known example of such a bound is the lower bound of $\frac{4}{3}$ on the competitive ratio of any algorithm in the model with multiple queues in a shared memory buffer and uniform packets (i.e., packets with identical value and required processing) [1], [12]. For the case of a single queue, previous works have considered two cases: variable value with uniform processing and variable processing with uniform values. In both cases, a single priority queue that orders packets with respect to the variable characteristic (largest value and smallest required processing first, respectively) is optimal, so there can be no nontrivial general lower bound regardless of transmission order. In the FIFO model, for the case of variable values and uniform processing there has been a line of adversarial lower bounds culminating in the lower bound of 1.419 that applies to all algorithms [21], with a stronger bound of 1.434 for the special case when $B = 2$ if all possible values are admissible [5], [37]. In the two-valued case, tight bounds are known: 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 an online algorithm that achieves competitive ratio r for arbitrary B and r_∞ for $B \to \infty$ [8]. In the case of variable processing with uniform values, no general lower bounds for FIFO order are known apart from a simple lower bound of $\frac{1}{2}(W + 1)$ for greedy non-push-out policies [24].

IV. Algorithms and Lower Bounds

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 have studied in detail the interrelations between policies with and without *push-out*, the capability to drop previously accepted packets from the buffer [11], [31]; 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 work 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 majority of previous works [11], [31]. 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 [14].

Note, however, that though 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 4.1: 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. Note that this definition, again, restricts the space of possible algorithms. In theory, we could separate admission and processing order, accepting and pushing out packets with respect to one ordering but processing and transmitting them with respect to a different ordering. In [24], in the setting with one characteristic and FIFO transmission order constraint, a similar idea—decouple transmission order from processing order—has led to significant improvements in competitive ratios, including a constant upper bound on the competitiveness of a policy which admitted and processed its packets as a priority queue but transmitted them in FIFO order. However, throughout this paper we simplify our considerations and assume that admission and processing orders are the same.

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;
(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.[1]

Fig. 1 shows a sample time slot of these priority queues; in this case $B = 3$, all policies start with $(5 \mid 2)$, $(4 \mid 3)$, and $(1 \mid 1)$ in their queues, and a $(6 \mid 3)$ packet arrives. $\mathrm{PQ}_{-w,v}$ rejects the $(6 \mid 3)$ since it has the largest processing requirement, $\mathrm{PQ}_{-v,w}$ pushes out $(1 \mid 1)$ since it has the smallest value, and $\mathrm{PQ}_{v/w}$ pushes out $(5 \mid 2)$ since it has the worst v/w ratio of $2/5$ compared to $3/4$, 1, and $3/6$ of the other three packets.

One of the goals of this work is to explore which order performs best. Our main result in this part is that, in general, priority queues fail to provide constant or even logarithmic competitiveness in the setting with two packet characteristics, as they do in cases, where there is only a single characteristic. For all three specific PQ policies shown above, we prove linear (in V and/or W) lower bounds on their competitive ratios against an optimal algorithm. This is an interesting and somewhat discouraging result since priority queues have proven to be efficient when each characteristic is considered separately, often with constant upper bounds on the competitive ratio or even shown to be optimal policies. Note that while it may seem intuitive that $\mathrm{PQ}_{v/w}$ should be best at least among these three, we will see lower bounds for all of them in this section, and later an even more counterintuitive result in a special case with two values.

For a lower bound, it suffices to present a hard sequence of packets on which the optimal algorithm outperforms the one in question; so in the theorems below we simply describe this sequence. We also show matching upper bounds when applicable.

Theorem 1: For a buffer of size B, maximal packet required processing W, and maximal packet value V, $\mathrm{PQ}_{-w,v}$ is at least V-competitive and at most V-competitive.

Proof: First, there arrive $B \times (1 \mid 1)$ packets, i.e., B packets with required processing 1 and value 1; $\mathrm{PQ}_{-w,v}$ accepts them while OPT does not. Then there arrive $B \times (2 \mid V)$ packets accepted by OPT; $\mathrm{PQ}_{-w,v}$ skips them since they have larger required processing than already admitted. No more packets arrive, so in $2B$ steps $\mathrm{PQ}_{-w,v}$ processes packets with total value B; OPT, with total value VB. The same sequence is repeated to get the asymptotic bound. The upper bound follows since PQ is optimal for uniform values and variable required processing; this means that $\mathrm{PQ}_{-w,v}$ processes as many packets as OPT, so it cannot lose by a factor of more than V. ■

[1]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.

Theorem 2: For a buffer of size B, maximal packet required processing W, and maximal packet value V, $\mathrm{PQ}_{v,-w}$ is at least $\left(\frac{(V-1)}{V}W - o(1)\right)$-competitive.

Proof: In the first burst, there arrive $B \times (W \mid V)$ which $\mathrm{PQ}_{v,-w}$ accepts but OPT does not. Then, over the next W time slots there arrives a $(1 \mid V-1)$ packet on every time slot. OPT accepts, processes, and transmits this packet immediately, while $\mathrm{PQ}_{v,-w}$ drops it since it has worse value than the ones in its queue. On time slot $W+1$, when $\mathrm{PQ}_{v,-w}$ has processed one $(W \mid V)$ packet, another $(W \mid V)$ arrives, to be accepted by $\mathrm{PQ}_{v,-w}$, and this brings us back to the same state as on the first time slot. Thus, over this sequence $\mathrm{PQ}_{v,-w}$ has processed packets with total value V, OPT has processed packets with total value $W(V-1)$, and the sequence can be repeated. After we repeat the sequence C times, we finish by "flushing" both buffers with $B \times (1 \mid V)$. Thus, both algorithms will end with VB more processed value, and the competitive ratio over this sequence is

$$\frac{(V-1)WC + VB}{V(B+C)}.$$

It remains to let $C \to \infty$. ∎

So is $\mathrm{PQ}_{v/w}$ that combines characteristics and aims for the best "value per timeslot" any better in the worst case? Unfortunately, no.

Theorem 3: For a buffer of size B, maximal packet required processing W, and maximal packet value V, $\mathrm{PQ}_{v/w}$ is at least $\min\{V, W\}$-competitive.

Proof: We denote $m = \min\{V, W\}$. In the first burst, there arrive $B \times (1 \mid 1)$ packets which $\mathrm{PQ}_{v/w}$ accepts but OPT does not. Then on the same time slot there arrive $B \times (m \mid m)$ packets which OPT accepts but $\mathrm{PQ}_{v/w}$ does not since they have work-to-value ratio worse than 1. Thus, in Bm steps OPT transmits total value mB, while $\mathrm{PQ}_{v/w}$ only transmits total value B, and this sequence can be repeated. ∎

V. General lower bounds

Theorems 1–3 have established that $\mathrm{PQ}_{-w,v}$, $\mathrm{PQ}_{v,-w}$, and $\mathrm{PQ}_{v/w}$ are non-competitive. But perhaps we have just failed to be inventive enough in designing these priority queues? Maybe we can devise a better priority queue, or simply a better online algorithm that will achieve constant competitiveness or even be optimal? In this section, we dash these hopes by proving *general lower bounds* on all online algorithms. 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.

Note that some of the bounds below are nontrivial only in the extreme cases of $W, V > B$ and even $W, V \gg B$, but it still shows that we cannot hope for constant upper bounds unless we explicitly assume $B > W, V$ and somehow use it in the proof. On the positive side, note that most of these lower bounds only need two kinds of packets, so they also work in restricted settings where value and/or work can only take some of the values in their respective intervals.

Later we will see that an important special case is the *two-valued case*, when required processing can be an arbitrary integer $1 \le w \le W$ but there are only two possible values, 1 and V; we will prove an upper bound for this case in Section VI. Therefore, we note special cases of lower bounds for this case as well. Note that all lower bounds trivially extend from the two-valued case to the general case but not vice versa.

We begin with a very simple case to illustrate basic ideas. It turns out that even reducing the buffer to a single slot does not let us construct a competitive online algorithm.

The basic idea for the following and most other general lower bounds is to have "light" and "heavy" packets such that it is x times better to process "light" packets per time slot, but a "heavy" packet is x times better than a single "light" packet, so if the algorithm pushes out the "heavy" packet, we can stop the arrivals and win x times the value again. This is where the $\sqrt{W}$ that appears here and in many further bounds comes from: since a "heavy" packet is x times worse per time slot but x times better overall, it must have x^2 times the processing of a "light" packet.

Theorem 4 ($B = 1$): 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 $B = 1$.

Proof: On the first step, two packets arrive, $(W \mid V)$ and $\left(1 \mid \frac{V}{\sqrt{W}}\right)$. If ALG accepts $\left(1 \mid \frac{V}{\sqrt{W}}\right)$, no other packets arrive, OPT accepts $(W \mid V)$ and wins by a factor of $\sqrt{W}$ in value. If ALG accepts $(W \mid V)$, the same pair of packets, $\left(1 \mid \frac{V}{\sqrt{W}}\right)$ and $(W \mid V)$, continue to arrive every tick, OPT processes light packets and earns value $V\sqrt{W}$ while ALG earns V. If ALG decides to switch to a lighter packet in the process, arrivals stop immediately, OPT accepts the current $(W \mid V)$, and the result is even worse for ALG. For the case when the value of $V/\sqrt{W}$ is either unavailable or is less than or equal to one, replace $\left(1 \mid \frac{V}{\sqrt{W}}\right)$ with $(1 \mid 1)$ and observe that in the first case we get a ratio of V and in the second, W/V. ∎

In traditional networking, most buffers implement FIFO processing order because of its simplicity and desired properties. The following theorem demonstrates that under the FIFO constraint on processing and transmission order, lower bounds may significantly deteriorate and become non-competitive. Note that the admission order is not constrained in the theorem, ALG is free to push out any packet.

Theorem 5 (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 order is at least $\frac{V+B-1}{W+B-1}$-competitive.

Proof: The construction is very similar: $(W \mid V)$ and $B \times \left(1 \mid \frac{V}{\sqrt{W}}\right)$ arrive on the first step. Then ALG either:

(a) drops $(W \mid V)$, in which case arrivals stop, and OPT earns total value $V + (B-1)\frac{V}{\sqrt{W}}$ while ALG earns $B\frac{V}{\sqrt{W}}$, for the total ratio of $\frac{V+(B-1)V/\sqrt{W}}{BV/\sqrt{W}} = \frac{\sqrt{W}}{B} + 1 - \frac{1}{B}$; or

(b) keeps processing $(W \mid V)$ while we feed OPT with more $\left(1 \mid \frac{V}{\sqrt{W}}\right)$s; then arrivals stop, ALG finishes the other $B-1$ of his packets and earns total value $V + (B-1)\frac{V}{\sqrt{W}}$

while OPT has earned $(W+B-1)\frac{V}{\sqrt{W}}$ in value; the ratio in this case is worse,

$$\frac{(W+B-1)V/\sqrt{W}}{V+(B-1)V/\sqrt{W}} = \frac{W+B-1}{\sqrt{W}+B-1} > \frac{\sqrt{W}}{B}+1.$$

For the second part, replace $\left(1 \mid \frac{V}{\sqrt{W}}\right)$ with $(1 \mid 1)$ and observe that in case (a) we get a competitive ratio of $\frac{V+B-1}{B}$, and in case (b) the competitive ratio is $\frac{V+B-1}{W+B-1}$, which is smaller. ■

Next, we turn to priority queues. We have already mentioned that priority queues arise naturally as candidates for best possible policies, but, again, by restricting our consideration to the class of priority queues (i.e., algorithms that have deterministic linear orderings on the packets) we get a larger general lower bound, regardless of what this order specifically is. We would like to emphasize that not all algorithms can be represented as PQ_f. For example, some algorithms may base their decisions on buffer occupancy or statistical data collected over previous timeslots. In particular, Theorem 4 is not a special case of the following theorem.

Theorem 6 (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 $V \le \sqrt{W}$, algorithm PQ_f is at least $\min\{V, W/V\}$-competitive.

Proof: Again, we only need two kinds of packets, $(W \mid V)$ and $\left(1 \mid V/\sqrt{W}\right)$. There are two cases:

(1) if PQ_f prefers $(W \mid V)$, i.e. $f(W,V) > f(1, V/\sqrt{W})$, we keep feeding the algorithms with both kinds of packets; PQ_f chooses and processes $(W \mid V)$s while OPT is processing $\left(1 \mid V/\sqrt{W}\right)$s, getting $\sqrt{W}$ times more value per time slot;

(2) if, on the other hand, PQ_f prefers $\left(1 \mid V/\sqrt{W}\right)$ to $(W \mid V)$, then $B \times \left(1 \mid V/\sqrt{W}\right)$ and $B \times (W \mid V)$ arrive on the first burst and then arrivals stop; ALG fills its buffer with light packets, OPT takes the heavy ones, and after BW time slots OPT has again transmitted $\sqrt{W}$ times more value.

For the second part, again, replace $\left(1 \mid \frac{V}{\sqrt{W}}\right)$ with $(1 \mid 1)$. ■

Finally, we consider general lower bounds for all deterministic online algorithms. We begin with the two-valued case. The idea of the following lower bound is to send in plenty of both "light" packets $(1 \mid 1)$ and "heavy" packets $(W \mid V)$. If ALG accepts few "heavy" packets, OPT can accept all of them, halt arrivals, and win a lot with the sequence. If ALG accepts a lot of "heavy" packets, OPT fully processes all "light" packets, winning over ALG that has to begin processing "heavy" ones, and then the buffers are flushed out with the "best" possible packets $(1 \mid V)$.

Theorem 7: For a buffer of size B, maximal packet required processing W, and available packet values 1 and $V > 1$, every online deterministic algorithm ALG is at least $\left(1 + \frac{V-1}{V^2} - O\left(\frac{1}{W}\right)\right)$-competitive.

Proof: On the first step there arrive $B \times (1 \mid 1)$ and $B \times (W \mid V)$. Suppose that ALG has accepted n of $(W \mid V)$ packets. Again, there are two cases.

- $\mathbf{n < \frac{V^2-1}{V^2+V-1}B}$: in this case, OPT chooses to accept $B \times (W \mid V)$, and no new packets arrive. After BW time slots, OPT has processed packets with total value VB, and ALG has processed at most $(B-n)+Vn$, yielding competitive ratio

$$\frac{VB}{B-n+Vn} = \frac{V}{1+(V-1)\frac{n}{B}},$$

which is at least $\frac{V^2+V-1}{V^2}$ for $\frac{n}{B} < \frac{V^2-1}{V^2+V-1}$.

- $\mathbf{n \ge \frac{V^2-1}{V^2+V-1}B}$: in this case, OPT accepts $B \times (1 \mid 1)$. After B time slots, OPT has transmitted total value B, while ALG has processed at most $(B-n) \times (1 \mid 1)$ and $B/W \times (W \mid V)$. Now $B \times (1 \mid V)$ arrive and then no more packets; after all buffers have been emptied OPT gets VB more value and ALG at most VB, which yields the ratio

$$\frac{B+VB}{(B-n)+VB/W+VB} = \frac{V+1}{V+1-n/B} - O\left(\frac{1}{W}\right) \ge$$
$$\ge \frac{V^2+V-1}{V^2} - O\left(\frac{1}{W}\right) \text{ for } \frac{n}{B} \ge \frac{V^2-1}{V^2+V-1}.$$

■

These same ideas can lead to a general lower bound on all deterministic online algorithms. We first show the proof for $B = 1$ here and then show the proofs for $B = 2$ as a characteristic special case and then for the general case of arbitrary B. These proofs essentially rely on the availability of a packet whose value is a specific fraction of V, and thus they are not applicable in the two-valued case. The proof for the general case is technically quite involved so we consider in detail the case of $B = 2$ and then show a proof sketch for arbitrary B. The basic idea of "light" and "heavy" packets remains the same, but for buffer size B we will need $B+1$ "levels" of different packets to get a recursive construction and an inductive proof of the bound. This leads to $\sqrt[4]{W}$ in the case of $B = 2$ and $\sqrt[2B]{W}$ in the general case.

Theorem 8 (arbitrary online ALG, $B = 1$): For $B = 1$, the competitive ratio of any deterministic online algorithm ALG is at least $\min\{\sqrt{W}, V\}$.

Proof: There are only two kinds of packets involved in the lower bound: "heavy" packets $(W \mid V)$ and "medium" packets $\left(\frac{W}{l^2} \mid \frac{V}{l}\right)$, where l is a parameter to be defined later. On the first burst, both packets arrive, $1 \times (W \mid V)$ and $1 \times \left(\frac{W}{l^2} \mid \frac{V}{l}\right)$, and then on every time slot $1 \times \left(\frac{W}{l^2} \mid \frac{V}{l}\right)$ arrives until ALG accepts it. Denote by t the time when ALG accepts a "medium" packet instead of $(W \mid V)$. There are two cases.

1. $t = W$, i.e., ALG processes $(W \mid V)$ to completion. In this case, we repeat the sequence by sending another $(W \mid V)$ after W time slots. OPT will process "medium" packets all the time, getting total value of Vl per W time slots while ALG obtains value V per W time slots.
2. At some $t < W$, ALG accepts a "medium" packet, pushing out the "heavy" one. In this case, "medium" packets immediately stop, and OPT processes only the "heavy" packet with value V while ALG processes one "medium" packet with total value $\frac{V}{l}$. Then both buffers become empty, and the sequence can be repeated.

We now take $l = \min\{\sqrt{W}, V\}$ to get the bound. ■

Theorem 9 (arbitrary online ALG, $B = 2$): For $B = 2$, the competitive ratio of any deterministic online algorithm ALG is at least $\frac{1}{2}\min\{\sqrt[4]{W}, \sqrt{V}\}$.

Proof: The basic idea is to preserve the following invariant: on every step except a small fraction OPT will obtain at least l times more value than ALG. There are three kinds of packets in the sequence: "heavy" $(W \mid V)$, "medium" $\left(\frac{W}{l^2} \mid \frac{V}{l}\right)$, and "light" $\left(\frac{W}{l^4} \mid \frac{V}{l^2}\right)$, where l is a parameter to be defined later. On the first burst, there arrive a "heavy" and a "medium" packet, $1 \times (W \mid V)$ and $1 \times \left(\frac{W}{l^2} \mid \frac{V}{l}\right)$, followed by two "light" packets, $2 \times \left(\frac{W}{l^4} \mid \frac{V}{l^2}\right)$. Then, on every time step two more "light" packets arrive, $2 \times \left(\frac{W}{l^4} \mid \frac{V}{l^2}\right)$.

There are several cases. Note that in what follows, OPT always keeps one "heavy" packet in its buffer, and "heavy" packets never arrive during the period we are counting the packets in. A new "heavy" packet only arrives when the sequence is repeated from the start.

1. ALG does not push out either "heavy" or "medium" packet in favor of "light" ones. Then OPT keeps one "heavy" packet in the buffer while it processes "light" packets, getting value $\frac{Vl^2}{W}$ per processing cycle while ALG is getting at most $\frac{Vl}{W}$ (from the "medium" packet). As soon as ALG finishes the "medium" or "heavy" packet, another packet of the same kind arrives, and the sequence is repeated. Note that this ratio of l in favor of OPT occurs every time OPT is able to process a "light" packet while ALG is processing a different one.
2. At some timeslot t, ALG pushes out the "heavy" packet for a "light" one. In this case, OPT accepts the last arriving "medium" packet (note that there may have been several "medium" packets arriving due to ALG processing them to completion), and there are no more arrivals after timeslot t. OPT finishes the "heavy" packet residing in its buffer and the last arriving "medium" packet while ALG can process at most a "medium" and a "light" one. As a result, before time t OPT had l times more value per timeslot by case 1, and after time t ALG has earned at most $V(\frac{1}{l} + \frac{1}{l^2})$ total value while OPT has earned $V(1 + \frac{1}{l})$, for the total ratio of l over the entire sequence.
3. At some timeslot t, ALG pushes out the "medium" packet for a "light" one. This is a slightly more complicated case, with two subcases depending on the time t' when the pushed out "medium" packet had arrived:
   (i) if $t - t' < \frac{W}{l^2}$ (the pushed out "medium" packet arrived less than $\frac{W}{l^2}$ timeslots ago), OPT accepts it at time t', and all arrivals stop until time $t' + \frac{W}{l^2}$, i.e., until OPT finishes processing it; over these $\frac{W}{l^2}$ timeslots:
   - OPT is getting value $\frac{Vl}{W}$ per time slot every time, for a total value $\frac{V}{l}$;
   - ALG can get total value $\frac{V}{l^2}$ once by processing this "light" packet, but on all other timeslots it could not get more than $\frac{V}{W}$ value per time slot (assuming it was processing the "heavy" packet); hence, over this period of time ALG gets no more than $\frac{V}{l^2} + \frac{V}{l^2}$ total value;

   hence, the competitive ratio over these timeslots is at least $\frac{l}{2}$;
   (ii) if $t - t' \geq \frac{W}{l^2}$ (the pushed out "medium" packet arrived at least $\frac{W}{l^2}$ timeslots ago), OPT is processing "light" packets all this time, and as soon as ALG has finished the "light" packet, another "medium" packet arrives, reverting to the original situation; in this case:
   - OPT has processed $\frac{(t-t')l^4}{W}$ "light" packets plus the one final "light" packet, obtaining total value at least $\frac{(t-t')Vl^2}{W} + \frac{V}{l^2}$;
   - ALG has obtained total value at most $\frac{(t-t')V}{W}$ over the past $t - t'$ timeslots (if ALG was processing the "medium" packet it is now worth nothing since it has been pushed out, so value can only come from the "heavy" packet) plus the final "light" packet, for a total of at most $\frac{(t-t')V}{W} + \frac{V}{l^2}$;

   since $t - t' \geq \frac{W}{l^2}$, the competitive ratio is at least

$$\frac{\frac{(t-t')Vl^2}{W} + \frac{V}{l^2}}{\frac{(t-t')V}{W} + \frac{V}{l^2}} \geq \frac{V + \frac{V}{l^2}}{2\frac{V}{l^2}} = \frac{l^2 + 1}{2}.$$

As a result, during the entire sequence the competitive ratio is never smaller than $\frac{l}{2}$, and the constraint on l is that $l \leq \min\{\sqrt[4]{W}, \sqrt{V}\}$. ■

Theorem 10 (general case): For arbitrary B, the competitive ratio of any deterministic online algorithm ALG is at least $\frac{1}{2}\left(\min\{\sqrt[2B]{W}, \sqrt[B]{V}\} - 1\right)$.

Proof of Theorem 10: We proceed by induction with a construction similar to the proof of Theorem 9. The induction base for $B = 1$ and $B = 2$ has already been considered in Theorems 8 and 9.

For the induction step, consider $k + 1$ types of packets that differ by a factor of v from each other in value and by a factor of v^2 in required processing: the first packet has value 1 and work 1, the second value v and work v^2, and so on until value v^k and work v^{2k}; for this to work we have to have $v \leq \min\{\sqrt[2B]{W}, \sqrt[B]{V}\}$. In the arrivals, we will preserve the invariant that ALG's buffer can only have two packets of identical value if they are the cheapest; i.e., we will not give "repeated" packets to the algorithm but send in a new packet of a certain size only after the previous one has finished processing or has been dropped; the cheapest and lightest packets are coming in with a steady stream, and in most cases OPT keeps processing them.

If ALG at some point pushes out the heaviest packet, arrivals stop immediately, and OPT finishes the heaviest packet; in this case, ALG loses by a factor of at least $\frac{1+v+v^2+\cdots v^{k-1}}{v^k}$. If ALG is working on the heaviest packet, OPT is working on the lightest packet and has unit gain over ALG by a factor of at least $v^k - 1$.

If, otherwise, ALG keeps the heaviest packet in the buffer and is working on some other packet, arrivals and OPT are operating as in the induction hypothesis for $B - 1$, with one slot in the buffer reserved for the heaviest packet. There is only one exception to this operation: if ALG has pushed out the second heaviest packet (which is heaviest in the $B - 1$ case), arrivals do not stop completely but cease temporarily for $v^{2(k-1)}$, i.e., for the time it takes to process the second

heaviest packet. Afterwards, we again send in one packet of each type except the heaviest.

The technicality here is that by sending in these packets to "reset" the buffer, we are violating the declared invariant. To fix this, we count all packets except the heaviest one as "processed" by the algorithm and assume that it has gotten full value for them (afterwards, if ALG is working on one of the "old" packets, we can simulate it as idle time). In total, OPT has processed v^{k-1} total value, and ALG has not processed more than $1 + v + v^2 + \cdots v^{k-2}$ which constitutes at most $v^k/v^2 = v^{k-2}$ of the "unit" cost of the heaviest packet.

Now the minimal competitive ratio for ALG out of the three cases is the last one:

$$\frac{v^{k-1}}{v^{k-2} + \sum_{0 \le i \le k-2} v^i} = \frac{v^{k-1}}{v^{k-2} + \frac{v^{k-1}-1}{v-1}} \ge \frac{1}{2}(v-1),$$

and we take $v = \min\{ \sqrt[2B]{W},\ \sqrt[B]{V}\}$ to obtain the lower bound. ■

Note that while this bound is not too large in practical cases, it is still non-constant, that is, we now cannot hope for an online algorithm with constant competitiveness unless we impose and use some additional constraints on the problem setting. One fruitful constraint turns out to be the constraint that the only two allowed values are 1 and V.

VI. Upper bound for the two-valued case

Given the pessimistic results of previous two sections, it remains only to impose additional constraints on at least one of the characteristic and try to distinguish important special cases under which a "good" upper bound may exist. In this section, we consider an important special case when there are only two possible packet values, 1 and V, so there are two kinds of packets, $(w \mid 1)$ and $(w \mid V)$; the required processing can still vary from 1 to W. This case often occurs in practice; for instance, $(w \mid 1)$ may represent "commodity" packets while $(w \mid V)$ corresponds to "golden" packets that have paid more to be processed. Similar special cases have been considered, e.g., in [23].

We will show in Theorem 12 that in this special case, the $\mathrm{PQ}_{v,-w}$ policy has an attractive upper bound on the competitive ratio, which implies a constant upper bound on the competitive ratio of both $\mathrm{PQ}_{v,-w}$ and $\mathrm{PQ}_{v/w}$ in case when $W < V$. This upper bound is fundamentally different from the lower bounds presented earlier: instead of showing that an algorithm (or a set of those) sometimes performs badly, it shows that these particular algorithms *always* perform well. However, we begin with negative results; Theorem 11 provides matching tight lower bounds for the main result that follows. A plot summarizing different lower bounds for the two-valued case is shown on Figure 2.

Theorem 11: Consider a buffer of size B with maximal required processing W and possible packet values 1 or V. Then:

(1) $\mathrm{PQ}_{-w,v}$ is at least V-competitive;
(2) if $W \ge V$ then $\mathrm{PQ}_{v/w}$ is at least V-competitive;
(3) $\mathrm{PQ}_{v,-w}$ is at least $\left(\frac{W}{V} + o(1)\right)$-competitive.

Proof:

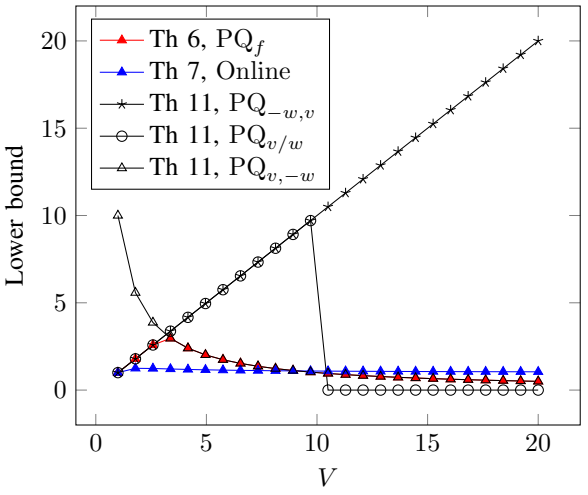


Fig. 2. Lower bounds on the competitive ratios for the two-valued case with fixed $W = 10$. Higher value of a lower bound is better.

(1) The construction from Theorem 1 uses only packets of values 1 and V.
(2) Again, we present a hard sequence of arrivals. In the first burst, there arrive $B \times (1 \mid 1)$ accepted by $\mathrm{PQ}_{v/w}$ but not OPT. Then, on the same time slot there arrive $B \times (W \mid V)$ which $\mathrm{PQ}_{v/w}$ has to miss since $\frac{W}{V} \ge \frac{V}{V} = \frac{1}{1}$ but which OPT accepts. Then, in BW steps, $\mathrm{PQ}_{v/w}$ will have processed total value B while OPT will have processed total value BV, which implies the bound.
(3) In the first time slot, there arrive $B \times (W \mid V)$, which $\mathrm{PQ}_{v,-w}$ accepts, but OPT does not. Then, over the next W time slots there arrives a $(1 \mid 1)$ on every time slot. OPT transmits it, and $\mathrm{PQ}_{v,-w}$ drops it. On time slot $W+1$, when $\mathrm{PQ}_{v,-w}$ has processed one $(W \mid V)$ packet, another $(W \mid V)$ packet arrives, which brings us back to the same state as on the first time slot. Over this sequence $\mathrm{PQ}_{v,-w}$ has processed packets with total value V, and OPT with total value W. Repeating this sequence C times, we get competitive ratio $\frac{WC+O(1)}{VC+O(1)}$ and let $C \to \infty$.

■

In the next theorem, we show one of the main results of this work, an upper bound on the competitive ratio of $\mathrm{PQ}_{v,-w}$. Note that the lower bounds from Theorem 11 and previous sections, which were linear in V, do not work for $\mathrm{PQ}_{v,-w}$ since in the two-valued case, it always processes packets with value V first and with value 1 last, so intuitively we cannot lose more than the worst possible packet with value V, $(W \mid V)$, against the best possible packet with value 1, $(1 \mid 1)$ (Theorem 11 (3) shows that we cannot lose any less). The following proof captures this intuition.

Theorem 12: Consider a buffer of size B with maximal required processing W and possible packet values 1 or V. Then $\mathrm{PQ}_{v,-w}$ is at most $\left(1 + \frac{W+2}{V}\right)$-competitive.

Proof: By the definition of the $\mathrm{PQ}_{v,-w}$ queue, any packet with value V pushes out any packet with value 1. This is the crucial property that we need to prove this upper bound. For brevity, throughout this proof we denote $\mathrm{PQ} = \mathrm{PQ}_{v,-w}$. We define the following sets of packets:

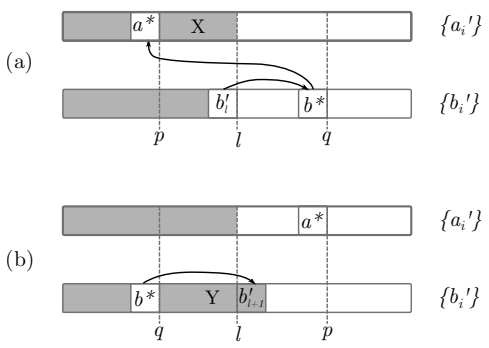


Fig. 3. Illustration for the subcases of Lemma 13: subcase $\mathbf{p} \leq \mathbf{q}$ is shown on (a), and subcase $\mathbf{q} < \mathbf{p}$ is shown on (b). Arrows represent $\leq$ relation, and shaded areas denote sets of elements that sum to either A_l or B_l (note that lemma's premise contains a $A_l \geq B_l$ inequality); dotted lines denote specific position in a sequence.

(1) $\mathrm{IB}_v^{\mathrm{ALG}} = \{p \in \mathrm{IB}^{\mathrm{ALG}} : \mathrm{value}(p) = v\}$ contains packets with value v in $\mathrm{IB}^{\mathrm{ALG}}$;
(2) $\mathrm{Trans}_v^{\mathrm{ALG}} = \{p \text{ transmitted by ALG} : \mathrm{value}(p) = v\}$ is the set of packets with value v transmitted by ALG; $\mathrm{Trans}^{\mathrm{ALG}} = \bigcup_v \mathrm{Trans}_v^{\mathrm{ALG}}$;
(3) $\mathrm{IBT}_v^{\mathrm{ALG}} = \mathrm{IB}_v^{\mathrm{ALG}} \cup \mathrm{Trans}_v^{\mathrm{ALG}}$ is the set of packets with value v either already transmitted by ALG or currently residing in its buffer; $\mathrm{IBT}^{\mathrm{ALG}} = \bigcup_v \mathrm{IBT}_v^{\mathrm{ALG}}$.

We also define $\Phi_v^{\mathrm{ALG}}(l) = \sum_{i=1}^{l} w(p_i)$, where p_i is the ith packet from $\mathrm{IBT}_v^{\mathrm{ALG}}$ in PQ order. Here $w(p)$ is the residual processing time of a packet at the current time moment; in particular, we let $w(p) = 0$ for already transmitted packets.

In the proof, we will sometimes force OPT to transmit certain packets immediately, "for free", thus improving its throughput. We denote the set of these packets at the current timeslot as $\mathrm{Free}^{\mathrm{OPT}}$; they do not fall into $\mathrm{Trans}^{\mathrm{OPT}}$ but rather contribute to the objective separately. The only requirement is that for any $p \in \mathrm{Free}^{\mathrm{OPT}}$ we must have $\mathrm{value}(p) = 1$, i.e., we only give out packets of value 1 for free. We begin with a technical statement.

