PROACTIVE CONGESTION CONTROL

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

Lavanya Jose

January 2019

© Copyright by Lavanya Jose 2019
All Rights Reserved

Lavanya Jose

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

(Nick McKeown) Principal Adviser

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

(Mohammad Alizadeh)

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

(Sachin Katti)

Approved for the Stanford University Committee on Graduate Studies

---

Abstract

Most congestion control algorithms rely on a reactive control system that detects congestion and then marches carefully toward a desired operating point (e.g., by modifying the window size or picking a rate). These algorithms often take hundreds of RTTs to converge—an increasing problem in networks with short-lived flows.

Motivated by the need for fast congestion control, this thesis focuses on a different class of congestion control algorithms, called proactive explicit rate-control (PERC) algorithms, which decouple the rate calculation from congestion signals in the network. The switches and NICs exchange control messages to run a distributed algorithm to pick explicit rates for each flow. PERC algorithms proactively schedule flows to be sent at certain explicit rates. They take as input the set of flows and the network link speeds and topology, but not a congestion signal. As a result, they converge faster and their convergence time depends only on fundamental "dependency chains," essentially couplings between links that carry common flows, that are a property of the traffic matrix and the network topology. We argue that congestion control should converge in a time limited *only* by fundamental dependency chains.

Our main contribution is s-PERC: a new practical distributed proactive scheduling algorithm. It is the first PERC algorithm to provably converge in bounded time without requiring per-flow state or network synchronization. It converges to the exact max-min fair allocation in $6N$ rounds (where N is the number of links in the longest dependency chain). In simulation and on a P4-programmed FPGA hardware test bed, s-PERC converges an order of magnitude faster than reactive schemes such as TCP, DCTCP, and RCP. Long flows complete in close to the ideal time, while short-lived flows are prioritized, making it relevant for data centers and wide-area networks (WANs).

To my parents.

Acknowledgments

Doing a Ph.D. is simultaneously the most difficult and the most enriching experience I have undertaken. I am deeply indebted to my adviser, Nick McKeown, for supporting me in every step of this journey and for providing just the right balance of independence and guidance. Nick, thank you for looking out for me. I have learned a lot from you—to take no assumption for granted, to distill every problem down to its essence, and many other valuable lessons which will hold me in good stead for my next adventure. I am glad to say that I have inherited from you a taste for real, practical problems and simple solutions that are grounded in theory.

I am grateful to Mohammad Alizadeh for the many insightful discussions we have had on congestion control in the last few years. Mohammad, thank you for introducing me to message passing algorithms, and for all the practical advice and feedback you have given. I also want to thank you for inviting me to the Dagstuhl workshop on "Network Latency Control in Data Centres." The breakout session on "Congestion Control in 100 Gb/s Networks," inspired by PERC, helped me to see the problem from expert perspectives across industry and academia.

I must also thank Sachin Katti for all the support from the beginning of my time at Stanford, when he reached out first to let me know I was accepted, until the very end, when he agreed to be my thesis reader. Sachin, thank you for taking me under your wing in my first quarter and advising my first rotation project, where I learned as much about the cellular packet core as I did about writing papers and working efficiently.

Many thanks to professors Balaji Prabhakar and Ramesh Johari for the insightful discussions during the last few months, which have enriched this thesis. Prof. Prabhakar, I would have loved to start our collaboration sooner. I have come away much wiser from each of our meetings, amazed at how good notation and ordering of arguments can make all the difference. Prof. Johari, thank you for sharing your optimization-theory perspective of the problem and for discussions that gave me much food for thought.

I am also grateful to George Varghese for mentoring me in my first two years at Stanford when we worked on the compiler project and the Fair (PERC) algorithm. George, your passion for problems is infectious, and

I thank you for the voice in the back of my head that nudges me to reach out across disciplines and find new perspectives.

The PERC approach started from discussions back in 2015, with Nick, Mohammad, George, Lisa Yan, and Isaac Keslassy about using the network in cool and interesting ways. Thanks to Isaac for conscientiously reading through early drafts of the proofs of the Fair algorithm. Thank you also to Steve Ibanez and Jonathan Perry for being a sounding board for my ideas about a practical PERC algorithm.

I count myself very lucky to have found a collaborator like Steve to work with me on s-PERC. Steve, this thesis wouldn't have been possible without your help. Thank you for helping to make s-PERC a reality with the P4-NetFPGA prototype and test bed; for demoing our work at Florianopolis, despite many bugging issues (Zika included) at the last minute; for reading through the first draft of this thesis; for being an unflappable, cheerful and super hard-working team mate through different iterations of s-PERC; and for being a great office mate with an inspiring work ethic.

I would like to extend my sincere thanks to Radhika Mittal, Srinivas Narayana, Sundar Iyer, Mani Kotaru, KK Yap, Lisa Yan, Sean Choi, Vimalkumar Jeyakumar, and Nandita Dukkipati for reading my drafts carefully and critically and providing invaluable feedback. I would like to thank Sabrina Leroe, for proofreading the final version of my thesis.

I would like to thank Rui Zhang, Jahangir Hasan, and Yaogong Wang for hosting me during my internships at Google, which helped me understand networking in the wild and see where it fits in the big picture. Many thanks to Jahangir and Yaogong for letting me try proactive congestion control in a real data-center network; it convinced me that I was on the right track with PERC. I also learned a lot from discussions with Hassan Wassel, Nandita Dukkipati, Abdul Kabbani, David Wetherall, Dina Papagiannaki, KK, Jon Zolla, and Amin Vahdat at Google.

A special thank you to Nandita for mentoring me through my time at Google over summers and beyond. Nandita, thank you for all your advice on life and work. I still cherish the day trip with Rong Pan and you following the Dagstuhl workshop, when I realized with awe—here are two women who were in my shoes 10 years ago, getting their Ph.D. in networking at Stanford, and are now making the Internet faster for everyone, through their contributions to networking, whether in the Cloud or in home routers across the world. Thank you, Nandita and Rong, for being inspiring role models for me.

I was lucky to have overlapped with many academic siblings during my time at Stanford and made lasting friendships—a huge thanks to James Zeng, Peyman Kazemian, KK, TY Huang, Glen Gibb, Yiannis Yiakoumis, Lisa, Steve, Sean Choi, Eyal Cidon, Catalin Voss, and Bruce Spang for all the ways you have enriched my Ph.D. journey. Lisa, thank you for being the best (and only!) office mate and sounding board during the

early years, and for collaborating with me on the compiler and PERC papers. Many thanks to my current officemates Colleen Josephson, Steve, Mani, Catalin and Jenny Han for making Gates 314 my favorite place to work. I'd like to recognize all the efforts of Lancy Nazaroff and our former group admin, Chris Hartung for making everything work smoothly.

I was a lucky undergraduate before I came to Stanford—I am deeply indebted to professors Jennifer Rexford, Christiane Fellbaum, and Moses Charikar for introducing me to research, and to the thrill of working on open problems that can give you (great) sleepless nights.

Finally, I would like to acknowledge how grateful I am for having a group of people in my life who are always ready to listen to my rants about research with a smile on their face. Sravya, Xiaoyang, Ramya, and Arun: thank you for all the wonderful conversations and for boosting my motivation by sharing your own Ph.D. stories with me. Ben: thanks for helping me put things in perspective. Thalia: it's finally done! I'm glad we kept in touch after the Tech Venture Formation class, our conversations about life, startups, poetry, and everything under the sun made my life much more colorful. A huge huge thank you to my sister, Betty, thank you for more than 65,000 minutes of pep talk and wisecracks over FaceTime over the last six years. As you know, all of it was completely unsolicited but absolutely essential to help me finish. Last but not least, I would like to thank my parents, to whom I dedicate this thesis. Thank you for believing in me more than I believe in myself. Thank you for encouraging me to make crazy life decisions, such as coming to the United States for college, all by myself, and then embarking on a roller-coaster journey that is a Ph.D., decisions that left me wiser and more grateful than ever before—for all the lessons I have learned, for all the mentors I have met, and for all the friends I have made.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 The Congestion Control Problem 2
1.2 The Solution Landscape 3
1.3 Distributed Algorithms and Dependency Chains 5
1.4 Why PERC Algorithms Converge Quickly 5
1.5 Why PERC Algorithms Now? 7
1.6 Why Do We Need New PERC Algorithms? 8
1.7 Overview of s-PERC 9
1.8 High-Speed Programmable Switches 10
1.8.1 Practical Considerations at High Speeds 11
1.9 Related Work 12
1.10 Outline 13
1.11 Previously Published Material 14

2 *Fair*: A PERC Algorithm with Per-Flow State **15**

2.1 Proactive (PERC) Algorithms 15
2.1.1 Model 16
2.1.2 Max-Min Fairness 17
2.2 *Fair*: a Max-Min PERC Algorithm with Per-Flow State 20
2.2.1 Description 20
2.3 The *Fair* Algorithm in Action 22

2.4 Properties of the *Fair* Algorithm . . . 23
2.5 Convergence of the *Fair* Algorithm . . . 24
2.6 A Tighter Convergence Bound for the $Fair$ Algorithm . . . 28
2.6.1 Invariants of $Fair$ based on the Induction Order . . . 28
2.6.2 The CPG Algorithm to Find Max-Min Fair Rates . . . 29
2.6.3 Dependencies from the CPG Algorithm . . . 31
2.7 Simulation Results . . . 32
2.7.1 Convergence Times . . . 33
2.7.2 Flow Completion Times . . . 34
2.7.3 Dependency Chains . . . 36
2.8 Problem with the *Fair* Algorithm . . . 38
3 s-PERC: A PERC Algorithm that Does Not Need Per-Flow State 39
3.1 n-PERC: a Naive PERC Algorithm Without Per-Flow State . . . 39
3.1.1 The n-PERC Algorithm in Action . . . 41
3.1.2 Transient Problems With n-PERC . . . 43
3.2 s-PERC: a Stateless PERC Algorithm with a Known Bounded Convergence Time . . . 44
3.2.1 Variables in the s-PERC Algorithm . . . 44
3.2.2 The s-PERC Algorithm . . . 46
3.2.3 When Should We Propagate the Bottleneck Rate? . . . 49
3.2.4 How Do We Approximate the Maximum $\hat{E}$ Allocation? . . . 50
3.2.5 s-PERC in Action . . . 53
3.3 Simulations of n-PERC and s-PERC . . . 55
3.4 Convergence Proof of s-PERC . . . 56
3.4.1 Centralized Water-Filling Algorithms . . . 56
3.4.2 Partitioning Bottleneck Links Using WF^{k} Algorithms . . . 57
3.4.3 Why Do We Need Different WF^{k} Algorithms? . . . 61
3.4.4 Using WF^2 to Reason About s-PERC . . . 62
3.4.5 Properties of the WF^2 Algorithm . . . 63
3.4.6 Invariants of s-PERC Based on the WF^2 Order . . . 67
3.4.7 Why Limit Rates of Flows Are at Least $\mathrm{R}(l)$ at Link l and Neighbors . . . 70
3.4.8 Why Flows Are Updated Correctly at Bottleneck Links . . . 71
3.4.9 Why Flows are Updated Correctly at Non-Bottleneck Links . . . 72

3.4.10 Dependencies from the WF^2 Algorithm . . . 74

4 Evaluating s-PERC for Data Centers **80**

4.1 Practical Design Considerations . . . 80

4.2 Hardware s-PERC Prototype . . . 86

4.2.1 NetFPGA s-PERC Switch . . . 86

4.2.2 MoonGen s-PERC End Host . . . 87

4.3 Evaluating s-PERC* for Data Centers . . . 87

4.3.1 Convergence Times . . . 87

4.3.2 Flow Completion Times . . . 88

4.3.3 Micro-Benchmarks . . . 92

4.4 NetFPGA Hardware Evaluation . . . 93

5 Future Work on PERC Algorithms **97**

5.1 Future Avenues for Research . . . 97

5.2 Summary . . . 100

6 Enabling Other Algorithms in Programmable Switches **101**

6.1 Introduction . . . 101

6.1.1 Packet Processing Languages . . . 103

6.1.2 Characteristics of Switches . . . 104

6.1.3 Approach and Contributions . . . 105

6.2 Problem Statement . . . 106

6.2.1 Table Dependency Graph . . . 106

6.3 Target Switches . . . 109

6.4 Integer Linear Programming . . . 111

6.4.1 Common Constraints . . . 111

6.4.2 Objective Functions . . . 112

6.4.3 Switch-Specific Constraints . . . 113

6.5 Greedy Heuristics . . . 115

6.5.1 Ordering Tables . . . 116

6.5.2 Single-Metric Heuristics . . . 117

6.5.3 Multi-Metric Heuristics . . . 117

6.6 Experiments . . . 118

6.7 Analysis of Results 121
6.7.1 ILP vs Greedy 121
6.7.2 Comparing Greedy Heuristics 122
6.7.3 Sensitivity Experiments 124
6.8 Related Work on Compilers 125
6.9 Summary 126

A Supplementary Material for Chapter 2 (*Fair*) 127
A.1 $Fair$ v/s d-CPG 127
A.2 Dependencies from the CPG Algorithm 128

B Supplementary Material for Chapter 3 (s-PERC) 130

Bibliography 132

List of Tables

1.1 Initial control packet for a flow in s-PERC. 10

2.1 Commonly used notation. 18
2.2 Initial Control Packet for Flow f_G in $Fair$. 20

3.1 Initial Control Packet for Flow f_G in n-PERC. 40
3.2 Initial Control Packet for Flow f_G in s-PERC. 45
3.3 Commonly used notation in the context of a WF^{k} algorithm. 57

4.1 Initial Control Packet for Flow f_W in s-PERC*. 84
4.2 Results for incast experiments using a NetFPGA test bed. 95

5.1 Convergence times for RCP and s-PERC in a WAN topology. 98

6.1 Mapping switch compiler dependencies to traditional compiler dependencies. 108
6.2 Logical program benchmarks for RMT and Flexpipe. N is the number of tables. 119
6.3 Benchmark results for 5-stage FlexPipe. 120
6.4 Benchmark results for RMT for L2L3-Complex. 120
6.5 Benchmark results for 32-stage RMT for L2L3-simple, L2L3-Mtag, and L3DC. 120

List of Figures

1.1 The congestion control problem. 2

1.2 Reactive control system. 3

1.3 Example topology, workload, and convergence behavior. 6

2.1 Example setup. 18

2.2 *Local* max-min fair calculation at link l for flow f. 22

2.3 Control packet updates for the first two rounds of the $Fair$ algorithm. 23

2.4 Example of CPG Algorithm for $Fair$. 32

2.5 Setup for convergence time experiments. 33

2.6 CDF of convergence times on an 8-link topology with 100 Gb/s links. 34

2.7 Spread FCTs on 10 Gb/s link with 60% load and 12 μs RTT. 35

2.8 Spread FCTs on 100 Gb/s link with 60% load and 12 μs RTT. 36

2.9 Spread FCTs on 10 Gb/s link with 60% load and 120 μs RTT. 36

2.10 Spread FCTs on 100 Gb/s link with 60% load and 120 μs RTT. 37

2.11 A dependency chain of three. 37

2.12 Time series of flow transmission rates in 3-level-bottleneck scenario. 38

3.1 Example setup for n-PERC in action. 42

3.2 Control packet updates for first three rounds of the n-PERC algorithm. 42

3.3 Example setup for s-PERC in action. 53

3.4 Control packet updates for first three rounds of the s-PERC algorithm. 53

3.5 An example for which n-PERC takes a long time to converge. 56

3.6 Three different WF^{k} algorithms, their respective partitions, and dependency graphs. 61

3.7 Examples of *direct* precedent links based on the WF^2 algorithm. 76

3.8 Examples of *indirect* precedent links based on the WF^2 algorithm. 78

4.1 CDF of convergence times. 88
4.2 FCT results at 60% load for search, and data-mining workloads. 90
4.3 FCT results at 80% load for search workload. 91
4.4 FCT results at 80% load for data-mining workload. 92
4.5 CDF of convergence times for for approximate division. 94
4.6 The topology and traffic pattern used for the two-level dependency chain experiment. 94
4.7 Example convergence behavior for TCP, DCTCP, and s-PERC running on a NetFPGA test bed. 96

6.1 A top-down switch design. 102
6.2 Compiler input and output. 103
6.3 A packet processing program named L2L3 describing a simple L2/L3 IPv4 switch. 105
6.4 TDG for the L2L3 program. 107
6.5 Switch configurations for RMT and FlexPipe. 108
6.6 Block layout features in different switches. 110
6.7 Dependency types and latency delays in RMT. 111
6.8 FlexPipe table sharing in detail. 114
6.9 Multiple-metric heuristics. 116
6.10 Greedy performing much worse than ILP (RMT.) 118
6.11 Greedy performing much worse than ILP (FlexPipe.) 118
6.12 FFL-16 and ILP solutions for L2L3-Complex. 123
6.13 FFLS and FFL solutions for L2L3-Mtag. 124

A.1 Sequence of updates in the CPG algorithm where it takes 1.5 round-trips for new information to propagate. 128
A.2 Dependent links in CPG that are not precedent links. 129
A.3 Dependent links in CPG that are not precedent links. 129

Chapter 1

Introduction

Congestion control algorithms allow different applications to share network bandwidth efficiently without oversubscribing any link. Congestion happens when a network link is oversubscribed by multiple applications: the buffers fill up, packets are dropped and retransmitted, and applications see a spike in latency or a drop in goodput. User-facing applications such as search and interactive web services care about latency, while back-end applications like e-mail backups care about goodput. There are different ways of sharing the network bandwidth to further adapt to the application mix; these correspond to different objectives for congestion control algorithms (max-min fairness, shortest-flow-first, etc.). Congestion control is important when the network is heavily loaded—that is, when bandwidth demands are high and there are always multiple applications that want to use the same links simultaneously. This thesis is motivated by the need for *fast congestion control*, by which we mean three things.

First, we want to control congestion even before it happens. This is important when the network speed or round-trip delay is high and buffers fill up quickly even before there is time to react. Consider two flows with 100 μs RTTs that share a single 100 Gb/s bottleneck link with 125 KB of buffering. It only takes 10 μs of line rate traffic to fill the buffers, whereas the servers can adjust their rates only at 100 μs time-scale, which is how long it takes to measure and react to congestion. When the reaction time is long, relative to the buffer size, it becomes hard to control congestion reactively. As the RTTs get longer or the link capacity gets higher, as in a WAN, without a corresponding increase in the buffer size, this problem only gets worse.

Second, we want to control congestion quickly relative to the lifetime of the network flows. For example, if most flows are short enough to finish in a few round-trip times (RTTs), a congestion control algorithm must quickly find a way to share the network bandwidth amongst all the flows, within this time budget. This is especially relevant in data centers today—a typical flow in a search workload would finish in less than 10

RTTs in an unloaded 100 Gb/s network.[1] Even if there are a few contending flows, it should still be able to finish in a few dozens of RTTs. As the link capacities increase and more flows become short-lived, fast congestion control becomes more important.

Finally, the time that it takes to control congestion should be independent of how dynamic the network traffic is—that is, a congestion control algorithm should be able to respond quickly to flash crowd scenarios, where a large number of flows start at once, or ON/OFF traffic patterns, where the network utilization varies between 1 and 0, just as quickly as it would given more stable workloads (such as Poisson arrivals at constant load.) As applications become more distributed, it is not uncommon to see such traffic patterns in data centers because of tightly synchronized network traffic between different servers. Data-center applications that stand to gain from fast congestion control include disaggregated storage and large-scale data processing, which demand high throughput and low latency and often involve network traffic between tightly coupled servers (or storage devices) where a large number of flows start at once.

In this thesis we explore a new class of fast congestion control algorithms that network operators can deploy in order to use the network at high loads, even as link speeds increase to hundreds of Gb/s. We call these "Proactive Explicit Rate Control" algorithms, or PERC algorithms. As we will see, PERC algorithms quickly converge to their objective even before congestion happens. This allows them to keep buffers close to empty while using the links efficiently, which in turn helps applications get low latency and high goodput in general.

1.1 The Congestion Control Problem

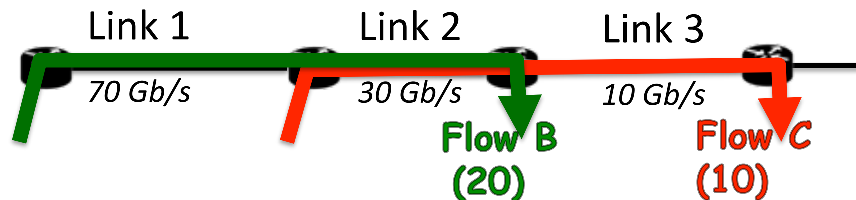


Figure 1.1: The congestion control problem. Ideal flow rates in parenthesis.