Lemma 13: Let $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_m$ be two sequences of numbers in nondecreasing order, and, moreover, suppose that $\forall l \in \{1, \ldots, m\}$ the prefix sums of length l satisfy the following inequality: $\sum_{i=1}^{l} a_i \geq \sum_{i=1}^{l} b_i$. Let also a^* and b^* be any two numbers, such that $a^* \geq b^*$. If a^* is inserted into $a_1, \ldots, a_m$, and b^* is inserted into $b_1, \ldots, b_m$ then the prefix sums of resulting sequences satisfy the same inequality. Formally, if the result of a^*'s insertion is $a_1', a_2', \ldots, a_{m+1}'$ and the result of b^*'s insertion is $b_1', b_2', \ldots, b_{m+1}'$, then we have, that $\forall l \in \{1, \ldots, m+1\}$ $\sum_{i=1}^{l} a_i' \geq \sum_{i=1}^{l} b_i'$.

Proof: Denote right and left hand sides of inequalities before (after) insertion as A_l and B_l (A_l' and B_l') respectively. Let p and q be positions of a^* and b^* in the new sequences. Assume that $l < \min\{p, q\}$, then inequalities hold since *none* of the inserted values lie in the prefix of length l, consequently, $A_l' = A_l$ and $B_l' = B_l$. If $l \geq \max\{p, q\}$ then *both* inserted values lie in the prefix of length l, thus we have $A_l' = A_{l-1} + a^*$ and $B_l' = B_{l-1} + b^*$, and it is easy to see that $A_l' \geq B_l'$. The remaining case splits into two subcases (see Figure 3).

$\mathbf{p \leq q}$. Denote $X = \sum_{i=p+1}^{l} a_i'$. See next that $A_l' = A_{p-1} + a^* + X$, $B_l' = B_{l-1} + b_l'$, and also $A_{p-1} + X = A_{l-1} \geq B_{l-1}$. Due to nondecreasing order: $a^* \geq b^* \geq b_l'$, and we easily get the required $A_l' \geq B_l'$.

$\mathbf{q < p}$. Denote $Y = \sum_{i=p+1}^{l} b_i'$. This gives us: $A_l' = A_l$, $B_l' = B_{p-1} + b^* + Y$, and we have $A_l \geq B_l = B_{p-1} + Y + b_{l+1}'$. Again, due to nondecreasing order: $b_{l+1}' \geq b^*$, and claimed inequality can be easily derived.

■

We now prove the crucial lemma for this upper bound.

Lemma 14: There exist an algorithm OPT that works no worse than the optimal algorithm on any sequence of inputs and such a choice of $\mathrm{Free}^{\mathrm{OPT}}$, that on every sequence of inputs at every time moment it holds that:

(1) $|\mathrm{IBT}_1^{\mathrm{PQ}}| \geq |\mathrm{IBT}_1^{\mathrm{OPT}}|$, and for all l, s.t. $l \leq |\mathrm{IBT}_1^{\mathrm{OPT}}|$ it holds that $\Phi_1^{\mathrm{OPT}}(l) \geq \Phi_1^{\mathrm{PQ}}(l)$;
(2) $|\mathrm{IBT}_V^{\mathrm{PQ}}| \geq |\mathrm{IBT}_V^{\mathrm{OPT}}|$, and for all l, s.t. $l \leq |\mathrm{IBT}_V^{\mathrm{OPT}}|$ it holds that $\Phi_V^{\mathrm{OPT}}(l) \geq \Phi_V^{\mathrm{PQ}}(l)$;

Proof: We prove these estimates by induction on the number of "events" such as receiving, processing, or transmitting a packet. At the initial time moment all conditions hold trivially. Note that whenever conditions (1) and (2) hold, it also necessarily holds that $|\mathrm{Trans}_v^{\mathrm{OPT}}| \leq |\mathrm{Trans}_v^{\mathrm{PQ}}|$ since $\Phi_v^{\mathrm{OPT}}(l) = 0$ for all $l \leq |\mathrm{Trans}_v^{\mathrm{OPT}}|$ and consequently $\Phi_v^{\mathrm{PQ}}(l) = 0$. We now remove from $\mathrm{IBT}_v^{\mathrm{PQ}}$ the $(|\mathrm{IBT}_v^{\mathrm{PQ}}| - |\mathrm{IBT}_v^{\mathrm{OPT}}|)$ packets with the lowest priority, denoting the resulting set by $\widetilde{\mathrm{IBT}}_v^{\mathrm{PQ}}$ and the corresponding set of packets in the buffer by $\widetilde{\mathrm{IB}}_v^{\mathrm{PQ}}$. Then all of the above implies that $|\mathrm{IB}_v^{\mathrm{OPT}}| \geq |\widetilde{\mathrm{IB}}_v^{\mathrm{PQ}}|$.

Upon acceptance we may mark a packet admitted to OPT buffer as "causing overflow". The set of such packets is denoted as $\mathrm{Over}^{\mathrm{OPT}}$, and it does not contribute to $\mathrm{IBT}^{\mathrm{OPT}}$. The induction step will guarantee that every packet in $\mathrm{Over}^{\mathrm{OPT}}$ is moved eventually to $\mathrm{Free}^{\mathrm{OPT}}$ and the following invariant holds: $|\mathrm{Over}^{\mathrm{OPT}}| \leq |\mathrm{IB}_V^{\mathrm{PQ}}| - |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|$.

Let us now consider all possible events one by one and show that none of them violates the conditions of the theorem.

Arrival of a new packet $\mathbf{p}$. There are two subcases.

$\mathbf{value(p) = V}$. If PQ has accepted the packet and has pushed out a packet from $\mathrm{IB}_1^{\mathrm{PQ}}$, we move the heaviest packet from $\mathrm{IB}_1^{\mathrm{OPT}}$ (if it is nonempty) to $\mathrm{Free}^{\mathrm{OPT}}$. Thus, inequalities for Φ_1 are not violated since we have removed largest elements from both $\mathrm{IB}_1^{\mathrm{OPT}}$ and $\mathrm{IB}_1^{\mathrm{PQ}}$. Further, if OPT accepts p, then $B > |\mathrm{IB}_V^{\mathrm{OPT}}| \geq |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|$, so the sequence $\mathrm{IBT}_V^{\mathrm{PQ}}$ will receive the lightest of packets $(\mathrm{IB}_V^{\mathrm{PQ}} \setminus \widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}) \cup \{p\}$ (according to the push-out rules). Therefore, by Lemma 13 inequalities for Φ_V still hold. The only time $|\mathrm{IB}_V^{\mathrm{PQ}}| - |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|$ increases is when $|\mathrm{IB}_V^{\mathrm{PQ}}| = B$ and OPT accepts, but $|\mathrm{IB}_V^{\mathrm{OPT}}| \geq |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|$ and $|\mathrm{Over}^{\mathrm{OPT}}| + |\mathrm{IB}_V^{\mathrm{OPT}}| < B$ together give $|\mathrm{Over}^{\mathrm{OPT}}| < |\mathrm{IB}_V^{\mathrm{PQ}}| - |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|$.

$\mathbf{value(p) = 1}$. We consider two subcases separately. (i) $|\mathbf{IB_V^{PQ}}| + |\widetilde{\mathbf{IB}}_\mathbf{1}^{\mathbf{PQ}}| < \mathbf{B}$. In this case, we add to $\widetilde{\mathrm{IB}}_1^{\mathrm{PQ}}$ the lightest packet from $(\mathrm{IB}_1^{\mathrm{PQ}} \setminus \widetilde{\mathrm{IB}}_1^{\mathrm{PQ}}) \cup \{p\}$ (according to push-out rules), and by Lemma 13 the inequalities are preserved. (ii) $|\mathbf{IB_V^{PQ}}| + |\widetilde{\mathbf{IB}}_\mathbf{1}^{\mathbf{PQ}}| = \mathbf{B}$. Then, since

$|\mathrm{IB}^{\mathrm{OPT}}| \geq |\widetilde{\mathrm{IB}}^{\mathrm{PQ}}|$ and $|\mathrm{IB}^{\mathrm{OPT}}| + |\mathrm{Over}^{\mathrm{OPT}}| < B$, we get that $|\mathrm{IB}_V^{\mathrm{PQ}}| - |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}| > |\mathrm{Over}^{\mathrm{OPT}}|$. Now, if OPT accepts p then p is added to the $\mathrm{Over}^{\mathrm{OPT}}$.

OPT **processes a packet p**. There are three subcases.

- $\mathbf{value(p) = V}$. This is a simple case. If $\mathrm{IB}_V^{\mathrm{PQ}} \neq \emptyset$ then each nonzero term in $\Phi_V^{\mathrm{PQ}}(l)$ reduces exactly by one, while each nonzero term in $\Phi_V^{\mathrm{OPT}}(l)$ reduces by at most one, so the inequalities are obviously preserved. If otherwise $\mathrm{IB}_V^{\mathrm{PQ}} = \emptyset$ then all $\Phi_V^{\mathrm{PQ}} = 0$.
- $\mathbf{value(p) = 1}$ **and** $\mathbf{IB_V^{PQ} = \emptyset}$. Similar to the previous.
- $\mathbf{value(p) = 1}$ **and** $\mathbf{IB_V^{PQ} \neq \emptyset}$. In this case, p is sent to $\mathrm{Free}^{\mathrm{OPT}}$. It remains to note that $\Phi_1^{\mathrm{OPT}}(l)$ do not decrease since we have merely removed an element from an ordered sequence.

Transmitting a packet. Inequalities on Φ obviously remain unchanged upon transmission; however, the value of $(|\mathrm{IB}_V^{\mathrm{PQ}}| - |\widetilde{\mathrm{IB}}_V^{\mathrm{PQ}}|)$ can decrease by one. If $\mathrm{Over}^{\mathrm{OPT}}$'s invariant is violated, we move an arbitrary packet from $\mathrm{Over}^{\mathrm{OPT}}$ to $\mathrm{Free}^{\mathrm{OPT}}$.

■

Lemma 15: The set $\mathrm{Free}^{\mathrm{OPT}}$ constructed in Lemma 14 satisfies the inequality $|\mathrm{Free}^{\mathrm{OPT}}| \leq (W+2)|\mathrm{Trans}_V^{\mathrm{PQ}}|$ after algorithms finish processing the input sequence.

Proof: It suffices to note that in the proof of Lemma 14 a packet may fall into $\mathrm{Free}^{\mathrm{OPT}}$ only when PQ receives, processes, or transmits a packet with value V. ■

Now, after both OPT and PQ have processed the entire sequence of packets, the total value of packets transmitted by PQ equals $|\mathrm{IBT}_1^{\mathrm{PQ}}| + V|\mathrm{IBT}_V^{\mathrm{PQ}}|$. The total value of packets transmitted by OPT is $|\mathrm{IBT}_1^{\mathrm{OPT}}| + V|\mathrm{IBT}_V^{\mathrm{OPT}}| + |\mathrm{Free}^{\mathrm{OPT}}|$. Thus, using the lemma's inequalities, the competitive ratio α can be bounded as follows:

$$\begin{aligned}\alpha &\leq \frac{|\mathrm{IBT}_1^{\mathrm{OPT}}| + V|\mathrm{IBT}_V^{\mathrm{OPT}}| + |\mathrm{Free}^{\mathrm{OPT}}|}{|\mathrm{IBT}_1^{\mathrm{PQ}}| + V|\mathrm{IBT}_V^{\mathrm{PQ}}|} \\ &\leq 1 + \frac{|\mathrm{Free}^{O}PT|}{|\mathrm{IBT}_1^{\mathrm{PQ}}| + V|\mathrm{IBT}_V^{\mathrm{PQ}}|} \leq 1 + \frac{|\mathrm{Free}^{O}PT|}{V|\mathrm{IBT}_V^{\mathrm{PQ}}|} \\ &\leq 1 + \frac{(W+2)|\mathrm{IBT}_V^{\mathrm{PQ}}|}{V|\mathrm{IBT}_V^{\mathrm{PQ}}|} \leq 1 + \frac{W+2}{V}.\end{aligned}$$

■

Corollary 16: If $W < V$, $\mathrm{PQ}_{v,-w}$ and $\mathrm{PQ}_{v/w}$ are at most $\left(2 + \frac{2}{V}\right)$-competitive.

Proof: Since $(W \mid V)$ pushes out $(1 \mid 1)$ in the $\mathrm{PQ}_{v/w}$ queue, any packet with value V pushes out any packet with value 1, so for $W < V$ $\mathrm{PQ}_{v/w}$ is equivalent to $\mathrm{PQ}_{v,-w}$. ■

Figure 4 shows a contour plot of the $\mathrm{PQ}_{v,-w}$ competitive ratio upper bound $1 + \frac{W+2}{V}$ for the two-valued case; naturally, for large V the bound is very good.

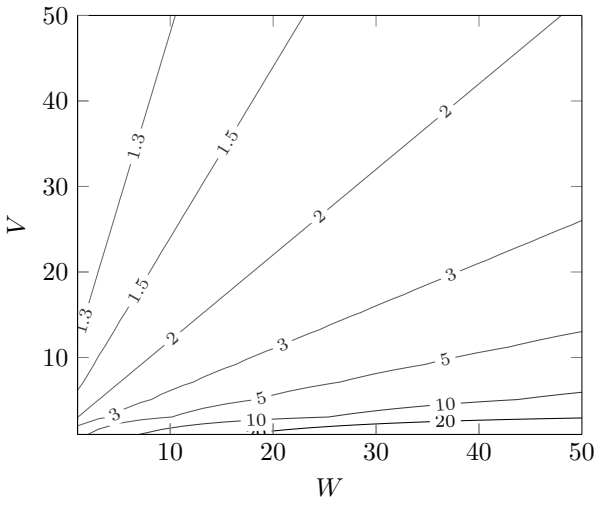


Fig. 4. Contour plot of the $\mathrm{PQ}_{v,-w}$ competitive ratio $1 + \frac{W+2}{V}$ as a function of W and V.

VII. The β-push-out case

In many networking systems, there arises an additional motivation to avoid pushouts and prioritize packets that are already in the buffer. For instance, newly admitted packets incur higher costs than packets that have already resided in the buffer since they require more access bandwidth to packet memories: a new packet incurs computational costs for constructing and updating the corresponding data structures in the network processor, and immediate push-out of a less-preferable packet can lead to increased computational overhead [14].

To represent these effects in the formal model, Keslassy et al. [14] introduced the notion of *copying cost* in the performance of transmission algorithms for packets with heterogeneous processing requirements but uniform values: if an algorithm accepts A packets and transmits packets with total value T, its transmitted value is $\max\{0, T - \alpha A\}$, i.e., each admitted packet incurs a cost α subtracted from the throughput in the objective function. Thus, in extreme cases the transmitted value of a push-out policy may even go down to zero; copying cost provides an additional control on the number of pushed out packets to avoid pathological cases. To implement such a control mechanism, Keslassy et al. [14] introduced the greedy push-out work-conserving policy PQ_β that processes a packet with minimal required work first and in the case of congestion such a policy pushes out only if a new arrival has at least β times less work then the maximal residual work in PQ_β.

However, the work [14] only dealt with a single packet characteristic, namely processing requirements. To generalize their ideas to our problem settings with two characteristics, we extend PQ_f to PQ_f^β and then show several lower bounds for β-push-out counterparts of our policies. Unfortunately, the proof of Theorem 12 cannot be directly applied to β-push-out policies; it remains an interesting open problem to show nontrivial (less than linear) upper bounds for β-push-out policies even in the two-valued case.

Definition 7.1: 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 17: Consider a buffer of size B with maximal required processing W and maximal packet value V. Then:

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

Proof: (1) In the construction from Theorem 1, packets are never pushed out from $\mathrm{PQ}_{-w,v}$ buffer, so the result still holds for $\mathrm{PQ}^{\beta}_{-w,v}$. (2) Construction from Theorem 3 also works because packets are never pushed out from $\mathrm{PQ}_{v/w}$ buffer in tnis construction. (3) This result can be seen as a relaxation of the second part of Theorem 11 since $\beta > 1$. The same construction works: $\mathrm{PQ}_{v/w}$ fills its buffer with $B \times (1 \mid 1)$ and then drops incoming $B \times (W \mid V)$. $\mathrm{PQ}^{\beta}_{v/w}$ also drops them since $\frac{V}{W} \leq \frac{\beta V}{V} = \beta$. (4) Again, the constructions from Theorem 2 and Theorem 11 work since they do not force packets to be pushed out from $\mathrm{PQ}_{-v,w}$ buffer. ■

VIII. Simulations

In this section, we present the results of a comprehensive simulation study intended to validate our theoretical results. Naturally, it would be desirable to compare the proposed algorithms on real life network traces. Unfortunately, available datasets such as CAIDA [10] are of little use for packet characteristics in our model since they do not provide data on required processing and intrinsic values of the packets. Nevertheless, we have used CAIDA traces [10] to model the incoming stream of packets, breaking down the timestamps into equal timeslots and counting the packets in each timeslot; hence, the intensity of the incoming stream below is measured in milliseconds, the size of a single timeslot.

We have conducted six series of experiments, studying how performance depends on maximal required processing W, buffer size B, maximal value V, size of a CAIDA trace timeslot t, and β (in the model with copying cost). The actual optimal online algorithm in our model would be computationally prohibitive, so to estimate and compare the competitive ratios of our algorithms we have used an algorithm which is actually better than optimal: a single priority queue with size BW that breaks each packet $(v \mid w)$ into "fractional" packets that each have required work 1 and value $\frac{v}{w}$ and then orders and processes them by this value. Since the priority queue has been proven optimal in the model with values and no required processing, it performs even better than optimal.

In our experiments, the values and processing ratios of packets were chosen uniformly from $\{1, \ldots, W\}$ and $\{1, \ldots, V\}$ respectively. We ran all experiments for $5 \cdot 10^5$ time slots with periodic "flushouts" (wait for all queues to finish their packets and then continue from an empty state), which in our experiments has proven to be sufficient for stable results. We have also performed simulations without flushouts; since the results are very close to the ones with flushouts in all settings, we do not show them separately. Note that all of our experiments venture into the values of parameters that yield high system load with large dropout rates for all algorithms; these are precisely the situations where we would like to compare performance since without heavy load and frequent congestion all reasonable algorithms perform identically. We have made the code for our experimental evaluation publicly available at GitHub [30].

Figure 5 shows simulation results presented in terms of the fraction of successfully transmitted packets: each graph shows the "better than optimal" reference algorithm in black alongside with the ratio of transmitted packets for other policies. There are five sets of experiments corresponding to the rows of Fig. 5 that will be described in subsections below; we have tested the four algorithms used in this work: $\mathrm{PQ}_{v/w,-w}$, $\mathrm{PQ}_{v/w,v}$, $\mathrm{PQ}_{-w,v}$, and $\mathrm{PQ}_{v,-w}$. Note that in all cases, $\mathrm{PQ}_{v/w,-w}$ and $\mathrm{PQ}_{v/w,v}$ are virtually indistiguishable across all settings. Thus, below we will sometimes refer to them collectively as $\mathrm{PQ}_{v/w}$.

A. Maximal required processing

In the first set of simulations (Fig. 5(1-3)), we study performance as a function of the maximal required processing W. As W grows, all algorithms deteriorate in absolute terms (packets become heavier), but it is clear that $\mathrm{PQ}_{-w,v}$, which pays more attention to required processing, fares better while $\mathrm{PQ}_{v,-w}$ loses badly as W grows. This is expected since $\mathrm{PQ}_{v,-w}$ cares little about W and therefore is likely to get stuck with very heavy packets. We see that $\mathrm{PQ}_{v/w}$ is uniformly the best policy, performing very close to OPT and deteriorating only slightly.

B. Buffer size B

In the second set of simulations (Fig. 5(4-6)), we study performance as a function of the buffer size B.

In this setting as well, $\mathrm{PQ}_{v/w,-w}$ and $\mathrm{PQ}_{v/w,v}$ remain indistinguishable, and since these experiments were done in the relatively low ranges of the W/V ratio, $\mathrm{PQ}_{v,-w}$ is also very close to $\mathrm{PQ}_{v/w}$. $\mathrm{PQ}_{v,-w}$, on the other hand, is able to store, in a larger buffer, more high-value packets and do so for longer, so as B increases and congestion decreases, $\mathrm{PQ}_{v,-w}$ becomes closer to the other three. Note, in this setting, all algorithms become significantly worse off compared to the fractional OPT through no fault of their own: the "unfairness" of fractional OPT becomes much more pronounced with large B (it has more and more extra space to store packets).

C. Maximal value V

In the third set of experiments (Fig. 5(7-9)), we look at performance as a function of the maximal value V. It turns out that while the relative performance to fractional OPT drops, the performance level of the three leading algorithms does not significantly depend on V in realistic cases, and only $\mathrm{PQ}_{v,-w}$ drops significantly in relative quality. This, again, can be explained by the fact that $\mathrm{PQ}_{v,-w}$ suffers from a wider variety of packets, getting stuck with valuable yet heavy ones. The relative order of algorithms remains unchanged.

D. Incoming stream intensity

The fourth set of experiments (Fig. 5(10-12)) shows how performance depends on the intensity of the packet source, expressed in terms of the timeslot size t (naturally, more packets on average arrive during a longer t). This setting lets us explore the most congested settings: all algorithms join together at the high end of intensity simply because now there are, on average, enough $(1 \mid V)$ packets arriving to keep all algorithms busy only with the obviously best packets. The relative standings of all algorithms remain the same throughout this increasing congestion.

E. β for β-push-out policies

The fifth set of experiments (Fig. 5(13-15)) studies a different situation; here, we have introduced nonzero copying cost α (on all three graphs, $\alpha = 0.3$) and have studied how performance depends on β for β-push-out counterparts of our policies (as introduced in Section VII); since the number of admitted packets is not well defined for our fractional OPT, OPT did not participate in these experiments; we have taken the results of $\mathrm{PQ}_{v/w,-w}$ for $\beta = 1$ as the starting point, dividing all the rest by this value. We see that in all cases, $\beta = 1$ appears to be the perfect or almost perfect choice in practice: sometimes $\beta = 1.2$ or $\beta = 1.3$ yield better results, but only slightly.

F. W for the two-valued case

The last, sixth set of experiments (Fig. 5(16-18)) deals with the two-valued case, when the intrinsic value of a packet can only take values in $\{1, V\}$ while required work can still be an arbitrary integer from 1 to W. We repeated the experiments from Section VIII-A with this additional restriction, and the results closely match our theoretical results from Section VI: contrary to the general case, now $\mathrm{PQ}_{v,-w}$ is not the obviously worst algorithm but performs on par with $\mathrm{PQ}_{v/w}$ policies, while $\mathrm{PQ}_{-w,v}$ diverges from them for larger W in exactly the same way as in the general case (compare to Section VIII-A, Fig. 5(1-3)). Since $\mathrm{PQ}_{v,-w}$ may be easier to implement than $\mathrm{PQ}_{v/w}$ (required work does not have to be considered or even known), for the two-valued case we recommend to use $\mathrm{PQ}_{v,-w}$. Again, as a side effect we see that the fractional OPT performs better (relative to other algorithms) when more buffer space is provided.

To summarize, in this section we have shown a comprehensive simulations study on synthetic traces. The main result is that the $\mathrm{PQ}_{v/w}$ policy that we have introduced in this work is uniformly the best policy across all tested settings, and there is little difference between tie-breaking variations of it, while in the two-valued case experimental results supported the theoretical conclusion that $\mathrm{PQ}_{v,-w}$ is a good policy.

IX. Conclusion

In this work, we have begun the study of buffer management for processing packets with two different characteristics: *processing requirement* and *value*. In these settings we have considered a single queue buffering architecture and have mostly studied algorithms based on priority queues; in the setting with two characteristics, there may be different reasonable priority queues that have different policies. We have investigated various packet processing orders and found that they have linear lower bounds on the competitive ratio, which makes them unattractive in the general case. However, we have provided positive results in the special case of two different values, 1 and V and heterogeneous processing requirements.

Processing policy	**General case**	**Two-valued case**	
Adversarial general lower bounds			
Any online algorithm	$\frac{\min\{\sqrt[2B]{W},\sqrt[B]{V}\}-1}{2}$	$1+\frac{V-1}{V^2}-O\left(\frac{1}{W}\right)$	
Any priority queue	$\sqrt{W}$	$\min\{V, W/V\}$	
Any FIFO online algorithm	$\left(\frac{\sqrt{W}}{B}+1-\frac{1}{B}\right)$	$\frac{V+B-1}{W+B-1}$	
Lower and upper bounds for specific algorithms			
	Lower bound	Lower	Upper
$\mathrm{PQ}_{-w,v}$, $\mathrm{PQ}^{\beta}_{-w,v}$	V	V	V
$\mathrm{PQ}_{v,-w}$, $\mathrm{PQ}^{\beta}_{v,-w}$	$\frac{W(V-1)}{V}-o(1)$	$\frac{W}{V}+o(1)$	$1+\frac{W+2}{V}$
$\mathrm{PQ}_{v/w}$, $W \geq V$	V	V	
$\mathrm{PQ}^{\beta}_{v/w}$, $\beta W \geq V$	V	V	
$\mathrm{PQ}_{v/w}$, $W < V$	W	$\frac{W}{V}+o(1)$	$2+\frac{2}{V}$

TABLE I
RESULTS SUMMARY: LOWER AND UPPER BOUNDS.

The results of our work are summarized in Table I; note that all algorithms in the table employ push-out (albeit with different heuristics for it). In the main result of this work, we have shown a $(1 + (W + 2)/V)$ upper bound for the buffer management policy $\mathrm{PQ}_{v,-w}$ that orders packets first by value and then by required processing. For $W < V$, this also becomes a constant upper bound on the competitive ratio of $\mathrm{PQ}_{v/w}$ which orders packets by unit processing (ratio of value to processing). This result has been somewhat counterintuitive since the intuition would be that the $\mathrm{PQ}_{v/w}$ policy that optimizes for value per timeslot would be best, but in the general two-valued case it has a non-competitive lower bound.

In addition, we have shown a number of general lower bounds, for the cases of any deterministic online algorithm with FIFO processing and transmission order, for any priority queue, and even for any deterministic online algorithm at all; while these lower bounds are relatively weak, they are non-constant and show that it is impossible to achieve constant upper bounds in these cases without additional assumptions on the relations between parameters such as B, V, and W.

For the two-valued case, we have shown tightly matching lower and upper bounds on the competitive ratio (they differ by $1+o(1)$). It still remains an interesting open problem to prove upper bounds for the general case of arbitrary values; another interesting problem would be to prove upper bounds for β-push-out policies. However, the really crucial question here is whether there exists a processing policy with better than linear competitive ratio for the general case of two characteristics: our general lower bounds are not constant but they are far from linear too. We suggest this problem for further study.

Acknowledgements: We thank both IEEE INFOCOM 2015 and *IEEE/ACM Transactions on Networking* reviewers for their insightful comments. This work was supported by the Russian Science Foundation grant 17-11-01276 "Networking and distributed systems and algorithms and related fundamental problems".

REFERENCES

[1] William Aiello, Alexander Kesselman, and Yishay Mansour. Competitive buffer management for shared-memory switches. *ACM Transactions on Algorithms*, 5(1), 2008.

[2] William Aiello, Yishay Mansour, S. Rajagopolan, and Adi Rosén. Competitive queue policies for differentiated services. In *INFOCOM*, pages 431–440, 2000.

[3] Nir Andelman. Randomized queue management for diffserv. In *SPAA*, pages 1–10, 2005.

[4] Nir Andelman and Yishay Mansour. Competitive management of non-preemptive queues with multiple values. In *DISC*, pages 166–180, 2003.

[5] Nir Andelman, Yishay Mansour, and An Zhu. Competitive queueing policies for QoS switches. In *SODA*, pages 761–770, 2003.

[6] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[7] Matthias Englert and Matthias Westermann. Considering suppressed packets improves buffer management in qos switches. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 209–218, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[8] Matthias Englert and Matthias Westermann. Lower and upper bounds on FIFO buffer management in QoS switches. *Algorithmica*, 53(4):523–548, 2009.

[9] Patrick Th. Eugster, Kirill Kogan, Sergey I. Nikolenko, and Alexander Sirotkin. Shared memory buffer management for heterogeneous packet processing. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 471–480, 2014.

[10] CAIDA The Cooperative Association for Internet Data Analysis. [Online] http://www.caida.org/.

[11] Michael Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.

[12] Ellen L. Hahne, Alexander Kesselman, and Yishay Mansour. Competitive buffer management for shared-memory switches. In *SPAA*, pages 53–58, 2001.

[13] B. Hajek. On the competitiveness of on-line scheduling of unit-length packets with hard deadlines in slotted time. In *In Proceedings of the 2001 Conference on Information Sciences and Systems*, 2001.

[14] Isaac Keslassy, Kirill Kogan, Gabriel Scalosub, and Michael Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[15] Alexander Kesselman, Kirill Kogan, and Michael Segal. Improved competitive performance bounds for CIOQ switches. In *ESA*, pages 577–588, 2008.

[16] Alexander Kesselman, Kirill Kogan, and Michael Segal. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.

[17] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[18] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in qos switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[19] Alexander Kesselman and Yishay Mansour. Loss-bounded analysis for differentiated services. *J. Algorithms*, 46(1):79–95, 2003.

[20] Alexander Kesselman and Yishay Mansour. Harmonic buffer management policy for shared memory switches. *Theor. Comput. Sci.*, 324(2-3):161–182, 2004.

[21] Alexander Kesselman, Yishay Mansour, and Rob van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43(1-2):63–80, 2005.

[22] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, Gabriel Scalosub, and Michael Segal. Balancing work and size with bounded buffers. In *Sixth International Conference on Communication Systems and Networks, COMSNETS 2014, Bangalore, India, January 6-10, 2014*, pages 1–8, 2014.

[23] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander Sirotkin. Multi-queued network processors for packets with heterogeneous processing requirements. In *Proceedings of the 5th International Conference on Communication Systems and Networks (COMSNETS 2013)*, pages 1–10, 2013.

[24] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander V. Sirotkin. A taxonomy of semi-FIFO policies. In *Proceedings of the 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012)*, pages 295–304, 2012.

[25] Kirill Kogan, Alejandro López-Ortiz, Sergey I. Nikolenko, and Alexander V. Sirotkin. Online scheduling FIFO policies with admission and push-out. *Theory Comput. Syst.*, 58(2):322–344, 2016.

[26] Fei Li. A near-optimal memoryless online algorithm for fifo buffering two packet classes. *Theoretical Computer Science*, 497:164 – 172, 2013.

[27] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A. Maltz. zupdate: updating data center networks with zero loss. In *SIGCOMM*, pages 411–422, 2013.

[28] Zvi Lotker and Boaz Patt-Shamir. Nearly optimal FIFO buffer management for two packet classes. *Comp. Netw.*, 42(4):481–492, 2003.

[29] Yishay Mansour, Boaz Patt-Shamir, and Ofer Lapid. Optimal smoothing schedules for real-time streams. *Distrib. Comp.*, 17(1):77–89, 2004.

[30] Sergey I. Nikolenko. Code for simulation experiments. http://github.com/snikolenko/sim-twocharacteristics.

[31] Sergey I. Nikolenko and Kirill Kogan. Single and multiple buffer processing. In *Encyclopedia of Algorithms*, pages 1988–1994. Springer, 2016.

[32] George Porter, Richard D. Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *SIGCOMM*, pages 447–458, 2013.

[33] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.

[34] Nourhan Sakr and Cliff Stein. An empirical study of online packet scheduling algorithms. In *Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685*, SEA 2016, pages 278–293, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[35] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[36] Tilman Wolf, Prashanth Pappu, and Mark A. Franklin. Predictive scheduling of network processors. *Comp. Netw.*, 41(5):601–621, 2003.

[37] An Zhu. Analysis of queueing policies in QoS switches. *J. Algorithms*, 53(2):137–168, 2004.

Pavel Chuprikov Pavel Chuprikov is a research assistant at the IMDEA Networks Institute, Spain and a Ph.D. student at the Steklov Institute of Mathematics at St. Petersburg, Russia. Previously, he worked as a software developer at JetBrains Research. Pavel received his M.Sc from St. Petersburg Academic University of Russian Academy of Sciences. His research interests include software defined networking, online algorithm design, and dependent types.

Alex Davydow Alex Davydow is a researcher at the Steklov Institute of Mathematics at St. Petersburg. He obtained his M.Sc. from the St. Petersburg Academic University and graduated from its Ph.D. studies. His research interests includes networking algorithms and systems, tropical algebraic geometry and its application to scheduling algorithms.

Sergey Nikolenko Sergey Nikolenko is a researcher at the Steklov Institute of Mathematics at St. Petersburg (PDMI RAS). He received his M.Sc. summa cum laude from St. Petersburg State University (2005); Ph.D., from PDMI RAS (2009). His research interests include networking algorithms and systems, machine learning and probabilistic inference, bioinformatics, and theoretical CS, with projects funded by major companies and research funds such as RSF, CRDF, INTAS, Mail.Ru, RFBR, RAS, and others.

Kirill Kogan Kirill Kogan is a Research Assistant Professor at IMDEA Networks Institute. He received his PhD from Ben-Gurion University (Israel) at 2012. He was a Technical Leader at Cisco Systems, where he worked in 2000-2012. He was a Postdoctoral Fellow at University of Waterloo and Purdue University during 2012-2014. His current research interests are in design, analysis, and implementation of networked systems, broadly defined.

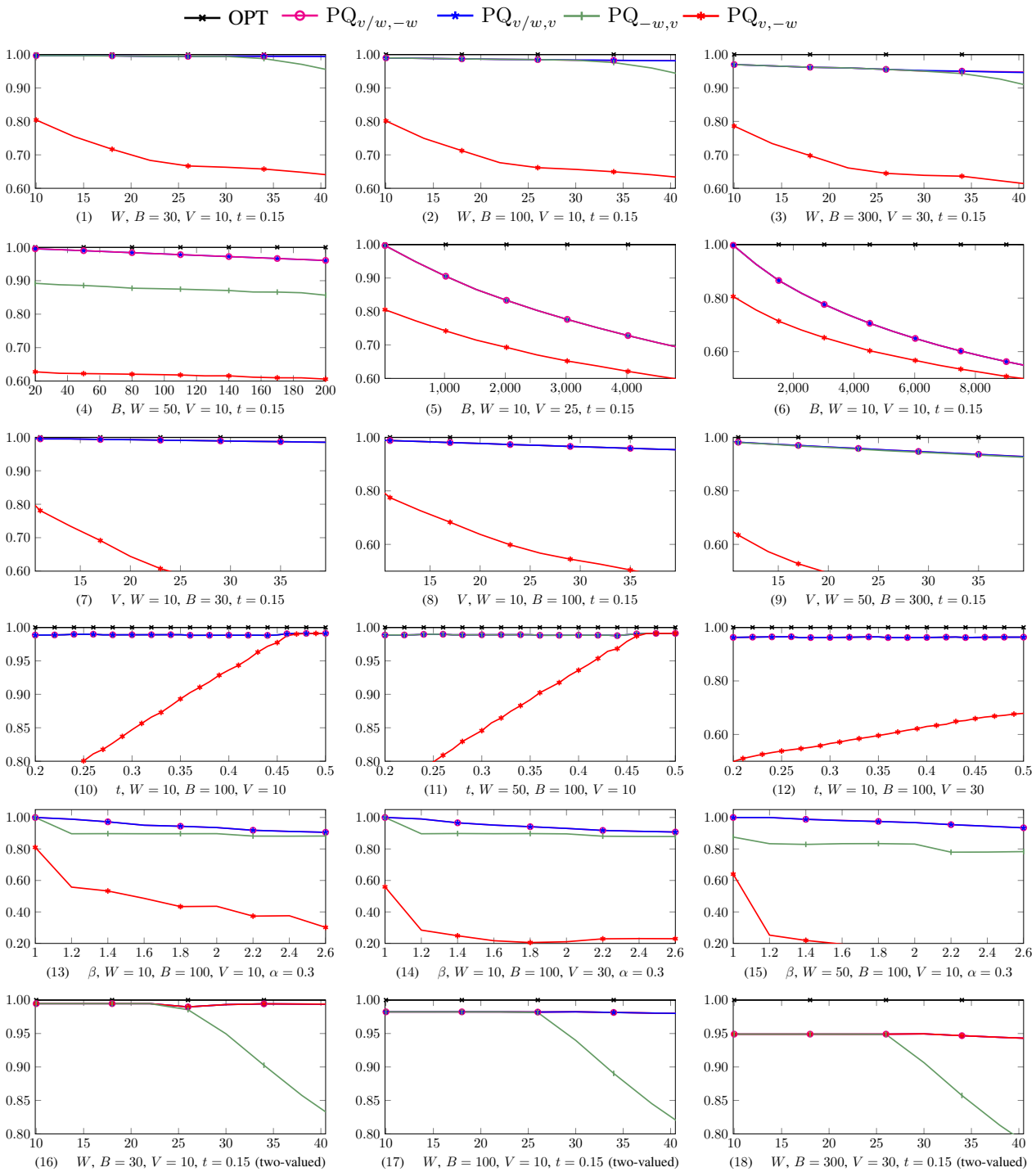


Fig. 5. Simulation results: share of successfully transmitted packets in the required processing model as a function of (1-3) maximal required processing W, (4-6) buffer size B, (7-9) maximal value V, (10-12) timeslot size t used for the CAIDA traces, (13-15) β parameter for β-push-out policies, (14-16) maximal required processing W in the two-valued case. Specific simulation parameters are shown in graph captions.

B Paper “General ternary bit strings on commodity longest-prefix-match infrastructure”

Authors. Pavel Chuprikov, Kirill Kogan, and Sergey Nikolenko

Abstract. Ternary Content-Addressable Memory (TCAM) is a powerful tool to represent network services with line-rate lookup time. There are various software-based approaches to represent multi-field packet classifiers. Unfortunately, all of them either require exponential memory or apply additional constraints on field representations (e.g, prefixes or exact values) to have line-rate lookup time. In this work, we propose alternatives to tcam and introduce a novel approach to represent packet classifiers based on ternary bit strings (without constraining field representation) on commodity longest-prefix-match (LPM) infrastructures. These representations are built on a novel property, prefix reorderability, that defines how to transform an ordered set of ternary bit strings to prefixes with lpm priorities in linear memory. Our results are supported by evaluations on large-scale packet classifiers with real parameters from ClassBench; moreover, we have developed a prototype in P4 to support these types of transformations.

General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures

Pavel Chuprikov*†, Kirill Kogan†, Sergey Nikolenko*

*Steklov Institute of Mathematics at St. Petersburg †IMDEA Networks Institute, Madrid

***Abstract*—Ternary Content-Addressable Memory (TCAM) is a powerful tool to represent network services with line-rate lookup time. There are various software-based approaches to represent multi-field packet classifiers. Unfortunately, all of them either require exponential memory or apply additional constraints on field representations (e.g, prefixes or exact values) to have line-rate lookup time. In this work, we propose alternatives to TCAM and introduce a novel approach to represent packet classifiers based on ternary bit strings (without constraining field representation) on commodity longest-prefix-match (LPM) infrastructures. These representations are built on a novel property, *prefix reorderability*, that defines how to transform an ordered set of ternary bit strings to prefixes with LPM priorities in linear memory. Our results are supported by evaluations on large-scale packet classifiers with real parameters from ClassBench; moreover, we have developed a prototype in P4 to support these types of transformations.**

I. Introduction

Packet classification is a core functionality for representing packet processing programs on the data plane. There are two major program categories: traffic forwarding between certain points in a communication network and service policies that guarantee desired traffic properties or track network behavior during forwarding (e.g., quality-of-service, access-control, firewall). Both can be captured as tuple matching with action sets, but they have distinct behavior and may rely on different invariants; e.g., forwarding tables can be represented by prefixes with priorities based on longest-prefix-match (LPM) while policies can consider general multi-field classifiers; forwarding tables may change frequently, while policies representing economic models or specific traffic signatures are designed mostly *a priori*. In this work, we concentrate on policies in the second category based on multi-field packet classifiers.

It is easier and more efficient to represent prefixes with LPM priorities than to use multi-field packet classifiers in software-based approaches [1]. 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 [2] , which makes them either too slow or too memory-intensive even with few prefix-fields.

Software-based approaches become even worse if classification rules are represented as general ternary bit strings, which is extremely useful in many applications [3], [4], [5]. Ternary content-addressable memory (TCAM) was introduced to overcome performance limitations of software-based solutions to represent multi-field packet classifiers and add a new level of expressiveness [6]. Unfortunately, TCAMs are expensive and power hungry [7], so TCAMs of a sufficient size are the *de facto* standard for classifier implementations only in high-end network elements [8], [9]. Most network elements efficiently implement prefix classifiers with LPM priorities at line-rate.

In this work, we explore alternatives to TCAMs and other software-based approaches and show how to represent multi-field packet classifiers on commodity LPM infrastructures (transparently to them) with line-rate performance. Various approaches to represent multi-field packet classifiers on LPM infrastructures exist, but all of them impose additional constraints on how fields are represented (e.g., prefixes or exact values) and most do not achieve desired worst-case guaranteed lookup time [10], [2], [11], [12]. Unlike prior art, we do not apply additional constraints on field representations and assume that classifier rules are ternary bit strings with general priorities as in TCAMs. We do not propose a specific classifier implementation but rather define an *abstraction layer* that chooses a subset of bit indices to be used in the lookup process. A classifier based on these bit indices can be transparently represented by other schemes, both in hardware and software.

The paper is organized as follows. Section II introduces the model; Section III, a novel structural property, *prefix reorderability*, with an optimal algorithm that transforms a given classifier into a prefix LPM classifier without extra memory (if possible). In Sections IV and V we show how to represent non-prefix-reorderable classifiers on existing LPM infrastructure without and with extra memory. Since classification width supported by LPM infrastructures is usually limited to 32 or 128 bits, in Section VI we show how to represent much wider classifiers on LPM infrastructures and study a composition of prefix reorderability with another structural property, *rule disjointness* (order independence [14]). Section VII discusses dynamic updates. In Section VIII, we evaluate our approach on ClassBench classifiers with real parameters [19]. Section IX outlines implementation details on top of the P4 domain-specific language [20]; we have released the code under an open source license. Section X discusses related prior art, and Section XI concludes the paper.

II. Model Description

In this section, we provide formal definitions for further exposition, starting with the basic notions of a packet header and classifier. A packet *header* $H = (h_1, \ldots, h_w)$ is a

978-1-5090-6501-1/17/$31.00

sequence of bits: each bit $h_i \in H$ has a value of either zero or one, $h_i \in \{0,1\}$, $1 \le i \le w$. For example, (1 0 0 0) is a 4-bit header. A *classifier* $\mathcal{K} = \{R_1, \ldots, R_N\}_\prec$ is an ordered (by $\prec$) set of rules, where each *rule* $R_i = (F_i, A_i)$ consists of a *filter* F_i and a pointer to the corresponding *action* A_i. A filter $F = (f_1, \ldots, f_w)$ is a sequence of, again, w values corresponding to bits in the headers, but this time the possible bit values are 0, 1, or $*$ ("don't care"). We illustrate classifiers by a table as in Example 1, where $R_i \prec R_{i+1}$ is implied.

Example 1: A classifier $\mathcal{K}$ on four bits (w = 4).

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	1	0	0	A_1
R_2	0	$*$	$*$	$*$	A_2
R_3	1	0	1	$*$	A_3
R_4	1	$*$	0	$*$	A_4

A classifier's main purpose is to find the action corresponding to the highest priority rule matching a given header. A header H *matches* a rule R if it matches R's filter, and it matches a filter F if for every bit of H the corresponding bit of F has either the same value or $*$. The set of rules has a non-cyclic priority ordering $\prec$; if a header matches both R and R' for $R \prec R'$, the action of rule R is applied. E.g., in Example 1 the header (0 1 0 0) matches both R_1 and R_2, but A_1 is applied. Filters F_1 and F_2 are *disjoint* if no single header matches both of them. Otherwise, F_1 and F_2 *intersect*, and rules have to be prioritized. Two rules *intersect* (are *disjoint*) if their filters intersect (are disjoint). In Example 1, R_1 and R_2 (as well as their filters) intersect (e.g., (0 1 0 0) matches both filters), while R_1, R_3, and R_4 are pairwise disjoint.

A classifier $\mathcal{K}$ is called a *prefix* classifier if in every filter $F \in \mathcal{K}$ all 0s and 1s precede all $*$ bits, i.e., they match prefixes of a header. A prefix classifier is called a *longest prefix match* (LPM) classifier if for every two intersecting rules R_1 and R_2, the one with the longer prefix in its filter takes precedence. LPM does not necessarily mean that a longer prefix is higher in the ordering: the constraint only concerns intersecting filters, so in Example 1 the first three rules comprise an LPM classifier.

Two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$ are *equivalent* if they choose the same action for every packet. Formally, for every header $H \in \{0,1\}^w$ H matches a rule in $\mathcal{K}_1$ iff it matches a rule in $\mathcal{K}_2$, and if H does match $R_1 = (F_1, A_1) \in \mathcal{K}_1$ and $R_2 = (F_2, A_2) \in \mathcal{K}_2$ then $A_1 = A_2$.

III. Prefix-reorderable classifiers

Our main objective is to find representations of classifiers with general priorities on commodity LPM infrastructures transparently to their implementations. To achieve this, we proceed in two steps: (1) reorder bit indices of ternary bit strings to transform them into prefixes (still with original priorities, not necessary LPM) and (2) convert the resulting prefix classifier (with general priorities) to an equivalent LPM classifier. Here we assume that the classification width w is at most w_{LPM} used in LPM infrastructures (e.g., 32 or 128 bits). We will return to the case $w > w_{\text{LPM}}$ in Section VI.

A. Ternary bit strings to prefixes

In this subsection, we identify exact conditions when a given classifier can be transformed by bit reordering into an equivalent prefix classifier. Moreover, we present an algorithm that constructs a required order.

Let $B = (b_1, b_2, \ldots, b_k)$, $k \le w$, be a sequence of distinct bit indices, $1 \le b_i \le w$, that represents a desired bit order. For a header $H = (h_1, h_2, \ldots, h_w)$, we denote by H^B the (sub)header $(h_{b_1}, h_{b_2}, \ldots, h_{b_k})$. E.g., for H = (0 1 0 0) and $B = (3, 2)$ H^B = (0 1). Similarly, for a filter F we have $F^B = (f_{b_1}, f_{b_2}, \ldots, f_{b_k})$, extending this notation to a rule $R = (F, A)$: $R^B = (F^B, A)$. Finally, for a classifier $\mathcal{K} = \{R_1, \ldots, R_N\}$ the *B-reordering* of $\mathcal{K}$, denoted $\mathcal{K}^B$, is obtained from $\mathcal{K}$ as follows: (i) replace each rule $R \in \mathcal{K}$ by R^B; (ii) for incoming packets, replace H by H^B prior to matching. E.g., in Example 1 for $B = (3, 1, 2)$ we have

$\mathcal{K}^B$	#3	#1	#2	*Action*
R_1^B	0	0	1	A_1
R_2^B	$*$	0	$*$	A_2
R_3^B	1	1	0	A_3
R_4^B	0	1	$*$	A_4

To match H = (0 1 0 0) in this classifier, H is first transformed to H^B = (0 0 1), which matches R_1^B.

Note that for H' = (0 1 0 1) in the above classifier the matching rule is also R_1^B, while in the original classifier it would match R_2 with a different action. Thus, $\mathcal{K}$ and $\mathcal{K}^B$ are not equivalent in general. Equivalence is obviously preserved, however, when no bits are omitted.

Observation 1: For every classifier $\mathcal{K}$ on w bits, if B is a permutation of $(1, \ldots, w)$ then $\mathcal{K}$ is equivalent to $\mathcal{K}^B$.

We call a classifier $\mathcal{K}$ on w bits *prefix-reorderable* if there exists a *permutation* B of $(1, \ldots, w)$ such that $\mathcal{K}^B$ is a prefix classifier (here $\mathcal{K}^B$ is always equivalent to $\mathcal{K}$).

Example 2: Classifier from Example 1 is not prefix but prefix-reorderable: for $B' = (1, 3, 2, 4)$ we have

$\mathcal{K}^{B'}$	#1	#3	#2	#4	*Action*
$R_1^{B'}$	0	0	1	0	A_1
$R_2^{B'}$	0	$*$	$*$	$*$	A_2
$R_3^{B'}$	1	1	0	$*$	A_3
$R_4^{B'}$	1	0	$*$	$*$	A_4

Unfortunately, not every classifier is prefix-reorderable.

Example 3: Consider the following classifier:

$\mathcal{K}$	#1	#2	*Action*
R_1	0	$*$	A_1
R_2	$*$	0	A_2

The key to prefix-reorderability criteria lies in the reason why Example 3 fails. Consider a filter $F = (f_1, \ldots, f_w)$; denote by $\text{exact}(F)$ the set of bit indices i such that $f_i \neq *$; we extend exact to rules as $\text{exact}((F, A)) = \text{exact}(F)$. In Example 3 we have $\text{exact}(R_1) = \{1\}$, and $\text{exact}(R_2) = \{2\}$, but in a prefix classifier $*$ must follow 0s and 1s in F^B.

Observation 2: For any filter F on w bits and any permutation B of $(1, \ldots, w)$, if F^B is a part of a *prefix* classifier, then indices from $\text{exact}(F)$ must precede in B all other indices.

Algorithm 1 `perm`($\mathcal{K}$)

1: $E'_1, \ldots, E'_{|\mathcal{K}|} \leftarrow$ `sort_by_size`($\text{exact}(\mathcal{K})$)
2: $B \leftarrow$ `sort`(E'_1)
3: **for** $i \in \{1, \ldots, |\mathcal{K}| - 1\}$ **do**
4: **if** $E'_i \not\subseteq E'_{i+1}$ **then**
5: **return** `NO`
6: $B \leftarrow B,$ `sort`$(E'_{i+1} \setminus E'_i)$
7: $B \leftarrow B,$ `sort`$(\{1, \ldots, w\} \setminus E'_{|\mathcal{K}|})$
8: **return** `YES`(`B`)

In Example 3, we cannot satisfy Observation 2 for both filters at once. This generalizes to any two filters with $\text{exact}(F_1) = E_1$ and $\text{exact}(F_2) = E_2$ such that there exist $x \in E_1 \setminus E_2$ and $y \in E_2 \setminus E_1$, i.e., $E_1 \not\subseteq E_2$ and $E_2 \not\subseteq E_1$. The next theorem states that to require either $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ is actually sufficient. The criterion is naturally formulated in terms of the structure of $\text{exact}(\mathcal{K}) = \{\text{exact}(R) : R \in \mathcal{K}\}$.

Theorem 1 (chain criterion): A classifier $\mathcal{K}$ is prefix-reorderable iff for every $E_1, E_2 \in \text{exact}(\mathcal{K})$ either $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ holds, i.e., $\text{exact}(\mathcal{K})$ can be reordered to form a "chain": $E_{i_1} \subseteq E_{i_2} \subseteq \ldots \subseteq E_{i_{|\mathcal{K}|}}$. The permutation of bit indices can be found in $O(|\mathcal{K}| \cdot w)$ time (if one exists).

Proof: ($\Rightarrow$) Necessity follows by the observation above: if we have $x \in E_i \setminus E_j$ and $y \in E_j \setminus E_i$ for some i, j, and a B-reordering of $\mathcal{K}$ is a prefix classifier, then both y must precede x in B and x must precede y, a contradiction.

($\Leftarrow$) Sufficiency. We need to present a permutation of bits that leads to prefix representations of all rules. We claim that Algorithm 1 constructs such a permutation or returns `NO` if the theorem's assumption does not hold. First, by assumption $\text{exact}(\mathcal{K})$ ordered by the size of $|E_i|$ is a chain, i.e., $E'_i \subseteq E'_{i+1}$ for all i, so the check on line 4 never succeeds. Next, since $E'_1 \subseteq \ldots \subseteq E'_{|\mathcal{K}|}$, after m iterations of the loop on line 3 B represents a permutation of E'_m. Finally, for every i in the resulting permutation, all bits from E'_i precede all bits from $\{1, \ldots, w\} \setminus E'_i$ since they get mapped later. Thus, B turns the classifier $\mathcal{K}$ into a prefix one. Algorithm 1 needs $O(|\mathcal{K}| \cdot w)$ operations to construct the set $\text{exact}(\mathcal{K})$; sorting can be done in time $O(w + |\mathcal{K}|)$ with radix sort, any set-theoretic operation (including sorting) requires $O(w)$ time, and the loop has $|\mathcal{K}| - 1$ iterations of time $O(w)$ each. ■

Note that in Example 3 the set $\text{exact}(\mathcal{K}) = \{\{1\}, \{2\}\}$ is not a chain. Algorithm `perm` exploits the criterion (Theorem 1) to construct a permutation of bit indices that results in a prefix-reorderable classifier, if one exists. To illustrate the behavior of `perm`, consider again the classifier $\mathcal{K}$ from Example 1. Here, if sorted by sizes, we have $\text{exact}(\mathcal{K}) = \{\{1\}, \{1, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\}\}$. Line 2 assigns $B = (1)$. On the first iteration, we have $E'_i = \{1\}$ and $E'_{i+1} = \{1, 3\}$, thus $B = (1, 3)$. On the second iteration, $E'_i = \{1, 3\}$, $E'_{i+1} = \{1, 2, 3\}$, so now $B = (1, 3, 2)$. Line 7 adds the last bit index 4, and the algorithm returns $B = (1, 3, 2, 4)$, leading to the same prefix classifier as in Example 2. Algorithm `perm` can now be used to find the equivalent prefix classifier if it exists; we let `general_to_prefix`($\mathcal{K}$) = `NO` if `perm`($\mathcal{K}$) = `NO` and `general_to_prefix`($\mathcal{K}$) = $\mathcal{K}^B$ if `perm`($\mathcal{K}$) = `YES`(B).

Algorithm 2 `prefix_to_lpm`($\mathcal{K}$)

1: **while** $\mathcal{K}$ is not LPM **do**
2: $R_1, R_2 \leftarrow$ rules violating LPM
3: **if** $|\text{exact}(R_1)| < |\text{exact}(R_2)|$ **then** ▷ R_1 has a shorter prefix
4: $\mathcal{K} \leftarrow \mathcal{K} \setminus R_2$
5: **else**
6: $\mathcal{K} \leftarrow \mathcal{K} \setminus R_1$
7: **return** $\mathcal{K}$

B. Prefix to LPM classifiers

Once a classifier is transformed to an equivalent prefix classifier $\mathcal{K}'$, the rule priorities of $\mathcal{K}'$ do not necessary conform to LPM priorities. In this part, we introduce a transformation of a prefix classifier (with general priorities) to an LPM classifier that does not add new rules and works in time $O(|\mathcal{K}| \cdot w)$.

Example 4: Consider a prefix classifier $\mathcal{K}_0$:

$\mathcal{K}_0$	#1	#2	#3	#4	*Action*
R_1	0	$*$	$*$	$*$	A_1
R_2	1	0	$*$	$*$	A_2
R_3	1	0	1	$*$	A_3

Observe that the last two rules in $\mathcal{K}_0$ do not conform to LPM since R_2 intersects with R_3, R_2 has a shorter prefix, but $R_2 \prec R_3$. In this case, the last rule is just redundant and can be removed since all packets matched by R_3 will always be matched by R_2 with higher priority. As a result, we get an equivalent prefix classifier that already has LPM priorities:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	0	$*$	$*$	$*$	A_1
R_2	1	0	$*$	$*$	A_2

It turns out that it suffices to remove redundant rules of this type to transform a prefix classifier into an LPM classifier, and Algorithm 2 does this without increasing the number of rules.

Theorem 2: `prefix_to_lpm` transforms a prefix classifier $\mathcal{K}$ into an equivalent LPM classifier $\mathcal{K}'$ with $|\mathcal{K}'| \leq |\mathcal{K}|$. Moreover, it can be implemented in time $O(w \cdot |\mathcal{K}|)$.

Proof: Since each iteration reduces the number of pairs of rules that violate LPM order, and there are finitely many such pairs, `prefix_to_lpm` terminates. To prove that it preserves equivalence, consider two rules: R_i with prefix x and R_{i+1} with prefix y; w.l.o.g. assume that $|x| < |y|$, so $y = uz$, where $|u| = |x|$. There are two cases: (1) $x = u$ and (2) $x \neq u$. In the former case, all packets matched by y would be already matched by x; thus, rule R_{i+1} is redundant and can be removed. In the latter case, the rules do not intersect and hence conform to LPM priorities.

A naive implementation of `prefix_to_lpm`, shown in Algorithm 2, works in $O(w \cdot |\mathcal{K}|^2)$ time. A faster approach can be based on bit tries: start with an empty bit trie T. Sort rules in non-decreasing order of $\text{exact}(R_i)$ in $O(|\mathcal{K}| + w)$ time using radix sort. Then, for every rule R_i in this order with 0-1 prefix p_i, check for every rule R_j such that its prefix p_j lies on the path from T's root to p_i that $R_i \prec R_j$; if the check does not fail, add p_i to T; if for some R_j $p_i = p_j$, leave in T the rule with higher priority. Figure 1 shows an example of adding the prefixes of $\mathcal{K}_0$ to this trie: the third rule's prefix (shown in gray) is not included since an earlier rule already has a shorter

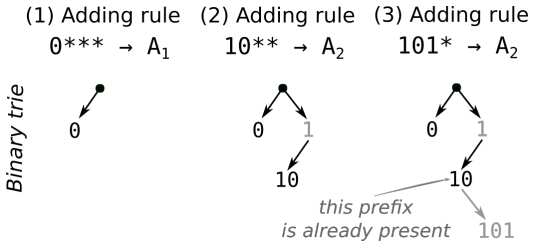


Fig. 1. Adding prefixes of $\mathcal{K}_0$ to a binary trie.

prefix. The resulting classifier $\mathcal{K}$ consists of all rules whose prefixes are left in T, together with the corresponding actions; the theorem follows since we always keep only top priority rules among intersecting ones. ■

In this section, we have shown how to transform prefix-reorderable classifiers to LPM classifiers. The proposed transformations, `perm` and `prefix_to_lpm`, are transparent to internal implementations of LPM infrastructures. Next we will see how to represent non-prefix-reorderable classifiers on commodity LPM infrastructures.

IV. Non-prefix-reorderable classifiers without extra memory

We introduce two approaches for dealing with non-prefix-reorderable classifiers. The first, which we consider in this section, requires prefix reorderability only on a subset of rules, exploiting the capability of a target architecture to perform multiple LPM lookups at line-rate and leading to multiple prefix-reorderable classifiers that are together equivalent to the original classifier.

A. Groupwise reorderability

In most cases, target architectures are able to perform multiple lookups to the LPM infrastructure at line-rate. Hence, we can partition rules into multiple prefix-reorderable classifiers and look up a header in each group, combining the outcomes to choose the highest priority rule as the final result. Since each rule appears only in one group, no extra memory is required.

Example 5: Consider the following (non-prefix-reorderable) classifier $\mathcal{K}$ that will be our running example in this section:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_3	$*$	1	0	0	A_3
R_4	0	0	$*$	$*$	A_4
R_5	$*$	0	1	$*$	A_5
R_6	$*$	1	0	$*$	A_6
R_7	$*$	0	$*$	$*$	A_7

$\mathcal{K}$ can be split into prefix-reorderable classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_4	0	0	$*$	$*$	A_4
R_7	$*$	0	$*$	$*$	A_7

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
R_3	$*$	1	0	0	A_3
R_5	$*$	0	1	$*$	A_5
R_6	$*$	1	0	$*$	A_6

Algorithm 3 $\texttt{mg}(H)$ for $\mathcal{K}$ and $\{\mathcal{K}_1, \ldots, \mathcal{K}_\beta\}$

1: **for** $i \in 1, \ldots \beta$ **do**
2: $\quad R_i^* \leftarrow \texttt{find\_match}(\mathcal{K}_i, H)$
3: $(F \Rightarrow A) \leftarrow \max_{1 \le i \le \beta} R_i^*$, w.r.t $\prec_{\mathcal{K}}$
4: **return** A

Algorithm 4 `min_group_partition`$(\mathcal{K})$

1: $\mathcal{E}_1, \ldots, \mathcal{E}_\beta \leftarrow \texttt{min\_chain\_partition}(\text{exact}(\mathcal{K}), \subseteq)$
2: $P \leftarrow \emptyset$
3: **for** $i \in \{1, 2, \ldots, \beta\}$ **do**
4: $\quad \mathcal{K}_i \leftarrow \{R \in \mathcal{K} : \text{exact}(R) \in \mathcal{E}_i\}$
5: $\quad P \leftarrow P \cup \{\mathcal{K}_i\}$
6: **return** P

If $\mathcal{K}_1$ and $\mathcal{K}_2$ both receive a header $H = (0\ 0\ 1\ 0)$, the former finds a matching rule R_2 and the latter finds R_5. The multigroup implementation compares priorities of matched rules and returns the one with higher priority, namely R_2.

Algorithm 7 defines the lookup procedure to a multigroup representation of a given classifier. Since all rules of the original classifier $\mathcal{K}$ participate in the lookup process, and the rule with highest priority is returned, the multigroup representation of a given classifier is equivalent to $\mathcal{K}$. Different objectives can be optimized in the construction of multigroup representations. We begin with a natural one that minimizes the number of prefix-reorderable groups.

B. Minimizing the number of groups

*Problem 1 (*MINGR*):* Find a partition of the rules of a given classifier $\mathcal{K}$ into a minimal number of disjoint prefix-reorderable groups.

Intuitively, the resulting number of groups should depend on the order in which rules are assigned to groups, and at first glance it looks like a hard problem. However, according to the chain criterion in Theorem 1, $\mathcal{K}'$ is a prefix-reorderable subset of $\mathcal{K}$ if and only if $\text{exact}(\mathcal{K}')$ is a chain. Thus, instead of partitioning $\mathcal{K}$ into prefix-reorderable groups we can partition $\text{exact}(\mathcal{K})$ into chains and then distribute rules to groups according to the chain partition. This transformation changes the problem in two important ways: first, the size of $\text{exact}(\mathcal{K})$ is usually much smaller than the number of rules in $\mathcal{K}$; second, most importantly, the minimal chain partition problem can be solved in polynomial time with the help of order theory [21].

The `min_group_partition` algorithm for MINGR, shown in Algorithm 4, exploits this idea. It calls the `min_chain_partition` function that returns a minimal size chain partition of a given ordered set based on Dilworth's decomposition theorem [22]. The implementation of `min_chain_partition` shown in Algorithm 5 is based on Fulkerson's proof of Dilworth's theorem [23].

Example 6: Fig. 2 illustrates Algorithm 4 with $\mathcal{K}$ from Example 5. Starting from $\text{exact}(\mathcal{K}) = \{\{2\}, \{1,2\}, \{2,3\}, \{1,2,3\}, \{2,3,4\}\}$, we order it by inclusion (Fig. 2a), construct the corresponding bipartite graph (Fig. 2b), find a maximal matching, in this case $\{(\{2\}, \{1,2\}), (\{1,2\}, \{1,2,3\}), (\{2,3\}, \{2,3,4\})\}$ (shown on Fig. 2b and c), and construct the chains by following this matching (Fig. 2c and d).

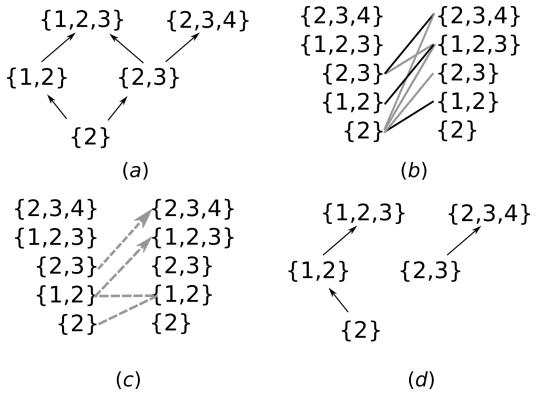


Fig. 2. Dilworth's decomposition theorem: (a) original ordered set; (b) the bipartite graph and its maximal matching; (c) chain decomposition in the bipartite graph; (d) chain decomposition in the original graph.

Algorithm 5 `min_chain_partition`$(X, \leq)$

```
1: G ← bipartite graph, s.t. V(G) = E(G) = ∅
2: for x ∈ X do
3:     G.add_left_vertex(x_L)
4:     G.add_right_vertex(x_R)
5: for x, x' ∈ X do
6:     if x < x' then
7:         G.add_edge(x_L, x'_R)
8: M ← max_matching(G)
9:
10: H ← graph, s.t. V(H) = X and E(H) = ∅
11: for (x_L, x'_R) ∈ M do
12:     H.add_edge(x, x')
13: return connected_components(H)
```

Theorem 3: The algorithm `min_chain_partition` finds an optimal solution for the MINGR problem in time $O\left(|\operatorname{exact}(\mathcal{K})|^{5/2} + |\mathcal{K}|^2 w\right)$.

Proof: For any prefix-reorderable partition of rules, by Theorem 1 the set of $\operatorname{exact}(\mathcal{K}_i)$ in each group $\mathcal{K}_i$ constitutes a chain, leading to a chain partition. Vice versa, if we have found a chain partition it suffices to group rules with respect to this partition to get a prefix-reorderable partition of the same size. Execution time of `min_pmgr` is dominated by the call to `min_chain_partition`, whose execution time is dominated by `max_matching`, so the Hopcroft–Karp algorithm [24] yields running time $O\left(|K|^{5/2}\right)$. It takes time $O\left(|\mathcal{K}|^2 w\right)$ to construct the ordered set $(\operatorname{exact}(\mathcal{K}), \subseteq)$: we have to compute $\subseteq$ for every pair of $|\mathcal{K}|$ elements in this set. ■

C. Mixed representations

The number of parallel and serial lookups to the LPM infrastructure at line rate is a property of the underlying target architecture. In the worst case, one can construct artificial instances of classifiers whose optimal representations require an exponential (in width w) number of disjoint groups.