At its core, the congestion control problem is as follows. We are given a network with links of different capacities and a set of flows, where each flow crosses a subset of the links. We need to find a rate allocation for each flow that is fair, for some notion of fairness, and use link capacity efficiently. Figure 1.1 shows

[1] A 1 MB flow would be larger than about 75% of all flows in a search workload[12] and finish in 80 μs if sent at 100 Gb/s, which is roughly 8 RTTs, assuming an RTT is 10 μs.

a network with two long-lived flows and three links, as well as a set of rates for the flows that satisfy the max-min fair objective. Flow C is bottlenecked to 10 Gb/s at Link 3, while Flow B is bottlenecked to 20 Gb/s at Link 2. If the flows are sent at these rates, they use Links 2 and 3 fully without causing congestion. Moreover, the applications sending these flows see high goodput, while any short flows (e.g., < 1 RTT) that also use the links see low latency. The question is therefore this: how do we figure out these rates?

There is a simple solution: Suppose we have a centralized controller that has full knowledge of the traffic matrix (i.e., which links are used by which flows and the individual link capacities). Then, we can run a simple program (e.g., the water-filling algorithm [22] for max-min fair rates, NUM for α-fair rates, or a greedy algorithm [19] for SRPT schedules) at the controller to calculate the ideal rates and communicate these rates back to the servers that send the flows. Although this solution is conceptually simple, it is hard to implement at scale and is therefore hardly used in today's networks.

1.2 The Solution Landscape

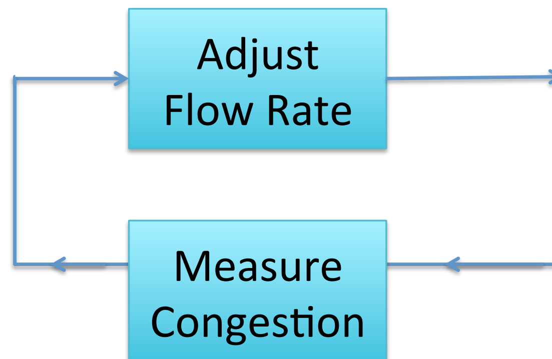


Figure 1.2: Reactive control system.

Reactive control-systems: Let's contrast the simple solution with a majority of existing algorithms (e.g., TCP [14], BBR [26], DCTCP [12], TIMELY [62], RCP [36], or XCP [53]). Almost all existing algorithms are completely distributed, either running at the servers or running at the servers and switches. However, the more important point of contrast is that the algorithms do not use explicit global information. That is, neither the servers nor the switches use any explicit knowledge of the traffic matrix or the link capacities. Instead, the servers independently start with arbitrary rates (or window sizes) and adjust these rates iteratively based on congestion signals. The congestion signals come in a variety of forms, ranging from packet drops and congestion markings (ECN), end-to-end delays measured at the end hosts as in TCP, DCTCP, or TIMELY, to traffic arrival or queuing rate measured at the individual links, as in RCP or XCP.

These algorithms, although seemingly very different, are in essence reactive control systems. They measure a congestion signal and react to it by adjusting the window size (TCP, DCTCP) or an explicit rate (TIMELY, RCP, XCP). This feeds back into the system, generating new congestion signals and so on (Figure 1.2). Because reactive control systems do not need to know anything about the network (size, capacity, etc.) or the number of flows, they are robust to a variety of operating conditions. However, this comes at a huge cost.

Reactive congestion control systems have two well-understood disadvantages. First, the control system needs to cause congestion in order to control it, which means we are constantly worried about queue occupancy, increased latency for short flows, and buffer overflows. Second, the control loop can only make small adjustments during each RTT to prevent the control loop from becoming unstable. Proceeding in small steps means the control system is always playing catch-up with current network conditions. As networks get faster, and more data fit into a few RTTs, more and more flows will finish before the congestion control loop has time to react, rendering the approach less useful in the future, particularly for short flows.

PERC algorithms: Notice that in the centralized solution, we decouple the rate calculation from the actual data traffic. We do not need to send data traffic or cause congestion in order to figure out the ideal rates. We calculate the ideal rates before the flows even start. We call this a centralized proactive approach. Flowtune [70], a recent proposal, implements such a system, where a central rate allocator receives requests from end hosts, calculates feasible and close-to-ideal rates, and communicates these rates to the servers. The rate allocator does the rate calculation again each time a flow departs or a new flow joins. Flowtune and proactive algorithms, in general, try to avoid congestion in the first place by scheduling rates for flows (or departure times for packets, as in Fastpass [71]) that do not depend on congestion signals, but instead fit within the known constraints of the network link speeds and topology.

Centralized proactive approaches like Flowtune are problematic at scale because the computation and bandwidth at the central allocator become bottlenecks. In this thesis, we focus on *distributed proactive* rate allocation algorithms, which we call Proactive Explicit Rate Control (PERC) algorithms. PERC algorithms distribute the rate calculation amongst the different links[2] and end hosts. Whenever a flow arrives or departs, the switches and end hosts exchange short messages, running a distributed algorithm to explicitly allocate a rate to each flow, including ongoing flows, to fit within the capacity constraint of the links. The end host then sends flows at the allocated rate. We use “data traffic" to refer to the flowsâĂŹ packets on the network and the term “control packets" to refer to the short messages that are exchanged to calculate rates. Unlike a reactive control system (such as RCP), a PERC algorithm does not depend on measuring congestion in order

[2]For each link, the link-specific calculations are performed by the switch at the head of the link, but we will refer to the link, rather than the switch, as a distributed agent for convenience.

to iteratively adjust rates. The distributed algorithm implemented by the control packets is decoupled from the data traffic because it is a rate calculation based on the set of flows and the network constraints, rather than any congestion signal. As a result, it can converge to the right rate allocation quickly.

1.3 Distributed Algorithms and Dependency Chains

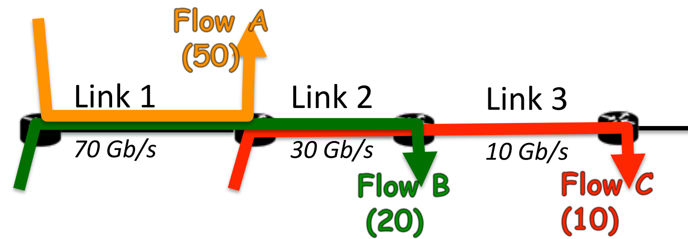
Note that because the rate calculation is distributed, we will need multiple rounds of communication amongst the switches in order to propagate bottleneck information from one link to another. Consider Figure 1.1 again. Flow C is bottlenecked to 10 Gb/s at Link 3, while Flow B is bottlenecked to 20 Gb/s at Link 2. If we consider a distributed rate calculation from a link's point of view, Link 2 cannot compute a bottleneck rate of 20 Gb/s for Flow B unless it knows that Flow C is limited to 10 Gb/s at Link 3. This is a coupling or *dependency* between the two links because they carry a common flow. We will rigorously describe different kinds of dependencies in the context of max-min fair rates in §3.4.10. For now, we say that link l depends on link k if they carry a common flow and the flow is bottlenecked at link k; for instance, Link 2 depends on Link 3 because they carry a common flow, C, which is bottlenecked at Link 3.

Such dependencies can span many links to form a *dependency chain* [76]. Typical heavily loaded data center topologies have dependency chains that span dozens of links. Any distributed rate calculation algorithm is fundamentally limited by the time (e.g., 1 RTT) it takes to carry bottleneck information from one link to another in a dependency chain [76]. In terms of convergence speed, a PERC algorithm is *only* limited by the dependency chain, whereas a reactive control system is limited by the dependency chain *and* the slow control loop. For a control system, it takes time to measure congestion in each control loop, and it takes many small adjustments over multiple control loops to move toward the fixed point in a stable way. Therefore, a PERC rate calculation should converge more quickly than a reactive control system. In the next section, we will walk through a simple example with a longer dependency chain to contrast the two kinds of convergence behavior.

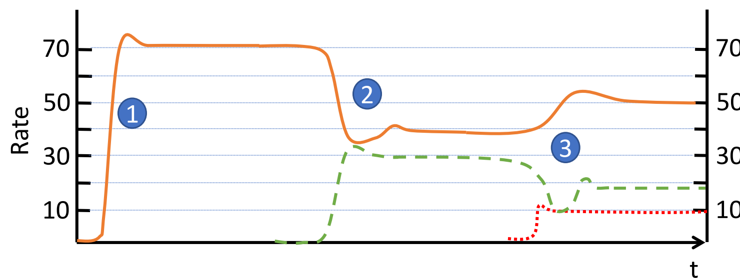
1.4 Why PERC Algorithms Converge Quickly

The lower bound on the convergence time for a distributed max-min fair algorithm is dictated by the longest dependency chain [76] and is key to understanding how PERC algorithms work.

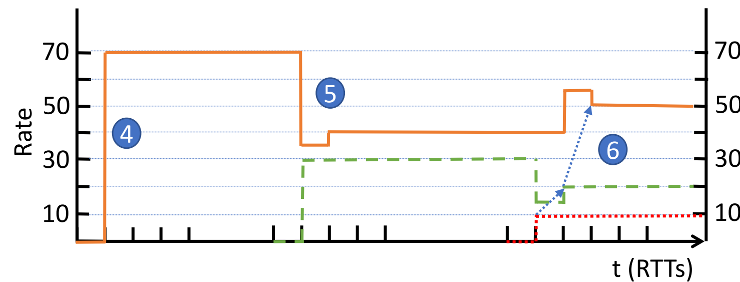
Figure 1.3a shows our original network with three flows. Flows B and C are still bottlenecked to 20 Gb/s and 10 Gb/s, respectively, while Flow A is bottlenecked to 50 Gb/s at Link 1. This setup with three flows has a longer dependency chain. Link 2 still depends on Link 1 because they carry Flow C, bottlenecked at Link



(a) 3-Level Dependency Chain Example.



(b) Example reactive algorithm rate behavior.



(c) Example PERC algorithm rate behavior.

Figure 1.3: Example topology, workload, and convergence behavior.

1. In addition, Link 3 depends on Link 2 because they carry Flow B, bottlenecked at Link 2. Hence, we have a dependency chain that spans all three links.

We will examine how reactive algorithms and PERC algorithms behave when flows A, B, and C are added one at a time. Note that we do not assume a particular reactive algorithm here, but just the general class, represented by, say, TCP, RCP, or DCTCP. The actual rates and shape of the convergence will depend on the algorithm. But for our purposes here, we treat them as a single class.

Figure 1.3b sketches how flow rates adapt with a reactive scheme. When flow A starts on Link 1, it ramps up to fill the link capacity 70 Gb/s; this is illustrated in event 1 in Figure 1.3b. When flow B is added (event 2), it is bottlenecked to 30 Gb/s by Link 2, leaving 40 Gb/s of capacity at Link 1 for flow A. Initially, however,

flow A moves toward a fair-share rate of 35 Gb/s on Link 1, before flow B finds its rate of 30 Gb/s on Link 2. Looking at it from the links' perspective, as Link 1 begins to indicate its fair-share rate to flow B, the bottleneck of flow B shifts to Link 2. While the flows are finding the correct rates, there is a dependency: flow A cannot converge before flow B. Notice that the dependency chain is a function of the set of flows and the network. At this point in time, with only flows A and B in the network, there is only a single dependency. The situation gets more complicated when flow C is added. Flows A, B, and C eventually converge to 50 Gb/s, 20 Gb/s, and 10 Gb/s, respectively. Flow C is bottlenecked to 10 Gb/s by Link 3, leaving 20 Gb/s spare capacity for flow B at Link 2, which leaves a spare capacity of 50 Gb/s for flow A at Link 1. The dependency chain is that Link 2 cannot determine the spare capacity until Link 3 has converged. Similarly, Link 1 cannot converge before Link 2 has converged.

This is an example of a three-level dependency chain that spans three links. How quickly the algorithms will converge on the final rates depends on how quickly they detect the congestion (e.g., TCP will detect a drop or duplicate ACK; DCTCP a queue threshold crossing and ECN markings; RCP a different fair-share rate); but all algorithms take a sequence of steps toward the stable operating point. As we will see later, this is typically over 100 RTTs, even for relatively simple topologies and traffic patterns.

In contrast, a PERC algorithm running in this topology can converge in three RTTs. Figure 1.3c illustrates the sequence, exposing the dependency chain more clearly. When flow A starts, it tells the switches, and the Link 1 switch can immediately tell it to send at 70 Gb/s; this is illustrated in event 4 in Figure 1.3c. When flow B starts, Link 1 and Link 2 update their fair-share rates to 35 Gb/s and 30 Gb/s, respectively, which is just their capacity divided evenly among the active flows. As a result, flow A's rate drops from 70 Gb/s to 35 Gb/s, and flow B is told the (final) rate of 30 Gb/s from Link 2. When this new rate is seen by Link 1, the link knows that flow B is limited at another link and can update its own fair share to 40 Gb/s (event 5). This new fair share can be picked by flow A 1 RTT after flow B is updated.

When flow C starts, a slightly longer chain of updates happens. First, flow C is told to send at 10 Gb/s by Link 3, and at the same time Link 2 gives flow B a fair-share rate of 15 Gb/s. One RTT later, Link 2 learns that flow C is bottlenecked elsewhere to 10 Gb/s and therefore increase its fair-share rate from 15 Gb/s to 20 Gb/s and tells flow B. Next, Link 1 learns that flow B is bottlenecked elsewhere to 20 Gb/s and will therefore decrease its fair-share rate from 55 Gb/s to 50 Gb/s (event 6).

1.5 Why PERC Algorithms Now?

Although a PERC algorithm was first proposed several decades ago for calculating a max-min fair-rate allocation [22], and many variants were explored for ATM virtual circuits in the 1990s [27], [76], [21], [11], it

was always considered too complicated to run in the network.

Three recent trends encourage a revival of research into PERC algorithms:

1. **Programmable switches**: Programmable switches make it possible to deploy and evolve new algorithms in switches (§1.8).

2. **Faster networks**: Faster networks, especially when coupled with small buffers, create a more pressing need for congestion control algorithms that avoid congestion rather than react to it (§1.2). This trend is particularly relevant to data center networks that have relatively small and fixed μs RTTs but whose network speeds have increased from 50x to 100x over the last ten years, driven by increasing demands from applications like disaggregated storage, large-scale data-processing, and interactive web service. Bandwidth per server is predicted to reach 100 Gb/s in the coming years.

3. **Data center build-out:** Cloud operators are building out even more data centers to be closer to the user and to comply with recent privacy laws that mandate that citizens' data stay within the country [81], [61], [20]. This calls for more distributed approaches for controlling the inter-data center traffic than those offered in existing solutions [56], [46]. PERC algorithms would allow network operators to use their expensive high-speed, high-RTT WAN links efficiently (at high loads) even as many more data centers are added to the network.

1.6 Why Do We Need New PERC Algorithms?

If a network could practically incorporate a PERC algorithm into its forwarding plane, then congestion could be more tightly controlled, without concern for stability. Before deploying such an algorithm in the network, the operator naturally expects that the algorithm demonstrably converges to the max-min fair rates in a known bounded time.

Unfortunately, all previous PERC algorithms with known convergence bounds [22], [76], [27], [21], [60] have limitations that make them impractical to deploy:

1. They require switches to either be synchronized[3] or to maintain state for individual flows, neither of which is scalable.

2. Almost[4] all of them allocate a max-min fair rate to each flow. While this makes sense for long-lived flows, we often want short-lived flows to have priority [45], [13], [43], especially in data centers.

[3] That is, the algorithm proceeds in iterations that have to be synchronized across all flows and switches.

[4] [60] allocates α-fair rates.

3. They do not address concerns about robustness (e.g., what happens if a scheduling message gets dropped, or if the rate calculations are imprecise?).

A recent PERC algorithm called PDQ [45] addresses some but not all of these issues. PDQ is asynchronous, aims to emulate shortest-job-first, and is shown to be robust. However, it needs to maintain state for individual flows.

Hence, we need new PERC algorithms for different objectives that provably converge in bounded time and are practical to deploy at high speeds.

1.7 Overview of s-PERC

In this thesis we introduce a practical PERC algorithm for max-min fair rates that can give priority to short flows as needed.

To the best of our knowledge, s-PERC ("stateless Proactive Explicit Rate Control") is the first practical PERC algorithm that converges to exact max-min rates in a known bounded time. This algorithm does not require switches to be synchronized or to maintain per-flow state, and it can be proved to converge in at most $6N$ RTTs (§3.4), where N is the length of the longest dependency chain. In addition, s-PERC can be implemented in hardware (e.g., a programmable switch) to run at 5 ns clock speeds (as evidenced by our 40 Gb/s NetFPGA prototype [85]), and we see no fundamental limitations that prevent it from running at 1 ns clock speeds (§4.2).

s-PERC runs in the switches and end-host NICs. The end-host NICs periodically send a control packet for each flow, which follows the same path as the flow for as long as the flow is active. The switches modify the control packets as they pass, but the switches do not maintain per-flow state. All flow-specific information is carried in the control packets.

A control packet is specific to a flow and carries four types of information for each link (Table 1.1). The fields are updated in the control packet as it passes each link: (1) Each link keeps a running estimate of its own bottleneck rate. (2) Links classify each flow depending on whether it is bottlenecked at this link or at another link; a flow may change classification as the algorithm converges. (3) Based on the classification, the link allocates bandwidth to the flow. (4) The link indicates its confidence in its fair-share rate. This is key to how s-PERC converges without per-flow state: if a link detects that its fair-share rate is too small, it alerts other links using the *ignore* bit. All four pieces of information are updated in control packets as they pass by. The end hosts use the allocations computed by the switches to adjust the rate at which they send the flows.

Table 1.1: Initial control packet for a flow in s-PERC. The control packet carries information for each link that the flow crosses. Each row corresponds to a link, while each entry in the row corresponds to different pieces of information about the link. For example, the first row corresponds to Link 1, and the first entry of the first row carries the bottleneck rate of Link 1, which is initially set to 0 at the end host.

Link	Bottleneck State (s) Allocation (a)	Bottleneck rate (b)	Ignore bit (ignore)
1	E @ 0	0	1
2	E @ 0	0	1

Deployment: Note that s-PERC is a clean-slate design and requires significant changes to existing commodity fixed-function switch chips, and programmable switches may provide an easier avenue for testing new algorithms like s-PERC. We believe that because s-PERC requires new protocols at both the servers and switches, it will initially be deployed in private networks where a single entity owns both the servers and switches. As we have seen with DCTCP [12], TIMELY [62], and BBR [26], network operators can and do deploy new congestion control algorithms in their data centers and private WANs to help their applications perform better. In this thesis, we will delve more deeply into the practical benefits of s-PERC for data center networks. We focus on data center networks because this is where most research is happening today. There is nothing about s-PERC that makes it specific to data centers. On the contrary, given how well it works for long-lived flows, we suspect s-PERC would be very well suited to long-haul networks where there is a need for well-behaved, non-congested networks that are fair among flows.

In the next section, we'll briefly discuss programmable switches and their advantages and highlight some constraints of running an algorithm in a programmable switch (or any hardware pipeline) at 1 ns clock speeds.

1.8 High-Speed Programmable Switches

Almost all commodity switches from the late 1990s to the early 2010s were made of fixed-function switch chips. Functions or protocols like ACL lookup and IP routing were baked into the switch chip. This made it hard to evolve an existing protocol (e.g., switch to IPv6) or deploy new protocols (e.g., MPLS) at scale quickly. A programmable switch requires more logic and wiring than a fixed-function chip. However, the overhead is modest (e.g., [24] shows that a programmable 64x10 Gb/s 28 nm switch chip imposes an area and power overhead of less than 15% over commodity fixed-function switch chips). Because transistors have gotten smaller, one can fit more logic in the same area, and the price of programmability has decreased. As a result, switches that run at Tb/s speeds and use programmable switch chips [2], [1] are already available. Such switches would allow network operators to implement new features and protocols easily and iterate

more quickly.

To understand the difference between a fixed-function switch and a programmable switch, we can view the fixed-function switch chip as a pipeline of stages, where each stage performs a fixed function. For example, a typical fixed-function switch has a Layer-2 switching function, followed by Layer-3 routing, followed by access control. Furthermore, each fixed function can be viewed as a match-action table; it matches on a fixed field of the packet header (e.g., match src and dst IP address) and then performs a fixed action (e.g., decide egress link).

In contrast, a programmable switch involves a pipeline of stages in which the memory and computation resources in each stage can be reallocated across different functions. This is possible because each stage is programmable: it contains modular memories that can be used to implement look-up tables that match any part of the header, and it contains modular action units (such as addition and subtraction) that modify different parts of the packet header simultaneously. Each stage also contains stateful registers that persist across packets and can be read/modified by the action units.

Together, the modular memories and action units provide the abstraction of a sequence (or tree) of match-action tables, each of which can be configured to match an arbitrary fixed part of the header and perform actions that modify the packet or stateful registers. In this abstraction, each tableâĂŹs action may actually be a composition of many primitive actions applied one after another.