Theorem 4: There exists a classifier $\mathcal{K}$ with $|\mathcal{K}| = O\left(\frac{1}{\sqrt{w}} 2^{\frac{w}{2}}\right)$ such that an optimal solution of the MINGR problem requires exactly $|\mathcal{K}|$ groups.

Proof: The idea of this worst-case example is to construct many filters with different sets of $*$ fields that are not subsets of each other (and hence cannot be reordered). Consider the set of ternary bit strings of length $\frac{w}{2}$ that contain exactly $\frac{w}{4}$ 1s and $\frac{w}{4}$ $*$. There are $O\left(2^{\frac{w}{2}}/\sqrt{w}\right)$ such strings. If we concatenate each of them with a different string from $\{0,1\}^{w/2}$, we get $O\left(2^{\frac{w}{2}}/\sqrt{w}\right)$ non-intersecting filters. Moreover, no two of these filters can belong to the same prefix-reorderable classifier, because their $\operatorname{exact}(F_i)$ sets are unequal due to different "don't care" positions in the first halfs of the filters. Assigning to each filter a different action, we get a classifier without redundant rules that cannot be prefix-reordered. ■

E.g., for the following classifier $\mathcal{K}$ with $w = 8$ we get six different combinations of two $*$ and two 1 bits in the left half:

$\mathcal{K}$	#1	#2	#3	#4	#5	#6	#7	#8	*Action*
R_1	$*$	$*$	1	1	0	0	0	0	A_1
R_2	$*$	1	$*$	1	0	0	0	1	A_2
R_3	$*$	1	1	$*$	0	0	1	0	A_3
R_4	1	$*$	$*$	1	0	0	1	1	A_4
R_5	1	$*$	1	$*$	0	1	0	0	A_5
R_6	1	1	$*$	$*$	0	1	0	1	A_6

While instances such as in Theorem 4 are rare, it often happens in practice that a small number of "bad" rules form a hard example and result in a large number of representing groups. To allow our methods to cover non-prefix-reorderable instances, in this section we introduce "mixed" representations that implement a part of the original classifier in a traditional way (e.g., in TCAM). The assumption is, again, that we can look up both multigroup and traditional parts of a given classifier and return the matched rule with highest priority, so the mixed representation is again equivalent to $\mathcal{K}$. We assume that traditional representations are more expensive than multigroup ones. Therefore, our objective is to maximize the number of rules from a given classifier used in the multigroup part, while still limiting the total number of groups.

*Problem 2 (*MAXCOV*):* Given a classifier $\mathcal{K}$ and a constant $\beta > 0$, assign the largest possible subset of $\mathcal{K}$'s rules into at most β prefix-reorderable disjoint groups.

Note that both MINGR and MAXCOV problems do not change the rules themselves and do not require additional memory. The mixed setting allows us to exclude "bad" rules from consideration when constructing a chain partition. To exclude an element E from $\operatorname{exact}(\mathcal{K})$ we must move every rule R with $\operatorname{exact}(R) = E$ to a traditional representation. Denoting a set of all such rules as $\mathcal{K}[E]$, we can associate with every removal of E a cost $|\mathcal{K}[E]|$; the goal now is to minimize the total cost. Algorithm 6 (`max_coverage_partition`) essentially follows the same logic as `min_group_partition`, but `max_coverage_partition` instead of a maximal matching uses a minimal weight matching of a fixed cardinality and introduces additional weighted edges on line 5. The cardinality $|M|$ of the matching determines the number of subgroups ($\beta = |\operatorname{exact}(\mathcal{K})| - |M|$). This lets MINGR minimize the number of groups but MAXCOV simply bounds it while another objective is being optimized.

Example 7: Fig. 3 shows `max_coverage_partition` on a classifier $\mathcal{K}$ from Section IV-A with $\beta = 1$; numbers of rules with each $\operatorname{exact}(R)$ are shown in top right corners (Fig. 3a). In the bipartite graph on Fig. 3b, `max_coverage_partition`

Algorithm 6 `max_coverage`$(\mathcal{K}, \beta)$

$G \leftarrow$ weighted bipartite graph, $V(G) = \emptyset$, $E(G) = \emptyset$
for $E \in \text{exact}(\mathcal{K})$ **do**
  G.`add_left_vertex`(E_L)
  G.`add_right_vertex`(E_R)
  G.`add_edge`$(E_L, E_R, \texttt{weight} = |\mathcal{K}[E]|)$
for $E, E' \in \text{exact}(\mathcal{K})$ **do**
  if $E \subset E'$ **then**
    G.`add_edge`$(E_L, E'_R, \texttt{weight} = 0)$
$M \leftarrow$ `min_weight_matching`$(G, |\text{exact}(\mathcal{K})| - \beta)$
$\mathcal{K}' \leftarrow \{R \in \mathcal{K} : (\text{exact}(R)_L, \text{exact}(R)_R) \in M\}$
return $\mathcal{K}'$

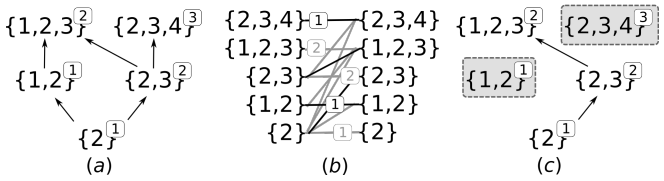


Fig. 3. `max_coverage_partition` example with $\beta = 1$: (a) ordered set $\text{exact}(\mathcal{K})$; (b) minimal weighted matching with additional edges; (c) resulting chains; shaded regions show excluded rule subsets.

adds weighted edges connecting each E with itself in the other part, with weight equal to the number of edges. Weighted edges in the minimal partition correspond to removed rule subsets, and the rest form the resulting chain (Fig. 3c).

Theorem 5: Algorithm `max_coverage_partition` produces an optimal solution for the MAXCOV problem in time $O(|\text{exact}(\mathcal{K})|^2|\mathcal{K}| + |\mathcal{K}|^2 w)$.

Proof: Correctness of `max_coverage_partition` follows from a one-to-one correspondence between solutions $\mathcal{E}$ for the MAXCOV problem and weighted matchings $M_\mathcal{E}$ of size $|\text{exact}(\mathcal{K})| - \beta$ in the bipartite graph with additional edges, and, moreover, $\sum_{E \in \text{exact}(\mathcal{K}) \setminus \mathcal{E}} |\mathcal{K}[E]| = w(M_\mathcal{E})$. This fact is clear by construction:

- every matching of size $|\text{exact}(\mathcal{K})| - \beta$ corresponds to β chains since in every chain, the number of vertices equals the number of edges plus one (in particular, a chain of size one corresponds to an unmatched vertex);
- by minimizing the weight of the matching, we minimize the total number of excluded rules.

Time complexity is similar to `min_group_partition`, but now `max_matching` is replaced with `min_weight_matching`; applying the augmenting paths approach, we get $|\text{exact}(\mathcal{K})|^2|\mathcal{K}|$ complexity since the maximal flow is bounded by $|\mathcal{K}|$, and there are $|\text{exact}(\mathcal{K})|$ vertices. ∎

Note also that an optimal algorithm for MAXCOV can be used to find an optimal solution for the MINGR problem, with an extra factor of $\log(|\mathcal{K}|)$ for a binary search on β.

V. Non-prefix-reorderable classifiers with extra memory

So far, all proposed representations did not change the number of rules or rules themselves. However, in many cases we can improve prefix reorderability by expanding some $*$ bits into all possible combinations of 0s and 1s.

Example 8: Consider again the non-prefix-reorderable classifier from Example 3. We can cover only one rule with a single prefix-reorderable group, but by expanding the $*$ bit in R_2 we get the following prefix-reorderable classifier:

$\mathcal{K}$	#1	#2	*Action*
R_1	0	$*$	A_1
R_2	1	1	A_2
R_3	0	1	A_3

Now R_3 is covered by R_1 and can be removed, so the end result is a single group covering all rules.

This example shows that sometimes it is worthwhile to trade a modest increase in memory footprint to cover more rules by prefix-reorderable groups. Naturally, to avoid exponential memory blowup one should control the expansion process, so we incorporate into the MAXCOV problem a constraint m on the number of expanded bits per rule. In the following definition, $\mathcal{K}^*$ denotes an expanded version of $\mathcal{K}'$.

*Problem 3 (*MAXCOV-m*):* Given a classifier $\mathcal{K}$, a constant $\beta > 0$, and a maximal number of rules M, find the largest subset $\mathcal{K}' \subseteq \mathcal{K}$ and a multigroup classifier $\mathcal{K}^*$ equivalent to $\mathcal{K}'$ with $|\mathcal{K}^*| \leq M$ such that $\mathcal{K}^*$ can be split into at most β prefix-reorderable groups.

In what follows we present a heuristic for MAXCOV-m. The general idea is to start with a solution for the MAXCOV problem (for some β) and then incorporate some of the extra rules into the prefix-reorderable groups by applying *bit expansion*. Specifically, we take a rule R (e.g., $((0\ *\ 1\ *), A)$) and a set of bit indices (e.g., $\{2\}$), and replicate R, replacing $*$ bits at given indices with all possible combinations of 0s and 1s: $R' = ((0\ 0\ 1\ *), A)$, $R'' = ((0\ 1\ 1\ *), A)$. We denote this procedure by `expand`$(\mathcal{K}, B)$, which applies bit expansion to all bits with indices from B in every rule in $\mathcal{K}$.

To understand how bit expansion can be useful for solving the MAXCOV-m problem, consider $\beta = 1$. Assume that `max_coverage_partition`$(\mathcal{K}, 1)$ produced a maximum prefix-reorderable group $\mathcal{K}' \subseteq \mathcal{K}$, so by Theorem 1 $\text{exact}(\mathcal{K}') = E_1 \subseteq E_2 \ldots \subseteq E_k$. We denote the "extra" rules represented in a traditional way as $\mathcal{K}_t = \mathcal{K} \setminus \mathcal{K}'$. Prefix-reorderability is determined solely by the structure of $\text{exact}(\mathcal{K}')$, so we can limit our consideration to $\mathcal{K}_t[E] = \{R \in \mathcal{K}_t : \text{exact}(R) = E\}$. To add some $\mathcal{K}_t[E]$ to the chain $\text{exact}(\mathcal{K}')$, we need to expand some bits in $\mathcal{K}_t[E]$ and/or possibly in $\mathcal{K}[E_i]$ so that the result is still prefix reorderable. For example, if $E \not\subseteq E_k$ (the maximal element of $\text{exact}(\mathcal{K}')$), we merge $\mathcal{K}_t[E]$ and $\mathcal{K}'[E_k]$ into $\mathcal{K}'[E_k \cup E]$ by replacing them with `expand`$(\mathcal{K}_t[E], E_k \cup E)$ and `expand`$(\mathcal{K}'[E_k], E_k \cup E)$.

The algorithm `expand_and_fit` (Algorithm 7) greedily selects $\mathcal{K}_t[E]$ to merge into a $\mathcal{K}'$ so that $|\mathcal{K}_t[E]|$ is maximized, since the less rules are left to a traditional representation (the subset $\mathcal{K}_t$) the better. To guarantee that bit expansion will not increase memory above M, we limit the number of bits that may be expanded in each rule by $\delta = \log_2(M/|\mathcal{K}|)$; the algorithm keeps track of already expanded bits from previous iterations in $\mathcal{K}'$ using #ex. The complexity of `expand_and_fit` without calls to `expand` and `min_chain_partition` is $O(w \cdot |\text{exact}(\mathcal{K}_t)| \cdot \log|\text{exact}(\mathcal{K}')|)$. An actual implementation of

Algorithm 7 `expand_and_fit`$(\mathcal{K}', \mathcal{K}_t, \delta)$

```
1: E ← sort exact(K_t) by 1/|K_t[E]|
2: for E ∈ E do
3:     (E_1 ⊆ ... ⊆ E_m) ← min_chain_partition(K')
4:     i* ← max{i : E ⊈ E_i}
5:     E_∪ ← E ∪ E_{i*}
6:     if |E_∪ \ E| > δ or #ex(E_{i*}) + |E_∪ \ E_{i*}| > δ then
7:         continue
8:     #ex(E_∪) ← max(|E_∪ \ E|, #ex(E_{i*}) + |E_∪ \ E_{i*}|, #ex(E_∪))
9:     K' ← (K'\K'[E_{i*}])∪expand(K'[E_{i*}], E_∪)∪expand(K_t[E], E_∪)
10:    K_t ← K_t \ K_t[E]
11: return (K', K_t)
```

the algorithm can postpone `expand` until the end by operating on pairs $(E, |\mathcal{K}'[E]|)$ instead of actual rule sets; moreover, the chain partition is usually known from from the preceding call to `max_coverage_partition`.

Example 9: Consider the classifier $\mathcal{K}$ from Example 5 and let $\beta = 1$ and $\delta = 1$. The maximum prefix-reorderable classifier $\mathcal{K}'$ is $\mathcal{K}_1$ and $\mathcal{K}_t$ is $\mathcal{K}_2$. We have $\text{exact}(\mathcal{K}') = (E_1 = \{2\}, E_2 = \{1,2\}, E_3 = \{1,2,3\})$; and $\text{exact}(\mathcal{K}_t) = (\{2,3\}, \{2,3,4\})$. First, `expand_and_fit` considers $E = \{2,3\}$; the largest E_i not containing E is $E_2 = \{1,2\}$, so $i^* = 2$. At this point, the check at line 6 succeeds, and $\mathcal{K}'$ gets augmented with `expand`$(\{R_5, R_6\}, \{1,2,3\})$. At line 6, $\text{exact}(\mathcal{K}')$ does not change but $\#ex(\{1,2,3\})$ is set to 1. Next, algorithm considers $E = \{2,3,4\}$ and selects $i^* = 3$. Here $|E_3 \setminus E_\cup| = 1$, but now $\#\text{ex}(E_3) = 1$, and the check at line 6 fails. The end result is the following:

$\mathcal{K}'$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_4	0	0	$*$	$*$	A_4
R_{50}	0	0	1	$*$	A_5
R_{51}	1	0	1	$*$	A_5
R_{60}	0	1	0	$*$	A_6
R_{61}	1	1	0	$*$	A_6
R_7	$*$	0	$*$	$*$	A_7

Note the redundancy in R_{50} that resulted from bit expansion.

VI. How to cut down classification width

We have already introduced several ways to represent ternary bit strings with general priorities on commodity LPM infrastructures. So far we assumed that classification width w of the classifiers is at most the classification width w_{LPM} of an implementing LPM infrastructure. In reality, w can be larger than supported by commodity LPM infrastructures (32 or 128 bits), so it is important to reduce classification width. The work [14] already introduced a structural property of classifiers called *rule disjointness* to tackle exactly this problem. A classifier is *rule-disjoint* iff all its rules are pairwise disjoint. If a given classifier $\mathcal{K}$ is still rule-disjoint on a subset of bit indices $B \subset \{1, \ldots, w\}$, the underlying lookup can be based only on these B bits, and then only a false-positive check is required on the remaining $\{1, \ldots, w\} \setminus B$ bits. Due to disjointness, only a single rule can be matched, so the false-positive check is not a lookup but just a bitwise comparison of bits from $\{1, \ldots, w\} \setminus B$ in the header and rule matched in the $\mathcal{K}^B$ classifier.

Example 10: Consider the following rule-disjoint classifier:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	1	$*$	0	A_1
R_2	1	$*$	0	$*$	A_2

Note that only a single bit is enough to keep an equivalent classifier $\mathcal{K}^B$ rule-disjoint, i.e., for $B = (1)$ we have

$\mathcal{K}^B$	#1	*Action*
R'_1	0	`false_positive_check`(R_1)
R'_2	1	`false_positive_check`(R_2)

Similarly to prefix reorderability, requiring rule disjointness on the entire classifier can be too restrictive, so multigroup representations are natural for both characteristics. In particular we want to have the width-bounded versions of the problems that we have defined earlier (e.g., MINGR or MAXCOV). We will not present here definitions for all of them (since they have similar changes) and limit ourselves to the following:

*Problem 4 (*MAXCOV-w*):* Given a classifier $\mathcal{K}$ and two positive numbers w_{LPM} and β, find a maximal subset of $\mathcal{K}$'s rules that can be assigned to at most β groups, where each group is prefix-reorderable and rule-disjoint on at most w_{LPM} bits.

At this point we need to understand the effects rule disjointness and prefix reorderability have on each other. The following two observations show that there is no interference.

*Observation 3 (*LPM-RD*):* If a classifier $\mathcal{K}^B$ is prefix-reorderable, $\mathcal{K}^{B'}$ is also prefix-reorderable for any $B' \subseteq B$.

*Observation 4 (*RD-LPM*):* Reordering of bit indices and expansion of $*$ bits preserve rule disjointness.

Thus, it is safe to combine transformations reducing classification width (RD-step) with those that aim for prefix reorderability (LPM-step) in any order. This is very important because it allows us to decouple them from each other. The only remaining question is in which order we should combine these steps: RD-LPM or LPM-RD? It turns out that there is no definite answer to this question.

Example 11: First, consider the hard classifier instance from Theorem 4 for $w = 4$:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	$*$	1	0	0	A_1
R_2	1	$*$	0	1	A_2

Assume that $w_{\text{LPM}} = 2$, and we intend to minimize the number of groups as in the PMGR problem. The RD-LPM approach will produce one group because it will first switch to $\mathcal{K}^{(3,4)}$, which is prefix-reorderable. But the LPM-RD approach will first split the rules into two groups, getting two groups instead of one.

On the other hand, consider the following example:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	$*$	0	1	0	A_2
R_3	1	0	$*$	$*$	A_3
R_4	$*$	1	0	$*$	A_4

Here the LPM-RD approach produces exactly two groups: $\{R_1, R_3\}$ and $\{R_2, R_4\}$, which are both order-independent on the first and second bit respectively. Although RD-LPM may use the same solution as LPM-RD, it is not required to

do so, and it may select the following two groups instead: $\{R_1, R_2\}$ and $\{R_3, R_4\}$. Each of those groups will require two subgroups to be represented on an LPM infrastructure producing four groups in the final representation. Thus, for the MINGR-w problem different approaches can work better for different instances of classifiers.

In the MAXCOV-w problem the objective is different: complete coverage is not required, but the number of rules is limited. The approach we suggest here is to greedily take the largest possible subset which is both rule disjoint and prefix-reorderable. It can be found found in two steps: first, we find a rule disjoint group using a greedy algorithm from [14]; second, we run MAXCOV with $\beta = 1$ to find a maximal prefix-reorderable subset. By Observation 4, the resulting group possesses both properties. We call this algorithm RD-MC. Another version of this algorithm, RD-MC-EXP, runs an additional `expand_and_fit` step after MAXCOV.

Another curious interaction between LPM and RD happens when we vary w_{LPM} (infrastructure limits) in the RD-LPM approach. On one hand, if RD reduces the number of bits, it is easier for LPM to find prefix-reorderable decomposition of rules (e.g., set $w_{\text{LPM}} = w/2$ in Theorem 4 and get a $|\mathcal{K}|$-fold decrease in the number of groups). On the other hand, RD itself may require a large number of groups, and the lower we set w_{LPM}, the more groups we will have. At the extreme, the following classifier is prefix-reorderable, but RD yields 4 groups even if $w_{\text{LPM}} = 4$:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	0	0	0	A_1
R_2	0	0	0	$*$	A_2
R_3	0	0	$*$	$*$	A_3
R_4	0	$*$	$*$	$*$	A_4

VII. DYNAMIC UPDATES

Another recurring theme in classifier representations is the support of dynamic updates. We have already mentioned two major service categories: traffic forwarding and service policies that represent economic models and traffic signatures designed in advance (e.g., quality of service, access control, firewall). Dynamic updates are not too important for the second category; offline computation can be valid in most cases. What if we still do need dynamic updates? Deletions and insertions that maintain groupwise representations are simple. If a new rule cannot be added to an existing group, or the traditional part is full, the multigroup part can be recomputed. Implementation of these cases is straightforward, so due to space constraints we do not go into further details here; the update procedure is similar to the one proposed in [14, Section 7.2]. There are two important factors that make a specific representation feasible: update resolution (that defines which part of a classifier is affected) and update frequency. Since there are cases when recomputation of a multigroup representation is required, we prefer to limit applicability of our representations only to the second group of services.

VIII. EVALUATION

To validate the proposed approach, we have run simulations on classifiers from the Classbench benchmark [19] generated with real parameters. Each classifier in the tables below has about 50,000 rules. The generated classifiers contained range-based fields, so in order to convert and operate on the entire multifield classifier we used the commonly used SRGE encoding scheme based on Gray coding [25]. Our code for processing and evaluating the classifiers is available at GitHub [26]. We did not measure lookup times since our approach serves as an abstract layer to find bit identities for lookup, independently of the underlying LPM infrastructure. Our evaluations are intended to validate the feasibility of the proposed approach, so we also did not compare our implementations with, for instance, software-based solutions intended to reduce size. We used four algorithms in tne evaluations: (1) RD is a greedy algorithm from [14] that iteratively removes bits that represent unique differences for the smallest number of rule pairs, breaking ties by the share of $*$ bits; (2) RD_{exact} is the same but removes only non-exact bits and does not stop until all bits are exact [14]; (3) RD-MC, our main heuristic, and (4) RD-MC-EXP have been discussed in Section VI.

Table I shows the results of our proposed heuristics without extra memory. We have covered five values of l: $l = 16$, $l = 24$, $l = 32$, and $l = 64$ for the composition of RD and LPM, and $l = 104$ which corresponds to "pure" LPM with no width limit, and have computed the number of rules

	Rules	$l=16$			$l=24$			$l=32$			$l=64$			$l=104$		
	$\beta=$	5	10	20	5	10	20	5	10	20	5	10	20	5	10	20
acl1	65331	85.1	96.0	99.8	93.5	97.8	99.9	93.8	97.7	99.7	92.1	98.6	98.7	72.6	86.2	95.1
acl2	96334	21.9	33.7	49.8	37.7	53.1	73.0	43.1	59.4	80.4	70.5	78.4	82.6	42.9	63.6	83.3
acl3	90208	56.4	77.5	85.8	73.9	89.9	97.7	75.7	91.3	97.6	72.9	82.1	86.2	49.8	67.2	83.5
acl4	84522	30.0	45.8	63.8	48.1	65.6	82.8	56.3	75.5	88.1	65.9	76.0	82.1	50.6	67.1	82.5
acl5	52240	13.5	25.8	46.4	33.5	54.1	72.4	47.1	68.4	85.4	88.4	98.2	100	76.0	90.4	98.7
fw1	165876	28.3	38.3	53.7	30.5	38.2	51.9	29.7	37.0	48.7	44.4	55.4	60.9	32.1	44.3	61.9
fw2	92730	61.5	81.5	83.2	84.3	86.3	87.9	80.9	87.5	92.7	40.5	46.3	49.3	73.7	89.8	98.5
fw3	128624	22.6	30.9	44.6	21.9	32.5	48.9	30.7	42.7	59.2	37.1	54.5	72.3	41.2	55.8	70.7
fw4	284747	28.0	41.0	48.1	51.2	53.0	54.6	51.6	62.9	74.7	13.4	17.4	23.3	31.5	44.8	61.4
fw5	106201	34.6	42.8	47.0	36.0	38.2	38.4	31.6	33.7	35.0	33.2	41.9	49.4	42.0	58.6	78.2
ipc1	67620	77.4	94.8	99.6	89.8	98.8	99.8	90.7	98.9	99.8	76.7	76.8	76.8	61.2	75.6	87.3
ipc2	50000	99.8	100	100	99.9	100	100	100	100	100	66.0	78.4	86.7	95.6	100	100

TABLE I
LPM RESULTS FOR DIFFERENT VALUES OF l AND β, COVERAGE %.

		Memory expansion by 4 bits								
		$l=16$			$l=24$			$l=32$		
	Rules	$\beta=5$	10	20	$\beta=5$	10	20	$\beta=5$	10	20
acl1	65331	85.1	96.1	99.8	93.7	97.1	99.7	93.9	97.7	99.7
acl2	96334	22.6	34.2	50.0	39.5	54.5	74.5	45.5	61.3	82.4
acl3	90208	57.5	78.4	86.2	74.6	91.2	98.9	77.6	92.6	98.5
acl4	84522	30.1	46.1	64.3	49.4	66.8	83.8	59.1	78.5	90.3
acl5	52240	13.5	25.8	46.4	33.6	54.3	73.1	47.7	69.1	86.7
fw1	165876	28.4	38.4	53.9	30.7	38.5	50.9	29.6	36.5	48.4
fw2	92730	61.5	81.5	83.2	85.6	90.4	93.6	82.9	87.7	91.7
fw3	128624	22.8	31.2	44.8	22.1	32.8	49.4	31.8	43.3	59.4
fw4	284747	28.0	41.1	48.2	48.7	50.4	52.0	62.2	73.4	84.7
fw5	106201	34.6	42.8	47.0	36.0	38.2	38.4	32.2	33.9	35.1
ipc1	67620	77.6	96.2	99.4	89.1	98.4	99.5	91.1	99.3	99.9
ipc2	50000	99.8	100	100	100.0	100.0	100.0	100.0	100	100
		Memory expansion by 8 bits								
		$l=16$			$l=24$			$l=32$		
	Rules	$\beta=5$	10	20	$\beta=5$	10	20	$\beta=5$	10	20
acl1	65331	85.1	96.1	99.8	93.7	97.1	99.7	93.9	97.9	99.9
acl2	96334	22.7	34.7	50.6	39.3	54.5	74.5	46.2	62.1	83.6
acl3	90208	57.5	78.5	86.6	75.3	90.8	97.3	78.3	95.4	99.0
acl4	84522	30.2	46.3	64.8	50.2	68.4	85.5	59.6	78.9	91.1
acl5	52240	13.5	25.8	46.4	33.6	54.3	73.1	47.7	69.1	86.7
fw1	165876	28.4	38.4	53.9	31.3	39.6	53.5	30.6	38.7	52.3
fw2	92730	61.5	81.5	83.2	94.9	98.9	99.9	94.2	99.3	100.0
fw3	128624	22.8	31.2	44.8	22.2	32.8	49.3	31.8	43.3	59.4
fw4	284747	28.0	41.1	48.2	48.7	50.4	52.0	62.4	74.2	86.3
fw5	106201	34.6	42.8	47.0	36.3	38.7	38.9	33.7	37.6	38.5
ipc1	67620	77.1	95.5	99.2	89.8	98.4	99.0	91.8	99.3	99.8
ipc2	50000	99.8	100	100	100.0	100.0	100.0	100.0	100	100

TABLE II
LPM RESULTS WITH BIT EXPANSION FOR DIFFERENT VALUES OF l AND β.

covered by top β groups for three different practical values of β: 5, 10, and 20. Note that for small values of l, the wider are the filters the larger the groups become and the fewer groups are needed; this is a property mostly inherited from the RD part since it is easier to find disjoint rules when their filters are wider. On the other hand, for extra large l it becomes significantly harder to preserve prefix reorderability, and we see that by the time we reach $l = 104$ the tension between these two effects becomes evident: in some examples pure LPM loses to RD-LPM composition for $l = 32$; in others, vice versa. This effect is also illustrated on Fig. 4 that shows rules coverage in more detail: we see that the coverage is far from monotone in l. Another effect is that depending on the LPM infrastructure implementation, wider groups (larger l) may use significantly, sometimes exponentially more memory in order to decrease lookup time [8], [9], so in reality a larger number of more narrow groups may be preferable to a smaller number of wider groups. In Table I we see that this kind of tradeoff is also possible: e.g., for *fw2* 10 groups with $l = 24$ cover more rules than 5 groups with $l = 32$, and so on. Table II shows the result with bit expansion for 4 and 8 bits. We see that this new allotted memory has virtually no effect in our example of practical classifiers at the level of β. The effect here is as follows: algorithm `min_pmgr_mstep` attempts to minimize the antichain size while trying to keep memory requirements as low as possible. This means that, in practice, `min_pmgr_mstep` does indeed significantly reduce the total number of groups, but does so by merging small groups, and they do not appear in the top 5-10 groups shown in Table II.

Figure 5 shows a comparison of the four algorithms in terms of rule coverage for three characteristic examples. Again, we see that the relative performance of algorithms highly depends on the specific instance, but the RD-MC-EXP heuristic proves to be most stable overall, across all examples. In all cases, we see that the proposed algorithms provide huge practical improvements in terms of TCAM size, covering, in most cases, a vast majority of the input classifier with but a few groups.

IX. Implementation in P4

We have implemented the proposed representations in the commonly used domain-specific programming language P4 that implements match-action pipelines [20]. We used the *Behavioral Model Version 2* (BMV2) framework that allows to implement a P4-programmable architecture as a virtual software switch; BMV2 is coming with the SIMPLE_SWITCH target architecture [27]. However, the problem with P4 is that one has to specify lookup tables, headers, etc. in advance, and the actual table content is added separately at "run time". Fixing bit indices implementing rule- disjointness and prefix reorderability before the actual table content is provided can significantly degrade efficiency for some instances of classifiers. To address this limitation, we extended the BMV2 table filling interface with the optimization engine that is target-independent; the only prerequisite is a P4 intermediate representation in JSON and the actual data. The engine replaces the original lookup table and its key structure specified in P4 with the equivalent representation based on data content, given a target specific maximal number of groups. We have released our implementation in open source [28].

X. Related Work

Research towards efficient implementations of packet classifiers falls into two major categories: algorithmic solutions (usually software based) and TCAM-based solutions; comprehensive surveys can be found in [29], [1]. Algorithmic solutions mainly rely on one of three techniques: decision trees, hashing, or coding-based compression. The works [30], [12] 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 [31]. 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 [32]. The works [33], [34] discuss hash-based solutions to match a packet to its possible matching rules. Efficient coding-based representations are shown in [35], [36]. TCAMs have no native support for range representations, so range encoding encompass an important direction in this domain [25], [37], [11]. Different approaches have been described to reduce number of entries: removing redundancies [38], [17], applying block permutations in the header space [39], transformations [13], [16], [40]. In particular [14], [15], [18] considered representations based on rule disjointness that we compose with prefix-reorderability to cut down classification width.

XI. Conclusion

We have proposed alternatives that allow to implement ternary bit strings with general priorities on commodity LPM infrastructure, completely transparently to its internals, while removing or significantly reducing the need in TCAM. Our approach is built around *prefix reorderability*, a novel structural property of classifiers. We extend our results to classifiers of arbitrary width by composing prefix reorderability with rule disjointness. Feasibility of the proposed representations is supported by evaluations on practical classifiers; in addition, we release a P4 implementation of our transformations.

Acknowledgments: This work was supported by the Russian Science Foundation grant no. 17-11-01276, "Networking and distributed systems and algorithms and related fundamental problems".

References

[1] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.

[2] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *SIGCOMM*, 1999, pp. 147–160.

[3] K. Huang, L. Ding, G. Xie, D. Zhang, A. X. Liu, and K. Salamatian, "Scalable tcam-based regular expression matching with compressed finite automata," in *ANCS*, 2013, pp. 83–93.

[4] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng, "Fast regular expression matching using small TCAM," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 94–109, 2014.

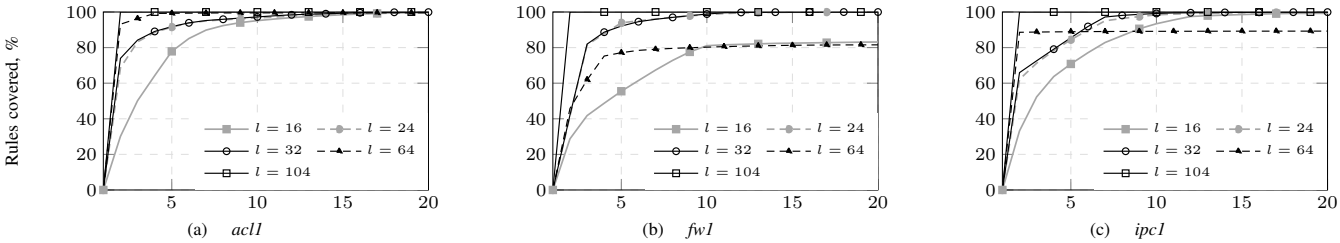


Fig. 4. A comparison of group sizes for the `min_group_partition` algorithm with different l: characteristic examples.

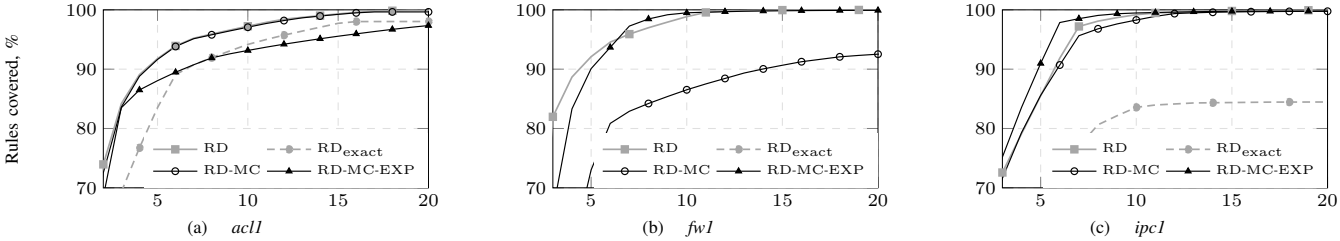


Fig. 5. A comparison of the algorithms with $l = 32$: characteristic examples.

[5] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *ANCS*, 2011, pp. 24–35.

[6] "Netlogic Microsystems. Content addressable memory," http://www.netlogicmicro.com.

[7] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *ISPASS*, 2006, pp. 120–129.

[8] "Cisco CRS forwarding Processor Cards," http://www.cisco.com/c/en/us/products/collateral/routers/ carrier-routing-system/datasheet-c78-730790.pdf.

[9] "The Cisco flow processor: Cisco's next generation network processor solution overview," http://www.cisco.com/c/en/us/products/collateral/routers/ asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html.

[10] F. Baboescu and G. V. G., "Scalable packet classification," in *SIGCOMM*, 2001, pp. 199–210.

[11] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM*, 1998, pp. 191–202.

[12] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput," in *SIGCOMM*, 2010, pp. 207–218.

[13] A. Kesselman, K. Kogan, S. Nemzer, M. Segal, "Space and speed tradeoffs in TCAM hierarchical packet classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.

[14] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting Order Independence for Scalable and Expressive Packet Classification," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1251–1264, 2016.

[15] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, O. Rottenstreich, "FIB efficiency in distributed platforms," in *ICNP*, 2016, pp. 1–10.

[16] K. Kogan, S. I. Nikolenko, P. Eugster, E. Ruan, 'Strategies for Mitigating TCAM Space Bottlenecks," in *HOTI*, 2014, pp. 25–32.

[17] K. Kogan, S. I. Nikolenko, w. Culhane, P. Eugster, E. Ruan, 'Towards efficient implementation of packet classifiers in SDN/OpenFlow," in *HotSDN*, 2013, pp. 153–154.

[18] S. I. Nikolenko, K. Kogan, G. Retvari, E. R. Bérczi-Kovács, A. Shalimov, "How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes," in *INFOCOM Workshops*, 2016, pp. 521–526.

[19] "ClassBench: A packet classification benchmark," http://www.arl.wustl.edu/classbench/.

[20] "The P4 language specification, version 1.0.3," http://p4.org/wp-content/uploads/2016/11/p4-spec-latest.pdf.

[21] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge: Cambridge university press, 1990.

[22] R. P. Dilworth, "A decomposition theorem for partially ordered sets," *Annals of Mathematics*, vol. 51, no. 1, pp. 161–166, 1950.

[23] D. R. Fulkerson, "Note on dilworth's decomposition theorem for partially ordered sets," *Proceedings of the American Mathematical Society*, vol. 7, no. 4, pp. 701–702, 1956.

[24] J. E. Hopcroft and R. M. Karp, "A n5/2 algorithm for maximum matchings in bipartite," in *SWAT*, 1971, pp. 122–125.

[25] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using Gray coding," *IEEE Trans. Computers*, vol. 61, no. 1, pp. 18–30, 2012.

[26] "Code for simulations," https://github.com/icnp2017/submission.

[27] "The Bmv2 simple switch target," https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.

[28] "Implementation on p4," https://github.com/icnp2017/submission.

[29] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[30] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, 2000.

[31] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *INFOCOM*, 2010.

[32] H. Song and J. S. Turner, "ABC: Adaptive binary cuttings for multidimensional packet classification," *IEEE/ACM Trans. Netw.*, vol. 21, no. 1, pp. 98–109, 2013.

[33] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *ACM/IEEE ANCS*, 2006.

[34] Y. Kanizo, D. Hay, and I. Keslassy, "Optimal fast hashing," in *Infocom*, 2009.

[35] A. Korösi, J. Tapolcai, B. Mihálka, G. Mészáros, and G. Rétvári, "Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously," in *ICNP*, 2014, pp. 332–343.

[36] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables for datacenter scalability," *IEEE JSAC*, vol. 32, no. 1, pp. 138–151, 2014.

[37] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *Infocom*, 2013.

[38] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *Infocom*, 2008.

[39] R. Wei, Y. Xu, and H. J. Chao, "Block permutations in Boolean space to minimize TCAM for packet classification," in *Infocom Mini-Conference*, 2012.

[40] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 237–250, 2011.

C Paper “How to implement complex policies on existing network infrastructure”

Authors. Pavel Chuprikov, Kirill Kogan, and Sergey Nikolenko

Abstract. Transport networks satisfy requests to forward data in a given topology. At the level of a network element, forwarding decisions are defined by flows. To implement desired data properties during forwarding, a network operator imposes economic models by applying policies to flows, ideally without dealing with underlying resource constraints. Policy splitting over multiple network elements under resource constraints is a hard optimization problem. We discuss limitations of the proposed methods and existing Boolean minimization techniques. The major contribution of this work is an optimal solution with linear time complexity at the price of a single bit forwarded in every packet. The results are supported by a comprehensive evaluation study that compares previous and currently proposed methods.

How to implement complex policies on existing network infrastructure

Pavel Chuprikov
IMDEA Networks Institute
Steklov Institute of Mathematics at St. Petersburg

Kirill Kogan
IMDEA Networks Institute

Sergey Nikolenko
Steklov Institute of Mathematics at St. Petersburg

ABSTRACT

Transport networks satisfy requests to forward data in a given topology. At the level of a network element, forwarding decisions are defined by flows. To implement desired data properties during forwarding, a network operator imposes economic models by applying policies to flows, ideally without dealing with underlying resource constraints. Policy splitting over multiple network elements under resource constraints is a hard optimization problem [6, 7]. We discuss limitations of the proposed methods and existing Boolean minimization techniques. The major contribution of this work is an optimal solution with linear time complexity at the price of a single bit forwarded in every packet. The results are supported by a comprehensive evaluation study that compares previous and currently proposed methods.

KEYWORDS

network management; software-defined networking

ACM Reference Format:
Pavel Chuprikov, Kirill Kogan, and Sergey Nikolenko. 2018. How to implement complex policies on existing network infrastructure. In *Proceedings of the ACM Symposium on SDN Research 2018 (SOSR'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The modern network edge has to perform tasks with heterogeneous complexity, including features such as advanced VPN services, deep packet inspection, firewall, intrusion detection, to list just a few. It is not always possible to use cloud resources to implement desired policies because of proximity issues and multiple management domains. Each required feature may require a different amount of processing at the level of individual network elements and may introduce new challenges for traditional architectures, leading to serious implementation and performance issues. In particular, increasing complexity of services implemented in-network can require additional computing and memory resources, which usually leads to the need to upgrade already existing (and expensive) network infrastructure. The challenge, however, is to find scalable and manageable methods to support these complexities without upgrading the capabilities of individual network elements. One possibility is to define a "virtual pipeline" by splitting required resources along the path of a packet flow using some knowledge of the network. The problem of splitting a policy into several lookup tables while minimizing the maximal local table size has been broadly studied in [6, 7] and found to be an intractable optimization problem. Heuristics proposed in [6, 7] suffer from three main problems: they have high computational cost, the resulting total classifier size can grow significantly (exponentially in the worst case), and finally, none of them can handle dynamic fields where a header can change along the routing path as a result of actions applied on a switch. The main contribution of this work is an optimal algorithm with linear time complexity that can handle dynamic fields at the price of a single bit of metadata prepended to every packet.

SOSR'18, March 2018, Los Angeles, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 MODEL DESCRIPTION

We begin with formal definitions needed for further exposition, starting with the basic notions of a packet header and classifier.

A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits: each bit $h_i \in H$ has a value of either zero or one, $h_i \in \{0, 1\}$, $1 \le i \le w$. For example, (1 0 0 0) is a 4-bit header. A *classifier* $\mathcal{K} = \{R_1, \ldots, R_n\}_{\prec}$ is an ordered set of rules with ordering $\prec$ (in all examples we assume that $R_i \prec R_{i+1}$), where each *rule* $R_i = (F_i, A_i)$ consists of a *filter* F_i and a pointer to the corresponding *action* A_i. A filter $F = (f_1, \ldots, f_w)$ is a sequence of, again, w values corresponding to bits in the headers, but this time possible bit values are 0, 1, and $*$ ("don't care").

Example 2.1. Consider a sample classifier $\mathcal{K}$ on $w = 4$ bits:

$\mathcal{K}$	#1	#2	#3	#4	$Action$
R_1	*	*	1	0	A_1
R_2	1	0	*	*	A_2
R_3	0	0	*	*	A_3
R_4	*	*	1	1	A_4

A classifier's main purpose is to find the action corresponding to the highest priority rule that matches a given header. A header H *matches* a rule R iff it matches R's filter, and it matches a filter F iff for every bit of H the corresponding bit of F has either the same value or $*$. The set of rules has a non-cyclic priority ordering $\prec$; if a header matches both R and R' for $R \prec R'$, the action of rule R is applied. In particular, in Example 2.1 the header (1 0 1 0) matches both R_1 and R_2, but A_1 is applied.

Two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$ are *equivalent* if they choose the same actions for every possible incoming packet; we assume that no action is applied if the packet does not match any rule.

3 SINGLE FLOW

As a building block in the automatic translation of network-wide policies to individual switch configurations, the work [6] introduced policy splitting over the path of the corresponding flow. To represent the equivalence between the original policy and its distributed representation, we define a *splitting* of a given classifier $\mathcal{K}$ to be a *sequence* of classifiers that is equivalent to $\mathcal{K}$. A sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ operates as follows: an incoming packet p is matched in each $\mathcal{K}_i$ in order, immediately applying actions of the matched rule. We define an *l-splitting of $\mathcal{K}$* as a splitting of $\mathcal{K}$ into l classifiers.

Originally, the work [6] introduced the following problem that captures policy splitting for a single flow.

Problem 1 (FlowSplit). *Given a classifier $\mathcal{K}$ and a sequence of switch capacities $c_1, c_2, \ldots, c_l$, find an l-splitting $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ of $\mathcal{K}$ such that $\mathcal{K}_i$ has at most c_i rules.*

Policy splitting becomes a nontrivial problem if there are rules R and R' in a classifier $\mathcal{K}$ that *intersect*, i.e., there exists a header that matches both rules. The reason is that a packet can now be matched twice, as the following example illustrates.

Example 3.1. Let us split $\mathcal{K}$ from Example 2.1 in two parts:

$\mathcal{K}_1$	#1	#2	#3	#4	$Action$
R_1	*	*	1	0	A_1
R_3	0	0	*	*	A_3

$\mathcal{K}_2$	#1	#2	#3	#4	$Action$
R_2	1	0	*	*	A_2
R_4	*	*	1	1	A_4

Now a header (1 0 1 0) in the sequence $(\mathcal{K}_1, \mathcal{K}_2)$ is matched twice, and both A_1 and A_2 are applied to a packet.

One way to fix Example 3.1 is to add a special **nop** action to $\mathcal{K}_2$ that does nothing but forward a packet further.

$\mathcal{K}'_2$	#1	#2	#3	#4	$Action$
R_1	*	*	1	0	**nop**
R_2	1	0	*	*	A_2
R_3	0	0	*	*	**nop**
R_4	*	*	1	1	A_4

Now the sequence $\mathcal{K}_1, \mathcal{K}'_2$ is a splitting of classifier $\mathcal{K}$ from Example 2.1.

4 SINGLE FLOW WITHOUT METADATA

Two major works on policy splitting, *Palette* (Palette) [7] and *One-Big-Switch* (OBS) [6], employ different methods to avoid the undesirable effect of rule intersection and preserve equivalence. In this section, we discuss the main ideas behind previously proposed methods. Next, we show how to apply *Boolean Minimization* (BM) techniques [1, 9] in a way that can give more flexibility but ultimately also limited. Finally, we discuss three major drawbacks that are common to all three approaches: slow running time, extensive rule duplication, and lack of support for dynamic fields.

4.1 Palette

Palette's idea [7] is to produce a splitting $\mathcal{K}_1, \ldots, \mathcal{K}_l$ that avoids any rule intersection between $\mathcal{K}_i$ and $\mathcal{K}_j$ for $i \neq j$. Two methods for building such a splitting were proposed: pivot-based and cut-based. Both methods expand "don't care" bits of possibly intersecting rules in order producing several non-intersecting sets of rules, as the following example shows.

Example 4.1. Expanding bit #1 in the classifier $\mathcal{K}$ from Example 2.1 yields two non-intersecting classifiers:

$\mathcal{K}^0$	#1	#2	#3	#4	$Action$
R^1_1	1	*	1	0	A_1
R_2	1	0	*	*	A_2
R^1_4	1	*	1	1	A_4

$\mathcal{K}^1$	#1	#2	#3	#4	$Action$
R^0_1	0	*	1	0	A_1
R_3	0	0	*	*	A_3
R^0_4	0	*	1	1	A_4

The rules from different subclassifiers do not intersect, hence, $\mathcal{K}^0$ and $\mathcal{K}^1$ form a 2-splitting of $\mathcal{K}$.

The pivot-based method is a top-down iterative approach. At every iteration, we consider the current splitting $\mathcal{K}_1, \ldots, \mathcal{K}_k$ of $\mathcal{K}$, starting from a single classifier $\mathcal{K}$. Among the current splitting, we choose the classifier $\mathcal{K}_{i^*}$ with the largest number of rules and search for a bit index j whose expansion results in $\mathcal{K}^0_{i^*}$ and $\mathcal{K}^1_{i^*}$ that minimize the value of $\max\{|\mathcal{K}^1_{i^*}|, |\mathcal{K}^0_{i^*}|\}$. Finally, we replace $\mathcal{K}_{i^*}$ with $\mathcal{K}^1_{i^*}$ and $K^0_{i^*}$. Example 4.1 in fact presents the result of the first iteration of this procedure; note that if bit #2 were expanded instead, the expansions

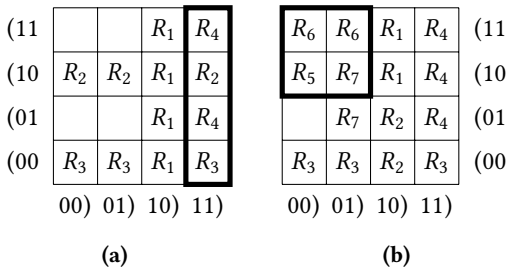


Figure 1: Classifiers from Example 2.1 and Example 4.2 represented in 2D. Bits #1 and #2 correspond to rows; bits #3 and #4, to columns. Each square shows the highest priority rule that matches the corresponding header.

would be of size 2 and 4. The time complexity of this algorithm is $O(nwl)$, where $n = |\mathcal{K}|$, w is the classification width, and l is the path length.

One of Palette's drawbacks is that it minimizes the maximal table size, which essentially assumes that switch capacities are equally utilized, $c_1 = c_2 = \ldots = c_n$. Since routing decisions of packet flows are decoupled from applying policies, this assumption significantly reduces the applicability of Palette in practice. Palette's cut-based approach uses graph-cut heuristics as a black box, and it is even harder to adjust for heterogeneous capacities.

4.2 One Big Switch

Palette not only optimizes a slightly different objective but also has fundamental limitations. First, an early decision to expand a bit propagates further to where it may be a suboptimal decision. Second, Palette always cuts the header space in half, which leads to worse performance when the path length is not a power of two. The *One Big Switch* (OBS) approach [6] avoids both limitations by taking a different *incremental* path.

The OBS algorithm also begins with a classifier $\mathcal{K}^{(0)} = \mathcal{K}$ but constructs the splitting $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ incrementally, one classifier at a time. At step i, the algorithm chooses a subregion r_i of the header space that overlaps with at most c_i rules from $\mathcal{K}^{(i-1)}$. These overlapping rules are put into $\mathcal{K}_i$; if an overlap is only partial then the rule is first cut by r's boundary. Finally, $\mathcal{K}^{(i)}$ is constructed by taking all rules from $\mathcal{K}^{(i-1)}$ that are not completely covered by r_i together with a special rule $(r_i, \textbf{nop})$ with the highest priority that guarantees that packets will not be matched more than once. Note that partially overlapping rules are present both in $\mathcal{K}_i$ and $\mathcal{K}^{(i)}$, which is similar to the rule duplication effect of Palette.

To illustrate this idea, we draw the header space in two dimensions (see Figure 1), with half of the bits shown along one axis, and the other half shown along the other axis. The classifier $\mathcal{K}$ from Example 2.1 can be represented in this space as in Figure 1a. Each square shows the highest priority rule from $\mathcal{K}$ that matches the corresponding header; e.g., for $H = (1\,0\,1\,1)$ the highest priority matching rules is R_2 (shown in the intersection of the second row and fourth column). A possible subregion $r_1 = (*\ *\ 1\ 1)$ for $c_1 = 3$ is shown in bold.

The OBS algorithm chooses r to be the rectangle in this two-dimensional space that maximizes the ratio $\frac{n_{\subseteq}-1}{n_{\cap}}$, where $n_{\subseteq}$ is the number of rules that lie inside r and $n_{\cap}$ is the number of rules that overlap with r. The time complexity of the overall algorithm is $O(n^5wl)$, where $n = |\mathcal{K}|$, w is the classification width, and l is the path length. In practice, however, we will see in Section 7 that OBS is not quite as slow as one might think.

Comparing OBS and Palette using the classifier from Example 2.1, it may seem that OBS is inferior since it cannot fit into the required capacity: any r_1 leaves four rules for the next switch. However, according to the evaluation study in [6] this is a rather rare case, and most of the time higher flexibility of OBS leads to better results, as in the example below.

Example 4.2. Consider a FlowSplit instance with $c_1 = 3$, $c_2 = 6$, and the following classifier:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	*	1	0	A_1
R_2	0	*	1	0	A_2
R_3	0	0	*	*	A_3
R_4	*	*	1	1	A_4
R_5	1	0	0	0	A_5
R_6	1	1	0	*	A_5
R_7	*	*	0	1	A_5

The pivot-based method will not satisfy the capacities, since expanding any bit gives a subclassifier with at least four rules. From the two-dimensional representation on Figure 1b we conclude that OBS can satisfy the requirements by taking $r_1 = (1\ *\ 0\ *)$, which covers R_5 and R_6 and partially overlaps with R_7. $\mathcal{K}_2$ is left with R_1, R_2, R_3, R_4, R_7, and $(r_1, \textbf{nop})$.

4.3 Boolean Minimization

The OBS is more flexible than Palette, but it requires the covered region to be a rectangle. We suggest an even more flexible *Boolean Minimization* (BM) approach that drops this requirement and allows to assign arbitrary subsets of rules to a switch, at the expense of adding more **nop** rules to subsequent switches.

Example 4.3. The classifier from Example 2.1 can be represented more efficiently than Palette with general **nop**

rules:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1
R_2	1	0	*	*	A_2
R_3	0	0	*	*	A_3

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
$R_{\textbf{nop}}$	*	0	*	*	**nop**
R_4	*	*	1	1	A_4

Now $R_{\textbf{nop}}$ is not a prefix on its first two bits, so it would not be a rectangle on Figure 1a.

We use an incremental approach similar to OBS. At step i, we allow an *arbitrary* subset $\mathcal{K}_i' \subseteq \mathcal{K}^{(i-1)}$ to be a candidate for $\mathcal{K}_i$. Once $\mathcal{K}_i'$ has been chosen, we consider $\mathcal{K}^{(i-1)}$, set the action to **nop** for all rules in $\mathcal{K}^{(i-1)} \setminus \mathcal{K}'$, and run one of the Boolean minimization heuristics [1, 9] to remove redundant rules (in Example 4.3 we choose $\mathcal{K}_1' = \{R_1, R_2, R_3\}$). If the result fits in c_i, it is selected as $\mathcal{K}_i$. To construct $\mathcal{K}^{(i)}$ we take $\mathcal{K}^{(i-1)}$, set actions for all rules in $\mathcal{K}' \cap \mathcal{K}^{(i-1)}$ to **nop**, and run Boolean minimization again (in Example 4.3 this leads to $\mathcal{K}^{(2)} = \mathcal{K}_2$).

We use heuristics for Boolean minimization since it is NP-hard to find the exact solution. The time complexity of those heuristics is $O(n^3 l w \log n)$, where n is the number of rules in the classifier, w is the rule width in bits, and l is the path length.

One disadvantage of BM is that when a low-priority rule intersects many higher-priority rules, the switch with the low-priority rule must contain all the intersecting higher-priority rules, either with **nop** or with the original action. Consider the following example:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	1	0	1	A_1
R_2	1	0	1	0	A_2
R_3	1	1	1	0	A_3
R_4	*	0	0	1	A_4
R_5	1	*	*	*	A_5

BM is not able to satisfy capacity requirements $c_1 = c_2 = 4$ since a switch with R_5 must also hold the other four rules to avoid spurious applications of A_5. Both OBS and Palette are able to find a feasible solution.

4.4 Incomparability of considered methods

We have seen that OBS, Palette, and BM are all incomparable in general. OBS potentially has greater flexibility than Palette but cannot be extended to arbitrary bit strings. BM has higher flexibility in rule placement than OBS and Palette but performs badly when one rule intersects too many others.

At the same time, there are limitations common to all three methods:

(1) *memory expansion* due to rule duplication from expanded bits (in Palette), partially overlapping rules (in OBS), or **nop** rules (in BM);
(2) *slow running time* due to the high complexity of underlying computational problems;
(3) *inability to handle dynamic fields*, i.e., fields that are changed by actions and participate in the classification process.

In the worst case, rule duplication may cause a memory increase proportional to the length of the path. Below we show such an example for Palette:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	1	*	*	A_1
R_2	1	*	1	*	A_2
R_3	1	*	*	1	A_3
R_4	*	1	1	*	A_4
R_5	*	1	*	1	A_5
R_6	*	*	1	1	A_6

Correctness under dynamic fields is a more subtle issue. Consider the splitting $(\mathcal{K}^0, \mathcal{K}^1)$ from Example 4.1. Assume that A_2 sets the first bit to zero, while A_3 sets it to one. Let us now try to match the header (1 0 0 0):

(1) (1 0 0 0) matches R_2 of $\mathcal{K}^0$ and changes to (0 0 0 0);
(2) (0 0 0 0) matches R_3 of $\mathcal{K}^1$, changing back to (1 0 0 0).

Dynamic fields can result in surprising interactions of seemingly independent actions, and equivalence can be violated in ways that are very hard to detect.

In the following, we propose a novel method that avoids all the above drawbacks by actually *exploiting* dynamic fields.

5 ONE-BIT PRICE FOR SIMPLICITY

In this section, we discuss the benefits of having a single extra bit of metadata that can be set, passed between network elements, and used in classification. In particular, we show how to satisfy any switch capacities once they sum up to at least $|\mathcal{K}|$. Moreover, the transformation will remain equivalent even in the presence of dynamic fields.

To understand how a single bit of metadata helps, we consider a very naive approach to FlowSplit. For a classifier $\mathcal{K}$ and a sequence of switch capacities $c_1, c_2, \ldots, c_l$ such that $\sum_i c_i \geq |\mathcal{K}|$, we simply put the first (top priority) c_1 rules to the first switch, then next c_2 rules to the second switch, and so on until all rules in $\mathcal{K}$ are covered. Unfortunately, a packet can be matched twice in a way similar to Example 2.1.

The remedy for this problem is obvious: do not let a packet to be matched twice! To implement this we use an extra metadata bit, which we call **matched**, that shows whether a packet was already matched. This bit is initially set to 0 and changed to 1 on any match. We perform classification for a packet only if **matched** is not set. Algorithm 1 (OneBit) formalizes this idea.

Example 5.1. Consider the classifier $\mathcal{K}$ from Example 2.1. The OneBit algorithm splits its rules into two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$, augmenting every non-default action of $\mathcal{K}_1$ with **matched** $\leftarrow 1$ and the default **nop** action with **matched** $\leftarrow 0$:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1, **matched** $\leftarrow 1$
R_2	1	0	*	*	A_2, **matched** $\leftarrow 1$
R_3	0	0	*	*	A_3, **matched** $\leftarrow 1$
default : **matched** $\leftarrow 0$					

$\mathcal{K}_2$ runs classification only if **matched** has not been set:

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
if matched $= 0$ **then**					
R_4	*	*	1	1	A_4, **matched** $\leftarrow 1$

Note that without the **matched** bit the header (1 0 1 1) would be erroneously matched to both R_2 in $\mathcal{K}_1$ and R_4 in $\mathcal{K}_2$.

Instead of checking the **matched** bit prior to classification we could incorporate this check into the classifier itself, similar to **nop**-actions of OBS and BM. But in this case we would be wasting processing cycles for packets that were already matched. One of the advantages of the **matched** bit approach is that it avoids these redundant efforts.

The following theorem demonstrates that OneBit operates correctly.

Theorem 5.2. *Given a FlowSplit's instance $(\mathcal{K}, \{c_i\})$ with $\sum_i c_i \geq |\mathcal{K}|$, the OneBit algorithm constructs, in time $O(|\mathcal{K}|)$, an l-splitting of $\mathcal{K}$ that remains correct in the presence of dynamic fields and has the same total number of rules $|\mathcal{K}|$.*

Proof. Assuming $|\mathcal{K}| \gg n$, the running time is dominated by line 5 that adds rules to $\mathcal{K}_i$. Since every added rule is removed from $\mathcal{K}$ in line 10, $\sum_i \mathcal{K}_i = |\mathcal{K}|$. Lines 7 and 9 use the **matched** bit to make sure that no packet is matched more than once; since none of $\mathcal{K}$'s rules depend on **matched**, dynamic fields do not violate this guarantee. Finally, consider an arbitrary header H that has been matched in $R \in \mathcal{K}$ and put into $\mathcal{K}_j$. H does not match any rule $R' \prec R$; thus, due to line 3 it is not a match to any $\mathcal{K}_{j'}$ with $j' < j$, and necessarily H is matched by R in $\mathcal{K}_j$. □

Hence, a single **matched** bit allows to construct a simple heuristic for the FlowSplit problem, which (1) does not expand rules, (2) has linear time complexity, and (3) remains correct even in the presence of dynamic fields.

6 NETWORK-WIDE POLICY SPLITTING

The FlowSplit problem considers the splitting of a policy controlling the behavior of a single flow. On the network-wide level, there are several flows, and each has its own policy and its own forwarding path. The authors of OBS suggested in [6] the following problem for joint multi-flow splitting.

Algorithm 1 OneBit$(\mathcal{K}_\prec, c_1, \ldots, c_l)$

```
1:  Initialize K_1, K_2, ... K_l to empty classifiers
2:  for i in 1, 2, ..., l do
3:      Let R_i be c_i highest priority rules of K
4:      for (F, A) ∈ sorted(R, ≺) do
5:          Append (F, [A, matched ← 1]) to K_i
6:      if i = 1 then
7:          Set default action to matched ← 0 in K_i
8:      else
9:          Make K_i conditioned on matched = 1
10:     Remove R from K
11: return K_1, K_2, ..., K_l
```

Problem 2 (MultiFlowSplit). *Given a set of nodes V, node capacities $c : V \to \mathbb{N}$, and a set of k path/classifier pairs $(P_i, \mathcal{K}_i)$, where P_i is a sequence of vertices $(v^i_1, \ldots, v^i_{l_i})$, find a capacity allocation $a : V \times [k] \to \mathbb{N}$ such that $\sum_{i=1}^{k} a(v, i) \leq c(v)$, and for each i the pair $(P_i, \mathcal{K}_i)$ is a solution for FlowSplit with $\mathcal{K} = \mathcal{K}_i$ and $c_j = a(v^i_j, j)$ for $j = 1, \ldots, l_i$.*

The approach of OBS to solving MultiFlowSplit [6] contains three steps that are repeated iteratively:

(1) estimate the total capacity η_i required by the ith flow;
(2) use η_i to allocate per-path per-node capacities by solving a linear programming problem;
(3) *try* to solve the FlowSplit problem for each flow.

If the OBS algorithm cannot solve FlowSplit for some flows on step (3), the algorithm goes back to (1), where η_i are increased for failed flows. Unfortunately, there is no clear bound on the number of iterations.

The main reason that iterations were needed at all is that η_i are mere estimations: the success of the OBS algorithm on each FlowSplit problem depends not only on the total capacity but also on the capacities of individual switches.

Fortunately, if we use OneBit with a similar approach, Theorem 5.2 implies that we succeed if and only if $\sum_i c_i \geq |\mathcal{K}|$. Thus, if we take $\eta_i = |\mathcal{K}_i|$ we will be able to solve MultiFlowSplit in a single iteration of steps (1)-(3), and the resulting algorithm is guaranteed to work in polynomial time.

7 EVALUATION RESULTS

To evaluate the proposed approaches for FlowSplit, we ran simulations on 12 classifiers from Classbench [18] generated with real parameters, each with $\approx$ 10K rules on 6 fields. In particular, we used the ACL test suite since it is a more likely example of a splittable policy. We investigated only the uniform version of FlowSplit, where all capacities are equal. Importantly, we applied the Boolean minimization procedure as described in [10] to make the comparison more fair for

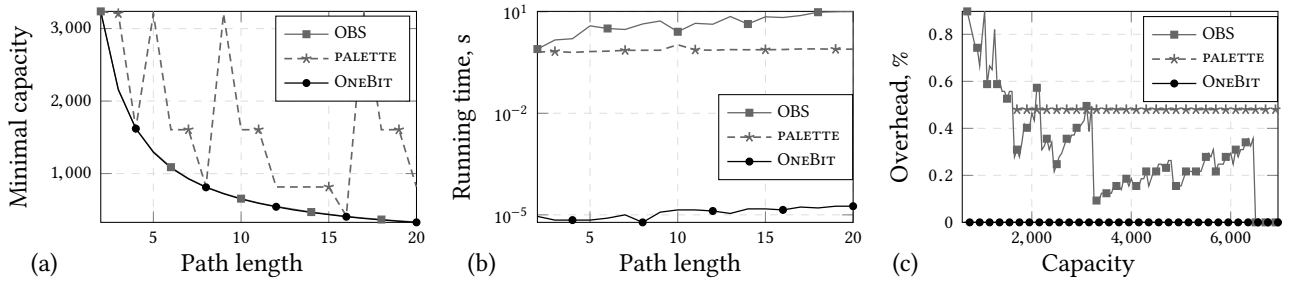


Figure 2: Experimental results on a ClassBench ACL classifier with 9928 rules.

those algorithms that do not use any classifier simplification internally. We used the pivot-based heuristic for Palette because the cut-based heuristic took too much time to complete. The code for our simulations is available at GitHub [4].

Results of our experimental evaluation are summarized in Figure 2, which shows detailed results for an ACL classifier, and Table 1, which shows other classifiers in our study; OneBit's overhead is not shown in Table 1 because it is always 0. In the first experiment (Fig. 2a, $c_{\min}$ column in Table 1), we investigated what is the minimal switch capacity for which an algorithm is able to accommodate the classifier, and how it depends on the path length. We computed these results with a binary search of different capacity values. We see that, as expected, OneBit is optimal, with OBS being several times worse in some cases (e.g., acl3 and ipc2 in Table 1), and Palette often imposing significant overhead in terms of extra capacity. The second experiment (Fig. 2b, t in Table 1) shows the running time of different algorithms. We see that OneBit, as expected, is several orders of magnitude faster than both Palette and OBS. In Table 1, we computed the runtimes on the same capacity value (the largest of minimal capacities) so that they would be comparable. The third experiment (Fig. 2c, OH in Table 1) shows the total (relative) overhead of different algorithms' representations in terms of actual increase in the total size of the resulting classifiers. OneBit is the optimal algorithm with zero overhead, OBS performs well in terms of this metric on the ACL classifier in Fig. 2c, but leads to a substantial overhead for several other classifiers (e.g., acl3 and fw2 in Table 1), and Palette is noticeably worse. Thus, evaluations support our theoretical conclusion that OneBit is the best possible with respect to all metrics.