Programming languages such as P4 provide a way to define custom packet headers and express the switch-chip logic using the abstraction of match-action tables that modify the packet header.

To implement a PERC algorithm such as s-PERC in a programmable switch, one would decompose the switch local algorithm into a series of match-action tables (possibly with trivial matches) that read or modify the control packet and stateful registers. As we will see later, match-action tables can be used to implement intermediate steps such as an approximate division.

A compiler figures out how to map the P4 program to a set of actual match-action tables [51] and sequence of primitive actions [80] that fit within the constraints (e.g., number of stages, size, and width of memories) of the target switch pipeline. Towards the end of this thesis, after we have discussed PERC algorithms, we will dig deeper into the problem of compiling general P4 programs to a programmable switch.

1.8.1 Practical Considerations at High Speeds

Although many distributed algorithms can be written as a P4 program, not all P4 programs can fit a given target switch. A P4 program must be compatible with the strict resource constraints imposed by the high-speed switch.

Memory and computation are obvious constraints. Look-up tables may be too large to fit in the switch memory. The computation (actions) may require more pipelined stages than are available in the switch. There may be a limit on the number of header fields that can be parsed at line rate. There may also be architecture-specific constraints (e.g., in some architectures, the stateful registers are private to a pipeline stage) [1]. Consequently, an innocuous step such as $r = r/(r-1)$, where r is a local register, would be infeasible because the new value of r depends on a computationally expensive function of the old value, which cannot fit in a single stage (or clock cycle). Such constraints may apply to both programmable switches and custom fixed-function pipelines that need to run at high line rates.

In §4.2, we discuss how we make s-PERC practical for Tb/s speeds given such constraints. Our P4→NetFPGA [85], [6] hardware prototype proves that it is easy to implement s-PERC in hardware at 4x10 Gb/s.

1.9 Related Work

Distributed proactive scheduling algorithms were first explored for ATM networks. Some schemes provably converge to the exact max-min rate allocation in a known bounded time [27], [76], [21], [11], [64], but all previous algorithms with bounded convergence time required per-flow state or synchronization [44], [48], [39], [60], [38], [11], [41], making them impractical in today's networks.

For example, in 1998, the authors of [17] proposed an approach without per-flow state that converges, but not necessarily to the exact max-min fair allocation. In 2008 the authors of [33] showed that an algorithm without per-flow state can stabilize, but they did not find an upper bound on the time it takes. Other schemes without per-flow state are not proved to converge [63], [54], [73], [52]. Ros-Giralt and Tsai [76] used the theory of dependency chains to establish a lower bound on the convergence times of a class of max-min fair schemes, which use per-flow state, and proposed a scheme that converges within twice the lower bound. Chrysos and Katevenis [30] use the theory of dependency chains to establish the stabilization time of a buffered cross-bar using a weighted fair-queuing (scheduler), following changes in offered load or weights. Their characterization of dependency chains, namely as a chain of flows in non-decreasing order of max-min rates, follows directly from Ros-Giralt and Tsai's definition of precedence links and the resulting chain of precedence links in a constrained precedence graph (CPG).

s-PERC builds upon [21], removing its dependence on per-flow state. Our opinion is that before deploying a new algorithm, an operator would require no per-flow state but would require proof the algorithm will converge in bounded time.

For data centers: pFabric [13] and PIAS[18] leverage priority queues in switches to approximate SRPT or SJF. The approach is complementary to s-PERC, and we borrow it for short flows.

PDQ [45] and D3 [83] are proactive congestion control algorithms, although their goal is to minimize mean FCT and the number of missed deadlines by dynamically prioritizing some flows over others. PDQ [45] is particularly interesting— it can emulate centralized scheduling algorithms such as shortest-job-first and earliest-deadline-first. PDQ requires some per-flow state for the top active flows, and in a general setting this state could be very large.

PASE [65] is a hybrid scheme, where flows get an initial rate from a software controller at the TOR switch and then switch to a reactive scheme.

NDP [43] is a receiver-driven transport protocol, where the sender's flow rate is dictated by the receiver, based on the sender's demand as well as the over-subscription at the receiver. s-PERC can complement NDP by providing explicit flow rates that also take into consideration hot spots *within* the network, not just at the edge.

ExpressPass [29] is a credit-based congestion control scheme that uses credit packets that traverse the same path as the data traffic, to aim for fast convergence. The difference is that in s-PERC the control packets are used by the network to calculate explicit rates directly, while in ExpressPass the credit packets must themselves go through a reactive but relatively aggressive feedback control loop in order to converge to credit rates that the data traffic can use.

1.10 Outline

PERC algorithms are based on the insight that we can calculate ideal rates quickly if we decouple the rate calculation from the actual data traffic and use explicit information (e.g., set of flows, network constraints) instead of congestion signals. In this paper, we assume the ideal rate allocation is the max-min fair allocation. The remainder of the thesis is organized as follows:

Chapter 2. Fair: A PERC algorithm with a per-flow state. In §2.1, we define explicitly what we mean by PERC algorithms and review max-min fairness for networks. We consider various PERC algorithms in this thesis in order to motivate s-PERC. We start with the *Fair* PERC algorithm in §2.2, which is able to converge to a global max-min fair rate allocation based only on *local* max-min fair calculations of the bottleneck rate at each link. While *Fair* is simple and fast, the local max-min fair calculation requires a per-flow state at the link, which does not scale to meet our needs.

Chapter 3. s-PERC: An algorithm that does not need a per-flow state. Our first attempt to get rid of the per-flow state is a simple algorithm called n-PERC, or "naive" PERC, in §3.1, where we replace the per-flow state of *Fair* with a few aggregate variables. As the name suggests, n-PERC has transient problems with the bottleneck rate calculations, and we cannot bound its convergence time. This leads us to s-PERC, or

"stateless" PERC, in §3.2, which overcomes the transient problems of n-PERC and converges in a provably bounded time. The trick is to recognize transient bad bottleneck rates and not propagate them. In §3.4, we present the formal proof of convergence for s-PERC.

Chapter 4. Evaluating s-PERC for data centers: In §4.1, we explain some design choices to make s-PERC practical and deployable and describe our 4x10 Gb/s NetFPGA prototype. In §4.3, we present our simulation results for s-PERC, particularly flow completion times (FCTs) in §4.3.2, compared to RCP, p-Fabric, and an ideal max-min rate allocator. We find that s-PERC is very close to an ideal max-min rate allocator for medium to large flows and provides the shortest flow completion times possible to the smallest flows. In §4.4, we describe real measurements of s-PERC, TCP, and DCTCP from our 40 Gb/s NetFPGA test bed. We find that for small incasts and traffic matrices with multiple bottlenecks, s-PERC converges 40–150 times faster than DCTCP.

Chapter 5. Future work In §5.1, we describe open questions and avenues for future research in PERC algorithms. We briefly discuss how s-PERC might be applicable in other networks such as private WANs or adapted for other objectives.

Chapter 6. Enabling other algorithms in programmable switches: Lastly, in §6, we dive into programmable switches and describe a compiler that can enable other algorithms such as s-PERC to run on a programmable switch. We describe how to map P4 programs (without state) to a programmable switch pipeline using the concept of a Table Dependency Graph and tools such as greedy heuristics and ILP.

1.11 Previously Published Material

- Chapter 2 revises a previous publication [50]: Jose, Lavanya, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. "High speed networks need proactive congestion control." In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, p. 14. ACM, 2015.
- Chapter 6 revises a previous publication [51]: Jose, Lavanya, Lisa Yan, George Varghese, and Nick McKeown. "Compiling Packet Programs to Reconfigurable Switches." In *NSDI*, pp. 103-115. 2015.

Chapter 2

Fair: A PERC Algorithm with Per-Flow State

2.1 Proactive (PERC) Algorithms

PERC ("Proactive Explicit Rate Control") algorithms figure out the max-min fair rate allocations by *proactively* exchanging messages (control packets) with switches along the path, over multiple rounds. The flow allocations are carried as *explicit rates* in the control packets. In this thesis we consider PERC algorithms for the max-min fairness objective. We first describe a set of standard assumptions about the setup for PERC algorithms and review the notion of max-min fairness, which allows us to prove properties about how and when different PERC algorithms will converge. In this chapter, we consider a simple PERC algorithm called $Fair$, which performs local max-min fair calculations at every link (based on "demands" of flows) in order to converge to the exact global max-min allocation. $Fair$ follows from an existing algorithm, called d-CPG, by Ros-Giralt et al. from 2001 [76].[1] $Fair$, however, requires per-flow state at the links and is not practical for networks with a large number of flows. Most of the memory in a switch is devoted to routing and forwarding while memory available for other functionality is at a premium. Moreover, it is hard to manipulate a large amount of memory at nanosecond time-scales, which is the norm for high-speed switches today. A practical solution should be able to scale to any number of flows, and not worry about per-flow state management.

[1] There are minor differences between $Fair$ and d-CPG, which we describe in section A.1 in the Appendix. These are related to assumptions about the setup and an edge case in the local switch algorithm. We use $Fair$ instead of the existing algorithm because it is easier to contrast with other PERC algorithms we consider.

2.1.1 Model

We make a set of simplifying assumptions about the setup for PERC algorithms. We can show that two PERC algorithms we consider— $Fair$ and s-PERC— can converge to exact max-min rates in $4N$ and $6N$ rounds (1 round ≈ 1 RTT), respectively, where N is the number of bottleneck links in the longest dependency chain. Some assumptions are stricter than others (e.g., the set of flows is fixed, a control packet is never dropped). In §4.1 we will describe how to relax the stricter assumptions (marked $*$) for a practical deployment (of s-PERC).

1. We are given an arbitrary network topology in the form of a graph, $G(V, E)$, where V is the set of switches and E is the set of links. The topology is fixed, and the links have fixed capacities defined by $\mathrm{C}\colon E \to \mathbb{R}_{>0}$.

$*$ 2. The network carries a fixed set of flows, F, where each flow $f \in F$ traverses a subset P_f of the links and each link $l \in E$ carries a subset Q_l of the flows. We use $J = |F|$ to refer to the number of flows and $K = |E|$ to refer to the number of links. We use the shorthand $\mathrm{N}(l) = |Q_l|$ to refer to the number of flows carried by link l.

$*$ 3. Data packets of flows are sent from a sender end-host to a receiver end-host at rates $\mathrm{X}\colon F \to \mathbb{R}_{\geq 0}$, where the rate of a flow $f \in F$ is limited only by the capacity of links in P_f. Hence, for all links l, we must have $\sum_{f \in Q_l} X(f) \leq \mathrm{C}(l)$. However, there are no limits imposed at the source or destination of the flows.

4. Each flow has exactly one outstanding control packet. The set of fields in the control packet vary from one algorithm to another. The control packet goes back and forth between the sender and the receiver and crosses the same set of links P_f as the flow's data packets. The control packet is seen and updated at each link in P_f as long as the flow is active. At any time $t \in \mathbb{R}_{\geq 0}$, we will use the notation $x_f(t)$ to refer to instantaneous value that flow f's control packet carries in field x.

$*$ 5. A control packet is never dropped or garbled.

6. A flow's control packet starts at the sender at some time t_0 and thereafter sees a random delay $d \sim D$ at each hop as it goes back and forth between the sender and the destination. Hence, every flow's control packet is seen (and modified) at discrete times $t_n, n \in \mathbb{N}$, where $t_n = t_{n-1} + d$.

   We use T_f to refer to all the discrete times when a control packet from flow f is seen at some link on its path P_f. We use $T_{fl} \subset T_f$ to refer to the times when the control packet is seen at a specific link $l \in P_f$. Control packets from different flows can be seen in any order at each link.

7. We assume that there is a finite upper bound to the time it takes for every flow's control packet to be seen at every link on the flow's path at least once. We use the term *round* to refer to this upper bound. A typical value of round would be one round-trip time (RTT).

∗ 8. For s-PERC, we assume that such an upper bound *round* is known ahead of time.

Note that there is no coordination between the links, and the arrival order of control packets at the links does not matter. The state associated with a flow f and link $l \in P_f$ is modified each time the flow's control packet crosses the link, and control packets from different flows arrive at a link independently. This model has been used in several previous works [27], [76], [21] and is easier to implement in practice than alternate models such as [44], [48], [39], [60], [41] or [38], [11], in which the algorithm proceeds in iterations $t \in \mathbb{N}$, or in atomic steps, that need to be synchronized across switches and flows.

We will analyze the convergence time of algorithms in rounds. Notice that the definition of round in (7) also implies that for any link l we can expect to see at least one control packet from every flow $f \in Q_l$ in any interval $[s, s + round]$.

Additionally, in all the algorithms we introduce in this thesis, it happens to be the case that the control packet is only modified by the links; the end host does not modify the control packet, except to signal that a flow is starting or ending. One of the fields in the control packet is the bandwidth allocated to the flow by each link. The source end host simply updates the rate $\mathrm{X}(f)$ at which it sends data packets to match the smallest allocation carried in the latest packet and then reflects the control packet back as is.

2.1.2 Max-Min Fairness

We start with an example of a max-min fair rate allocation. Consider the setup in Figure 2.1. There are $J = 2$ flows and $K = 3$ links. The max-min fair allocation is 12 Gb/s for flow f_G, which is bottlenecked at link l_{12}, and 18 Gb/s for flow f_B, which is bottlenecked at link l_{30}. Link l_{20} has spare capacity since the only flow it carries, f_B, is bottlenecked at another link.

Definition: We say that flow f is bottlenecked at link l, we mean that the link l is fully used and the flow f gets the maximum rate of all flows in Q_l. We say that a rate allocation for the flows is max-min fair *iff* every flow is bottlenecked at some link l. We note that there always exists a unique max-min fair allocation [58].

We note that there are several equivalent definitions of max-min fairness in literature including the following by Bertsekas et al [22]: a feasible rate allocation X is max-min fair if and only if for any other feasible allocation Y, if $Y(f) > X(f)$ for some flow f, then there must exist some other flow f' such that $X(f') \leq X(f)$ and $Y(f') < X(f')$ (see [58] for proof that the two definitions are equivalent).

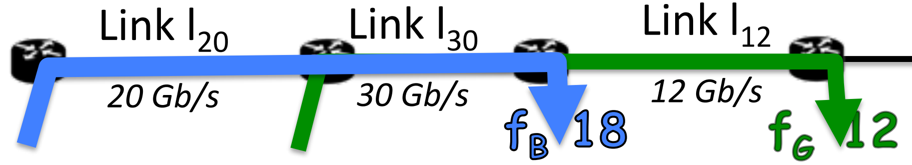


Figure 2.1: Example setup. There are $J = 2$ flows and $K = 3$ links. Each flow crosses a subset of the links. The green flow f_G is bottlenecked to 12 Gb/s at link l_{12}, while the blue flow f_B is bottlenecked to 18 Gb/s at link l_{30}. We show here the link capacities and the max-min fair rate allocations for the flows.

Table 2.1: Commonly used notation for a given set of flows and links. The second set of rows contain notation relevant to the max-min fair allocation for the given set of flows and links.

$\mathrm{C}(l)$	capacity of link l
P_f	path of flow f
Q_l	flows carried by link l (or by set of links l)
N(l)	number of flows carried by link l
J	Total number of flows
K	Total number of links
$\mathrm{A}(f)$	max-min fair allocation of flow f
$\mathrm{B}(l)$	set of flows bottlenecked at link l
$\mathrm{E}(l)$	set of flows carried by l, bottlenecked elsewhere
$\mathrm{R}(l)$	max-min fair rate of link l (or set of links l, when they have identical rates)
SumE(l)	sum of max-min fair allocations of $\mathrm{E}(l)$ flows
NumB(l)	number of flows bottlenecked at link l
L_i	set of links with ith smallest max-min fair rate, by convention L_0 is the empty set
N	total number of bottleneck links
W	total number of distinct max-min fair rates
LL_n	set of links with max-min fair rates less than or equal to the first n rates, that is, $L_0, L_1, .., L_n$
FL_n	set of flows carried by links in LL_n
$FL_n(l)$	subset of flows in FL_n carried by link l
LG_n	set of links with max-min fair rates greater than the first n rates, that is, $L_{n+1}, .., L_W, L_{W+1}$
FG_n	set of flows carried by links in LG_n that don't cross LL_n
$FG_n(l)$	subset of flows in FG_n carried by link l

Notation: See Table 2.1 for a summary of commonly used notation. We use $\mathrm{A}(f)$ to refer to the max-min fair allocation of a flow f. If the rate allocation is max-min fair, then all flows crossing a link belong to one of two sets. We use $\mathrm{B}(l)$ to refer to the set of flows bottlenecked at link l ("**B**ottlenecked here") and $\mathrm{E}(l)$ to refer to the set of flows bottlenecked at other links ("not bottlenecked here, but bottlenecked **E**lsewhere").

The max-min rate of a link l, $\mathrm{R}(l)$, refers to the max-min fair allocation of its bottleneck flows, $\mathrm{B}(l)$. This is well defined for links that are fully used. If a link is not fully used, we define by convention that $\mathrm{R}(l) = \infty$.

We can write the max-min rate of a link l as follows:

$$\mathrm{R}(l) = \frac{C(l) - \sum_{f \in \mathrm{E}(l)} \mathrm{A}(f)}{|\,\mathrm{B}(l)|}. \tag{2.1}$$

We use the shorthand $\mathrm{SumE}(l) = \sum_{f \in \mathrm{E}(l)} \mathrm{A}(f)$ for the total allocation of flows bottlenecked elsewhere, and $\mathrm{NumB}(l) = |\,\mathrm{B}(l)|$ for the number of flows bottlenecked at link l to get:

$$\mathrm{R}(l) = \frac{C(l) - \mathrm{SumE}(l)}{\mathrm{NumB}(l)}. \tag{2.2}$$

In other words, each bottleneck flow gets an equal share of the remaining capacity after we remove the flows bottlenecked elsewhere.

We use $l_1, l_2, ..., l_K$ to refer to links in increasing order of their max-min fair rates, with ties broken arbitrarily. Since multiple links may have the same max-min rate, we use L_i to refer to the group of links with the ith max-min rate ($1 \leq i \leq W$ when there are W distinct max-min rates) and $R(L_i)$ to refer to the ith max-min rate. We use the convention that L_0 is just the empty set. Additionally, we will use Q_{L_i} to refer to the flows carried by links in L_i i.e., $Q_{L_i} = \bigcup_{l \in L_i} Q_l$.

LG_n is the set of links with max-min rates greater than the first n rates. FG_n is the set of flows carried by links in LG_n, that do not cross any link in $L_0, ..., L_n$. We use $FG_n(l)$ to refer to the subset of FG_n flows carried by link l— that is, $FG_n(l) = FG_n \cap Q_l$. Analogous to LG_n and FG_n, we use LL_n and FL_n to refer to the set of links with the n smallest max-min rates (rates less than or equal to the nth max-min rate), and the set of flows carried by them. We use the convention that LL_0 is just the empty set.

$$\begin{aligned}
LG_n &= L_{n+1}, ..., L_{W+1} \\
LL_n &= L_0, ..., L_n \\
FG_n &= Q_{LG_n} \setminus Q_{LL_n} \\
FL_n &= Q_{LL_n}
\end{aligned}$$

An operational definition: We can use Equation 2.1 to define the max-min fair rate of link $l \in L_{n+1}$ in terms of the rates of the previous links. Any flow in Q_l that is bottlenecked elsewhere to a smaller max-min rate must be a flow that is carried by links in LL_n. All other flows in Q_l, namely $FG_n(l)$, are bottlenecked at the link. Substituting $\mathrm{E}(l) = FL_n(l)$ and $\mathrm{B}(l) = FG_n(l)$, we get the bottleneck rate of link $l \in L_{n+1}$ and the allocation of its B flows:

Table 2.2: Initial Control Packet for Flow f_G from Figure 2.1 in $Fair$, that uses per-flow state at links and in control packet

Link	Allocation (a)	Bottleneck Rate(b)
l_{30}	0	0
l_{12}	0	0

$$R(l) = \frac{C(l) - \sum_{f \in FL_n(l)} A(f)}{|FG_n(l)|},$$
$$A(f) = R(l) \text{ for } f \in FG_n(l).$$

For $l \in L_1$, all flows in Q_l are bottlenecked at the link, and we have:

$$R(l) = \frac{C(l)}{\mathrm{N}(l)},$$
$$A(f) = R(l) \text{ for } f \in Q_l.$$

Later, we will prove that the distributed algorithms we consider converge to exactly these expressions, and hence we will conclude that they converge to the max-min fair rates.

2.2 *Fair*: a Max-Min PERC Algorithm with Per-Flow State

$Fair$ is an algorithm that uses per-flow state at each link to calculate local max-min fair rates at every link. The algorithm provably converges to the global max-min fair rates in $4N$ rounds, where N is the number of bottleneck links.[2] $Fair$ and its dependency-chain based analysis follow directly from existing work by Ros-Giralt et al. [76], [75], [74].