		Palette			OBS			OneBit	
	Size	$c_{\min}$	OH,%	t,s	$c_{\min}$	OH,%	t,s	$c_{\min}$	t,s
acl1	9928	2480	2.66	0.68	1030	3.49	11.03	997	$\leq 10^{-5}$
acl2	7433	2308	47.50	0.54	1214	60.85	105.81	746	$\leq 10^{-5}$
acl3	9149	3622	123.49	0.63	2687	157.15	1638.02	919	$\leq 10^{-5}$
acl4	8059	2870	98.52	0.59	1706	106.82	434.41	809	$\leq 10^{-5}$
acl5	9072	2265	0.04	0.63	912	0.20	2.60	910	$\leq 10^{-5}$
fw1	8902	3776	132.33	0.67	2423	159.35	80.42	891	$\leq 10^{-5}$
fw2	9968	4308	160.03	0.70	3688	250.71	651.36	997	$\leq 10^{-5}$
fw3	8029	3877	180.91	0.59	2818	243.77	433.64	804	$\leq 10^{-5}$
fw4	2633	1196	173.15	0.31	800	192.10	31.63	268	$\leq 10^{-5}$
fw5	8136	2651	81.31	0.58	1780	113.03	62.38	814	$\leq 10^{-5}$
ipc1	8338	2260	26.52	0.58	1088	29.89	42.49	837	$\leq 10^{-5}$
ipc2	10000	4348	134.10	0.67	2406	111.51	300.37	1002	$\leq 10^{-5}$

Table 1: Experimental results on ClassBench classifiers with path length $l = 10$: min capacity $c_{\min}$, overhead OH, and runtime t.

8 RELATED WORK

In addition to Palette [7] and OBS [6], there are other works that deal with application of network-wide policies; unfortunately, they do not consider switch resource constraints and as a result are of limited applicability in practice [2, 5, 21]. Other works consider the policy splitting while changing the routing [14, 15, 20]. This setting is different from Palette, OBS, and our approach, where routing is not affected as a result of policy splitting. Optimizing classifiers on a single switch is a well-developed line of research, with numerous heuristics suggested in works such as [3, 8, 11–13, 16, 17, 19]. In this light it is interesting to explore applications of Boolean minimization methods to reduce memory requirements that we apply on input classifiers [1, 9] prior to running policy splitting methods.

9 CONCLUSION

In this work, we have studied the automatic translation of a network-wide policy to individual switch configurations. In particular, we have explored efficient implementations of the path splitting algorithm used for such translations. We show that at the price of a single bit per packet the intractable combinatorial problem previously considered in [6, 7] can be solved optimally in linear time.

Acknowledgements. We are grateful to the authors of [7] and [6] for providing us with the source code for their algorithms. This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation.

REFERENCES

[1] Eric Allender, Lisa Hellerstein, Paul McCabe, Toniann Pitassi, and Michael E. Saks. 2008. Minimizing Disjunctive Normal Form Formulas and AC^0 Circuits Given a Truth Table. *SIAM J. Comput.* 38, 1 (2008), 63–84.

[2] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. 2009. Rethinking enterprise network control. *IEEE/ACM Trans. Netw.* 17, 4 (2009), 1270–1283.

[3] Pavel Chuprikov, Kirill Kogan, and Sergey I. Nikolenko. 2017. General ternary bit strings on commodity longest-prefix-match infrastructures. In *ICNP*. 1–10.

[4] Code for simulations 2017. Code for simulations. https://github.com/distributedpolicies/submission. (2017).

[5] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. 2000. Implementing a distributed firewall. In *CCS*. 190–199.

[6] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*. 13–24.

[7] Yossi Kanizo, David Hay, and Isaac Keslassy. 2013. Palette: Distributing tables in software-defined networks. In *INFOCOM*. 545–549.

[8] Alexander Kesselman, Kirill Kogan, Sergey Nemzer, and Michael Segal. 2013. Space and speed tradeoffs in TCAM hierarchical packet classification. *J. Comput. Syst. Sci.* 79, 1 (2013), 111–121. DOI: https://doi.org/10.1016/j.jcss.2012.06.001

[9] Subhash Khot and Rishi Saket. 2008. Hardness of Minimizing and Learning DNF Expressions. In *FOCS*. 231–240.

[10] Kirill Kogan, Sergey I. Nikolenko, Patrick Eugster, and Eddie Ruan. 2014. Strategies for Mitigating TCAM Space Bottlenecks. In *HOTI*. IEEE, 25–32.

[11] Kirill Kogan, Sergey I. Nikolenko, Patrick Eugster, Alexander Shalimov, and Ori Rottenstreich. 2017. Efficient FIB Representations on Distributed Platforms. *IEEE/ACM Trans. Netw.* 25, 6 (2017), 3309–3322.

[12] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2016. Exploiting Order Independence for Scalable and Expressive Packet Classification. *IEEE/ACM Trans. Netw.* 24, 2 (2016), 1251–1264.

[13] Chad R. Meiners, Alex X. Liu, and Eric Torng. 2012. Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs. *IEEE/ACM Trans. Netw.* 20, 2 (2012), 488–500.

[14] Masoud Moshref, Minlan Yu, Abhishek B. Sharma, and Ramesh Govindan. 2012. vCRIB: Virtualized Rule Management in the Cloud. In *HotCloud*.

[15] Masoud Moshref, Minlan Yu, Abhishek B. Sharma, and Ramesh Govindan. 2013. Scalable Rule Management for Data Centers. In *NSDI*. 157–170.

[16] Sergey I. Nikolenko, Kirill Kogan, Gábor Rétvári, Erika R. Bérczi-Kovács, and Alexander Shalimov. 2016. How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes. In *INFOCOM Workshops*. 521–526.

[17] Eric Norige, Alex X. Liu, and Eric Torng. 2013. A Ternary Unification Framework for optimizing TCAM-based packet classification systems. In *ANCS*. 95–104.

[18] David E. Taylor and Jonathan S. Turner. 2007. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.* 15, 3 (2007), 499–511. DOI: https://doi.org/10.1109/TNET.2007.893156

[19] Rihua Wei, Yang Xu, and H. Jonathan Chao. 2012. Block permutations in Boolean Space to minimize TCAM for packet classification. In *INFOCOM*. 2561–2565.

[20] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable flow-based networking with DIFANE. In *SIGCOMM*. 351–362.

[21] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. 2006. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *S&P*. 199–213.

D Paper “Elastic capacity planning for service auto-scaling”

Authors. Pavel Chuprikov, Sergey Nikolenko, and Kirill Kogan

Abstract. Cloud computing allows on demand elastic service scaling. The capability of a service to predict resource requirements for the next operational period defines how well it will exploit the elasticity of cloud computing in order to reduce operational costs. In this work, we consider a capacity planning process for service scale-out as an online pricing model. In particular, we study the impact of buffering service requests on revenues in various settings with allocation and maintenance costs. In addition, we analyze the incurred latency implied by buffering service requests. We believe our insights will allow to significantly simplify predictions and mitigate the unknowns of future demands on resources.

On Demand Elastic Capacity Planning for Service Auto-Scaling

Pavel Chuprikov*†
*Steklov Mathematical Institute at St. Petersburg, Russia
†IMDEA Networks Insitute, Madrid, Spain

Sergey Nikolenko‡§
‡National Research University Higher School of Economics, St. Petersburg, Russia
§Steklov Mathematical Institute at St. Petersburg, Russia

Kirill Kogan
IMDEA Networks Institute, Madrid, Spain

***Abstract*—Cloud computing allows on demand elastic service scaling. The capability of a service to predict resource requirements for the next operational period defines how well it will exploit the elasticity of cloud computing in order to reduce operational costs. In this work, we consider a capacity planning process for service scale-out as an online pricing model. In particular, we study the impact of buffering service requests on revenues in various settings with allocation and maintenance costs. In addition, we analyze the incurred latency implied by buffering service requests. We believe that our insights will allow to significantly simplify predictions and mitigate the unknowns of future demands on resources.**

I. Introduction and Motivation

One of the biggest advantages of cloud service platforms such as Amazon EC2, Microsoft Azure, and IBM Smart Cloud Enterprise is the elastic usage of resources that can be translated into greater flexibility. This elasticity allows to reduce operational costs of implemented services. However, to exploit the elasticity of resource allocation, capacity planning should be as precise as possible: for instance, in scale-in/out of implemented services any significant difference between required and planned resources leads either to extra operational costs in case of over-provisioning or to missed revenues for rejected service requests in case of under-provisioning.

Since different types of resources (memory, bandwidth, processing power etc.) need to be allocated with different resolutions, we consider virtual *resource units*. To make operational settings more real, we assume that every resource unit has *maintenance* and *allocation costs*, there is an upper bound on the number of simultaneously allocated resources, every *service request* requires one virtual resource unit, and service requests are independent. We do not presume any specific distribution or pattern in the arriving stream; arrivals can be adversarial. In this environment, resource capacity planning for the next operational period becomes very complicated and interesting problem. Unlike other objectives such as average response time, latency, etc., where average case analysis is preferable, worst case guarantees become extremely important in economic settings: average case guarantees may need a long run of operation to be actually realized, and negative revenues may occur in intermediate points of operation, a loss that often cannot be afforded by service providers.

Our Contributions: In this work, we explore the effect of delaying service requests on revenues in various settings. There is a fundamental tradeoff between the ability to delay service requests in time (giving a better opportunity for capacity planning) and potentially missed opportunities to service requests if resource units available at a given time are limited. Delaying service requests may introduce additional latency, and a resource management policy can heavily depend on relations between allocation and maintenance costs. We study these effects in detail and propose efficient online policies with worst case performance guarantees. In our analytic study, we use *competitive analysis* that compares the performance of an online policy with an optimal clairvoyant offline algorithm [1]. We also supplement our theoretical results with a comprehensive simulation study.

This paper is organized as follows. In Section II, we summarize related work, covering three different existing approaches to auto-scaling. Section III introduces two discrete time models, with and without service requests deadlines, and the desired objectives. Section IV shows the simplest policies that do not have a buffer (or, equivalently, deal with urgent jobs that cannot be delayed). In Section V, we provide a comprehensive theoretical study of policies with buffers, showing a general lower bound and introducing two new online policies. Section VI shows results on the latency (both maximal and average) of the proposed online policies. In Section VII we consider an updated model, where latency constraints are embedded on the model level. The proposed policies are evaluated in Section VIII. We conclude with Section IX.

II. Related Work

There is a long prior art that considers buffer management policies in various settings [2], [3]. For a single queue architecture, throughput maximization is considered in [4], [5], [6], [7], [8]. More complex buffering architectures are considered in [9], [10], [11], [12], [13], [14]. A recent survey [15] classifies auto-scaling techniques into several major categories: static policies, threshold-based polices, reinforcement learning, control theory, and time-series analysis. This classification can be viewed in terms of three major research lines. The

first line deals with reactive mechanisms that do not try to anticipate future needs. A decision to increase or decrease capacity is based on the arrival patterns during last time slots (or values of other important variables). There is a long line of research that deals with benchmarking frameworks that assist in the evaluation of resources in cloud systems for service auto-scaling, either micro-architecturally [16], [17], [18] or at the system level [19], [20], [21], [22], [23]. Second, Amazon AWS AutoScaling and other cloud service brokers such as RightScale and Enstrautus implement rule-based auto-scaling. These mechanisms are applicable when service providers understand their resource demands. In this case service providers are responsible for defining reasonable scaling rules, and auto-scaling without explicit user intervention is hard. Naturally, there have been works that attempt to predict future resource requirements based on historical data of resource usage; usually, some machine learning model is trained on historical data and used to predict future loads, making auto-scaling decisions based on its predictions; to predict workload, researchers have used both nonparametric techniques [24], [25] and supervised learning methods such as neural networks and linear regression [26]. The works [27], [28], [29] attempt to predict patterns of service requests. Finally, one can take a hybrid path; for instance, the work [30] proposed a hybrid scaling technique where scaling up is done based on reactive rules while a supervised learning is used to scale down. [28] presented an online resource demand prediction model that achieves adaptive resource allocation. Cloud computing economics are considered in [31], [32], [33]. To represent leasing, the *parking permit problem* was introduced in [34], with [35] presenting the leasing model more generally for many infrastructure problems such as *facility location* and *set cover*. To our best knowledge, no related work has considered the economic effect of delaying service requests.

III. Model Description

We assume that time is slotted, and deploying a service for every request requires one unit of allocated resource per time slot. Note that a unit of virtual resource is defined by the implemented service (e.g., a service request has specific bandwidth requirements). Each resource unit has *allocation cost* α (including deallocation costs) and *maintenance cost* β per time slot. Each request has an intrinsic value gained by the service if the service request is serviced by an available allocated resource unit. In most cases, the behavior of an allocation process can be formulated as a pricing model, when at the end of every time slot a capacity planning process allocates resources for the next time slot to maximize revenue, given α and β costs.

Formally speaking, on every time slot t there arrives a sequence $\mathcal{J}_t$ of service requests. Each request j has an intrinsic positive value v_j, and we scale values so that $\min_j v_j = 1$. We assume that each request requires one virtual *resource unit*; virtual units can represent processing cores, virtual machines, or storage resources. Each time slot consists of three phases: (1) *admission*: new requests arrive, and the allocation policy decides whether to drop or admit each request; (2) *processing*: admitted requests are assigned to resource units for processing (at most one to each resource unit); (3) *prediction*: the allocation policy determines the *processing capacity* (number of resource units) allocated for the next time slot.

Changing capacity for time slot t costs c_t; we define c_t as a function of *additionally* allocated resource units and assume that deallocation cost is also amortized in c_t, so it is always free to decrease capacity for the next time slot. In most cases, we assume $c_t = \alpha n_t$, where n_t is the number of additional resource units. A_t is the set of requests admitted at time slot t. The goal is to maximize total revenue $\sum_t(\sum_{j\in A_t} v_j - \beta \cdot N_t - c_t)$, where N_t is the number of allocated resource units at time slot t. We call this model *Bufferless Heterogeneous values* (BLH) and consider it in Section IV. Fig. 1 shows sample timeslots of an allocation policy that predicts as many resource units as requests arrived on this timeslot; dropped requests are shaded in gradient. On the first timeslot, all requests are dropped since there are no resource units. On the second, two highest valued requests are admitted by an allocation policy; then processing capacity is overestimated, and two extra resource units are allocated for the last timeslot. Note that the revenue we just defined can be negative, so it is important to clarify the notion of α-competitiveness. Specifically, we say that an online algorithm A is α-competitive, if there is some universal constant c such that for any sequence of requests σ, A's revenue $\mathrm{A}(\sigma)$ and OPT's revenue $\mathrm{OPT}(\sigma)$ satisfy $\mathrm{A}(\sigma) + c \geq \mathrm{OPT}(\sigma)/\alpha$.

In the second model, we assume that a resource allocation system has a possibility to buffer at most δB requests, $\delta > 0$. We will see that these properties allow for efficient allocation policies and at the same time let us simplify predictions significantly. We call this model *Buffered with Heterogeneous values* (BH), with the corresponding uniform version *Buffered with Uniform values* (BU), and consider it in Section V.

Finally, in the last model we assume that each incoming request i in addition to value has a deadline $d(i)$ ($d(i) \geq 1$). A request i that arrived at timeslot t will be automatically dropped at the end of timeslot $t + d(i)$. In this model the latency constraints are embedded and not a property of specific algorithms as before. We call this model *Bounded Delay* (BD) and consider it in Section VII.

IV. Bufferless Allocation

In the *bufferless* model BLH, it is impossible to postpone servicing, so the maximal number of serviced requests equals the number of resource units allocated at the end of the previous time slot. The next theorem demonstrates that in the BLH model, online algorithms are non-competitive.

Theorem 1. *If $\alpha < 1 - \beta$, then in the* BLH *model any deterministic online allocation policy is non-competitive.*

Proof. Consider an arbitrary online deterministic algorithm A. Assume that A allocates m resource units at the first time slot. Then km service requests arrive at the first time slot, and A's

revenue is at most $m(1-\alpha-\beta)$, while OPT's is $km(1-\alpha-\beta)$ if it allocates km resource units for the first time slot, and the sequence can be repeated. Thus, the competitive ratio is $\geq k$ for an arbitrary k. □

To avoid the rather trivial obstacle of Theorem 1, we could try to impose a natural constraint: suppose that the maximal number of resource units that can be allocated at a given time slot is bounded by B, which is also an input to the algorithm. We call the resulting model *Bounded Bufferless with Heterogeneous Values* (BBLH). Unfortunately, even under the BBLH constraint a similar adversarial sequence still works.

Theorem 2. *If $\alpha < 1-\beta$, then in the* BBLH *model any deterministic online allocation policy is non-competitive.*

Proof. Consider an arbitrary online deterministic algorithm A. Let $k = \sqrt{B\beta}$, i.e., $B = k^2/\beta$. On the first time slot, if A allocates $\geq k/\beta$ resource units and no requests arrive, A spends $\geq \beta k/\beta = k$, while OPT allocates nothing and pays zero. Otherwise, if A allocates less than k/β resource units and there arrive B requests, its revenue is at most $k/\beta(1-\alpha-\beta)$, while if OPT allocates B resource units, its revenue is $B(1-\alpha-\beta)$, and the competitive ratio is at least k, which is arbitrary large. □

Note that the maximal number of allocated resource units is expected to be high because higher number of resource units means, in general, more requests serviced and larger revenue. Taking B as an input to the algorithm, we are able to provide guarantees independent of this large number.

V. Buffered Allocation

A. General lower bound

In the previous section, we have shown that without a buffer we get high competitive ratios even with limited number of allocated resource units. Unfortunately, the BU model still does not allow us to build optimal online algorithms.

Theorem 3. *If $\alpha < 1-\beta$, then in the* BU *model every online algorithm is at least $1+\frac{1}{\delta}\left(1-\frac{\alpha}{1-\beta}\right)$-competitive.*

Proof. First, note that no competitive deterministic online algorithm A allocates resource units if no requests have arrived, or at least does so finitely many times, because otherwise it will process the empty input sequence with an arbitrarily low (negative) objective function. Let t_0 be the first time slot when A allocates no resource units, with no requests arriving before t_0. Then consider the following sequence: at time slot t_0 there arrive $(1+\delta)B$ unit-valued requests. OPT predicts B machines at time slot t_0-1 and thus immediately runs B requests at time slot t_0; the δB requests are stored in the buffer to service them on the next time slot. Now OPT's revenue is at least $B(1+\delta)(1-\beta)-B\alpha$. On the other hand, A is able to service at most δB requests, and its revenue is no more than $\delta B(1-\beta)$. Thus, the resulting competitive ratio is $\frac{B(1+\delta)(1-\beta)-B\alpha}{\delta B(1-\beta)} = 1+\frac{1}{\delta}\left(1-\frac{\alpha}{1-\beta}\right)$, as needed. □

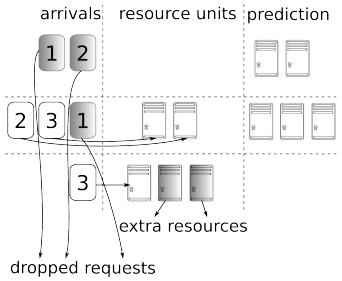


Fig. 1. Sample operation of a bufferless policy that predicts as many resources as there have been requests on the last time slot.

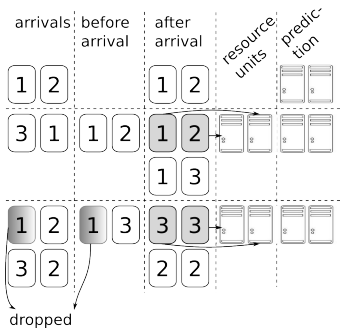


Fig. 2. Sample operation of the NRAP algorithm with buffer size 4.

B. Next round allocation policy

Now we present the first of the two allocation policies that we study in this work in detail. This policy, NRAP, simply predicts as many resource units for the next time slot as there are service requests currently residing in the buffer.

Definition V.1. *The* Next Round Allocation Policy *(*NRAP*):*

- *accepts incoming requests are as long as there is room in the buffer;*
- *pushes out the lowest valued request in the buffer by a higher valued arrival;*
- *predicts k resource units for the next time slot if there are k requests in the buffer by the end of a time slot;*
- *services either requests from the previous timeslot or those that pushed them out.*

Fig. 2 shows sample NRAP operation over three time slots with buffer size 4; gradient shading shows dropped service requests; gray, requests that will be processed on the current time slot. Note that while NRAP by definition does not spend extra resources, it can drop many incoming requests and spend a lot on (de)allocation. On the other hand, NRAP is simple, has low memory requirements, and, as the next theorem shows, has constant competitive ratio for some values of β and α.

Theorem 4. *If $\alpha < 1-\beta$, then* NRAP *is at most $2\left(1+\frac{\alpha}{1-\alpha-\beta}\right)$-competitive.*

Proof. For any incoming sequence, we map every request serviced by OPT to a request serviced by NRAP in such a way that the value of the image is no less than the value

of its preimages, and at most two requests accepted by OPT are mapped to every request serviced by NRAP. OPT never pushes requests out, so "accepted by OPT" is equivalent to "serviced by OPT". To see that this mapping will imply the necessary bound, note that for each request of value v accepted (and later serviced) by OPT, OPT gets revenue at most $v-\beta$. By the mapping, there also exists a request with value $v' \geq v$ serviced by NRAP with revenue at least $v'-\beta-\alpha \geq v-\beta-\alpha$. Half of this revenue corresponds to j. Hence, for each request we have the ratio of $\frac{v-\beta}{2(v-\beta-\alpha)} \leq \left(1+\frac{\alpha}{1-\beta-\alpha}\right)/2$.

Next we show the mapping itself. It preserves the following invariant: the preimage of each request in the buffer at the beginning of a timeslot has size one. Consider the buffer state after arrival. Due to push-out, the ith highest valued request in NRAP's buffer has value no less than the ith highest valued request admitted by OPT, and we map these requests one to one. Some requests in NRAP's buffer that had another request mapped to them before arrival could be pushed out; in this case, we map their preimages to those requests that pushed them out. Requests with preimage of size two are exactly those that will be processed in the processing phase. □

This is an impressive result, given the simplicity of the policy, but, unfortunately, this upper bound can be arbitrarily high in case $\alpha \approx 1$ and $\beta \approx 0$, because NRAP has to pay allocation cost for each processed request and thus cannot profit from it.

Theorem 5. *Assume that $\alpha < 1-\beta$. Then in the BU model* NRAP *is at least $\left(2+\frac{\alpha}{1-\alpha-\beta}\right)$-competitive.*

Proof. Consider the following sequence: on the first and second time slots, there arrive B requests. NRAP drops the second burst of B requests, with total revenue $B(1-\beta-\alpha)$. OPT could predict B resource units for both timeslots with total revenue $2B(1-\beta)-B\alpha$, getting the bound. □

Although NRAP is nearly optimal when the arrival pattern is uniform and has acceptable competitiveness when allocation cost is low, the policy still has a major drawback: it always allocates a resource unit to service a single request and never reuses computational resources. This makes NRAP lose its competitiveness guarantees when allocation cost is high.

C. Amortizing Allocation Policy

Here we propose a different strategy that tries to avoid NRAP's limitations by amortizing allocation costs across several requests, performing well for high allocation costs. We denote by $\mathrm{PreAssigned}(v)$ the subset of requests in the buffer preassigned to an allocated resource unit v.

Definition V.2. *Upon arrival of k requests, the* Amortizing Allocation Policy *ρ-AAP with parameter $\rho > 1$:*

- *accepts incoming requests as long as the remaining buffer capacity plus the number of allocated resource units is more than $(B-\lceil\frac{\rho\alpha}{1-\beta}\rceil)/(\lceil\frac{\rho\alpha}{1-\beta}\rceil+1)-\lceil\frac{\rho\alpha}{1-\beta}\rceil$ (we assume it to be a positive integer for simplicity); only requests accepted on the current time slot can be pushed out.*

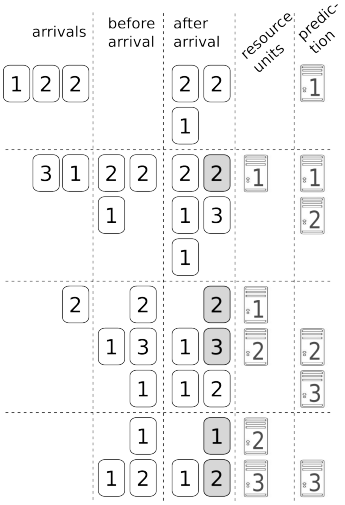


Fig. 3. Sample operation of the ρ-AAP algorithm.

- *while the total value of non-preassigned requests residing in the buffer (those that do not belong to any $\mathrm{PreAssigned}(v)$) is at least $\lceil\frac{\rho\alpha}{1-\beta}\rceil$, repeats:*
  - *predict an additional resource unit v' to be needed for the next time slot;*
  - *move as few as possible non-preassigned requests with total value at least $\lceil\frac{\rho\alpha}{1-\beta}\rceil$ to $\mathrm{PreAssigned}(v')$;*
- *then predicts for each resource unit v such that $\mathrm{PreAssigned}(v) \neq \emptyset$ ρ-AAP that it will need v for the next time slot; every allocated resource unit with $\mathrm{PreAssigned}(v) = \emptyset$ is deallocated.*

Figure 3 shows the sample operation of the ρ-AAP algorithm for buffer size 7 with $\lceil\frac{\rho\alpha}{1-\beta}\rceil = 2$. In all columns except the first, each row represents either an allocated resource unit v (shown with its index in the last two columns) together with its corresponding $\mathrm{PreAssigned}(v)$ (second and third columns) or a set of non-preassigned requests. In the first timeslot, three requests arrive, two of them get a resource unit, and the last one remains non-preassigned (since the AAP threshold is 2 in this example). On the last timeslot, resource unit 1 has processed its queue entirely and is deallocated. The next theorem provides an upper bound for the ρ-AAP competitiveness; note how α no longer appears in the denominator.

Theorem 6. *For $\alpha \geq (1-\beta)$, ρ-AAP with an increased buffer of size $B+\lceil\frac{\rho\alpha}{1-\beta}\rceil$ has competitive ratio at most $\frac{\rho}{\rho-1}\left(\left\lceil\frac{\rho\alpha}{1-\beta}\right\rceil+1\right)$ against the optimal algorithm* OPT *with a buffer of size B.*

Proof. We denote $c = \lceil\frac{\rho\alpha}{1-\beta}\rceil$. Note that requests are preassigned to a resource unit in batches, with total value of a batch at least c. We get at least $c-\beta n-\alpha$ from servicing those requests, where n the number of requests in a batch. Note that $n \leq c$, so revenue per value unit is at least $\frac{\rho-1}{\rho}(1-\beta)$.

Next, assume that the number of preassigned requests is m. Then the servicing capacity is at least m/c, and there may

be admitted at least $B\frac{\lceil \frac{\rho\alpha}{1-\beta} \rceil}{\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1} - m\left(1 - \frac{1}{c}\right)$ requests. Since $m \leq B\lceil \frac{\rho\alpha}{1-\beta} \rceil / (\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$ (by the first step of the algorithm), we have a lower bound on this value as at least $B/(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$. This means that ρ-AAP takes at least $1/(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$ fraction of total value admitted by OPT due to push-outs. Suppose now that OPT accepts and services L requests in total. Obviously, its revenue is at most $L(1-\beta)$. Now we can bound from below the number of requests serviced by δ-AAP as $L/(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$, and the above remark gives us ρ-AAP's revenue as at least $\frac{L(\rho-1)(1-\beta)}{\rho(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)}$. This results in an upper bound on the competitive ratio as $\frac{\rho}{\rho-1}(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$. Since there may be some "tail" requests never serviced by ρ-AAP (less than c of them), we have non-strict competitiveness. □

We finish this section with a matching lower bound for the ρ-AAP (up to a factor of $\rho/(\rho-1)$).

Theorem 7. *In the* BU *model the ρ-AAP algorithm with an increased buffer of size $B + \lceil \frac{\rho\alpha}{1-\beta} \rceil$ is at least $(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$-competitive against* OPT *with a buffer of size B.*

Proof. Consider a long sequence of B requests per time slot, which OPT services with B resource units. Since ρ-AAP never predicts more than $B/(\lceil \frac{\rho\alpha}{1-\beta} \rceil + 1)$ resource units, the competitive ratio follows. □

VI. Latency considerations

A. Worst-case analysis

For a given service request i accepted at time slot t_a and processed at time slot t_p, its *latency* is $\mathrm{lat}(i) = t_p - t_a$. The latency of an algorithm A is defined as $\mathrm{lat}(A) = \sup_{\mathcal{J}} \max_i \mathrm{lat}(i)$, where $\mathcal{J}$ is an input sequence and i is any request from $\mathcal{J}$ serviced by A. If no requests have arrived we set $\max_i \mathrm{lat}(i) = 0$. The goal is to minimize $\mathrm{lat}(A)$ while keeping the competitive ratio low. First, note that NRAP is very good when it comes to latency: every accepted request is immediately processed or pushed out in the next time slot.

Theorem 8. *In the* BH *model* $\mathrm{lat}(\mathrm{NRAP}) = 1$.

Unfortunately, since ρ-AAP waits to process packets, its latency becomes arbitrarily high in the general BH model, so we make an additional assumption that at least one request arrives on each time slot in the sequence from first to last; previous theorems obviously hold in this case. Under this restriction, we modify ρ-AAP to minimize maximal latency.

Definition VI.1. *The algorithm ρ-AAPm (modified ρ-AAP) behaves exactly like ρ-AAPm, but if no requests arrive at a given time slot then ρ-AAPm immediately drops all non-preassigned packets from its buffer.*

This modification lets us compute the latency of ρ-AAP.

Theorem 9. *If at least one request arrives on each time slot,* $\mathrm{lat}(\rho\text{-}AAPm) = \left\lceil \frac{\rho\alpha}{1-\beta} \right\rceil$.

Proof. Assume that before a request arrives there are at least k non-preassigned requests. Then we need to wait at most $\lceil \frac{\rho\alpha}{1-\beta} \rceil - k - 1$ time slots before enough requests are collected and a resource unit is allocated (or the request sequence ends). Next, due to FIFO order, this request waits for $k+1$ time slots until it is finally processed. Adding up these delays, we get the bound. It is easy to check that if exactly one request arrives per time slot then the bound is realized. □

B. Average-case analysis

While it is important to have worst-case guarantees for processing latency, the average case is also important, providing better guarantees for most requests while perhaps tolerating high latency for a small fraction. For an algorithm A, we define the average latency as $\mathrm{lat}_{\mathrm{avg}}(\mathrm{A}) = \sup_{\mathcal{J}} \frac{\sum_{i \in \mathcal{J}} \mathrm{lat}(i)}{|\mathcal{J}|}$, where $\mathcal{J}$ is an incoming sequence of requests.

NRAP is still straightforward to analyze, and we can also bound the average latency for ρ-AAPm.

Theorem 10. *In the* BH *model,* $\mathrm{lat}_{\mathrm{avg}}(\mathrm{NRAP}) = 1$.

Theorem 11. *If at least one request arrives on each time slot,* $\mathrm{lat}_{\mathrm{avg}}(\rho\text{-}AAPm) \geq \frac{1}{2}\lceil \frac{\rho\alpha}{(1-\beta)} \rceil + \frac{1}{2}$.

Proof. Consider $\lceil \frac{\rho\alpha}{1-\beta} \rceil$ service requests assigned to the same resource unit. If a request j is the ith to be processed among them, then $\mathrm{lat}(j) \geq i$. Summing over i and dividing by $\lceil \frac{\rho\alpha}{1-\beta} \rceil$, we get average latency $\frac{1+2+\ldots+\lceil \frac{\rho\alpha}{1-\beta} \rceil}{\lceil \frac{\rho\alpha}{1-\beta} \rceil} = \frac{1}{2}\left\lceil \frac{\rho\alpha}{1-\beta} \right\rceil + \frac{1}{2}$. It remains to note that the average over a set is no less than the minimum average among a partition of this set. □

VII. Bounded-delay allocation

In the previous section, incoming requests had only a value characteristic, and the incurred latency was a property of specific algorithms. In this case we were able to estimate upper bounds for maximum and average delays that a request may experience. Here, we consider the Bounded Delay model, where the latency constraints are embedded on the model level: packets have both value and allowed delay.

A. General lower bound

Our first result shows that this model is very different from BH and BU: it is now impossible to find an online algorithm with good competitiveness if $\alpha + \beta > 1$ (in the BH model ρ-AAP had an upper bound on competitiveness independent of B).

Theorem 12. *If $\alpha + \beta > 1$, then any online algorithm is non-competitive in the BD model. The statement still holds even if all values and delays are equal to one.*

Proof. Assume that we have some c-competitive online algorithm A. Since A is competitive then there is some lower bound l (possibly negative) on its revenue. We will build an adversarial sequence of arrivals in such a way, that A's revenue is non-positive, while OPT's revenue can be arbitrary high.

Now, if A has no resource units allocated for a timeslot then there arrives a single request i with $d(i) = 1$ and $v(i) = 1$, otherwise no requests arrive. First, note that every resource unit allocated by A processes at most one request

(the one that arrived on the timeslot immediately before its allocation). Since $1-\alpha-\beta<0$, it means that every time A allocates a resource unit it pays, and since A's revenue is bounded from below, A allocates only a finite number of resources units. Hence, after some timeslot T the sequence has a request arriving on each time slot. Requests arriving after T may be processed by OPT with a single resource unit, getting an arbitrarily high revenue (allocation cost α becomes negligible). Since A's revenue is non-positive and OPT's may be arbitrarily high, A is in fact non-competitive. □

Having proved the theorem, in the following we assume $\alpha+\beta<1$. Note that the lower bound implies that there is no way to get more than a constant factor advantage of processing multiple requests by a single request unit.

B. Unbounded resources

Again, we begin with a default setting with no upper bound on the number of allocated resources units (BD model). Again, we are able to show a general lower bound.

Theorem 13. *If $\alpha+\beta<1$, then in the* BD *model every online algorithm is at least $\left(1+\frac{\alpha}{2(1-\alpha-\beta)}\right)$-competitive.*

Proof. Consider an arbitrary competitive online algorithm A; again, if no packets arrive then there is some timeslot t for which A has not allocated any resource units. Then the adversarial sequence has no requests until t and then on timeslot t two requests with unit delay and value arrive. A clearly is unable to gain more than $2(1-\beta-\alpha)$, since after timeslot $t+1$ all requests will be dropped, and at time t A has no resource units. OPT, in turn, can predict a single resource unit on timeslot $(t-1)$ and process both requests on this resource unit, getting revenue $(2-2\beta-\alpha)$, as needed. □

On the positive side, NRAP can be adapted for the BD settings in such a way that its competitiveness is at most a constant factor away from the global lower bound. To achieve this, we assume that NRAP: (1) predicts the number of allocated resource units for the next timeslot equal to the number of requests arrived on this timeslot, and (2) processes exactly those requests that arrived on the previous timeslot.

Theorem 14. *If $\alpha+\beta<1$, then in the* BD *model* NRAP *is at most $(1+\frac{\alpha}{1-\beta-\alpha})$-competitive.*

Proof. Every arriving request will be processed by NRAP on the next time slot with a revenue at least $(1-\beta-\alpha)$. If the total number of arrived requests is L, then NRAP's total revenue is no less than $L(1-\beta-\alpha)$, while OPT's revenue is clearly no more than $L(1-\beta)$. □

C. Bounded resources

We now bound the number of simultaneously allocated resource units by B and call it the *Limited Bounded Delay* (LBD) model. Interestingly, in this case competitiveness becomes provably worse. The problem is that with limited resources available for allocation over a given time period, if an algorithm fails to predict the needed resource units at the start of the period, it cannot compensate by allocating more in the future; the optimal algorithm always predicts correctly.

Theorem 15. *If $\alpha+\beta<1$, then in the* LBD *model any online algorithm is at least $(2+\frac{\alpha}{1-\beta-\alpha})$-competitive.*

Proof. Again, for a competitive algorithm Athere exists a time slot t when it allocates no resources. At time t, there arrive $2B$ requests with unit delay and value. Again, similar to Theorem 13, A is unable to process more than B requests on the next time slot (the maximal number of resource units is B, so A's total revenue is at most $B(1-\beta-\alpha)$. OPT predicts B resource units for timeslot t and processes $2B$ requests by the end of timeslot $t+1$ with revenue $2B(1-\beta-\alpha/2)$. □

NRAP's performance remains close to the absolute bound.

Theorem 16. *If $\alpha+\beta<1$, then in the* LBD *model* NRAP *is at most $3(1+\frac{\alpha+\beta}{1-\beta-\alpha})$-competitive.*

Proof. Consider an arbitrary sequnce of requests σ. Let O be the set of requests processed by OPT; N, by NRAP. To prove the bound we map requests from $O\setminus N$ to requests in N in such a way that the preimage of every request $i\in N$ is of size at most 2 and the value of each request in the preimage is at most $v(i)$. If we can construct this mapping, the claim follows since $\sum_{i\in O\setminus N}v(i)\le 2\sum_{i\in N}v(i)$, so $\sum_{i\in O}v(i)\le 3\sum_{i\in N}v(i)$. Because OPT pays at least β for each processed request and NRAP pays at most $\beta+\alpha$, the competitive ratio is at most

$$\frac{3\sigma-\beta|O|}{\sigma-(\alpha+\beta)|I|}\le 3+\frac{\beta(3|I|-|O|)+3|I|\alpha}{\sigma-(\alpha+\beta)|I|}\le 3+\frac{3(\beta+\alpha)}{1-\alpha-\beta}$$

since $\sigma\ge|I|$, where $\sigma=\sum_{i\in I}v(i)$.

For the mapping, we subdivide $O\setminus N$ into two disjoint subsets: D with requests served on the time slot when their deadline expires and all other requests $O\setminus D$. We will map each subset injectively to N. Assume that a request $i\in O\setminus D$ is processed by OPT on timeslot t; then it remains alive on timeslot $t+1$ because t is not i's deadline. We map i to any request i' processed by NRAP on timeslot $t+1$ that has no requests mapped to it; it is possible because at least as many resource units are allocated for timeslot $t+1$ as requests from $O\setminus D$ processed on timeslot t. Moreover, $v(i')\ge v(i)$ because otherwise i would be processed instead of i'. Next, assume that a request $j\in D$ is processed by OPT on timeslot t. Since j's deadline is t, it was alive at time $t-1$, and following the same logic we map it to a request processed by NRAP on timeslot t. Since mappings from both D and $O\setminus D$ are injective, in their combination every preimage is of size at most 2. □

Theorem 17. *If $\alpha+\beta<1$, then in the* LBD *model* NRAP *is at least* 3*-competitive.*

Proof. As always, we present an adversarial sequence. Fix $V>1$; on timeslot $t=0$, there arrive $2B$ requests with delay 1 and value $(V-1)$ and B requests with delay 2 and value V. It is easy to see that NRAP processes B requests with value V (on timeslot $t=1$), while OPT is able to process all $3B$ requests, getting competitive ratio $3-1/V$ for arbitrary V. □

VIII. Simulations

A. Experimental Setting

To validate our theoretical results, we have conducted an extensive simulation study. It would be valuable to test the policies on real life traces, but unfortunately we have no access to real life datasets for resource demands. In addition, interesting interrelations between delaying service requests and allocation/maintenance costs are difficult to cover. Therefore, we have decided to test our algorithms on synthetic traces in a series of four experiments; two of them study how the objective function (total transmitted value less the cost of resource unit maintenance and allocation) depends on buffer size B and resource allocation cost α in the BH model, the third considers average latency among processed service requests, and the fourth adds delays in the BD model.

Note that the actual optimal algorithm in our model has prohibitive computational cost even if we assume that the optimal algorithm is clairvoyant: for $\alpha > 0$ it is a hard optimization problem to find the optimal sequence of (de)allocations. Therefore, as OPT we simply use the maximal objective function that can be generated, i.e., a clairvoyant algorithm that exactly predicts the need for capacity at the next turn and, moreover, gets to increase its capacity for free. Apart from OPT, NRAP, and three different versions of ρ-AAP for $\rho = 1.5$, 2.0, and 2.5, we have also added for comparison three other algorithms: Const5 simply always allocates 5 resource units, while Avg50 and Median50 use as their prediction the average and median value of buffer occupancy over the last 50 time slots (a value chosen by experimental cross-validation).

To generate traffic, we have, again lacking real life information regarding job distribution, followed the ideas of network traffic simulation for simulating incoming jobs. Network traffic has a long-tail distribution due to its bursty nature; its has long-range dependencies [36] that are not always perfectly captured by Poisson models [37]. Mathematical models for simulating long-tail traffic include Markov-modulated Poisson processes [38], [39] and fractional Brownian motion [40]; here, we use the Poisson Pareto burst process (PPBP), a recent successful traffic model [41], [42]. In PPBP, traffic is modeled with multiple overlapping bursts whose lengths conform to a Pareto (long-tail) distribution. We use PPBP to model the stream of incoming jobs, drawing each job's value from a Zipf (discrete inverse polynomial) distribution. We expect Avg50 and Median50 to perform well on average since the incoming stream distribution does not change during simulations, and they can learn the true mean quickly.

B. Objective function in the BH model

Figure 4 shows the results of our experiments for variable B; in the figure, the top row (Fig. 4, (1)–(3)) deals with the case of uniform values, and the bottom row (Fig. 4, (4)–(6)) adds values into consideration. The figures show that in our experiments, the proposed algorithms, NRAP and ρ-AAP, start by losing to the benchmark algorithms Avg50 and Median50 for small values of B but then overcome them as B grows, approaching 1 much faster than the algorithms whose predictions are based on average values. This effect is clearly visible for both uniform and variable values (Fig. 4, (1) and (4)), and it becomes even more pronounced as the input stream intensity increases, increasing congestion (Fig. 4, (3) and (6)). However, for larger values of the allocation cost α (Fig. 4, (2) and (5), where $\alpha = 1.0$) the proposed algorithms lose to the benchmark ones. For NRAP, this is simply due to the fact that it allocates and deallocates resources very unevenly on every time slot, spending a lot on new allocations. For ρ-AAP, this is due to the fact that as α grows, the algorithm becomes more cautious, its allocation threshold grows, and therefore it allocates fewer cores out of the available set. We explore this effect in more detail in the next set of simulations.

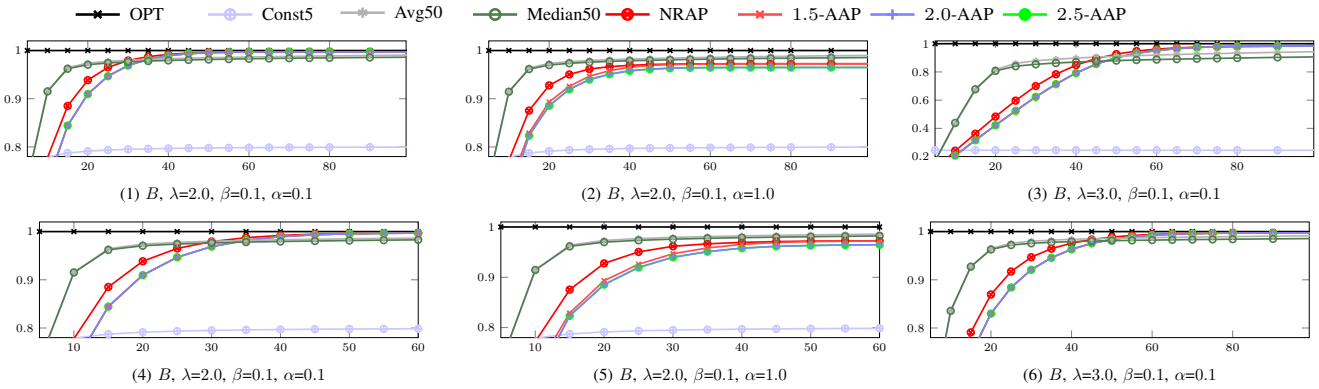
In the second set of simulations (Fig. 5), we studied how the objective function changes with α: uniform values in the top row (Fig. 5, (1)-(3)) and variable values in the bottom row (Fig. 5, (4)-(6)). We see the same effect: both NRAP and ρ-AAP begin to lose to Avg and Median as α grows; for extreme values of α they even drop below the simplest benchmark algorithm, Const5.

C. Latency in the BH model

Finally, in the last set of simulations (shown on Figure 6) we study the average latency among processed service requests. There is no need to distinguish between uniform and heterogeneous values here since the value does not influence our objective function. The top row of graphs (Fig. 6, (1)-(3)) shows how latency changes as B grows. We see that the average latency of Const, Avg, and Median grows virtually linearly with B. As expected for small α, the average latency of ρ-AAP for all three considered values of ρ is virtually indistinguishable from 1, so for larger B the latency of Avg and Median grows significantly larger than that of ρ-AAP. The most interesting behaviour is exhibited by NARP: at first its latency decreases, reflecting the increasing processing power, but then it begins to increase due to the increased usage of the buffer. Still, its latency remains safely below the average latency of all other considered algorithms and approaches 1 from below as B grows. The bottom row of graphs (Fig. 5, (4)-(6)) shows the average latency as α grows. As expected, no algorithms except ρ-AAP care for α at all, and ρ-AAP's latency begins to grow for $\alpha > 1$ and further due to the effect explained above.

D. The BD model

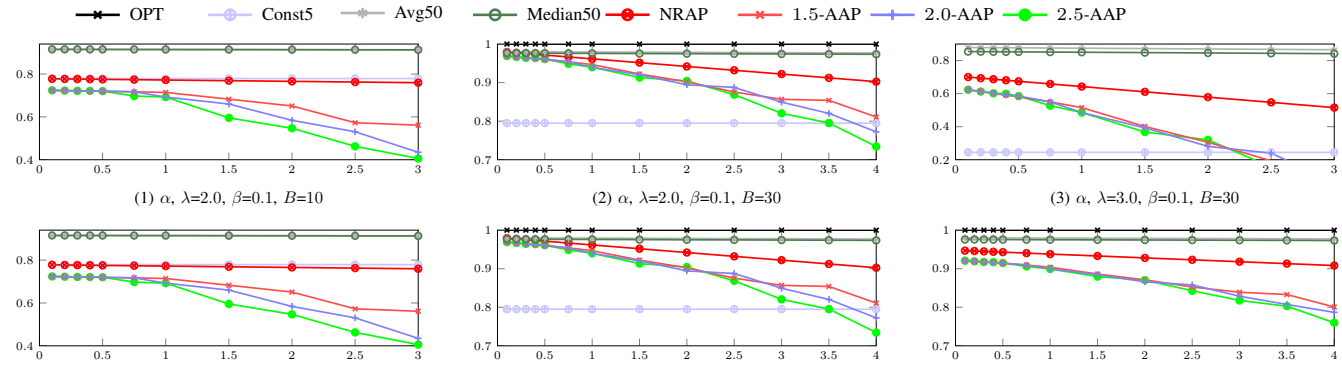
Figure 7 shows how the algorithms behave in the BD model, with delays chosen according to a Zipf distribution. Note that in this model, a packet has two characteristics so the priority queue which is at the heart of every policy can come with two different priorities, sorting packets either first by slack and then by values (denoted with suffix S) or first by value and then by slack (denoted with suffix V). The results are very similar to the BH model; note, however, that algorithms that sort first by value consistently and significantly outperform their slack-preferring counterparts.



(1) B, λ=2.0, β=0.1, α=0.1 (2) B, λ=2.0, β=0.1, α=1.0 (3) B, λ=3.0, β=0.1, α=0.1

(4) B, λ=2.0, β=0.1, α=0.1 (5) B, λ=2.0, β=0.1, α=1.0 (6) B, λ=3.0, β=0.1, α=0.1

Fig. 4. Simulation results in the BH model with variable B. Top row: uniform values; bottom row: variable values.



(1) α, λ=2.0, β=0.1, B=10 (2) α, λ=2.0, β=0.1, B=30 (3) α, λ=3.0, β=0.1, B=30

(4) α, λ=2.0, β=0.1, B=10 (5) α, λ=2.0, β=0.1, B=30 (6) α, λ=3.0, β=0.1, B=30

Fig. 5. Simulation results in the BH model with variable α. Top row: uniform values; bottom row: variable values.

IX. Conclusion

Service providers operational environments introduce a fundamental tradeoff: between resource over-provisioning and missed service requests. In this work, we have studied the effects of delaying service requests to maximize revenues of implemented services with competitive analysis, i.e., in an adversarial setting without any assumptions on the arrival distributions. In addition, we have analyzed the incurred latency as a result of buffering service requests. We believe that this study can help to identify desired properties of resource allocation policies that optimize revenue in economic constraints.

Acknowledgements: Th work of Pavel Chuprikov and Sergey Nikolenko was partially supported by the Government of the Russian Federation (grant 14.Z50.31.0030) and President grant for Young Ph.D.'s MK-7287.2016.1.

References

[1] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. Cambridge University Press, 1998.

[2] M. Goldwasser, "A survey of buffer management policies for packet switches," *SIGACT News*, vol. 41, no. 1, pp. 100–128, 2010.

[3] S. Nikolenko and K. Kogan, "Single and multiple buffer processing," in *Encyclopedia of Algorithms*, 2015.

[4] P. Chuprikov, S. Nikolenko, and K. Kogan, "Priority queueing with multiple packet characteristics," in *INFOCOM*, 2015, pp. 1418–1426.

[5] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," *Trans. Netw.*, vol. 20, no. 6, pp. 1895–1909, 2012.

[6] K. Kogan, A. López-Ortiz, S. Nikolenko, A. Sirotkin, and D. Tugaryov, "FIFO queueing policies for packets with heterogeneous processing," in *MedAlg*, 2012, pp. 248–260.

[7] K. Kogan, A. López-Ortiz, S. Nikolenko, and A. Sirotkin, "A taxonomy of semi-fifo policies," in *IPCCC*, 2012, pp. 295–304.

[8] K. Kogan, A. López-Ortiz, S. Nikolenko, G. Scalosub, and M. Segal, "Balancing work and size with bounded buffers," in *COMSNETS*, 2014, pp. 1–8.

[9] A. Kesselman, K. Kogan, and M. Segal, "Packet mode and qos algorithms for buffered crossbar switches with FIFO queuing," *Distributed Computing*, vol. 23, no. 3, pp. 163–175, 2010.

[10] ——, "Improved competitive performance bounds for CIOQ switches," *Algorithmica*, vol. 63, no. 1-2, pp. 411–424, 2012.

[11] ——, "Best effort and priority queuing policies for buffered crossbar switches," *Chicago J. Theor. Comput. Sci.*, vol. 2012, 2012.

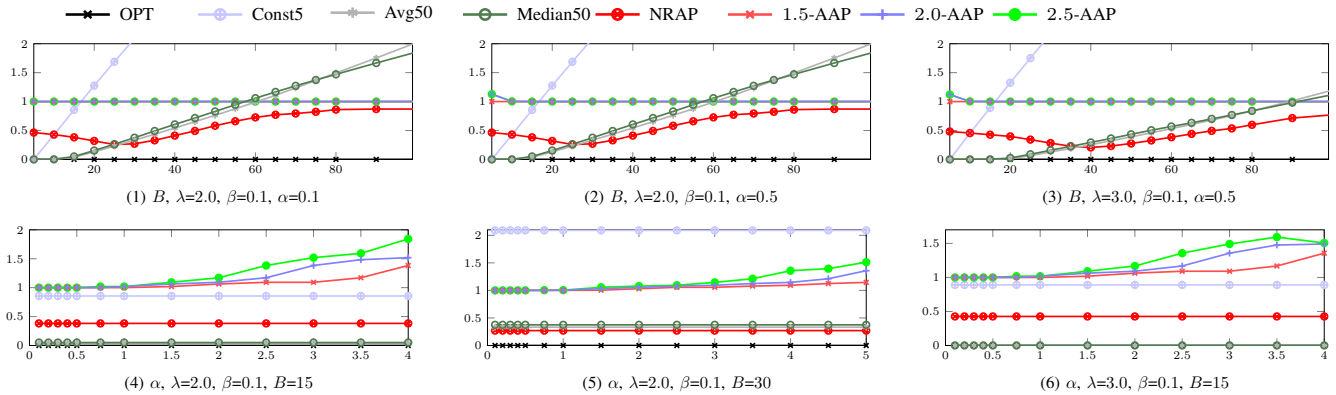
[12] K. Kogan, A. López-Ortiz, S. Nikolenko, and A. Sirotkin, "Multi-queued network processors for packets with heterogeneous processing requirements," in *COMSNETS*, 2013, pp. 1–10.

[13] P. Eugster, K. Kogan, S. Nikolenko, and A. Sirotkin, "Shared memory buffer management for heterogeneous packet processing," in *ICDCS*, 2014, pp. 471–480.

[14] P. Eugster, A. Kesselman, K. Kogan, S. Nikolenko, and A. Sirotkin, "Essential traffic parameters for shared memory switch performance," in *SIROCCO*, 2015, pp. 61–75.

[15] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

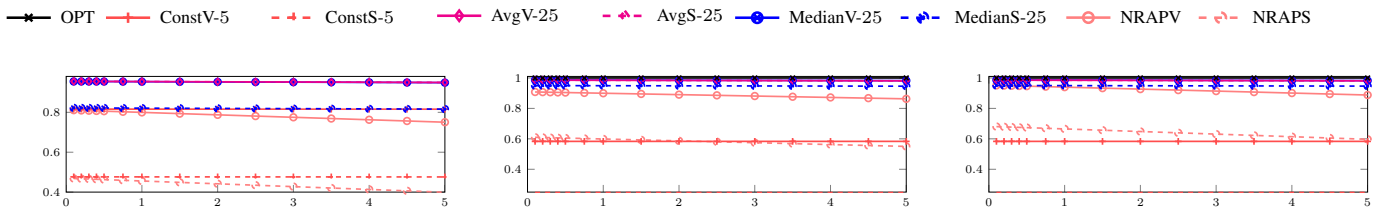
[16] M. Ferdman and et al., "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012, pp. 37–48.



(1) B, λ=2.0, β=0.1, α=0.1 (2) B, λ=2.0, β=0.1, α=0.5 (3) B, λ=3.0, β=0.1, α=0.5

(4) α, λ=2.0, β=0.1, B=15 (5) α, λ=2.0, β=0.1, B=30 (6) α, λ=3.0, β=0.1, B=15

Fig. 6. Simulation results in the BH model, average latency among processed service requests.



(1) α, λ=1.0, β=0.1, B=10 (2) α, λ=5.0, β=0.1, B=30 (3) α, λ=5.0, β=0.1, B=50

(4) B, λ=15.0, β=0.1, α=0.5 (5) B, λ=15.0, β=0.1, α=0.75 (6) B, λ=15.0, β=0.1, α=1.0

Fig. 7. Simulation results in the BD model with deadlines.

[17] V. Reddi and et al., "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *ISCA*, 2010, pp. 314–325.

[18] L. Tang and et al., "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011, pp. 283–294.

[19] B. Cooper and et al., "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010, pp. 143–154.

[20] C. Kozyrakis and et al., "Server engineering insights for large-scale online services," *Micro*, vol. 30, no. 4, pp. 8–19, 2010.

[21] A. Li and et al., "Cloudcmp: comparing public cloud providers," in *SIGCOMM*, 2010, pp. 1–14.

[22] A. Li, X. Yang, and M. Zhang, "Cloudcmp: Shopping for a cloud made easy," in *HotCloud*, 2010.

[23] V. Soundararajan and J. Anderson, "The impact of management operations on the virtualized datacenter," in *ISCA*, 2010, pp. 326–337.

[24] R. Han and et al., "Lightweight resource scaling for cloud applications," in *CCGrid*, 2012, pp. 644–651.

[25] H. Goudarzi, M. Ghasemazar, and M. Pedram, "Sla-based optimization of power and migration cost in cloud computing," in *CCGrid*, 2012, pp. 172–179.

[26] S. Islam and et al., "Empirical prediction models for adaptive resource provisioning in the cloud," *FGCS*, vol. 28, no. 1, pp. 155–162, 2012.

[27] Z. Gong, X. Gu, and J. Wilkes, "PRESS: predictive elastic resource scaling for cloud systems," in *CNSM*, 2010, pp. 9–16.

[28] Z. Shen and et al., "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *SOCC*, 2011, p. 5.

[29] E. Caron, F. Desprez, and A. Muresan, "Pattern matching based forecast of non-periodic repetitive behavior for cloud clients," *J. Grid Comput.*, vol. 9, no. 1, pp. 49–64, 2011.

[30] W. Iqbal, M. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *FGCS*, vol. 27, no. 6, pp. 871–879, 2011.

[31] D. Dash, V. Kantere, and A. Ailamaki, "An economic model for self-tuned cloud caching," in *ICDE*, 2009, pp. 1687–1693.

[32] R. Pal and P. Hui, "Economic models for cloud service markets," in *ICDCN*, 2012, pp. 382–396.

[33] M. Armbrust and et al., "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[34] A. Meyerson, "The parking permit problem," in *FOCS*, 2005, pp. 274–284.

[35] B. Anthony and A. Gupta, "Infrastructure leasing problems," in *IPCO*, 2007, pp. 424–438.

[36] Z. Xiaoyun, Y. J. Yu, and J. Doyle, "Heavy tails, generalized coding, and optimal web layout," in *INFOCOM*, vol. 3, 2001, pp. 1617–1626 vol.3.

[37] V. Paxson and S. Floyd, "Wide area traffic: The failure of poisson modeling," *Trans. Netw.*, vol. 3, no. 3, pp. 226–244, Jun. 1995.

[38] W. Fischer and K. Meier-Hellstern, "The markov-modulated poisson process (mmpp) cookbook," *PER*, vol. 18, no. 2, pp. 149–171, 1993.

[39] H. Heffes and D. Lucantoni, "A markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance," *JSAC*, vol. 4, no. 6, pp. 856–868, Sep 1986.

[40] I. Norros, "On the use of fractional brownian motion in the theory of connectionless networks," *JSAC*, vol. 13, no. 6, pp. 953–962, Aug 1995.

[41] R. Addie, T. Neame, and M. Zukerman, "Performance evaluation of a queue fed by a poisson pareto burst process," *Computer Networks*, vol. 40, no. 3, pp. 377–397, 2002.

[42] M. Zukerman, T. Neame, and R. Addie, "Internet traffic modeling and future technology implications," in *INFOCOM*, vol. 1, March 2003, pp. 587–596 vol.1.

E Paper “Formalizing compute-aggregate problems in cloud computing”

Authors. Pavel Chuprikov, Alex Davydow, Kirill Kogan, Sergey Nikolenko, and Alexander Sirotkin

Abstract. Efficient representation of data aggregations is a fundamental problem in modern big data applications, where network topologies and deployed routing and transport mechanisms play a fundamental role to optimize desired objectives: cost, latency, and others. We study the design principles of routing and transport infrastructure and identify extra information that can be used to improve implementations of compute-aggregate tasks. We build a taxonomy of compute-aggregate services unifying aggregation design principles, propose algorithms for each class, analyze them, and support our results with an extensive experimental study.

Formalizing Compute-Aggregate Problems in Cloud Computing*

Pavel Chuprikov[1,2], Alex Davydow[1], Kirill Kogan[2], Sergey Nikolenko[1], and Alexander Sirotkin[1,3]

[1] Steklov Institute of Mathematics at St. Petersburg, Russia adavydow@gmail.com, sergey@logic.pdmi.ras.ru
[2] IMDEA Networks Institute, Madrid, Spain pavel.chuprikov@imdea.org, kirill.kogan@imdea.org
[3] National Research University Higher School of Economics, St. Petersburg, Russia avsirotkin@hse.ru

Abstract. Efficient representation of data aggregations is a fundamental problem in modern big data applications, where network topologies and deployed routing and transport mechanisms play a fundamental role to optimize desired objectives: cost, latency, and others. We study the design principles of routing and transport infrastructure and identify extra information that can be used to improve implementations of compute-aggregate tasks. We build a taxonomy of compute-aggregate services unifying aggregation design principles, propose algorithms for each class, analyze them, and support our results with an extensive experimental study.

Keywords: Compute-aggregate problems · Cloud computing · Competitive analysis.

1 Introduction

Data centers store data at different interconnected locations. Modern big data applications are highly distributed, and requests need to satisfy various objectives: latency, cost efficiency, etc. [2, 6, 14]. *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. Traditionally, applications have little control over how network transport handles the data. Latency optimization should account for properties of underlying transports in order to avoid, e.g., the *incast problem* [8, 27], 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 work, we assume that each compute-aggregate task should conform to a budget constraint since different cloud tenants are able to invest different

* This work was supported by the Russian Science Foundation grant 17-11-01276.

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 in (1) 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 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 jobs is 40.3% of the initial data sizes in Google [11], 8.2% in Yahoo, and 5.4% in Facebook [7].

In this work, we define a model for constructing an aggregation plan under budget constraints that requires applications to specify only one aggregation property: the (approximate) size of two data chunks after aggregation. Properties of aggregation functions can have a significant effect on the aggregation plan. We classify compute-aggregate tasks into several categories with respect to this property, propose algorithms for these optimization problems, and analyze their properties, proving a number of results on their performance and complexity, both positive (polynomial algorithms with good approximation ratios) and negative (inapproximability results).

The paper is organized as follows. Section 2 summarizes prior art, Section 3 introduces the model. In Section 4 we classify aggregate functions with regard to their effect on input data chunks and study their computational properties, i.e., hardness and approximability of optimization problems. Section 5 presents our experimental results, and Section 6 concludes the paper.

2 Related work

Various frameworks split computations into multiple phases: Map-Reduce-Merge [23] extends MapReduce to implement aggregations, Camdoop [9] assumes that an aggregation's output size is a specific fraction of input sizes, Astrolabe [17] collects large-scale system state and provides on-the-fly attribute aggregation, and so on. Like other data-flow systems [11, 24, 25], Naiad [16] offers the low latency of stream processors together with the ability to perform iterative and incremental computations. The work [25] introduces a distributed memory ab-

straction for fault-tolerant in-memory computation on large clusters, with orders of magnitude better latency than disk accesses. Other stream processing frameworks support low-latency dataflow computations over a static dataflow graph [1, 15, 20], while [10] explores optimal tree overlays to optimize latency of compute-aggregate tasks under specified budget constraints.

3 Motivation and model description

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 work, 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, is fully decoupled from transport implementation, and will be formalized below.

3.1 Compute-aggregate tasks

We model a network as an undirected connected graph $G = (V, E)$, where V is the 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 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 transmit $\boxed{x}$ through e one must pay $c(e) \cdot \mathrm{size}(\boxed{x})$. A simple example of a compute-aggregate task is shown on Fig. 1a. Costs are shown on the edges, square brackets denote chunks, and the root vertex is marked by t.

3.2 Move to root

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 on Fig. 1 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

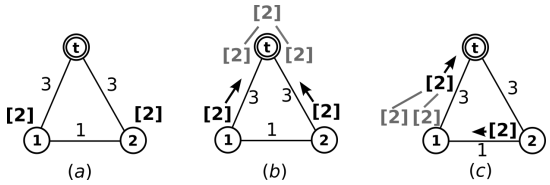


Fig. 1. A sample compute-aggregate task with three vertices t (target), u, and v: (a) the problem; (b) "move to root" plan with cost 12; (c) optimal aggregation plan with cost 11. Transmission cost of every edge is specified near the middle of the edge (e.g., $c(u, v) = 1$).

total cost 12 (Fig. 1b: two chunks of size 2 each moving along edges of cost 3), while on Fig. 1c 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 [5, 26] 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 [7, 11] 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 [22]. 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 [11]. The TCP-incast problem, on the other hand, can be mitigated by carefully spreading chunk transmissions in time (to reduce overlap), which requires complex synchronization on the part of the transport layer. Low-latency in-RAM applications also have to synchronize data transmissions to avoid too many data chunks "in the air" at the same time that cannot be aggregated; this is hard to implement in a distributed system,

and if **aggr** is not commutative and associative this leads to more constraints since transmissions must occur in a specific order.

All of the above suggests that it is hard for the "move to root" heuristic to reconcile network transport limitations with storage constraints and the distributed environment. Hence, in this work we explore aggregations at intermediate nodes.

3.3 Exploiting intermediate nodes

The basic principle of *data locality optimization*, which lies at the heart of the *Hadoop* framework [21], 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 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: $\mathrm{v}(\boxed{z}) = t$. E.g., Figs. 1b and 1c show aggregation plans for the problem on Fig. 1a. Aggregation plans are fully decoupled from the transport layer, producing instructions and constraints that the transport layer must satisfy.

An aggregation plan has an associated transmission cost $\mathrm{cost}(P)$, which is the sum of costs of all operations in P; here $\mathrm{cost}(\mathbf{aggr}(\boxed{x}, \boxed{y})) = 0$ (there is no data transmission), and $\mathrm{cost}(\mathbf{move}(\boxed{x}, v)) = \mathrm{size}(\boxed{x}) \cdot d(\mathrm{v}(\boxed{x}), v)$, where $d(u, v)$ is the total cost of the cheapest path from u to v.

This approach of "moving aggregation to data" has some important advantages over "move to root". First, the TCP-incast problem becomes less pronounced because inbound traffic is spread among different nodes, and fewer nodes need to be synchronized. Moreover, the total number of transmitted bits is reduced due to earlier aggregations (we usually expect an aggregation result to be smaller than the total input size). Second, storage capacity is now less of a constraint since less data has to be collected per node. Last but not least, data transmission cost is also reduced (cf. examples on Fig. 1). Note, however, that in practice not all nodes may be used for data aggregation. For example, we may be restricted to nodes where initial data chunks reside because it is expensive to allocate additional compute nodes; or it can be a security concern to perform computation on intermediate nodes (e.g., initial nodes belong to a private cloud, and the rest are transit nodes). This question must be carefully answered, and in what follows we assume that the overlay graph G reflects this answer and maps **aggr** operations to appropriate nodes.

3.4 Aggregation size function

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

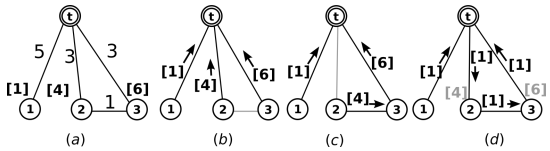


Fig. 2. Different μ lead to different plans: (a) a sample task; (b) optimal plan for $\mu(a,b) = a + b$; (c) optimal plan for $\mu(a,b) = \max(a,b)$; (d) optimal plan for $\mu(a,b) = \min(a,b)$.

aggregation result sizes can lead to very different solutions. For example, if on Fig. 1 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 on Fig. 1c 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, μ should also have these properties. Some examples of μ for practical problems include: $\mu(a,b) = \text{const}$ 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; $\max(a,b) \le \mu(a,b) \le a + b$ for set union (word count).