2.2.1 Description

The **control packet** (Table 2.2) carries two fields for each link l: the allocation $a[l]$ and the bottleneck rate $b[l]$.

[2]There are K links in the network, of which N are bottleneck links that are fully used.

The **state at the link** is a table of limit rates for every flow in Q_l. The limit rate $e[f]$ for a flow f is stored at link l and indicates the rate that the flow is limited to, by other links in its path $P_f \setminus l$.

The **algorithm at the link** (Algorithm 2) is as follows. Consider the update of flow f's control packet at link l. The link l computes a *bottleneck rate* b for f (Lines 4—5). In order to compute the bottleneck rate, the link first assumes the flow is not limited anywhere. Then it does a "local max-min fair rate" calculation given the limit rates of all flows in Q_l. For a given flow f, the "local max-min fair rate" is a max-min fair rate of the link in a topology in which the flow f is limited only by link l, and all other flows traverse both link l and an additional link that has capacity equal to their limit rate. It is the unique solution to $b = (\mathrm{C}(l) - \sum_{f:e[f]<b} e[f])/(\sum_{f:e[f]\geq b} 1)$ after we replace $e[f] = \infty$. The rate can be computed explicitly at the link using Algorithm 1. To summarize the algorithm, the link first sorts limit rates in increasing order, and then iteratively expands its estimate of the set of flows that are bottlenecked at other links, until it gets the maximum rate for its own bottleneck flows. The final bottleneck rate satisfies $b = \frac{C-\sum_{f:e[f]<b} e[f]}{\sum_{f:e[f]\geq b} 1}$.

Algorithm 1 Algorithm to compute a local max-min rate b at link l given $\mathrm{N}(l)$ limit rates e. Note that we assume that the sum of the limit rates exceeds the link capacity and hence a local max-min rate is well defined.

1: Step 1: Sort all $\mathrm{N}(l)$ limit rates in increasing order from e_1 to $e_{\mathrm{N}(l)}$
2: Step 2: Iteratively add flows to $E(l)$ as long as R exceeds their limit rates
3: $SumE \leftarrow 0$
4: $NumB \leftarrow \mathrm{N}(l)$
5: $\mathrm{R} \leftarrow (\mathrm{C}(l) - SumE)/NumB$
6: $i \leftarrow 1$
7: **while** $e_i < \mathrm{R}$ **do**
8: $\quad SumE \leftarrow SumE + e_i$
9: $\quad NumB \leftarrow NumB - 1$
10: $\quad \mathrm{R} \leftarrow (\mathrm{C}(l) - SumE)/NumB$
11: $\quad i \leftarrow i + 1$
return R

Next, the link calculates the latest limit rate of the flow (Line 6). Note that the limit rate e of the flow is the smallest *bottleneck rate* that it gets from any *other* link (as deduced from the control packet). The actual bandwidth allocated to the flow is $a = min(e, b)$, which is e if the flow is limited to a smaller bottleneck rate elsewhere, and b if the flow is bottlenecked at link l (Line 7). Finally, the link updates the local limit rate stored for the flow, and the bottleneck rate and allocation stored in the flow's control packet.

Note that only the end hosts look at the allocations stored in the control packet. Over time, we expect the bottleneck rates and allocations to converge to the max-min fair rates.

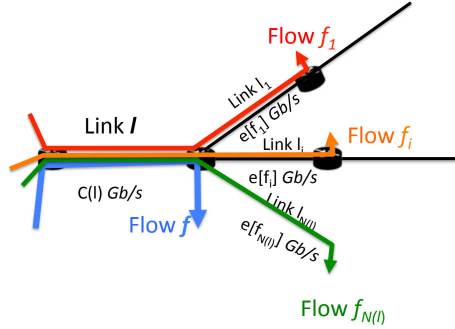


Figure 2.2: *Local* max-min fair calculation at link l for flow f given limit rates $e\colon Q_l \to \mathbb{R}_{\geq 0}$ and capacity $\mathrm{C}(l)$. We defined it as the max-min fair rate of link l in a topology, where flow f is limited only by link l, and all other flows traverse both link l and an additional link that has capacity equal to their limit rate. It is the unique solution to $b = (\mathrm{C}(l) - \sum_{f:e[f]<b} e[f])/(\sum_{f:e[f]\geq b} 1)$ after we replace $e[f] = \infty$.

Algorithm 2 Control Packet Processing at Link l for $Fair$.

1: $e = \{\}$: table of limit rates for flows that cross link l ▷ Initial State at link l
2: $e[f] \leftarrow \infty$ ▷ Update link state to assume flow is going to be bottlenecked
3: Find b such that $b = \frac{\mathrm{C}(l) - \sum_{f:e[f]<b} e[f]}{\sum_{f:e[f]\geq b} 1}$ ▷ Local max-min fair calculation, see Algorithm 1
4: $e \leftarrow \min_{\{l \in P_f \setminus \text{this link}\}} b[l]$ (or ∞ if there is no other link in P_f) ▷ Find flow's new limit rate
5: $a \leftarrow \min(\mathrm{b,e})$ ▷ Find flow's new allocation
6: $b[l] \leftarrow b, \quad a[l] \leftarrow a$ ▷ Update control packet
7: **if** flow is leaving **then** delete $e[f]$ ▷ Update link state to remove flow
8: **else** $e[f] \leftarrow e$ ▷ Update link state to reflect flow's new limit rate

Let's walk through an example to understand how the rates evolve in the $Fair$ algorithm (see Figure 2.3). We will later contrast this with how the rates evolve (badly) when we don't have per-flow state.

2.3 The *Fair* Algorithm in Action

Flow f_B is first seen at link l_{20}. The link assumes the flow is not limited anywhere else, computes a bottleneck rate of 20 Gb/s, and allocates this to the flow. Flow f_B is then seen at link l_{30} during update 2. The link computes a bottleneck rate of 30 Gb/s, sees that the flow is limited to 20 Gb/s, and allocates 20 Gb/s.

When flow f_G is seen at link l_{30} during update 3, the link knows that there is one other flow limited to 20 Gb/s. It assumes flow f_G is not limited ($e = \infty$), and a local max-min fair calculation yields 15 Gb/s for each flow. In terms of Equation 2.2, the link concludes that both flows are bottlenecked at the link ($NumB = 2, SumE = 0, R = C - 0/2$). Since the bottleneck rate is smaller than the flow's limiting rate, the link allocates 15 Gb/s. Then when the flow is seen at link l_{12}, its limit rate 15 Gb/s exceeds the bottleneck

rate 12 Gb/s, and the flow gets it correct max-min fair allocation during update 4.

Round	Time	Event	e	b	a
1	1	Flow f_B @ Link l_{20}	∞	20	20
1	2	Flow f_B @ Link l_{30}	20	30	20
1	3	Flow f_W @ Link l_{30}	∞	15	15
1	**4**	**Flow f_W @ Link l_{12}**	**15**	**12**	**12**
l_{20}: e[f_B]=∞ l_{30}: e[f_B]=20, e[f_W]=∞ l_{12}: e[f_W]=15			f_B: b[l_{20}]=20 b[l_{30}]=30 f_W: b[l_{30}]=15 b[l_{12}]=12		
2	**5**	**Flow f_W @ Link l_{30}**	**12**	**15**	**12**
2	6	Flow f_W @ Link l_{12}	15	12	12
2	7	Flow f_B @ Link l_{20}	30	20	20
2	**8**	**Flow f_B @ Link l_{30}**	**20**	**18**	**18**
l_{20}: e[f_B]=30 l_{30}: e[f_B]=20, e[f_W]=12 l_{12}: e[f_W]=15			f_B: b[l_{20}]=20, b[l_{30}]=15 f_W: b[l_{30}]=18, b[l_{12}]=12		

Figure 2.3: Control packet updates for the first two rounds of the $Fair$ algorithm. We show the link state (limit rates) and the control packet state (just the bottleneck rates) at the end of each round.

We will skip the discussion of the subsequent updates for flow f_B and directly proceed to discuss some properties of the algorithm.

2.4 Properties of the *Fair* Algorithm

There are two properties of the $Fair$ algorithm, which we will later see are lacking in algorithms that don't use per-flow state.

1. The bottleneck rate calculation at each link is *locally max-min fair*. During update 3, given a limit rate of 20 Gb/s for flow f_B and $e = \infty$ for flow f_G, an allocation of the bottleneck rate 15 Gb/s for both flows is max-min fair— they are both bottlenecked at the link.

2. Suppose we say that a link "estimates" a flow as bottlenecked here, that is, in B, if and only if its limit rate exceeds its local max-min fair rate (which is a function of the link state in $Fair$). Then there can be multiple flows, which in the link's estimate, move from E to B during a single update. After update

3 at link l_{30}, given the new limit rates, the link estimates both flows to be bottlenecked at the link (local max-min fair rates are 15 Gb/s for each). Before the update however, given limit rate 20 Gb/s for flow f_B, the link estimated f_B to be in E ($e = 20$ Gb/s is smaller than local max-min fair rate 30 Gb/s). Hence, with update 3, not only did the link identify the new flow f_G as bottlenecked, it also changed its estimate of the other flow f_B from E to B.

2.5 Convergence of the *Fair* Algorithm

Theorem 2.5.1. *Once the set of flows in the network stabilizes,* Fair *is guaranteed to converge to the max-min fair allocation in less than or equal to* $4N$ *rounds, where* N *is the number of bottleneck links in the max-min fair allocation.*

The proof follows by induction on the links in increasing order of their max-min fair rates. Hence, we first define, recursively, what it means for a link to converge.

Definition 2.5.2. *A link* l *has converged by time* T *if*

- *the bottleneck links of its* E *flows have converged by time* T*;*
- *for its* B *flows, for all updates after time* T*,*
  - *at link* l*, the flow's "limit rate elsewhere" is at least the max-min rate of link* l*, which is exactly the value of the bottleneck rate at link* l $e \geq R(l) = b$*;*
  - *at other links* m*, the flow's "limit rate elsewhere" is exactly the max-min rate of link* l*, which is less than the bottleneck rate at link* m $e = R(l) \leq b$*.*

Note that Definition 2.5.2 is recursive because 1) for a link $l \in L_1$, all flows in Q_l are B flows; and 2) when $n \geq 1$, for a link $l \in L_{n+1}$ any E flow is bottlenecked at link(s) in $L_1, ..., L_n$, which have smaller max-min rates.

Proof. The proof follows by induction on the sets of links $L_0, L_1, .., L_W$. Note that L_0 is the empty set and W is the number of distinct max-min rates, which is less than the total number of bottleneck links, that is, $W \leq N$.

Base case The base case for induction is that the links in L_0 converge. This is trivially true.

Induction step We show that for $0 \leq n < W$, if $L_0, ..., L_n$ have converged by time T (induction hypothesis), then any link $l \in L_{n+1}$ will converge within four rounds of T.

1. From time T, flows carried by $L_1, ..., L_n$ present limit rates at links $m \in LG_n$, which are exactly their max-min fair allocation. This follows directly from the induction hypothesis.

2. From time $T + 1\ round$, the limit rates of all flows carried by $L_1, ..., L_n$, as *stored* at links $m \in LG_n$ equal their max-min rates. This is because every flow in Q_m is seen at least once at link m in the round following time T. Given this, we can show that from time $T + 1\ round$, flows in $\mathrm{B}(l)$ pick up bottleneck rates from other links in LG_n, which are at least $R(l)$. Consider an update of a flow $f \in \mathrm{B}(l)$ at a link $m \in LG_n$. Because link m is in LG_n, it has at least $R(l)$ for each of its flows that remain after removing the max-min rates of flows crossing $L_1, ..., L_n$ (see Lemma 2.5.3). As a result, the local max-min fair rate computed at m for f is at least $R(l)$ (see Lemma 2.5.5).

3. From time $T + 2\ rounds$, l's B flows present limit rates at l, which are at least $R(l)$. This is true because the limit rate of l's B flows is the smallest bottleneck rate that the flow gets at any other link in LG_n.

4. From time $T + 3\ rounds$, the limit rates of all flows in $\mathrm{B}(l)$, as *stored* at link l, are at least $\mathrm{R}(l)$. This is because every flow in Q_l is seen at least once at link l in the round following time $T + 2\ rounds$. Given this, we can show that from time $T + 3\ rounds$, flows in $\mathrm{B}(l)$ pick up a bottleneck rate from l that is exactly $R(l)$. Consider an update of flow $f \in \mathrm{B}(l)$ at link l. $R(l)$ is exactly the value that a local max-min fair calculation at l for f would yield, given that the B flows are limited to at least $R(l)$ and the E flows (carried by the first n links) are limited to their max-min rates (see Lemma 2.5.4). Hence, from time $T + 3\ rounds$, any update of a $B(l)$ flow at l satisfies the condition $e \geq R(l) = b$.

5. From time $T + 4\ rounds$, l's B flows present limit rates at other links that equal $R(l)$, because the bottleneck rate that they get at l is the smallest of all links on their path. Hence, from time $T+4\ rounds$, any update of a B flow at $m \in LG_n$ satisfies the condition $e = \mathrm{R}(l) \leq b$.

□

Lemma 2.5.3. *For any link $m \in LG_n$, define $CR_n(m)$ as the capacity of link m after removing the max-min allocations of flows carried by $L_1, ..., L_n$, namely, $CR_n(m) = C(m) - \sum_{f \in FL_n(m)} A(f)$. Then links $l \in L_{n+1}$ have the smallest remaining capacity per-flow $CR_n(l)/FG_n(l)$ out of all links in LG_n, and this is exactly their max-min rate:*

$$
\begin{aligned}
R(l) &= \frac{C(l) - \sum_{f \in FL_n(m)} A(f)}{|FG_n(l)|} \\
&= \frac{CR_n(l)}{|FG_n(l)|} \\
&\leq \frac{CR_n(m)}{|FG_n(m)|}, m \in LG_n
\end{aligned}
$$

Proof. We can observe from the water-filling algorithm [22] that the set of links with the $n{+}1$th smallest max-min rates L_{n+1} is exactly the set of links that have the smallest remaining capacity per-flow after removing the max-min allocations of flows carried by $L_1, ..., L_n$. □

Lemma 2.5.4. *Consider a local max-min fair calculation at link $l \in L_{n+1}$ for a flow $f \in \mathrm{B}(l)$ (Line 5 of Algorithm 2). Given that the limit rates stored at l for flows crossing $L_1, ..., L_n$ are exactly their max-min rates and for flows in $FG_n(l)$ are at least $\mathrm{R}(l)$, the local max-min fair rate b computed at link l for f is exactly $\mathrm{R}(l)$.*

Proof. We defined the max-min fair rate of link $l \in L_{n+1}$ as follows:

$$
\mathrm{R}(l) = \frac{C(l) - \sum_{f \in FL_n(l)} A(f)}{|FG_n(l)|}
$$

The set of flows crossing $L_1, ..., L_n$ is exactly the set of flows whose max-min rates are smaller than $R(l)$. Moreover, we are given that their limit rates equal their max-min rates and hence satisfy $e[f] < R(l)$, while the limit rates of other flows in Q_l exceed $\mathrm{R}(l)$. So we can substitute $\bigcup_{i=1}^{n} Q_l \cap L_i = \{f \in Q_l, e[f] < \mathrm{R}(l)\}$ and for each of these flows $A(f) = e[f]$ to get:

$$
\mathrm{R}(l) = \frac{\mathrm{C}(l) - \sum_{f \in Q_l, e[f] < \mathrm{R}(l)} e[f]}{|FG_n(l)|}
$$

We are given that the limit rates of flows in $FG_n(l)$ are at least $R(l)$ and hence satisfy $e[f] \geq R(l)$, while the limit rates of other flows in Q_l are less than $\mathrm{R}(l)$. So we can substitute $|FG_n(l)| = \sum_{f \in Q_l, e[f] \geq \mathrm{R}(l)} 1$ to get:

$$
\mathrm{R}(l) = \frac{\mathrm{C}(l) - \sum_{f \in Q_l, e[f] < \mathrm{R}(l)} e[f]}{\sum_{f \in Q_l, e[f] \geq \mathrm{R}(l)} 1}
$$

Hence, $\mathrm{R}(l)$ is also the local max-min fair rate at the link for the given limit rates. □

Lemma 2.5.5. *Let link $l \in L_{n+1}$. Consider a local max-min fair calculation at link $m \in LG_n$ for a flow $f \in \mathrm{B}(l)$ (Line 5 of Algorithm 2). Given that the limit rates stored at m for flows crossing $L_1, ..., L_n$ are*

exactly their max-min rates, the local max-min fair rate b computed at link m for f is at least $\mathrm{R}(l)$.

Proof. We know that the local max-min fair rate b computed at m for $f \in \mathrm{B}(l)$ satisfies:

$$b = \frac{\mathrm{C}(m) - \sum_{f \in Q_m, e[f] < b} e[f]}{\sum_{f \in Q_m, e[f] \geq b} 1}$$
$$\mathrm{C}(m) = \sum_{f \in Q_m, e[f] \geq b} b + \sum_{f \in Q_m, e[f] < b} e[f]$$

Some flows with limit rates at least b are in $FG_n(m)$, while others are carried by $L_0, ..., L_n$:

$$\begin{aligned}
\sum_{f \in Q_m, e[f] \geq b} b &= \sum_{e[f] \geq b, f \in FG_n(m)} b + \sum_{e[f] \geq b, f \in FL_n(m)} b \\
&\leq \sum_{e[f] \geq b, f \in FG_n(m)} b + \sum_{e[f] \geq b, f \in FL_n(m)} e[f] \qquad \text{(since } e[f] \geq b\text{)} \\
&\leq \sum_{e[f] \geq b, f \in FG_n(m)} b + \sum_{e[f] \geq b, f \in FL_n(m)} A(f) \qquad \text{(using induction hypothesis)} \qquad (2.3)
\end{aligned}$$

Some flows with limit rates less than b are in $FG_n(m)$, while others are carried by $L_0, ..., L_n$:

$$\begin{aligned}
\sum_{f \in Q_m, e[f] < b} e[f] &= \sum_{e[f] < b, f \in FG_n(m)} e[f] + \sum_{e[f] < b, f \in FL_n(m)} e[f] \\
&\leq \sum_{e[f] < b, f \in FG_n(m)} b + \sum_{e[f] < b, f \in FL_n(m)} A(f) \qquad \text{(using induction hypothesis)} \qquad (2.4)
\end{aligned}$$

Hence, we see that if we were to assign b to the $FG_n(m)$ flows and the max-min fair rates to flows in $FL_n(m)$, then the total bandwidth used will be at least the link capacity.

If we assume that the local max-min fair rate $b < \mathrm{R}(l)$, we get a contradiction.

$$\begin{aligned}
\mathrm{C}(m) &= \sum_{f \in Q_m, e[f] \geq b} b + \sum_{f \in Q_m, e[f] < b} e[f] \qquad \text{(property of local max-min rate } b \text{ at } m\text{)} \\
&\leq \sum_{f \in FG_n(m)} b + \sum_{f \in FL_n(m)} A(f) \qquad \text{(using Equations 2.3 and 2.4)} \\
&< \sum_{f \in FG_n(m)} R(l) + \sum_{f \in FL_n(m)} A(f) \qquad \text{(assuming } b < \mathrm{R}(l)\text{)} \\
&\leq \mathrm{C}(m) \qquad \text{(using Lemma 2.5.3}
\end{aligned}$$

In the last step, we used Lemma 2.5.3, which says the link m has least $\mathrm{R}(l)$ for every flow in $FG_n(m)$ after removing the max-min allocations of flows carried by $L_0, ..., L_n$.

Hence, using a proof by contradiction, we showed that the local max-min fair rate b at m for flow $f \in \mathrm{B}(l)$ must be at least $\mathrm{R}(l)$.

□

2.6 A Tighter Convergence Bound for the $Fair$ Algorithm

In this section we describe how to get a tighter convergence bound for the $Fair$ algorithm. This follows directly from the proof in [74]. We present it here for instructional value because we will adapt this technique to prove the convergence of the s-PERC algorithm in the next chapter.

2.6.1 Invariants of $Fair$ based on the Induction Order

In the proof of Theorem 2.5.1, we used induction on the links ordered by their max-min rates, that is on the sets $L_0, L_1, .., L_W$, where W is the number of distinct max-min rates. We shall refer to this as the water-filling order, since this is also the order in which links are bottlenecked in the classic centralized water-filling [22] algorithm . The induction hypothesis assumes that links in $L_0, ..., L_n$ have converged by time T. The proof of convergence uses this hypothesis to establish two invariants about updates at a link $l \in L_{n+1}$ and at its neighbors. First, l's B flows pick up bottleneck rates from other links on their path that are at least $\mathrm{R}(l)$ from $T + 1$ $round$. Second, l itself computes a bottleneck rate for its B flows that is exactly $\mathrm{R}(l)$ from $T + 3$ $rounds$. We use $\cup_{f \in \mathrm{B}(l)} P_f$ to denote the set of links that carry link l's bottleneck flows. Note that $\cup_{f \in \mathrm{B}(l)} P_f$ is a subset of LG_n since by definition LG_n is the set of links with max-min rates that are at least $\mathrm{R}(l)$.

Lemma 2.5.5 shows that the first invariant follows specifically from the induction hypothesis that for any $m \in \cup_{f \in \mathrm{B}(l)} P_f$, the limit rates stored at m for flows carried by $L_0, ..., L_n$ are exactly their max-min rates, and the following property of m, which says that the remaining capacity per-flow of m after removing the max-min rates of flows carried by $L_0, ..., L_n$ is at least the max-min rate of link l:

For $m \in \cup_{f \in \mathrm{B}(l)} P_f$,

$$CR(m)/FG_n(m) = \frac{C(m) - \sum_{f \in FL_n(m)} A(f)}{|FG_n(m)|} \geq R(l) \tag{2.5}$$

Lemma 2.5.4 shows that the second result then follows specifically from the induction hypothesis that the limit rates stored at l for flows carried by $L_0, ..., L_n$ are exactly their max-min rates, and for flows in $\mathrm{B}(l)$ are at least $\mathrm{R}(l)$ and the following property of l, which says that the remaining capacity per-flow of l after removing the max-min rates of flows carried by $L_0, ..., L_n$ is exactly the max-min rate:

For $l \in L_{n+1}$:

$$\begin{aligned} CR(l)/FG_n(l) &= \frac{C(l) - \sum_{f \in FL_n(l)} A(f)}{|FG_n(l)|} \\ &= R(l) \end{aligned} \tag{2.6}$$

2.6.2 The CPG Algorithm to Find Max-Min Fair Rates

Ros-Giralt et al. [75] describe a parallel variant of the classic water-filling algorithm called the CPG (constrained precedence graph) algorithm that calculates max-min fair rates and identifies bottleneck links in D (fewer than W) iterations (Algorithm 3). Both algorithms computer fair share rates for all links in every iteration. The classic water-filling algorithm identifies and 'freezes' bottleneck links, one (rate) at a time, by checking if a link's fair share rate in a given iteration is smaller than any other link globally. This yields the $L_1, .., L_W$ partition of the bottleneck links, one set for each bottleneck rate. The CPG algorithm identifies and 'freezes' bottleneck links, multiple links (and rates) at a time, by checking if a link's fair share rate in a given iteration is smaller than its neighbors. This yields a different smaller partition $L_1^1, ..., L_D^1$, which we shall call the CPG order. We shall use the notation $R_n^1[l]$ to refer to the the fair share rate computed by a link l in iteration n of the CPG algorithm, and distinguish this from the link's max-min rate $\mathrm{R}(l)$.

Algorithm 3 CPG Algorithm for $Fair$. For any link l that is removed in round n, we allocate $A[f] = R[l] = C[l]/N[l]$ as computed in round n to each of its flows in Q_l during round n, and remove the flow.

```
1: L : set of all links in the network that have at least one flow
2: C : remaining link capacities, N: number of flows                ▷ arrays indexed by link
3: R : remaining link capacity per-flow, Q: active flows             ▷ arrays indexed by link
4: A: bandwidth allocation to flow, P: array of links per flow       ▷ arrays indexed by flow
5: wfRound ← 0
6: while L is not empty do
7:     wfRound ← wfRound + 1
8:     for all l ∈ L do R[l] ← C[l]/N[l]
9:     links_to_remove ← {}, flows_to_remove ← {}
10:    for all l ∈ L do
11:        minRate ← min_{m: l,m share a flow} R[m]
12:        if R[l] == minRate then
13:            add l to links_to_remove
14:            for all f ∈ Q[l] do
15:                add f to flows_to_remove
16:                A[f] = R[l]
17:    for all f ∈ flows_to_remove do
18:        for all l ∈ P[f] do
19:            C[l] ← C[l] − A[f]
20:            N[l] ← N[l] − 1
21:            remove f from Q[l]
22:    remove links_to_remove from L
```

If we replace L_i with L_i^1 and replace FG_n with FG_n^1, which in the context of the CPG order, is defined as the set of flows not carried by $L_1^1, ..L_n^1$, then Equation 2.5 and Equation 2.6 follow immediately from the CPG algorithm. As a result, one can show that the two invariants about the $Fair$ algorithm hold in the context of the CPG order as well. Hence, a proof by induction on the CPG order yields a tighter convergence bound of $4D$ rounds where D is the number of iterations of the CPG algorithm:

Theorem 2.6.1. *Once the set of flows in the network stabilizes,* Fair *is guaranteed to converge to the max-min fair allocation in less than or equal to* $4D$ *rounds, where* D *is the number of iterations that the CPG algorithm takes for the given set of flows and links.*

Note that the analogous convergence theorem for the distributed d-CPG algorithm in [76] is in terms of the number of levels in a constrained precedence graph, which they show is exactly equal to the number of iterations in the CPG algorithm. In the next section, we review Ros-Giralt et al.'s definition of the constrained

precedence graph and their proof of why the depth of the graph is equal to the number of iterations in the CPG algorithm.

2.6.3 Dependencies from the CPG Algorithm

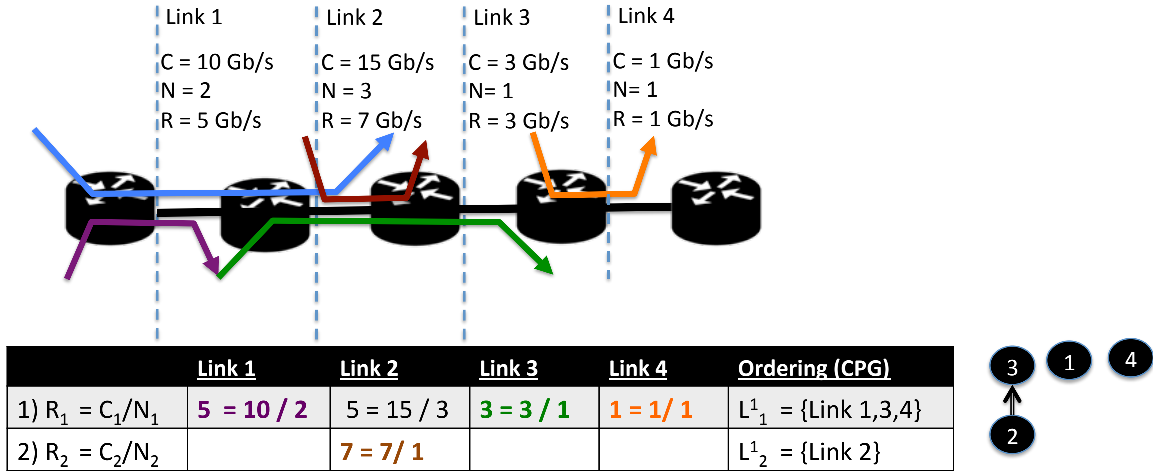
It follows from the CPG algorithm that in order for a link $l \in L^1_{n+1}$ to "converge" (in a centralized algorithm or in a distributed algorithm like $Fair$), it is sufficient for links $L^1_0, .., L^1_n$ in the CPG order to converge, rather than all links with smaller max-min rates than l. Ros-Giralt et al. further prove that it is in fact necessary for at least one link in L^1_n to converge. They call this the *precedent link relationship*. We paraphrase their result here (Lemma 2 in [75])

(Precedent Link Relationship): For a given set of flows and links, let j be a link that is removed at iteration $n + 1$ of the centralized CPG algorithm. Let i be a link that shares a flow with link j in iteration n so that $R^1_n[i] < R^1_n[l]$. Note that such a link must exist, otherwise link j would be removed at iteration n. Then, it must be that either link i becomes a bottleneck at iteration n or there exists at least one link k that shares at least one flow with link i such that it becomes a bottleneck at iteration n.

The definitions of a *direct/ indirect precedent link* follow directly from this Lemma. If link i becomes a bottleneck then i is called a *direct precedent link* of j. On the other hand, if i does not become a bottleneck but the link k that shares at least one flow with link i does, then k is called an *indirect precedent link* of j (and link i is called the *medium link*).

The precedent link relationship implies that for a link $j \in L^1_{n+1}$ there is at least one precedent link relationship with some link in L^1_n. Hence, the precedent link relationships form a directed acyclic graph of depth D, where D is the number of iterations that the CPG algorithm takes for the given set of flows and links. This directed acyclic graph is called the constrained precedence graph (CPG).

The *precedent link relationship* is an example of a dependency, and any path in the constrained precedence graph is a dependency chain. The precedent link relationship implies that in order for a link l to converge in the $Fair$ algorithm (to be removed in the centralized CPG algorithm), it is necessary for all ancestors of the link in the constrained precedence graph to converge (to be removed) first. See Figure 2.4 for an example of the CPG algorithm and the resulting constrained precedence graph for a set of flows and links. See §A.2 in the Appendix for more notes on dependencies.



	Link 1	Link 2	Link 3	Link 4	Ordering (CPG)
1) $R_1 = C_1/N_1$	**5 = 10 / 2**	5 = 15 / 3	**3 = 3 / 1**	**1 = 1/ 1**	L^1_1 = {Link 1,3,4}
2) $R_2 = C_2/N_2$		**7 = 7/ 1**			L^1_2 = {Link 2}

Figure 2.4: Example of CPG Algorithm for $Fair$. We show the progress of the CPG algorithm for the given set of flows and links. A cell in a row is bolded when a link is bottlenecked (i.e., it has the smallest rate among neighbors, that is, links that share a flow) in the respective iteration. Link 3 is a direct precedent of link 2 because link 2 is removed in iteration 2 while link 3 which is removed in iteration 1, shared a (green) flow with link 2 in the iteration and had a smaller fair share rate (3 Gb/s vs 5 Gb/s).

2.7 Simulation Results

We performed small-scale packet-level simulations of the $Fair$ algorithm to evaluate three main claims about the benefits of PERC algorithms relative to reactive algorithms as well as a previous PERC algorithm we'll call $Charny$ [27]. Before describing our evaluation, we contrast $Fair$ and $Charny$. Like $Fair$, $Charny$ also requires per-flow state, and it has been proven to converge in the $4N$ RTTs. $Charny$ differs from $Fair$ in the way that the bottleneck rate is calculated at the switch. A link l in maintains an estimate of whether each flow is in $\mathrm{B}(l)$ of $\mathrm{E}(l)$ and computes a bottleneck rate that is consistent with the $E(l)$ estimates but not necessarily locally max-min fair. Another difference is that the $Charny$ control packet carries a single rate field that is updated by every link on the flow's path. This can lead to slow updates, when a bottleneck changes during the course of the Charny algorithm– typically the old bottleneck link must update (increase) the rate carried in the packet before the new bottleneck link can fill in its rate.

We perform the following experiments to evaluate three main claims about PERC and reactive algorithms:

(1) **Convergence Times**: PERC algorithms converge faster than reactive algorithms; in particular $Fair$ converges faster other state-of-the-art PERC algorithms. We compared Convergence Times of $Fair$ with reactive algorithm RCP and $Charny$.

(2) **Flow Completion Times (FCTs)**: Slow convergence times lead to long Flow Completion Times (FCTs), especially with higher speeds and round-trip delays. We compared FCTs of $Fair$ with reactive

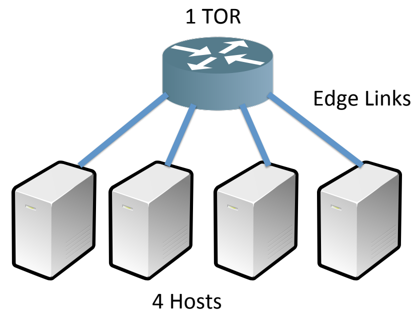


Figure 2.5: Setup for convergence time experiments: one TOR connected to four hosts with 100 Gb/s up/down links.

algorithms RCP and DCTCP and PERC algorithm $Charny$.

(3) **Dependency Chains**: All algorithms take longer to converge when dependency chains between flows get longer; but reactive algorithms perform worse than PERC algorithms.

Our evaluations are based on OMNET++ [82] implementations of reactive (RCP [36]), PERC ($Fair$, $Charny$ [27]), ideal algorithms for max-min rate allocation, and ns2 [4] simulations of DCTCP [12].

2.7.1 Convergence Times

Fair *converges faster than reactive and state-of-the-art PERC algorithms.*

Setup: We consider a topology with one TOR connected to four hosts (Figure 2.5). All 8 links— one up-link and one down-link per host— have capacity 100 Gb/s each. The RTT between hosts is 12 μs. We start with 32 flows between random pairs of hosts, and then alternately add a new random flow or remove an old one (randomly) after the transmission rates for the old set of flows have converged.

Metric: We define convergence as the first time when each of the flow rates has been within 10% of the optimal value for at least ten consecutive RTTs. We look at the number of RTTs to converge for each of 3000 such flow changes.

Algorithms compared: We compare the empirical Cumulative Distribution Function (CDF) of convergence times, as observed across all flow changes, for RCP and the two PERC algorithms $Fair$ and $Charny$ in Figure 2.6.

Results and Analysis

Fair *converges faster than other PERC algorithms*: With $Fair$, more than 99% of the flow changes take less than ten RTTs to converge, whereas with $Charny$ convergence takes 50% longer. The median shows

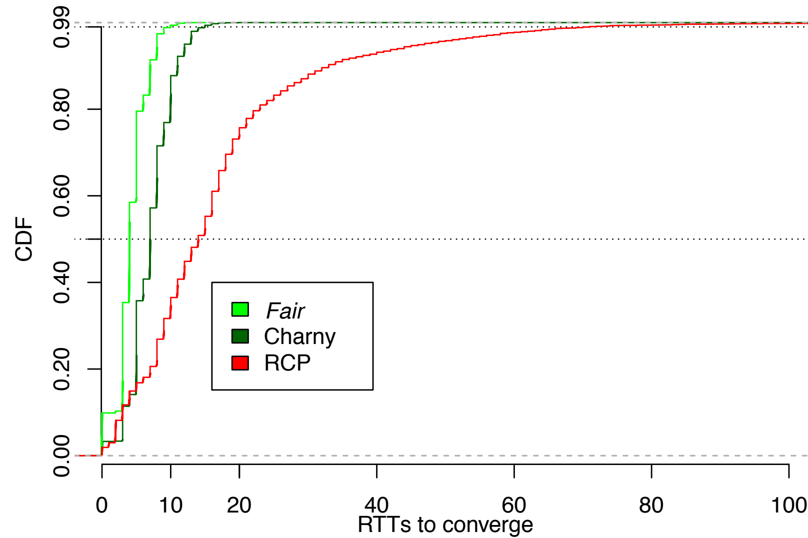


Figure 2.6: **CDF of convergence times for $Fair$, $Charny$, and RCP on an 8-link topology with 100 Gb/s links. RTT is 12 μs.**

a similar trend. $Fair$ converges faster because in the $Fair$ algorithm, control messages carrying demands from flows to links carry more information. In $Charny$, control messages contain a single stamped rate for all links, which is updated by a link if its fair share is smaller. On the other hand, in $Fair$, a flow expresses a different demand for *each link*, which is the maximum it can send based on what it has heard from *all other links.*

PERC algorithms converge faster than reactive algorithms: The median convergence time for RCP is 14 RTTs, which is twice that of $Charny$ and more than three times that of $Fair$. The 99th percentile convergence time of RCP is even worse; at 71 RTTs, it is more than 5 times longer than $Charny$ and 7 times longer than $Fair$. RCP's tail performance is particularly poor if there are long dependency chains, as we show in the next example.

2.7.2 Flow Completion Times

Slow convergence times lead to long (FCTs), especially with higher speeds and round-trip delays.

We test our claim by comparing FCTs of reactive and PERC algorithms for various flow sizes on a single link. We explore the effect of higher link capacities and longer RTTs.

Setup: We use the same workload based on a Microsoft web search data center cluster used in prior work [12], [13]. Flows arrive according to a Poisson process, and the flow size is chosen based on empirically observed distributions. The workload has a diverse mix of small (1 kB-10 kB), medium (10 kB-1 MB), and large (1

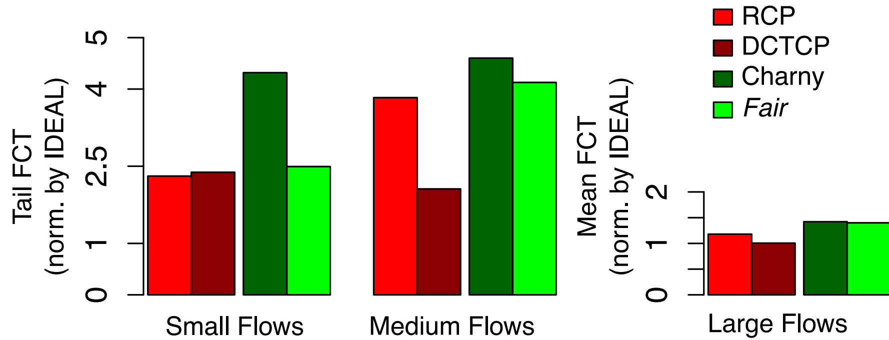


Figure 2.7: Spread FCTs on 10 Gb/s link with 60% load and 12 μs RTT: 99th percentile for small (< 10 kB) and medium (10 kB- 1000 kB) flows, and mean for large (> 1 MB) flows. Normalized by respective statistics for ideal instantaneous max-min rate allocation.

MB- 100MB) flows. By this definition, 14% of the flows are small, 56% are medium, and 30% are large. We generate the workload over a simple dumbbell topology with a single bottleneck link. We present results for an average load of 60%; the results at other loads are qualitatively similar.

Algorithms compared: We compare the FCT for small, medium, and large flows for reactive (RCP [36], DCTCP [12]) and PERC ($Charny$ [27], $Fair$) congestion control algorithms. We normalize the results for each algorithm with an IDEAL reference algorithm in which all flows always transmit at the correct max-min rate. The normalization gives us a common benchmark and lets us see how close different algorithms are to the ideal max-min FCT. In all of our experiments, RCP uses a default initial advertised rate of 0.5 times the link capacity, and the α and β parameters for the rate update are set to 0.5 each (as suggested in [36]).

Results and Analysis

Fair *performs reasonably for moderate link speeds and round-trip times:* With a 10 Gb/s link speed and 12 μs RTT (Figure 2.7), $Fair$'s FCT is comparable to that of RCP across all flow sizes. DCTCP performs better than RCP and $Fair$ in this case, but its performance degrades significantly at higher link speeds and round-trip delays (see below).

Fair *performs much better than RCP/DCTCP at higher link speeds:* If we increase link speed to 100 Gb/s (Figure 2.8), $Fair$ outperforms the reactive algorithms, keeping the tail FCT very low for medium flows. Because flows can be sent much faster, medium flows that previously took 60 RTTs now finish in as few as 6 RTTs. $Fair$ quickly converges to the optimal rates within the shorter lifetime of flows, but the reactive algorithms have a much harder time. For some flow rates DCTCP never reaches the fair share, and this shows up in the long tail FCT. For RCP, all flows get the same rate, but it might be too low or too high (causing long

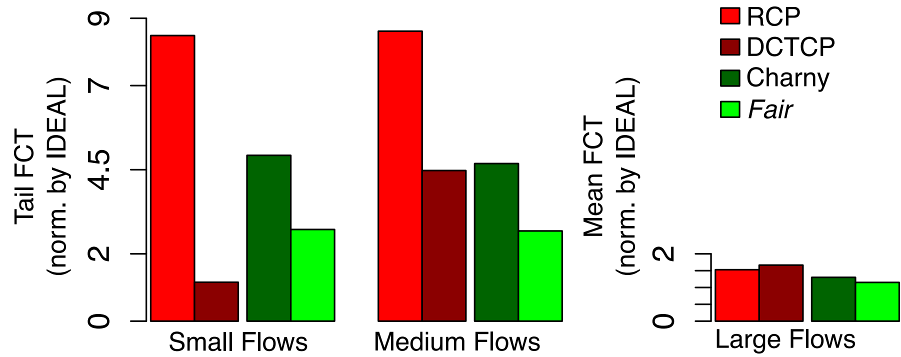


Figure 2.8: Spread FCTs on 100 Gb/s link with 60% load and 12 μs RTT: 99th percentile for small (< 10 kB) and medium (10 kB- 1000 kB) flows, and mean for large (> 1 MB) flows. Normalized by respective statistics for ideal instantaneous max-min rate allocation.