Fig. 2 shows how the choice of μ can affect the optimal aggregation plan. Fig. 2a 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. 2b). 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. 2c). 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. 2d). Thus, even in a simple example the aggregation plan can change drastically depending on μ. We get the following optimization problem.

*Problem 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$[\mu]$ problem is to find an aggregation plan P such that $\text{cost}(P)$ is minimized.

Interestingly, unless both the aggregation size function is well-behaved and there are constraints on the graph structure, there is not much we could do in the worst case.

Theorem 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.*

Proof. We can encode an NP-hard problem in choosing the correct order of merging for a non-associative μ. For example, consider an instance of the knapsack problem with weights $w_1, \ldots, w_n$, unit values, and knapsack size W; then we have n chunks of size $w_1, \ldots, w_n$, and μ is defined as follows: if either $x = 0$, $y = 0$, or $x + y = W$ then $\mu(x, y) = 0$; else $\mu(x, y) = x + y$. This way, if we can fill the knapsack exactly the total resulting weight will be zero, and if not, it will be greater than zero, leading to unbounded approximation ratio unless we can solve the knapsack problem.

In this work, we investigate two main degrees of freedom that the CAM problem has: network topology graph G and aggregation size function μ. Let us begin with μ.

4 A Taxonomy of Aggregation Functions

There are different types of big data applications, a large variation in data-center network topologies, and countless data distributions, 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). E.g., if the network graph is a tree, it might be possible to construct an aggregation plan in linear time; or a better (in terms of the resulting cost) algorithm 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 are common network topologies, such as *hypercube*, *fat-tree*, or *jellyfish*, there are plenty of variations and exceptions. So, while algorithms specifically tailored for, e.g., the hyper-cube topology [9] remain a valid topic for future study, in this work 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

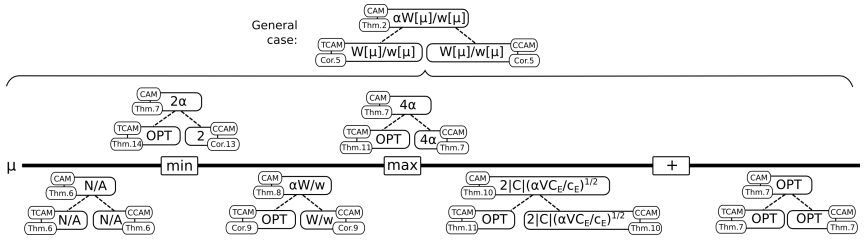


Fig. 3. The structure of our results in relation to aggregation size function μ and problem instance (CAM, TCAM, CCAM).

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). Our theoretical results are summarized in Fig. 3; the horizontal axis corresponds to how fast μ grows, and each small tree of results shows approximation ratios for CAM, TCAM, and CCAM, referring to specific theorems below.

4.1 General case

In this subsection we assume no constraint on the behavior of an aggregation size function μ. As an example, consider a simpler setting where all chunks have size x, and $\mu(x, x) = x$. In this case, when paths of two chunks intersect, it 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 [13], MSTT, that has many constant approximation algorithms. Using one of those we build our first aggregation plan construction algorithm `steiner_rec` (Alg. 2). If there is a polynomial α-approximate algorithm for MSTT, then `steiner_rec` provides an α-approximation for the special case when size$(x) = S$ for any $x \in C$, and $\mu(S, S) = S$. The `steiner_rec` algorithm has a number of interesting properties; e.g., it does not require any knowledge of μ or even chunk sizes. The infrastructure can run `steiner_rec` even before preprocessing (in map-reduce terminology, before a *map* phase). It turns out that in the general case, the price of using `steiner_rec` does not exceed the *ratio between the largest and smallest intermediate chunk*. We denote by $W_C[\mu]$ the maximal aggregate size of a subset of chunks from the set C, $W_C[\mu] = \max_{C' \subseteq C}\{\mu(C')\}$; it is well defined since μ is associative and commutative. We also denote by $w_C[\mu]$ the corresponding minimal aggregate size, $w_C[\mu] = \min_{C' \subseteq C}\{\mu(C')\}$.

Algorithm 1 `steiner`$(G, V' \subseteq V(G))$

if G is a tree **then**
    return unique subtree $T_{V'}$ covering V'
else if $V(G) = \{\mathrm{v}(x) : x \in C\}$ **then**
    return `min_spanning_tree`(G)
else
    return `steiner_tree_approx`$(G, \{\mathrm{v}(x) : x \in C\})$

Algorithm 2 `steiner_rec`(G, t, C)

$P \leftarrow (); T \leftarrow$ `steiner`$(G, \{\mathrm{v}(x) : x \in C\})$
for $v \in T$ in decreasing order of depth(v) **do**
    ▷ Denote $C(v) = \{x \in C : \mathrm{v}(x) = v\}$
    while $|C(v)| > 1$ **do**
        P.`append`(**aggr**(x, y)), where $x, y \in C(v)$
        C.`update`(**aggr**(x, y))
    if $\exists\, x \in C(v)$ **and** $\exists$ parent(v) **then**
        P.`append`(**move**$(x, \mathrm{parent}(v))$)
        C.`update`(**move**$(x, \mathrm{parent}(v))$)
return P

Theorem 2. *If there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial algorithm that solves* CAM$[\mu]$ *with approximation factor* $\alpha \frac{W_C[\mu]}{w_C[\mu]}$.

Proof. First, note that any algorithm, even optimal, has to traverse at least the Steiner tree of G in total size, and has to carry at least weight w_c over each edge. The approximate algorithm begins by constructing the approximate Steiner tree with approximation ratio α, and then carries all chunks along this tree to the root, merging the chunks at first opportunity; in this process, the maximal possible chunk size is W_c, and it is carried over at most α times longer distance than in the actual Steiner tree, getting the approximation bound. □

There is a well-known 2-approximation to MStT based on a minimum spanning tree (MST) of the distance closure G^* of G; a much more involved construction leads to the best known approximation ratio of $\ln 4 + \varepsilon \leq 1.39$ [4]. Although `steiner_rec` does not depend on either μ or chunk sizes, the approximation factor in Theorem 2 includes both. This result improves for special cases of TCAM and CCAM.

Theorem 3. *If every vertex in G contains a data chunk, then* MSTT *can be solved exactly in polynomial time.*

Proof. In this case, MSTT is equivalent to MST. □

Theorem 4. *If G is a tree,* MSTT *can be solved exactly in polynomial time.*

Proof. There is only one subtree in G that connects a given set of vertices, and it can be found in polynomial time. □

Theorem 3 and Theorem 4 essentially say that in these special cases we have 1-approximation algorithms for MSTT. Theorem 2 and this observation together imply the following.

Corollary 1. *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 2 and Corollary 1 provide 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 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 $P = NP$.*

Proof. Consider a complete graph G where the root r contains an infinitely large chunk, all non-root vertices are terminals, edges between two terminal vertices cost 1, and edges between a terminal vertex and the root cost ∞. Given an instance of Set Cover, where a set S must be covered with a minimal number of m subsets $S_i \subseteq S$, we define $n(S_i)$ as the number with binary representation equivalent to $S \setminus S_i$ (for some fixed order of elements in the set). We encode S by a chunk of size 0 and any other subset $A \subset S$ by a chunk of size $n(A) + 4n \times 2^{|S|}$. The aggregation size function for two chunks corresponding to subsets A and B produces a chunk of size $n(A \cup B)$. Now, if there exists a set cover $S_{i_1}, S_{i_2}, \cdots, S_{i_k}$ then there is a solution to CAM$[\mu]$ of size $k \times 4n \times 2^{|S|} + c$, where $c \leq n \times 2^{|S|}$ (we can aggregate S_{i_j} in any order and then aggregate the rest with a zero chunk we obtained). On the other hand, if there exists a solution to CAM$[\mu]$ of size $g \times 4n \times 2^{|S|} + c$, where $c \leq n \times 2^{|S|}$, then there exists a solution to Set Cover of size g (to achieve this solution of CAM$[\mu]$ we have to obtain 0 in at most g agregations). Thus, a constant approximation for CAM$[\mu]$ implies a constant approximation of Set Cover which is impossible unless $P = NP$. For TCAM$[\mu]$, consider the following transformation of G to a tree T_G: remove all the edges; introduce a new vertex c; connect c with r by an edge of weight ∞ and with the rest of G's vertices by edges of weight 1. Changing G to T_G does not increase cost more than twice (we traverse two edges now). Thus, the transformation preserves approximations, which again implies that TCAM$[\mu]$ does not have constant approximations unless $P = NP$. □

Since CCAM is a strict subset of CAM, there is no constant-approximation solution for CAM either.

4.2 Range-bounded aggregation size functions

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. Theorem 5 showed that aggregation size functions that reduce size too much are provably hard. On the other side of the spectrum, where $\mu(x, y) \geq x + y$, there is an optimal solution: bring all chunks to the sink.

Theorem 6. *If $\mu(a, b) \geq a+b$ for all a, b then there exists a polynomial optimal algorithm for* CAM[μ], CCAM[μ], *and* TCAM[μ]; *for* TCAM[μ] *the running time is* $O(|C| + |G|)$.

Proof. In this case, it does not make sense to merge chunks at all; the optimal algorithm is to bring all chunks separately to the sink. Formally, consider an optimal aggregation plan for CAM that merges two chunks not at the sink. Next, consider a transformed plan that carries both chunks separately and treats them separately until the final vertex. Since $\mu(a, b) \geq a + b$, the total cost will not increase in this transformation, and we can sequentially get a plan without any merging without increasing the costs. The optimal strategy without merging is to move all chunks to the root along shortest paths, which can be computed in polynomial time. Because TCAM and CCAM are strict subsets of CAM, `SPT` is optimal for them too. For TCAM there is no need to calculate shortest paths since paths are unique, and the running time becomes linear. □

We have found that for $\mu(x, y) \in (-\infty, \min\{x, y\}]$ the problem is inapproximable (Theorem 5), and for $\mu(x, y) \in [x + y, \infty)$ there is an optimal algorithm (Theorem 6). We split the remaining range $[\min\{x, y\}, x + y]$ at $\max\{x, y\}$ for two reasons. First, in practice max is a valid bound for many applications: set intersection, set union, outer join (symmetric or asymmetric); thus, the infrastructure often knows on which side of max μ lies. Second, theoretic results below show that max is an interesting demarcation line for worst-case guarantees: below max chunk sizes are a primary factor, and above max the graph structure starts to dominate. If $\mu(x, y) \in [\min(x, y), \max(x, y)]$, we can replace the ratio $W_C[\mu]/w_c[\mu]$ (Theorem 2), which depends on μ, with a simpler one that depends only on chunk sizes. In the next theorem $W_C = \max_{x \in C}\{\text{size}(x)\}$, $w_c = \min_{x \in C}\{\text{size}(x)\}$.

Theorem 7. *If* $\min\{a, b\} \leq \mu(a, b) \leq \max\{a, b\}$ *for all a, b and there exists a polynomial α-approximate algorithm for* MSTT, *then there exists a polynomial algorithm that solves* CAM[μ] *with approximation factor* $\alpha \frac{W_C}{w_C}$.

Corollary 2. *If* $\min\{a, b\} \leq \mu(a, b) \leq \max\{a, b\}$ *then there exist polynomial algorithms that solves* CCAM[μ] *and* TCAM[μ] *with approximation factor* $\frac{W_C}{w_C}$.

For $\mu(x, y) \in [\max\{x, y\}, x + y]$, the last remaining range, we employ a mix of `SPT` and `steiner_rec`: merge chunks above a certain threshold with `steiner_rec`; below, with `SPT`.

Theorem 8. *If for all a and b $\max(a, b) \le \mu(a, b) \le a + b$, then there is an $2NV^{1/2}\sqrt{\alpha\frac{c_{\max}}{c_{\min}}}$-approximate polynomial algorithm for* CAM$[\mu]$*, which we call* `RECH_MStTSplit`*, where V is the number of vertices in G, N is the number of chunks, $c_{\max}$ is the cost of the most expensive edge in G, $c_{\min}$, of the cheapest edge, and α is an approximation factor for* MStT.

Proof. The idea of the algorithm is as follows. We split all chunks C into two sets: chunks with weight at least δM go into set C_1 and chunks with weight smaller than δM go into C_2, where M is the weight of the maximal chunk and δ is a constant to be defined later, so $C = C_1 \cup C_2$. Next we solve two separate CAM problems. For C_1 we run the general algorithm from Theorem 2, and for C_2 we run the algorithm from Theorem 6 that we used for μ such that $\mu(a, b) \ge a + b$. The first algorithm yields an $\frac{\alpha N}{\delta}$-approximate solution, and the total weight of the second solution does not exceed $\delta MVNc_{\max}$, where N is the number of chunks. Let W be the weight of the optimal solution. Now, since $\max(a, b) \le \mu(a, b)$, and W is at least the weight of the optimal solution for C_1, we can conclude that the weight of the solution for C_1 is at most $\frac{\alpha V}{\delta}W$. On the other hand, since $W \ge Mc_{\min}$, the weight of the solution for C_2 is at most $\delta V^2\frac{c_{\max}}{c_{\min}}W$. Now if we choose $\delta = \frac{\sqrt{\alpha c_{\min}}}{V^{1/2}\sqrt{c_{\max}}}$ to minimize the total result, the total weight of both solutions will be $2NV^{1/2}\sqrt{\alpha\frac{c_{\max}}{c_{\min}}}W$. □

To improve the above theorem we cannot apply Theorem 3 to get rid of α for CCAM or TCAM since it uses Steiner tree only for a subset of chunks. But, remarkably, we can do better for TCAM: the following theorem proves that between *max* and "+" `steiner_rec` is optimal for TCAM. The algorithm is similar to Theorem 2.

Theorem 9. *There exists a polynomial optimal algorithm for the* TCAM$[\mu]$ *problem for any μ such that $\forall a, b$ $\max(a, b) \le \mu(a, b) \le a + b$.*

Proof. The algorithm is similar to Theorem 2: move chunks towards the vertex t, merging them in intermediate nodes. Consider an arbitrary subtree T of G. All data chunks from T have to be eventually moved upwards using parent edge e (if $T \ne G$). Minimal cost of this operation is clearly $\le s_T$, where s_T is the size of the aggregation result of all chunks from $C|_T$. Can it be less? Assume the opposite: there is a set of data chunks X that will be moved upwards through e s.t. $C_T \subseteq X^* = \bigcup_{x\in X} x^*$ and $\sum_{x\in X}\text{size}(x) < s_T$, where x^* is the set of initial chunks that contributed to x. If $C_T \subsetneq X^*$, we can throw away $X^* \setminus C_T$ without any increase in the cost because μ is at least max. Also, since μ does not exceed the sum, we can aggregate X with no cost increase. The resulting chunk has size s_T, which is a contradiction: by construction, each upward edge e from a subtree T will add exactly s_T. □

4.3 Specific aggregation size functions

Sometimes we can improve performance further if we know μ exactly. This is especially interesting for the "junction points" between previous results; Theorem 6 covers sum, here we concentrate on min and max.

Theorem 10. *If there exists a polynomial α-approximate algorithm for* MStT*, then there exists a polynomial 2α-approximate algorithm for* CAM[min].

Proof. Given an instance (G, t, C) of the CAM[min] problem, first we find an α-approximation T to the MStT instance $(G, V' = \{t\} \cup \{\mathrm{v}(x) : x \in C\})$. Then, we construct an aggregation schedule by taking a data chunk with the smallest size and walking it through T. The resulting cost does not exceed $2m \cdot w(T)$, where m is the size of the smallest chunk. Similar to Theorem 13, an aggregation schedule defines a subgraph $H \supseteq V'$, and so incurs the cost of at least $m \cdot w(H)$. A sample solution for this algorithm is shown on Fig. 4. □

Corollary 3. *There exists a polynomial* 2*-approximate algorithm for* CCAM[min].

TCAM[min] can be solved in polynomial time with dynamic programming.

Theorem 11. *There is an optimal algorithm for* TCAM[min] *running in polynomial time.*

Proof. The optimal algorithm uses dynamic programming. Consider an instance $(G = (V, E), t, C)$ of the TCAM[min] problem, where G is a tree with a root t. For every vertex $v \in V$ we compute $\mathrm{mc}(v)$, the size of the smallest chunk in a subtree T_v rooted at v, and for every $c \in C$ we compute $\mathrm{dp}(v, \mathrm{size}(c))$, an optimal solution for T_v with an additional chunk of size $\mathrm{size}(c)$ at v. We can find $\mathrm{mc}(v)$ for every vertex in linear time by running depth first search. If $\mathrm{dp}(v, \mathrm{size}(c))$ are known for every $u \in \texttt{children}(v)$ then $\mathrm{dp}(v, \mathrm{size}(c))$ can be computed as $\mathrm{dp}(v, \mathrm{size}(c)) = \sum_{u \in \texttt{ch}(v)} \min\{2 \cdot \mathrm{size}(c) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{size}(c)), \mathrm{mc}(u) \cdot \mathrm{d}(v, u) + \mathrm{dp}(u, \mathrm{mc}(u))\}$. Now $\mathrm{dp}(t, \mathrm{mc}(t))$ contains the cost of an optimal aggregation plan, which can be found with backtracking. □

Our second approximate algorithm, for CAM[max], is also based on the MStT problem, but with a different construction.

Theorem 12. *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`.

Proof. Let the maximal chunk size be equal to M. We separate data chunks into several subsets: the first with chunks with sizes in $(M/2, M]$, the second in $(M/4, M/2]$, and so on. The idea is to build an approximate solution for the first subset, extend it to a solution for the first two subsets, and so on. First, we build a Steiner tree for the root and chunks in the first subset and solve the problem on this tree. This solution is at least 2α-competitive, where α is the MStT approximation factor: edges used by a different solution must connect

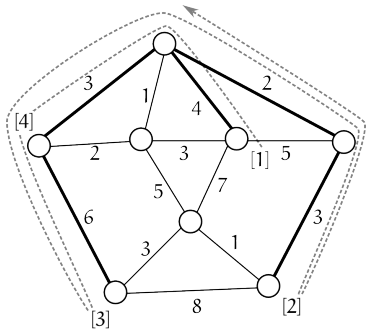


Fig. 4. Sample solution from Theorem 10. Steiner tree approximation is shown with bold lines, and the resulting path of the smallest chunk is shown by a dotted arrow. Note that this chunk never visits one edge more than twice.

every chunk to the root, so their total cost is at least the cost of a minimum Steiner tree, and their sizes are at least $M/2$. Next, we merge the tree obtained on the first iteration into a single vertex, throw away the first subset, build a Steiner tree for the second subset, solve the problem for this tree, and so on. Suppose that there were k such subsets. Since we move chunks of size at most $M/2^{i-1}$, and merging vertices does not increase the weight of a Steiner tree, the cost of the ith subset does not exceed $M/2^{i-1}\alpha\mathrm{ST}(i, i-1)$, where $\mathrm{ST}(i, j)$ is the optimal Steiner tree weight for chunks with sizes in $(M/2^i, M/2^j]$. Thus, the total cost does not exceed $2\alpha M \sum_{i=1}^{k} \mathrm{ST}(i, i-1)/2^i$. For the lower bound, we count the cost of all data movements across every edge. The total cost of all edges with chunks of mass at least $M/2$ moved along them is bounded by $\mathrm{ST}(1, 0)$, so the cost is bounded by $\frac{M}{2}\mathrm{ST}(1, 0)$; repeating the process for $\frac{M}{4}$, $\frac{M}{8}$ and so on, we get in total $M \sum_{i=1}^{k} \mathrm{ST}(i, 0)/2^i$. Some of the edges are counted more than once: an edge with the largest chunk of size $M/2^j$ moved along it has been counted once with a factor of $M/2^j$, once with $M/2^{j+1}$, and so on, but the real lower bound for this edge is $M/2^j$. Thus for every edge we have an extra factor of $1 + 1/2 + 1/4 + \ldots \leq 2$, and the optimal cost is at least $M/2 \sum_{i=1}^{k} \mathrm{ST}(i, 0)/2^i$. Since $\mathrm{ST}(i, 0) \geq \mathrm{ST}(i, i-1)$, the total approximation factor is $\frac{2\alpha M}{M/2} = 4\alpha$. □

Theorem 9 implies that TCAM[max] has an optimal solution since $\max\{x, y\}$ lies trivially in $[\max\{x, y\}, x + y]$. However, results for CAM[min] and CAM[max] cannot be significantly improved because both are NP-hard.

Theorem 13. *If there exists $a > 0$ s.t. $\mu(a, a) = a$ then* CAM$[\mu]$ *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.*

Proof. The proof is by the reduction from the MSTT problem. Given a MSTT instance $(G, w, V' \subseteq V)$, we place data chunks of size a in each vertex of V'

except one, which becomes the sink. Any aggregation schedule defines a connected subgraph H of G that contains all vertices from V'. The minimal cost is $a \cdot w(H)$, where $w(H) = \sum_{e \in E(H)} w(e)$, since all transmitted data chunks have size a, and we can always avoid transmitting more than one chunk across one link. Any spanning tree of H defines a Steiner tree for V', and vice versa, any Steiner tree T defines an aggregation schedule with cost $a \cdot w(T)$. □

Theorem 14. *The* CAM[min] *problem is NP-hard even if all edge weights are equal, and each vertex is required to contain a data chunk.*

Proof. This time we reduce the Hamiltonian cycle problem. Given a Hamiltonian cycle problem instance G, we choose the sink arbitrarily, place a chunk of size 1 in the sink and chunks of size $|V|^2$ in every other vertex. The optimal solution travels with weight 1 along a Hamiltonian cycle. □

5 Evaluation

To evaluate the relative performance of the proposed heuristics, we have compared them in practical settings on network topologies generated by the *topobench* library [12]. We analyzed topologies that can serve for both server adjacency and switch adjacency, namely *JellyFish* [19], *Hypercube*, and *Small World Datacenter Ring* (SWDC Ring) [18]. We compare seven algorithms: `CAMH`, a greedy algorithm that on every step chooses a pair of chunks $\boxed{x}$ and $\boxed{y}$ and vertex v such that after $\boxed{x}$ and $\boxed{y}$ are merged at v the cost of moving all resulting chunks along shortest paths plus the cost of moving $\boxed{x}$ and $\boxed{y}$ to v is minimized; `MCAMH`, a variation of `CAMH` where $v \in \{\mathrm{v}(\boxed{x}), \mathrm{v}(\boxed{y})\}$; $\texttt{MCAMH}_{\leq}$, a restriction of `MCAMH` where we always move the smaller chunk; `RECH_MST`, a variation of Alg. 2 that merges along the minimum *spanning* tree; `RECH_MStT` (Alg. 2); `RECH_SPT`, another variation of Alg. 2 that uses a tree of shortest paths; `RECH_MStTMax` (Thm. 12); `RECH_MStTSplit` (Thm. 8). An implementation of all algorithms and sample topologies can be found at [3]. Selected results are shown on Fig. 5; the plots show the number of data chunks $n_C = |C|$, maximal edge weight W, and range s of chunk sizes, $s = [s_{\text{avg}} - \Delta_s, s_{\text{avg}} + \Delta_s]$. To generate a CAM instance for a given topology, we uniformly select weights from 1 to W and uniformly place n_C data chunks in the vertices, each of size chosen uniformly from s. We used three aggregation functions: $\mu(x, y) = \exp(\log|x| + \log|y|)$ (Fig. 5a-c), which roughly estimates joining two dictionaries, WordCount, $\mathbf{aggr}(x, y) = \text{SetUnion}(x, y)$ (Fig. 5d-i), and $\mathbf{aggr}(x, y) = \text{ChooseBest}(x, y)$ (Fig. 5j-o), which compares the (random) priorities of two chunks. For each point, we averaged 1000 random experiments.

The results show that among tree-based heuristics, the best are `RECH_MStT`, `RECH_MStTMax`, and `RECH_MStTSplit`. This is expected: unlike `RECH_MST`, `RECH_MStT` works on the local level and does not optimize the part of the tree not directly invloved in aggregation, but also does not optimize every chunk separately as `RECH_STP` does. Among potential-based heuristics, `CAMH` clearly wins for

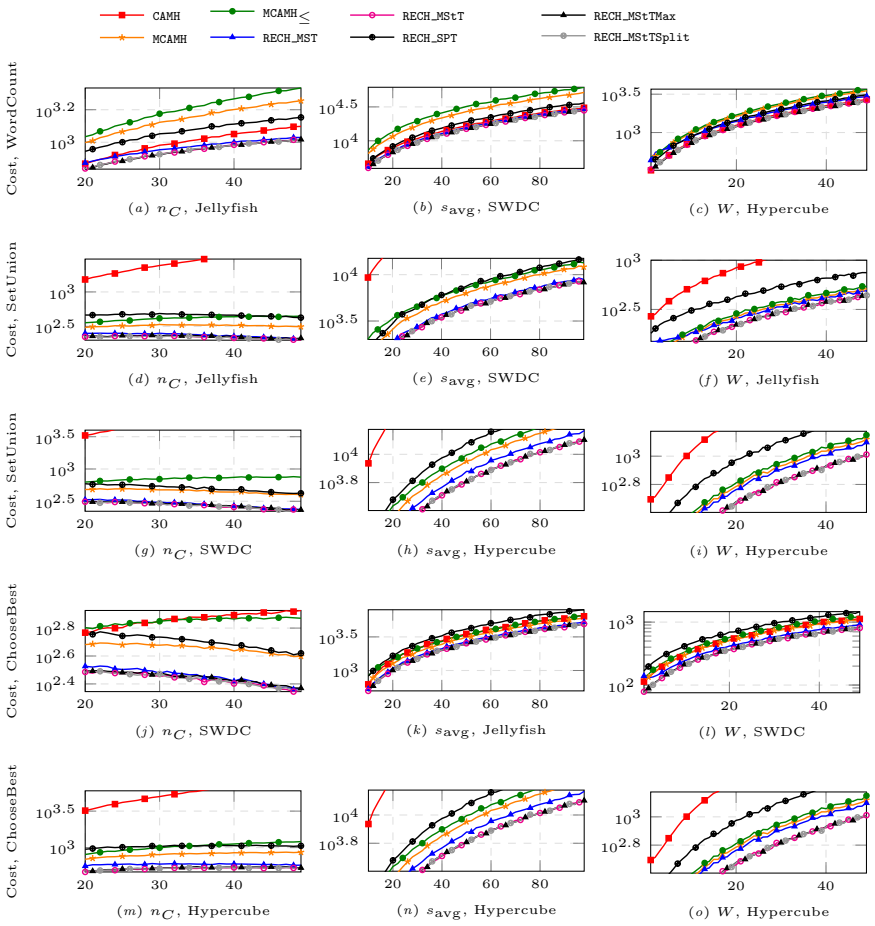


Fig. 5. Costs for different aggregation functions: (a-c) $\mu(x, y) = \exp(\log x + \log y)$; (d-i) $\mathbf{aggr}(x, y) = \text{SetUnion}(x, y)$; (j-o) $\mathbf{aggr}(x, y) = \text{ChooseBest}(x, y)$. Labels show X-axis value and topology. Parameters in left column: $W = 15$, $s = [10, 30]$; middle column: $n_C = 20$, $W = 100$; right column: $n_C = 10$, $s = [10, 30]$.

WordCount (it is even better than `RECH_STP`) but fares much worse for SetUnion and ChooseBest. `CAMH`, `MCAMH`, and $\texttt{MCAMH}_{\leq}$ use the value of μ, so their behaviour changes significantly between SetUnion and ChooseBest: the latter can bring a larger improvement if we get lucky and get a light high-priority chunk. The relative performance of other algorithms does not change significantly with aggregation function. `CAMH`, `MCAMH`, and $\texttt{MCAMH}_{\leq}$ represent a tradeoff between expressivity (larger search space) and a higher chance to end up in a worse solution. `MCAMH` serves as the middle ground and takes the leading place almost everywhere among these three heuristics; `CAMH` wins on WordCount, where merges are more regular and predictable, and the local behavior of **aggr** does not mislead `CAMH`. In

general, heuristics based on Steiner trees are the best throughout all experiments, with `RECH_MStT`, `RECH_MStTMax`, and `RECH_MStTSplit` occupying the top three places almost everywhere, even though they do not pay that much attention to **aggr**. `RECH_MStTMax` starts losing for the ChooseBest aggregation function since it was designed specifically for **aggr** = max, and ChooseBest is very different. As the number of chunks grows, `RECH_MST` becomes closer to `RECH_MStT` and its variations: chunks cover more vertices, and Steiner trees become more similar to spanning trees. `RECH_SPT` is worse than `RECH_MstT`, as expected; it even loses to `MCAMH` and $\texttt{MCAMH}_{\leq}$ in many cases but regains some ground for more chunks. Interestingly, `RECH_SPT` works better on SWDC Ring; this may be because this topology is bounded (it is a cycle with four random chords coming out of each vertex), so the tree of shortest paths covers most edges in the cycle and more chunks.

There are two main conclusions to be drawn from the evaluation study. First, for a given topology heuristics designed for specific aggregation functions perform significantly better in their own domain. Second, heuristics designed for the same range of aggregation functions for a specific topology perform better than more general heuristics independent from network topologies. This confirms analytic results from Section 4 and allows designers of compute-aggregate infrastructures to extend the taxonomy proposed in Fig. 3 if better performance is needed. This demonstrates a fundamental tradeoff between simplicity of infrastructure and its performance.

6 Conclusion

In this work, we have introduced a model to find a schedule of aggregations that satisfies budget constraints rather than directly optimizing desired objectives such as latency or throughput. We believe that this approach will allow to decouple optimization problems from underlying transports and provide fine-grained control to exploit network infrastructure. Our primary contribution is a classification of aggregation functions with a theoretical and practical analysis that together lead to unified design principles of "perfect" aggregations.

References

1. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. PVLDB **6**(11), 1033–1044 (2013)
2. Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A.: Hedera: Dynamic flow scheduling for data center networks. In: USENIX. pp. 281–296 (2010)
3. Anonymous: Formalizing compute-aggregate problems in cloud computing code. https://github.com/CrKJdwbRAe/infocom2017
4. Byrka, J., Grandoni, F., Rothvoß, T., Sanità, L.: An improved lp-based approximation for steiner tree. In: Proceedings of the Forty-second ACM Symposium on Theory of Computing. pp. 583–592. STOC '10, ACM, New York, NY, USA

(2010). https://doi.org/10.1145/1806689.1806769, http://doi.acm.org/10.1145/1806689.1806769
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink^{TM}: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data (awarded best paper!). In: OSDI. pp. 205–218 (2006)
7. Chen, Y., Ganapathi, A., Griffith, R., Katz, R.H.: The case for evaluating mapreduce performance using workload suites. In: MASCOTS. pp. 390–399 (2011)
8. Chen, Y., Griffith, R., Liu, J., Katz, R.H., Joseph, A.D.: Understanding TCP incast throughput collapse in datacenter networks. In: WREN. pp. 73–82 (2009)
9. Costa, P., Donnelly, A., Rowstron, A.I.T., O'Shea, G.: Camdoop: Exploiting in-network aggregation for big data applications. In: NSDI. pp. 29–42 (2012)
10. Culhane, W., Kogan, K., Jayalath, C., Eugster, P.: Optimal communication structures for big data aggregation. In: INFOCOM. pp. 1643–1651 (2015)
11. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
12. Jyothi, S.A., Singla, A., Godfrey, P.B., Kolla, A.: Measuring and Understanding Throughput of Network Topologies. Tech. rep. (2014), http://arxiv.org/abs/1402.2531
13. Kaklamanis, C., Chlebk, M., Chlebkov, J.: Algorithmic aspects of global computing the steiner tree problem on graphs: Inapproximability results. Theoretical Computer Science **406**(3), 207 – 214 (2008)
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review **44**(2), 35–40 (2010)
15. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD. pp. 135–146 (2010)
16. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SIGOPS. pp. 439–455 (2013)
17. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. **21**(2), 164–206 (2003)
18. Shin, J.Y., Wong, B., Sirer, E.G.: Small-world datacenters. In: SOCC. pp. 2:1–2:13. SOCC '11 (2011)
19. Singla, A., Hong, C.Y., Popa, L., Godfrey, P.B.: Jellyfish: Networking Data Centers Randomly. In: USENIX (2012)
20. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. **15**(3), 555–568 (2003)
21. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc., 1st edn. (2009)
22. Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., Zhou, L.: Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 44–53. ICSE Companion 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2591062.2591177, http://doi.acm.org/10.1145/2591062.2591177
23. Yang, H., Dasdan, A., Hsiao, R., Jr., D.S.P.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD. pp. 1029–1040 (2007)

24. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI. pp. 1–14 (2008)
25. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI. pp. 15–28 (2012)
26. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: A unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664, `http://doi.acm.org/10.1145/2934664`
27. Zhang, Y., Ansari, N.: On architecture design, congestion notification, TCP incast and power consumption in data centers. IEEE Communications Surveys and Tutorials **15**(1), 39–64 (2013)