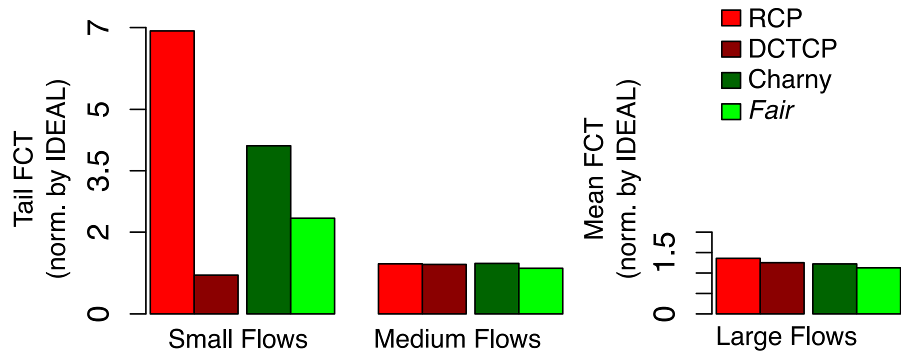


Figure 2.9: Spread FCTs on 10 Gb/s link with 60% load and 120 μs RTT: tail statistics for small/medium flows, and mean for large flows, normalized by respective statistics for an ideal max-min algorithm.

queues).

The difference in tail FCT between Fair*and reactive algorithms is even more significant at higher RTTs:* At higher RTTs (Figure 2.9), reactive algorithms react more slowly since the time from measurement to reaction increases. It takes more steps to converge, and FCT grows. Note that the large bandwidth-delay product at 100 Gb/s and 120 μs RTT causes DCTCP to completely break down because the ramp-up time for medium and large flows becomes prohibitively slow.

2.7.3 Dependency Chains

Convergence times of reactive algorithms scale poorly with long dependency chains

While PERC algorithms scale linearly with the length of dependency chains [76], our initial simulations suggest that reactive algorithms are much worse.

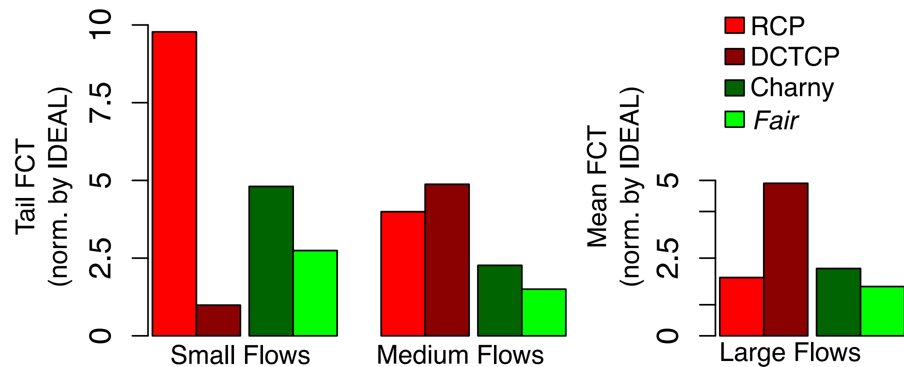


Figure 2.10: Spread FCTs on 100 Gb/s link with 60% load and 120 μs RTT: tail statistics for small/medium flows, and mean for large flows, normalized by respective statistics for an ideal max-min algorithm.

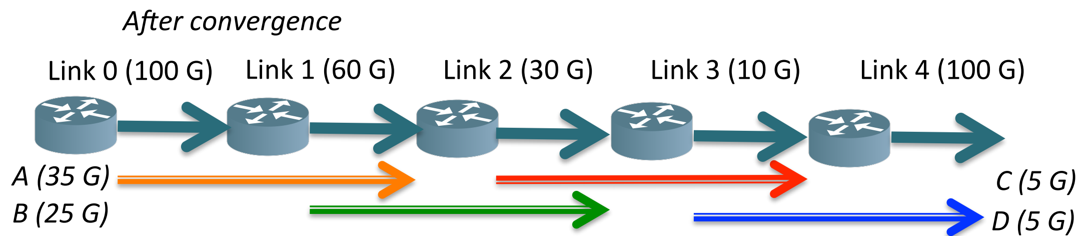


Figure 2.11: A dependency chain of three. A 60 Gb/s link is shared by flows A (orange) and B (green); a 30 Gb/s link by flows B (green) and C (red); a 10 Gb/s link by flows C (red) and D (blue). RTT is 24 μs.

For example, consider the three bottleneck-link scenario shown in Figure 2.11. Links 1—3 are each shared by two flows, of which one is bottlenecked at a later link (except flow D). There is a dependency chain that spans three links. Figure 2.12 shows the time series of the sending rates of the four flows for $Fair$, RCP, and IDEAL.

$Fair$ converges to the correct rates for all flows within 100 μs. At around 50 μs, flow C converges to 5 Gb/s at the bottleneck, and then at 100 μs, flows A and B use up all the spare capacity on the 30 Gb/s and 60 Gb/s links. As the dependency chains grow, the bottleneck information takes longer to propagate, growing linearly with chain length.

RCP takes much longer to converge: flows C and D take almost 600 μs to converge, and flows A and B oscillate within 20% of the target rates for milliseconds (Figure 2.12 only shows up to 1 ms). The bottleneck information takes time to propagate in RCP too, but the convergence behavior is much worse with longer dependency chains.

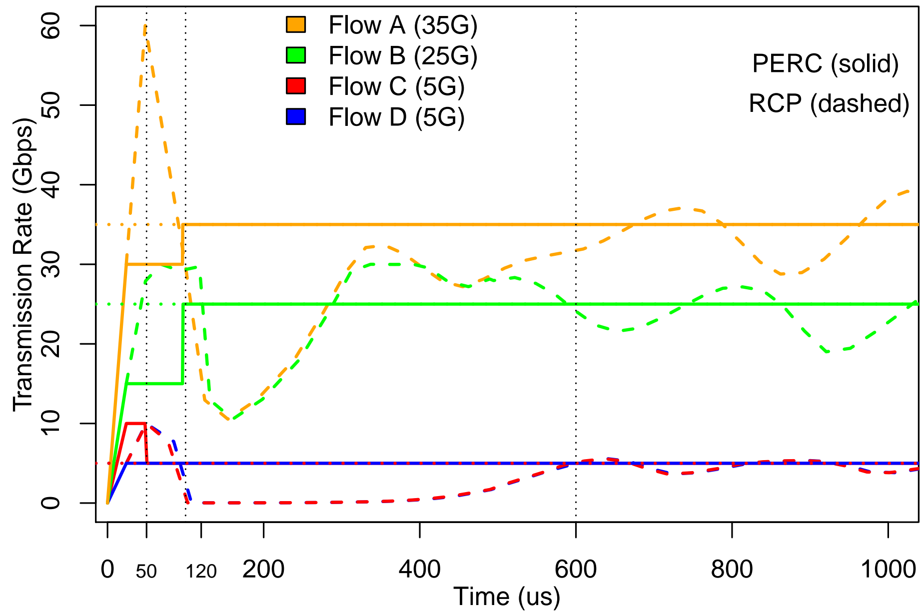


Figure 2.12: Time series of flow transmission rates in 3-level-bottleneck scenario, for RCP (dashed), $Fair$ (solid), and IDEAL (dotted).

2.8 Problem with the *Fair* Algorithm

The small-scale simulations are encouraging and suggest that PERC algorithms like $Fair$ converge much faster than reactive algorithms, and this can yield better Flow Completion Times (FCTs) especially when link speeds are high or round-trip delays are long. Additionally, it appears that proactive schemes scale well as dependency chain lengths increase (our proof shows that the worst-case convergence time scales linearly with the dependency chain length), while reactive algorithms perform worse.

However, the $Fair$ algorithm is not practical. It requires per-flow state at the link for the local max-min fair calculation. This is because the local max-min fair rate calculation in each update requires the link to store the limit rates for each flow.

In the next chapter we explain how to eliminate per-flow state at the links in a way that the new algorithm is still guaranteed to converge in a bounded time.

Chapter 3

s-PERC: A PERC Algorithm that Does Not Need Per-Flow State

In this chapter, we introduce s-PERC, a PERC algorithm for max-min rates that does not need per-flow state. We will start by describing the problems that arise when we eliminate per-flow state naively. Using an algorithm called n-PERC, we will motivate s-PERC's technique of propagating bottleneck rates only when they are high enough. We will then walk through an example of s-PERC and contrast it with the n-PERC example. We will present simulation results that indicate that n-PERC can take an arbitrarily long time to converge in the worst case. Finally, we will present a convergence proof of s-PERC in terms of dependencies that exist between links for a given set of flows.

3.1 n-PERC: a Naive PERC Algorithm Without Per-Flow State

The n-PERC algorithm eliminates per-flow state at the links. Each link l has an estimate of the set of flows bottlenecked at the link and elsewhere, and we will call these $\hat{B}(l)$ and $\hat{E}(l)$ to distinguish them from the true sets. Whether a flow is in $\hat{B}(l)$ or $\hat{E}(l)$ is carried in the control packet of the flow, for each link l.

The link does not need to store per-flow state. Recall that in the $Fair$ algorithm the link uses per-flow state to store the limit rate of every flow, that is, the rate that the flow is limited to by the rest of the network. In the n-PERC algorithm, the link stores two aggregate values only, called $SumE$ and $NumB$. $SumE$ is the sum of the allocations of flows in $\hat{E}(l)$, where each flow is allocated exactly its limit rate, and $NumB$ is the number of flows in $\hat{B}(l)$, which are all limited at l.

While these values of $SumE$ and $NumB$ may not yield the correct local max-min fair rate, they do yield some rate R using Equation 2.2, replacing the actual values of SumE and NumB with $SumE$ and $NumB$:

$$R = \frac{C - SumE}{NumB}.$$

This rate R in turn can be used to reclassify a flow into $\hat{B}$ and $\hat{E}$ based on its latest limit rate. The question to ask is whether the rates converge over time to the ideal max-min rates.

The **control packet** (Table 3.1) carries three fields for each link l: the bottleneck state $s[l]$, which identifies whether the flow is in $\hat{B}$ or $\hat{E}$, the allocation $a[l]$, and the bottleneck rate $b[l]$.

Table 3.1: Initial Control Packet for Flow f_G in n-PERC. f_G crosses two links l_{30} and l_{12}, and the information pertaining to each link is in the first and second row, respectively.

Link	Bottleneck State (s) Allocation (a)	Bottleneck rate (b)
l_{30}	E @ 0	∞
l_{12}	E @ 0	∞

The control packet is only modified by the links; the end host does not modify the control packet, except to indicate when the flow is finished. In the common case, it simply updates the rate at which it sends data packets to match the lowest allocation carried in the control packet, and then reflects the control packet back unmodified.

The **algorithm at the link** (Algorithm 4) is as follows. A link l starts with initial values of $SumE = 0$ and $NumB = 0$. When it sees a flow it uses Equation 2.2 to compute a bottleneck rate for the flow, assuming it is going to be bottlenecked here. For a new flow f, this would be $b = (C - SumE)/(NumB)$ after $NumB$ has been incremented (lines 3—6).

The limit rate e of the flow is the lowest *bottleneck rate* that it gets from any *other* link (line 7). If the link computes a bottleneck rate that is strictly greater than the flow's limit rate, the link classifies the flow into $\hat{E}$; otherwise it classifies the flow in $\hat{B}$. Then it updates $SumE$ and $NumB$ based on the new classification. For an $\hat{E}$ flow the limit rate e becomes part of $SumE$ (line 16); otherwise the flow is remains in $NumB$ (line 5). For the next flow, the link uses the new values of $SumE$ and $NumB$, which will yield a new bottleneck rate.

At the end of the update, the link l updates the bottleneck rate b, the allocation a, and the bottleneck state field s in the control packet (line 11).

Note that other links (i.e., links other than l) only look at the bottleneck rate $b[l]$ field of link l. The allocation $a[l]$ and bottleneck state $s[l]$ fields are used only by link l to correctly update running estimates $NumB$ and $SumE$. For example, if the flow moves from E to B, the link looks up the flow's old allocation

Algorithm 4 Control Packet Processing at Link l for n-PERC.

1: $SumE = 0$: sum of allocations of flows bottlenecked elsewhere ▷ Initial state at link l
2: $NumB = 0$: number of flows bottlenecked here
3: **if** $s[l] == \mathrm{E}$ **then** ▷ Flow was last bottlenecked elsewhere
4: $\quad SumE \leftarrow SumE - a[f]$ ▷ Update link state to assume flow is going to be bottlenecked
5: $\quad NumB \leftarrow NumB + 1$
6: $b \leftarrow \frac{C - SumE}{NumB}$
7: $e \leftarrow min_{m \in P_f \setminus l} b[m]$ (or ∞ if there is no other link in P_f) ▷ Find flow's new limit rate
8: $a \leftarrow \min(b, e)$ ▷ Find flow's new allocation
9: **if** $b <= e$ **then** $s \leftarrow \mathrm{B}$ ▷ Flow is now bottlenecked here; classify as $\hat{B}$
10: **else** $s \leftarrow \mathrm{E}$ ▷ Flow is now bottlenecked elsewhere; classify as $\hat{E}$
11: $b[l] \leftarrow b, \quad a[l] \leftarrow a, \quad s[l] \leftarrow s$ ▷ Update control packet
12: **if** flow is leaving **then** ▷ Update link state to remove flow
13: $\quad NumB \leftarrow NumB - 1$
14: **else if** $s == \mathrm{E}$ **then** ▷ Update link state to reflect flow is in $\hat{E}$
15: $\quad NumB \leftarrow NumB - 1$
16: $\quad SumE \leftarrow SumE + a$

in the control packet, subtracts it from $SumE$, and increments $NumB$ (lines 4—5).

Over time, we hope that sets $\hat{E}$ and $\hat{B}$ at each link converge to the true E and B sets, and the bottleneck rate b at the link converges to the correct max-min fair allocation for the B flows.

We call this algorithm n-PERC, for naive Proactive Explicit Rate Control.

3.1.1 The n-PERC Algorithm in Action

Let's walk through an example to understand how the rates evolve in the n-PERC algorithm (see Figure 3.2). We will use the same topology and workload that are used for demonstrating the $Fair$ algorithm, and we reproduce the figure here for convenience (Figure 3.1). We will examine the first four updates, which are sufficient for the discussion that follows.

Flow f_B is first seen at link l_{20}. The link assumes the flow is not limited anywhere else, computes a bottleneck rate of 20 Gb/s, and allocates this to the flow. Flow f_B is then seen at link l_{30}. The link computes a bottleneck rate of 30 Gb/s, sees that the flow is limited to 20 Gb/s, and allocates 20 Gb/s. So far the allocations are exactly the same as the $Fair$ algorithm.

When Flow f_G is seen at link l_{30} during update 3, the link knows that the total allocation of E flows is 20 Gb/s. It assumes flow f_G is not limited ($e = \infty$) and uses Equation 2.2—where $NumB = 1$, and $SumE = 20$—to compute a bottleneck rate of $b = 10$ Gb/s for flow f_G. Since the limit rate of flow f_G

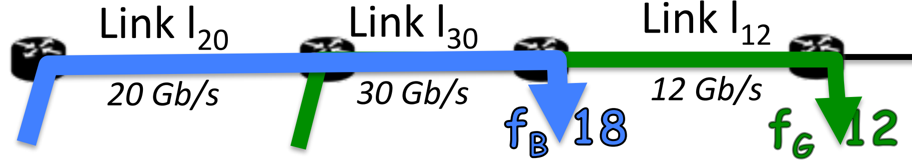


Figure 3.1: Example setup for n-PERC in action. There are $J = 2$ flows and $K = 3$ links. Each flow crosses a subset of the links. The green flow f_G is bottlenecked to 12 Gb/s at link l_{12}, while the blue flow f_B is bottlenecked to 18 Gb/s at link l_{30}. We show here the link capacities and the max-min fair rate allocations for the flows.

Round	Time	Event	e	b	a	state
1	1	Flow f_B @ Link l_{20}	∞	20	20	B
1	2	Flow f_B @ Link l_{30}	20	30	20	E
1	3	Flow f_W @ Link l_{30}	∞	10	10	B
1	4	Flow f_W @ Link l_{12}	10	12	10	E
l_{20}: NumB=1,SumE=0 l_{30}: NumB=0,SumE=20 l_{12}: NumB=1,SumE=0			f_B: b[l_{20}]=20, b[l_{30}]=30 f_W: b[l_{30}]=10, b[l_{12}]=12			
2	5	Flow f_W @ Link l_{30}	12	10	10	B
2	6	Flow f_W @ Link l_{12}	10	12	10	E
2	7	Flow f_B @ Link l_{20}	30	20	20	B
2	8	Flow f_B @ Link l_{30}	20	15	15	B
l_{20}: NumB=1,SumE=0 l_{30}: NumB=2,SumE=0 l_{12}: NumB=1,SumE=0			f_W: b[l_{30}]=10, b[l_{12}]=12 f_B: b[l_{20}]=20, b[l_{30}]=15			
3	**9**	**Flow f_W @ Link l_{30}**	**12**	**15**	**12**	**E**
3	**10**	**Flow f_W @ Link l_{12}**	**15**	**12**	**12**	**B**
3	**11**	**Flow f_B @ Link l_{30}**	**20**	**18**	**18**	**B**
3	**12**	**Flow f_B @ Link l_{20}**	**18**	**20**	**18**	**E**
l_{20}: NumB=0,SumE=18 l_{30}: NumB=1,SumE=12 l_{12}: NumB=1,SumE=0			f_W: b[l_{30}]=15, b[l_{12}]=12 f_B: b[l_{30}]=18, b[l_{20}]=20			

Figure 3.2: Control packet updates for first three rounds of the n-PERC algorithm. We show the limit rate e computed for the flow and the control packet state (the bottleneck rate b, allocation a, and the bottleneck state s after the update). We also show the link state $(NumB, SumE)$ at the end of each round.

is ∞, the link concludes that the flow is a bottleneck flow, allocates $a = b = 10$ Gb/s and updates the control packet. Notice that this is different from $Fair$, which computed a bottleneck rate of $b = 15$ Gb/s and allocated $a = b = 15$ Gb/s to the flow.

When flow f_G is next seen at its actual bottleneck link l_{12} during update 4, its limit rate is $e = 10$ Gb/s, which is the lowest bottleneck rate at any other link. This is lower than the link's bottleneck rate $b = 12$ Gb/s and the link fails to recognize its bottleneck flow. The link assumes the flow is bottlenecked elsewhere and

allocates only 10 Gb/s. This is a problem because despite having the lowest max-min rate of all links, l_{12} cannot immediately recognize it as a bottleneck flow.

The difference between n-PERC and $Fair$ gives some insight into why it is hard to bound n-PERC's convergence time. If we consider flow f_G's update at link l_{30} (where it is labeled B and allocated $b = 10$ in update 3) and the subsequent update at l_{12} (where it is labeled E and allocated $e = 10$), then we can in fact identify two transient problems with the n-PERC algorithm that we describe below. Extensive numerical simulations suggest that the algorithm will still eventually stabilize despite the transient problems (see §3.3), but there is no known upper bound for how long it can take. A similar algorithm has been proved to stabilize eventually in [33], but it has no upper bound either.

3.1.2 Transient Problems With n-PERC

1. **Suboptimal local rates**: The first problem is that the rate allocation at a link can be suboptimal—that is, it is not locally max-min fair. The B flows are allocated a lower rate than if the link had used per-flow state to store the limit rates and calculate a local max-min fair rate. To see this, consider the rate allocation at link l_{30} at the end of update 3. Link l_{30} considers flow f_B bottlenecked elsewhere to 20 Gb/s and flow f_G bottlenecked at the link to 10 Gb/s. A B flow is allocated less than an E flow. It is not max-min fair given the limit rates, because we can increase f_G's allocation at the expense of flow f_B that gets more than f_G.

2. **Bad bottleneck rate propagation**: The second problem is that when suboptimal bottleneck rates at one link for a flow are propagated to the actual bottleneck of the flow, they may prevent the bottleneck link from identifying that flow is indeed bottlenecked and thus delay its convergence. In the example, f_G is actually bottlenecked at link l_{12}. However, it picks up a rate of $b = 10$ Gb/s from link l_{30}, which becomes its limit rate at l_{12}. At l_{12}, since the limit rate $e = 10$ Gb/s is even lower than the bottleneck rate $b = 12$ Gb/s, the flow is wrongly labeled E and allocated only $e = 10$ Gb/s. Link l_{12} does not immediately recognize the flow as a bottleneck flow and must wait until update 9, when l_{30}'s bottleneck rate for flow f_G $b = 15$ Gb/s is high enough. It appears from [33] that the labels may oscillate for a long time but they eventually stabilize.

While the first problem of suboptimal rates cannot be fixed unless we use per-flow state to store every flow's limit rate, we describe next how we can easily fix the problem of bad rate propagation.

The $Fair$ algorithm does not propagate bad rates. Since l_{12} has the lowest max-min rate 12 Gb/s, by definition any other link has at least 12 Gb/s available for each of its flows, including l_{12}'s bottleneck flow.

So the local max-min fair rate at any other link is at least 12 Gb/s. Notice that the limit rates of the flows at link l_{30} during update 3 are identical for both the $Fair$ algorithm and n-PERC. Yet, while $Fair$ computes bottleneck rate 15 Gb/s, which is higher than l_{12}'s max-min fair rate, n-PERC computes a bottleneck rate that is lower. A local max-min fair rate is always a good bottleneck rate and high enough to propagate. But it is impossible to calculate a local max-min fair rate without storing the rate of every flow.

An alternative is to identify bottleneck rates that may be too low and not propagate them. Suppose link $l \in L_n$ (using the notation from §2.1.2) is updating the control packet for a flow f. If f is a bottleneck flow of l, we do want to propagate the bottleneck rate b once it is correct and equals $R(l)$, in order to make progress. If the rate *is not* correct yet, especially if f is bottlenecked at some other link $l' \in L_m$, which has a lower max-min rate ($m < n$), we would rather say that the flow is not limited by l than have l propagate a low rate that could potentially delay the true bottleneck link l' from correctly identifying its bottleneck flow. This leads us to s-PERC.

3.2 s-PERC: a Stateless PERC Algorithm with a Known Bounded Convergence Time

In s-PERC, we maintain an additional variable at each link called $MaxE$ that is updated each time l classifies a flow as $\hat{E}$. At any time $MaxE$ is at least as high as the allocation of the highest $\hat{E}(l)$ flow. We propagate a bottleneck rate b only when $b \geq MaxE$. Why might this work? The bottleneck is just the remaining capacity after removing all $\hat{E}$ allocations $C - SumE$, divided evenly among the $\hat{B}$ flows. We can see that if one of the $\hat{E}$ allocations is more than the bottleneck rate (such as in the n-PERC example) then that flow in $\hat{E}$ may actually need to be reclassified as a $\hat{B}$ flow. In other words, a bottleneck link rate $b < MaxE$ is inconsistent with the set of flows we have assumed to be in $\hat{E}$; hence, the bottleneck rate $\frac{C-SumE}{NumB}$ might be too low to propagate. We confirm this hypothesis in Lemma 3.2.2.

3.2.1 Variables in the s-PERC Algorithm

The **control packet** (Table 3.2) carries four fields for each link l. The first three are the same as in n-PERC: the bottleneck state $s[l]$, which identifies whether the flow is in $\hat{B}$ or $\hat{E}$, the allocation $a[l]$, and the bottleneck rate $b[l]$. In addition, the packet carries an "ignore" bit $i[l]$, which is cleared if link l wants to propagate its bottleneck rate.

We clarify some s-PERC specific notation:

Table 3.2: Initial Control Packet for Flow f_G in s-PERC. f_G crosses two links l_{30} and l_{12}, and the information pertaining to each link is in the first and second row, respectively.

Link	Bottleneck State (s) Allocation (a)	Bottleneck rate (b)	Ignore bit (ignore)
l_{30}	E @ 0	0	1
l_{12}	E @ 0	0	1

1. We say that the allocation of a flow f at link l at time t is the value of $a[l]$ carried in the f's control packet at time t, and we refer to it as $a^l_f(t)$. Similarly, the bottleneck state of a flow f at link l at time t is the value of $s[l]$ carried in f's control packet at time t, and we refer to it as $s^l_f(t)$. If the control packet is modified at the link precisely at time t, we will use the convention that t^- and t^+ refer to the time just before and just after an update respectively.

2. We will use $\hat{\mathrm{E}}(t)$ to refer to the set of flows that are considered by link l to be in $\hat{E}$ or "bottlenecked elsewhere" at time t. This information is carried in $s[l]$ in the control packets of flows in Q_l, where Q_l is the set of all flows that cross link l:

$$\hat{\mathrm{E}}(t) = \{f : f \in Q_l, s^l_f(t) = \mathrm{E}\}.$$

3. We will use $\hat{\mathrm{B}}(t)$ to refer to the set of flows that are considered by link l to be in $\hat{B}$ or "bottlenecked here" at the link l at time t. This information is carried in $s[l]$ in the control packets of flows in Q_l:

$$\hat{\mathrm{B}}(t) = \{f : f \in Q_l, s^l_f(t) = \mathrm{B}\}.$$

The state at the link comprises four variables. The first two are the same as in n-PERC: $SumE$, the sum of allocations of flows that are considered by link l to be bottlenecked elsewhere, and $NumB$, the number of flows that are considered by link l to be bottlenecked at the link. There are two additional variables, $MaxE$ and $MaxE'$, which are used to estimate the maximum allocation of a flow that is considered by link l to be bottlenecked elsewhere.

We characterize the link state for a link l at any time t, explicitly in terms of the information carried in the control packets of link l's flows (these relations follow from Algorithm 6):

1. $SumE(t)$ is the sum of allocations of flows that are considered by link l to be in $\hat{\mathrm{E}}$ or "bottlenecked elsewhere" at time t. This information is stored in the $SumE$ variable at the link:

$$SumE(t) = \sum_{f \in \hat{\mathrm{E}}(t)} a^l_f(t).$$

2. $NumB(t)$ is the number of flows that are considered by link l to be in $\hat{\mathrm{B}}$ or "bottlenecked here" at the link l at time t. This information is stored in the $NumB$ variable at the link:

$$NumB(t) = |\hat{\mathrm{B}}(t)|.$$

3. We use $M(t)$ to refer to the maximum allocation of an $\hat{E}$ flow at link l at time t. We do *not* track this explicitly at the link because it would require per-flow state. Instead, we maintain two variables, $MaxE$ and $MaxE'$, at the link such that $MaxE$ is an upper bound to M.

$$M(t) = \max_{f \in \hat{\mathrm{E}}(t)} a^l_f(t).$$

3.2.2 The s-PERC Algorithm

Algorithm 6 describes the **algorithm at the link** that runs each time a control packet is seen. In addition, Algorithm 5 is run periodically at the link algorithm to reset variables $MaxE$ and $MaxE'$.

The per-packet algorithm at the link is similar to n-PERC except for the following differences. The limit rate e of the flow is the lowest *propagated bottleneck rate* that it gets from any *other* link. If all other links have their ignore bits set $i[l] = 1$, the limit rate is ∞. After the update, the link checks if the bottleneck rate is too low. If $b < MaxE$, it sets the ignore bit to True since the rate may be too low to propagate.

In order to estimate the maximum allocation of an $\hat{E}$ flow, the link needs $MaxE$ and an additional variable, $MaxE'$. Both $MaxE'$ and $MaxE$ are updated when a flow is classified as $\hat{E}$ (lines 21–22 in Algorithm 6) so that they both increase each time a flow is classified as $\hat{E}$ and are allocated a higher value. They are also reset every round. $MaxE'$ is reset to 0, so that it starts afresh every round, while $MaxE$ is reset to the old value of $MaxE'$ so that it starts with the maximum $\hat{E}$ allocation seen in the last round. Since we defined round to be long enough that every flow is seen at every link at least once, $MaxE$ is an upper bound to M. We formalize this in §3.2.4.

Algorithm 5 Timeout action at link l for s-PERC, every round starting from time $T_0(l)$

1: $MaxE \leftarrow MaxE'$
2: $MaxE' \leftarrow 0$

Algorithm 6 Control Packet Processing at Link l for s-PERC. See also Algorithm 5.

1: $SumE = 0$: sum of allocations of flows bottlenecked elsewhere, that is, in $\hat{E}$ ▷ Initial state at link l
2: $NumB = 0$: number of flows bottlenecked here, that is, in $\hat{B}$
3: $MaxE = 0$: maximum allocation of flows moved to $\hat{E}$ since last round
4: $MaxE' = 0$: maximum allocation of flows moved to $\hat{E}$ in this round
5: **if** $s[l] == \mathrm{E}$ **then** ▷ Flow was last bottlenecked elsewhere
6: $\quad SumE \leftarrow SumE - a[f]$ ▷ Update link state to assume flow is going to be bottlenecked
7: $\quad NumB \leftarrow NumB + 1$
8: $b \leftarrow \frac{C - SumE}{NumB}$
9: $e \leftarrow min_{m \in P_f \setminus l, \; \mathrm{i}[m]=0\}} b[m]$ (or ∞ if there is no other link in P_f with ignore bit unset) ▷ Find flow's new limit rate
10: $a \leftarrow \min(b, e)$ ▷ Find flow's new allocation
11: **if** $b <= e$ **then** $s \leftarrow \mathrm{B}$ ▷ Flow is now bottlenecked here, classify as $\hat{B}$
12: **else** $s \leftarrow \mathrm{E}$ ▷ Flow is now bottlenecked elsewhere, classify as $\hat{E}$
13: $b[l] \leftarrow b, \quad a[l] \leftarrow a, \quad s[l] \leftarrow s$ ▷ Update control packet
14: **if** $b < MaxE$ **then** $i[l] \leftarrow 1$ ▷ Bottleneck rate is low; do *not* propagate it
15: **else** $i[l] \leftarrow 0$ ▷ Bottleneck rate is high enough; propagate it
16: **if** flow is leaving **then** ▷ Update link state to remove flow
17: $\quad NumB \leftarrow NumB - 1$
18: **else if** $s == \mathrm{E}$ **then** ▷ Update link state to reflect flow is in $\hat{E}$
19: $\quad NumB \leftarrow NumB - 1$
20: $\quad SumE \leftarrow SumE + a$
21: $\quad MaxE \leftarrow \max(MaxE, a)$
22: $\quad MaxE' \leftarrow \max(MaxE', a)$

There are two properties of the s-PERC algorithm, which we will prove before diving into an example, and then the convergence proof.

1. The bottleneck rate calculation (Lines 5–8 in Algorithm 5) ensures that when a flow is moved to $\hat{E}$, the new value of $C - SumE/NumB$ at the link is consistent with (higher than) the limit rate of the flow, even if it is not locally max-min fair and even if it is possibly inconsistent with (lower than or equal to) the limit rates of flows previously moved to $\hat{E}$ (Lemma 3.2.1.)

2. The check before propagating a bottleneck rate (Lines 14–15 in Algorithm 5) ensures that propagated bottleneck rates are high enough to allow the true bottleneck link to recognize its bottleneck flows, that is, once links $L_1, ..L_n$ have converged, any rate propagated by links in LG_n is at least $\mathrm{R}(l)$ for $l \in L_{n+1}$, which is high enough for l to recognize its bottleneck flows (Lemma 3.2.2.)

As we have seen in the n-PERC example, the link state following an update need not be consistent with the

set of $\hat{E}$ flows at the link–the bottleneck rate can drop below the bandwidth allocated to an $\hat{E}$ flow previously. However, because the bottleneck rate calculation for a flow assumes that the flow is going to bottlenecked, we can at least guarantee that if a link classifies a flow f as $\hat{E}$, the link state following an update is consistent with flow f's allocation, even if it is inconsistent with flows previously classified as $\hat{E}$. We will use this property in s-PERC's proof of convergence.

Lemma 3.2.1. *If a flow f moves to $\hat{E}$ during an update at link l, after the update, either $NumB = 0$ and $C > SumE$ or $R = (C - SumE)/NumB$ exceeds the flow's limit rate. In the first case, all flows are in $\hat{E}$ and there is spare capacity at the link. In the second case, the bandwidth available to the $\hat{B}$ flows after removing the allocations of the $\hat{E}$ flows exceeds the latest $\hat{E}$ flow's limit rate.*

Proof. Let $NumB = NumB(t), SumE = SumE(t)$ represent the link state at line 8 in the Algorithm, as they are used to compute b. Since the flow moved to $\hat{E}$, we know that:

$$b = \frac{C - SumE(t)}{NumB(t)} > e$$

$$C - SumE(t) > NumB(t) \cdot e \tag{3.1}$$

Following the update, the new link state is $NumB(t^+) = NumB(t) - 1$ and $SumE(t^+) = SumE(t) + e$.

If there are no more flows marked bottlenecked, that is, $NumB(t) = 1$, then we are in the first case since $NumB(t) - 1 = 0$ and $C > SumE(t^+)$, from Equation 3.1. Otherwise, the new value of $(C - SumE)/NumB$ following the update is:

$$\begin{aligned} \frac{C - SumE(t^+)}{NumB(t^+)} &= \frac{C - (SumE(t) + e)}{(NumB(t) - 1)} \\ &= \frac{(C - SumE(t)) - e}{(NumB(t) - 1)} \\ &> \frac{NumB(t) \cdot e - e}{(NumB(t) - 1)} && \text{(using Equation 3.1)} \\ &= e \end{aligned}$$

□

3.2.3 When Should We Propagate the Bottleneck Rate?

Given that at any link $MaxE$ is an upper bound to the maximum allocation of an $\hat{E}$ flow, we can show that once the set of flows and links stabilize, within a $round$ any propagated rate will be at least $\mathrm{R}(l)$, where l is the link with the lowest max-min rate. This is helpful for l to recognize its bottleneck flows. More generally, once links with the n lowest rates have converged, any propagated rate will be high enough to enable a link $l \in L_{n+1}$ to recognize its bottleneck flows $f \in \mathrm{B}(l)$:

Lemma 3.2.2. *Let l be any link in L_{n+1} and f be any flow in $\mathrm{B}(l)$. Suppose $L_1, ..., L_n$ have converged at time T. The bottleneck rate for f from a link $x \in P_f$ is either not propagated or at least $\mathrm{R}(l)$ from time $T + 1\, round$.*

Corollary 3.2.3. *The limit rate of f at link l is at least $\mathrm{R}(l)$ from time $T + 2\, rounds$.*

Proof. Flow f is seen at every link in P_f at least once between times $T + 1\, round$ and $T + 2\, rounds$ and picks up a bottleneck rate that is either not propagated or $\mathrm{R}(l)$. Flow f's limit rate at link l is the lowest propagated bottleneck rate from any link in $P_f \setminus l$ and hence is at least $\mathrm{R}(l)$ from time $T + 2\, rounds$. □

Proof of Lemma 3.2.2. Consider an update for a flow $f \in \mathrm{B}(l)$ at a link $x \in P_f$ at some time after $T + 1\, round$. Suppose $b \geq MaxE$ and the rate is propagated (line 15 of Algorithm 6). If $MaxE \geq \mathrm{R}(l)$, the proof is done. So let us consider the case when $MaxE < \mathrm{R}(l)$.

Let C be the capacity of link x and let $SumE$ and $NumB$ represent the link state of link x at line 8 of Algorithm 6 just before it computes a bottleneck rate for f. We will break down aggregate link state in terms of the contribution of the flows carried by $L_1, ..., L_n$, that is, $FL_n(x)$, and the remaining flows $FG_n(x)$. The bottleneck rate computed for the flow f satisfies the following (omitting x for simplicity):

$$
\begin{aligned}
b &= (C - SumE)/NumB \\
b &= \frac{C - \sum_{f \in \hat{\mathrm{E}} \cap FL_n} a_f - \sum_{f \in \hat{\mathrm{E}} \cap FG_n} a_f}{|\hat{\mathrm{B}} \cap FL_n| + |\hat{\mathrm{B}} \cap FG_n|} \\
&= \frac{C - \sum_{f \in FL_n} \mathrm{A}(f) - \sum_{f \in \hat{\mathrm{E}} \cap FG_n} a_f}{|\hat{\mathrm{B}} \cap FG_n|}
\end{aligned}
$$

In the last step, we used the fact that the flows carried by $L_1, ..., L_n$ are all classified correctly as $\hat{E}$ and allocated their max-min rates at link x after time $T + 1\, round$.

Rearranging the terms, we get:

$$
\begin{aligned}
C - \sum_{f \in FL_n} \mathrm{A}(f) &= b \cdot |\hat{\mathrm{B}} \cap FG_n| + \sum_{f \in \hat{\mathrm{E}} \cap FG_n} a_f \\
&\leq b \cdot |\hat{\mathrm{B}} \cap FG_n| + MaxE \cdot |\hat{\mathrm{E}} \cap FG_n| \quad \text{(using Lemma 3.2.4, } \max_{f \in \hat{\mathrm{E}}} a_f = M \leq MaxE\text{)} \\
&\leq b \cdot |\hat{\mathrm{B}} \cap FG_n| + \mathrm{R}(l) \cdot |\hat{\mathrm{E}} \cap FG_n| \quad \text{(we are considering the case } MaxE < \mathrm{R}(l)\text{)}
\end{aligned} \tag{3.2}
$$

Since f is bottlenecked at link l, any other link $x \in P_f$ must have a max-min rate of at least $\mathrm{R}(l)$. We again use the property of link x, which we used in our analysis of the $Fair$ algorithm—link x has at least $\mathrm{R}(l)$ for each of the flows in FG_n, after removing the max-min allocation of flows carried by $L_1, ..., L_n$ (Lemma 2.5.3):

$$
C - \sum_{f \in FL_n} A(f) \geq \mathrm{R}(l) \cdot |FG_n| \tag{3.3}
$$

Combining Equations 3.2 and 3.3, and noting that when we compute b, there is at least one flow in $\hat{B}$ ($|\hat{\mathrm{B}} \cap FG_n| > 0$), we see that the bottleneck rate computed at link x for $f \in \mathrm{B}(l)$ is at least $R(l)$, that is, $b \geq \mathrm{R}(l)$.

$$
\begin{aligned}
b \cdot |\hat{\mathrm{B}} \cap FG_n| + \mathrm{R}(l) \cdot |\hat{\mathrm{E}} \cap FG_n(x)| &\geq \mathrm{R}(l) \cdot |FG_n| && \text{(from Equations 3.2 and 3.3)} \\
b \cdot |\hat{\mathrm{B}} \cap FG_n| &\geq \mathrm{R}(l) \cdot |\hat{\mathrm{B}} \cap FG_n| \\
b &\geq \mathrm{R}(l) && \text{(since } |\hat{\mathrm{B}} \cap FG_n| > 0\text{)}
\end{aligned}
$$

□

3.2.4 How Do We Approximate the Maximum $\hat{E}$ Allocation?

Why should $MaxE$ be an upper bound to M? Maintaining the exact value of M at a link in an online fashion requires per-flow state since the set of $\hat{E}$ flows and their allocations may be changing with every control packet update at the link [15]. Instead, as described in §3.2.2 every link l maintains variables $MaxE$ and $MaxE'$, that are updated each time a flow is moved to $\hat{E}(l)$ and reset every $round$ starting from some time $T_0(l)$ (Algorithm 5.)

We will derive explicit expressions for $MaxE(T)$ and $MaxE'(T)$ in terms of the values carried by the control packet at or before time T in order to prove two properties of $MaxE(T)$: first, that it is an upper

bound to $M(T)$, the maximum allocation of any flow in $\hat{E}(T)$; second, that it does reflect $M(T)$ after the set of $\hat{E}$ allocations stabilize. This allows us to show that the propagated rate is high enough, and once the allocations and the bottleneck rate stabilize to their correct values, the propagated rate is exactly the bottleneck rate.

Lemma 3.2.4. *Consider a link l. $MaxE(T) \geq M(T) = \max_{f \in \hat{E}(T)} a^l_f(T)$ for all times T following $T_0(l) + 1\ round$.*

Lemma 3.2.5. *Suppose the set of $\hat{E}$ allocations at link l stabilize by some time T_s. In other words, the value of the maximum $\hat{E}$ allocation M is unchanged for all times from T_s. Then $MaxE$ at link l stabilizes to the maximum $\hat{E}$ allocation $M(T_s)$ within two rounds of T_s. Moreover, if $T_u < T_s$ is the last time that a flow was classified as $\hat{E}$ and allocated more than $M(T_s)$, then $MaxE$ stabilizes to $M(T_s)$ after two rounds following T_u. (Note that for simplicity, we assume that $T_u, T_s > T_0(l) + 1\ round$, so that all flows have been seen at least once following the link's first reset.)*

$MaxE'(T)$ reflects the maximum bandwidth allocated to an $\hat{E}$ flow in the current round at the switch (that is, since the last reset before time T) while $MaxE(T)$ reflects the maximum bandwidth allocated to an $\hat{E}$ flow in the last two rounds at the switch. We already offered an explanation for why this is true in §3.2. Here, we formally derive explicit expressions for $MaxE'(T)$ and $MaxE(T)$ in terms of the allocations carried in the control packets in the two rounds preceding T.

We consider $MaxE'$ first.

Notice that every link l starts independently at some time $T_0(l)$ before any of the flows have started and resets $MaxE'$ and $MaxE$ every $round$ following $T_0(l)$. During the course of the algorithm, resets at different links will be happening at different times; they are not synchronized. Given an arbitrary time $T \geq T_0(l)$, we use $\lfloor T \rfloor_l$ to refer to the most recent time of reset at link l at or before T.

$$\lfloor T \rfloor_l = T_0(l) + \text{round}.\lfloor \frac{T - T_0(l)}{\text{round}} \rfloor \tag{3.4}$$

We shall use the notation T_{fl} to refer to the set of discrete times when a flow f carried by link l is updated at the link.

Suppose we want to calculate the value of $MaxE$ and $MaxE'$ at some time T. When $T < T_0(l)$, $MaxE$ and $MaxE'$ reflect their initial value of 0. If $T \geq T_0(l)$, we know from Algorithm 6 that $MaxE'$ was last reset to 0 at time $\lfloor T \rfloor_l$. Thereafter, it was updated every time a flow's control packet was seen at the link and the flow was labeled $\hat{E}$. Hence, for $T \geq T_0(l)$:

$$
\begin{aligned}
MaxE'(t) &= max(MaxE'(\lfloor T \rfloor_l), max_{\substack{t \in T_{lf}, f \in \hat{E}(t^+) \\ \lfloor T \rfloor_l < t < T}} a_f^l(t)) \\
&= max_{\substack{t \in T_{lf}, f \in \hat{E}(t^+) \\ \lfloor T \rfloor_l < t < T}} a_f^l(t) \qquad \text{(replacing } MaxE'(\lfloor T \rfloor_l) = 0\text{)}
\end{aligned} \tag{3.5}
$$

Next, we derive an expression for $MaxE$. Given any time T, we know from Algorithm 6 that $MaxE$ was last reset to $MaxE'$ at time $\lfloor T \rfloor_l$ to the value of $MaxE'$ just before the reset. Thereafter, it was updated every time a flow's control packet was seen at the link and the flow was labeled $\hat{E}$. Hence,

$$
\begin{aligned}
MaxE(t) &= max(MaxE'(\lfloor T \rfloor_l^-), max_{\substack{t \in T_{lf}, f \in \hat{E}(t^+) \\ \lfloor T \rfloor_l < t < T}} a_f^l(t)) \\
&= max_{\substack{t \in T_{lf}, f \in \hat{E}(t^+) \\ T_0 < t < T}} a_f^l(t)) \qquad \text{(when } \lfloor T \rfloor_l = T_0, \text{ then } MaxE'(T_0^-) = 0\text{)}
\end{aligned} \tag{3.6}
$$

$$
= max_{\substack{t \in T_{lf}, f \in \hat{E}(t^+) \\ \lfloor T \rfloor_l - \text{round} < t < T}} a_f^l(t) \qquad \text{(otherwise, replacing } T = \lfloor T \rfloor_l^- \text{ in Equation 3.5)} \tag{3.7}
$$

This allows us to prove Lemma 3.2.4 and Lemma 3.2.5.

Proof of Lemma 3.2.4. From Equation 3.7, we can see that $MaxE(T)$ includes all updates in the round before T. Hence, it includes the latest allocations of flows that are in $\hat{E}(T)$. So $MaxE(T) \geq M(T)$. □

Proof of Lemma 3.2.5. From Equation 3.7, we can see that for $T \geq T_s + 2\, rounds$, $MaxE(T)$ includes all updates in the round preceding T. So it includes the latest allocations of flows that are in $\hat{E}(T) = \hat{E}(T_s)$. Moreover, we can see that it cannot include any update more than two-rounds-old. So it excludes updates before time T_s that may be inconsistent with the final $M(T_s)$. Hence, $MaxE(T) = M(T_s)$ for $T \geq T_s + 2\, rounds$. We define an inconsistent update with respect to $M(T_s)$, as an update where a flow is classified as $\hat{E}$ and allocated more than $M(T_s)$. If the time of the last inconsistent update T_u is known, then the same argument applies for $T > T_u + 2\, rounds$ as for $T \geq T_s + 2\, rounds$. Notice, however, that if $T < T_u + 2\, rounds$ and $\lfloor T \rfloor - round < T_u$ then $MaxE(T)$ would include the inconsistent allocation made at time T_u and exceed $M(T_s)$.

□

3.2.5 s-PERC in Action

Let's walk through an example to understand how the rates evolve in the s-PERC algorithm (Figure 3.4). We will use the same topology and workload we used for demonstrating n-PERC and $Fair$, which we reproduce here for convenience (Figure 3.3.)

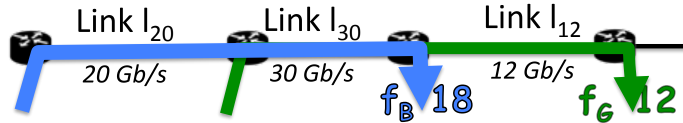


Figure 3.3: Example setup for s-PERC in action. There are $J = 2$ flows and $K = 3$ links. Each flow crosses a subset of the links. The green flow f_G is bottlenecked to 12 Gb/s at link l_{12}, while the blue flow f_B is bottlenecked to 18 Gb/s at link l_{30}. We show here the link capacities and the max-min fair rate allocations for the flows.

Round	Time	Event	MaxE	e	b	a	state	ignore bit
1	1	Flow f_B @ Link l_{20}	0	∞	20	20	B	
1	2	Flow f_B @ Link l_{30}	0	20	30	20	E	
1	3	Flow f_W @ Link l_{30}	20	∞	10	10	B	T
1	**4**	**Flow f_W @ Link l_{12}**	**0**	**∞**	**12**	**12**	**B**	
l_{20}: NumB=1,SumE=0,MaxE=0,MaxE'=0 l_{30}: NumB=1,SumE=20,MaxE=20,MaxE'=0 l_{12}: NumB=1,SumE=0,MaxE=0,MaxE'=0					f_B: b[l_{20}]=20, b[l_{30}]=30 f_W: b[l_{30}]=20(**T**), b[l_{12}]=12			
2	5	Flow f_W @ Link l_{30}	20	12	10	10	B	T
2	6	Flow f_W @ Link l_{12}	0	∞	12	12	B	
2	7	Flow f_B @ Link l_{20}	0	30	20	20	B	
2	8	Flow f_B @ Link l_{30}	20	20	15	15	B	T
l_{20}: NumB=1,SumE=0,MaxE=0,MaxE'=0 l_{30}: NumB=2,SumE=0,MaxE=0,MaxE'=0 l_{12}: NumB=1,SumE=0,MaxeE=0,MaxE'=0					f_W: b[l_{30}]=10(**T**), b[l_{12}]=12 f_B: b[l_{20}]=20, b[l_{30}]=15(**T**)			
3	**9**	**Flow f_W @ Link l_{30}**	0	**12**	**15**	**12**	**E**	
3	10	Flow f_W @ Link l_{12}	0	15	12	12	B	
3	**11**	**Flow f_B @ Link l_{30}**	**12**	**20**	**18**	**18**	**B**	
3	**12**	**Flow f_B @ Link l_{20}**	**0**	**18**	**20**	**18**	**E**	
l_{20}: NumB=0,SumE=18,MaxE=18,MaxE'=0 l_{30}: NumB=2,SumE=0,MaxE=12,MaxE'=0 l_{12}: NumB=1,SumE=0,MaxE=0,MaxE'=0					f_W: b[l_{30}]=15, b[l_{12}]=12 f_B: b[l_{20}]=20, b[l_{30}]=18			

Figure 3.4: Control packet updates for first three rounds of the s-PERC algorithm. We show the link state ($MaxE$ before the update), the limit rate e computed for the flow, and the control packet state (the bottleneck rate b, allocation a, bottleneck state s, and ignore bit after the update). We also show the link state ($SumE$, $NumB$, $MaxE$) at the end of each round.

Flow f_B is first seen at link l_{20}. The link assumes the flow is not limited anywhere else, computes a

bottleneck rate of 20, and allocates this to the flow. Flow f_B is then seen at link l_{30}. The link computes a bottleneck rate of 30 Gb/s, sees that the flow is limited to 20 Gb/s, and allocates 20 Gb/s. So far the updates are exactly the same as the $Fair$ algorithm.

When flow f_G is seen at link l_{30} during **update 3**, the link knows that the total allocation of E flows is 20 Gb/s. It assumes flow f_G is not limited ($e = \infty$) and uses Equation 2.2—where $NumB = 1$ and $SumE = 20$—to compute a bottleneck rate of $b = 10$ Gb/s for flow f_G. Since the limit rate $e = \infty$ is greater, the link concludes that the flow is bottlenecked at the link, and allocates $a = b = 10$ Gb/s. However, because $MaxE = 20$ Gb/s is higher than b, the link guesses that b may be too low to propagate to the next link and sets the ignore bit to True in the control packet.

When flow f_G is next seen at its actual bottleneck link l_{12} during **update 4**, its bottleneck rate 10 Gb/s from link l_{30} is ignored and the link assumes flow f_G is not limited ($e = \infty$). The link computes a bottleneck rate $b = 12$ Gb/s. Since the limit rate is greater, the link correctly assumes the flow is bottlenecked at the link and allocates $b = 12$ Gb/s. Hence, link l_{30} avoids propagating its bad bottleneck rate 10 Gb/s to link l_{12} and enables link l_{12} to immediately identify its bottleneck flow f_G.

The allocation at link l_{30} for flow f_G is still 10 Gb/s however. It is not until after update 8 (when both flows are B) that the bottleneck rate at link l_{30} for flow f_G increases from $b = 10$ Gb/s to $b = 15$ Gb/s. During **update 9**, $b = 15$ Gb/s exceeds $e = 12$ Gb/s and flow f_G is correctly marked E at link l_{30}. We say the flow f_G has converged at this point, since we can show that it is forever marked B at its bottleneck link l_{12}, E at link l_{30} and allocated exactly its max-min fair rate 12 Gb/s at both links.

Let's consider flow f_B next. Note that during update 8 at link l_{30}, flow f_B is allocated a locally max-min fair rate based on the limit rates, but because $MaxE > b = 15$ Gb/s, the link does not propagate the bottleneck rate to link l_{20}.

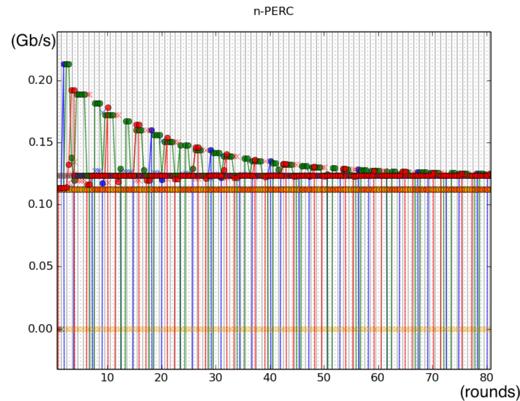
$MaxE$ is reset at the end of the round to 0 since no flows at link l_{30} were marked E. After update 9, once flow f_G is marked E, it stabilizes to $MaxE = 12$ Gb/s. During the next update of flow f_B at link l_{30} (**update 11**), $MaxE = 12$ Gb/s is less than $b = 18$ Gb/s and not only does link l_{30} correctly mark flow f_B as B at $b = 18$ Gb/s, but it also propagates the bottleneck rate to link l_{20}. As a result, during **update 12** link l_{20} correctly marks the flow f_B E at its max-min fair rate. We say flow f_B has converged at this point, since we can show that it is forever marked B at its bottleneck link l_{30}, E at link l_{20} and allocated exactly its max-min fair rate 18 Gb/s at both links.

3.3 Simulations of n-PERC and s-PERC

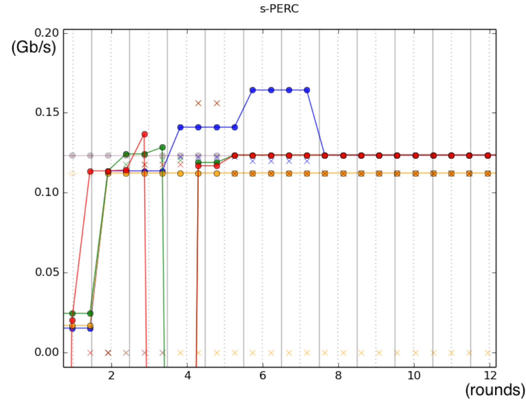
We used numerical simulations (in Python) of n-PERC and s-PERC in a fully connected network of K links, where each link has a uniform capacity of 10 Gb/s and a uniform delay of 10 us (with some random jitter on the order of a nanosecond to allow reordering of packets). For each workload, we randomly generate a set of K long-lived flows, where each flow traverses the same number of links P (for simplicity.) Note that this is not a packet-level simulation, and we only perform the calculations involved in control packet processing at intervals dictated by the link delays and configured timeouts (e.g., every round) using an event-queue. Moreover, we use a static value of $round$ instead of dynamically adjusting it based on the number of flows.

We ran thousands of different network and flow setups for different configurations of J, K, P such as $J = K = 1000$ and $J = K = 100$ and $P \in \{10, 40, 50, 80\}$ and different random seeds for the ordering of packets, and the paths of flows. We verified that s-PERC takes no more than six rounds per bottleneck link rate to converge. For n-PERC, while we did not find a simulation that failed to converge, we did find that for some setups, the convergence time can get much longer than six rounds for each bottleneck link. The worst case convergence time we observed was 60 rounds for four bottleneck links (two distinct bottleneck rates) (Figure 3.5).

Figure 3.5a shows a time series of the bottleneck rates at the links for n-PERC's worst example. This setup has $K = 100$ links carrying $J = 100$ flows. We used a uniform capacity of 10 Gb/s and a uniform delay of $10\mu s$ for all links. Every flow crosses $P = 80$ random links. The value of $round$ is configured to $880\mu s$. It turns out that all flows are bottlenecked at one of four links. The bottleneck rates are identical for three of the four links. The n-PERC algorithm takes almost 60 rounds for all four links (two distinct bottleneck rates) to converge, while s-PERC converges within six rounds total for the same example (Figure 3.5b).



(a) Time series of fair share rates (o) and $MaxE$ (x) at bottleneck links for n-PERC.



(b) Time series of fair share rates (o) and $MaxE$ (x) at bottleneck links for s-PERC.

Figure 3.5: An example for which n-PERC takes a long time to converge. Note that we use $(C - SumE/NumB)$ as the value of the fair share rate (-1 when $NumB = 0$). The faded disks (o) denote the ideal max-min fair share rates.

3.4 Convergence Proof of s-PERC

3.4.1 Centralized Water-Filling Algorithms

The centralized water-filling algorithm [22] and the CPG algorithm [75] are two instances of a class of centralized algorithms to compute max-min fair rates, which we call $\mathrm{WF^k}$ algorithms. A $\mathrm{WF^k}$ algorithm is an iterative algorithm (see Algorithm 7), where in each iteration, links compute a fair share rate (which we call WF rate to distinguish from the max-min rate), and then the algorithm picks a set of links to declare bottlenecked and remove from the network. A link is removed in an iteration if it has the lowest WF rate of all neighbors up to k-degrees, where a (first-degree) neighbor of a link l is defined as a link that shares a flow with l in the given iteration. Hence, $k = \infty$ corresponds to the centralized water-filling algorithm, where a link is removed if it has the lowest WF rate out of *all* links that remain in the iteration, while $k = 1$ corresponds to the CPG algorithm, where a link is removed if it has the lowest rate of its first-degree neighbors. When a link is removed, the flows carried by the link in that iteration are allocated the WF rate of the link from that iteration and also removed from the network. The algorithm terminates when there are no more flows. As we will show, all $\mathrm{WF^k}$ algorithms compute max-min fair rates for the flows, and s-PERC's convergence behavior can be understood in terms of WF^2.

Notation: We summarize notation related to $\mathrm{WF^k}$ algorithms in Table 3.3. We use $R^k_n[l]$ to denote the fair share rate computed by link l in iteration n of some $\mathrm{WF^k}$ algorithm. In general, we use the term *WF rate* when talking about $R^k_n[l]$ rather than the term "fair share rate," to distinguish it from the max-min fair rate

Table 3.3: Commonly used notation in the context of a WF^{k} algorithm run for a given set of flows and links, where $k = 1$ for centralized water-filling algorithm, $k = 1$ for CPG algorithm, and $k = 2$ for the algorithm we introduce to analyze s-PERC. We also note in brackets how these new terms relate to terms introduced in §2.1, when we described the setup for PERC algorithms. The first set of rows correspond directly to variables in the WF^{k} algorithm.

Notation	Meaning
$R_n^k[l]$	(WF rate) fair share rate computed by link l in iteration n
$N_n^k[l]$	number of flows carried by link l in iteration n
$C_n^k[l]$	remaining capacity of link l in iteration n
$A[f]$	bandwidth allocated to flow f
$X1_n^k(l)$	(first-degree neighbors) set of links that share a flow with l in iteration n
$X2_n^k(l)$	(second-degree neighbors) set of links that share a flow with links in $X1_n^k(l)$
W^k	total number of iterations (Note that $W^\infty = W$ and $W^1 = D$)
L_i^k	set of links removed in iteration i, by convention L_0^k is the empty set
	and $L_{W^k+1}^k$ is the set of links that are never removed (Note that $L_i^\infty = L_i$)
LL_n^k	set of links removed in iterations up to and including n, that is, $L_0^k, ..., L_n^k$
FL_n^k	set of flows carried by links in LL_n^k
$FL_n^k(l)$	subset of flows in FL_n^k carried by link l
LG_n^k	set of links that remain in iteration $n+1$, that is, $L_{n+1}^k, ..., L_{W^k+1}^k$
FG_n^k	set of flows carried by links in LG_n^k that do not cross LL_n^k
$FG_n^k(l)$	subset of flows in FG_n^k carried by link l

of the link. Notice that variables in the algorithm that describe links—such as the set of active links L, the remaining capacity C, and the number of flows per active link N—are updated in every iteration while the allocation of a flow f $A[f]$ is filled in once, when the flow is removed. We use LG_n^k and FG_n^k to denote the set of links and flows that remain at beginning of iteration n+1. Using $FG_n^k(l)$ to denote the set of flows carried by link l in iteration $n+1$, we define a first-degree neighbor of a link l in an iteration $n+1$ as the set of links that share a flow with l in the iteration:

$$X1_{n+1}^k(l) = \cup_{f \in FG_n^k(l)} P_f$$

A second-degree neighbor, which we denote using $X2_{n+1}^k(l)$, shares a flow with a first-degree neighbor and so on.

3.4.2 Partitioning Bottleneck Links Using WF^{k} Algorithms

It can be shown that all WF^{k} algorithms compute max-min fair rates for the flows, and the WF rate of a link when it is removed is exactly the max-min fair rate of its bottleneck flows.

We can prove the following property of a WF^{k} algorithm:

Algorithm 7 The $\mathrm{WF^k}$ algorithm to compute max-min rates. For any link l that is removed in iteration n, we allocate $A[f] = R[l] = C[l]/N[l]$ as computed in iteration n to each of its flows in Q_l during iteration n, and remove the flow.

1: L : set of all links in the network that have at least one flow
2: C : remaining link capacities, N: number of flows ▷ arrays indexed by link
3: R : remaining link capacity per flow, Q: active flows ▷ arrays indexed by link
4: A: bandwidth allocation to flow, P: array of links per flow ▷ arrays indexed by flow
5: $iteration \leftarrow 0$
6: **while** L is not empty **do**
7: $\quad iteration \leftarrow iteration + 1$
8: $\quad$ **for all** $l \in L$ **do** $R[l] \leftarrow C[l]/N[l]$
9: $\quad links_to_remove \leftarrow \{\}, flows_to_remove \leftarrow \{\}$
10: $\quad$ **for all** $l \in L$ **do**
11: $\quad\quad$ **if** $\mathrm{WF^k} == \mathrm{WF^\infty}$ **then** $minRate = \leftarrow \min_{x \in L} R[x]$
12: $\quad\quad$ **else** $minRate = min^{\star}_{i=1} \min_{x \in Xi(l)} [R[x]$ ▷ Xi(l) denotes ith degree neighbor of l
13: $\quad\quad$ **if** $R[l] == minRate$ **then**
14: $\quad\quad\quad$ add l to $links_to_remove$
15: $\quad\quad\quad$ **for all** $f \in Q[l]$ **do**
16: $\quad\quad\quad\quad$ add f to $flows_to_remove$
17: $\quad\quad\quad\quad$ $A[f] = R[l]$
18: $\quad$ **for all** $f \in flows_to_remove$ **do**
19: $\quad\quad$ **for all** $l \in P[f]$ **do**
20: $\quad\quad\quad$ $C[l] \leftarrow C[l] - A[f]$
21: $\quad\quad\quad$ $N[l] \leftarrow N[l] - 1$
22: $\quad\quad\quad$ remove f from $Q[l]$
23: $\quad$ remove $links_to_remove$ from L

Lemma 3.4.1. [Monotonic Rate Behavior] *Let link $l \in L_n^k$. The WF rate computed for link l (line 8 of Algorithm 7) is non-decreasing from iterations 1 through n.*

Proof. Consider any iteration i between 1 and n. There are two possibilities. If no flows are removed from link l during iteration i, the rate is unchanged. Otherwise, let $F = FG_{i-1}^k(l) \cap Q_{L_i^k}$ denote the set of flows carried by l that are removed in iteration i and let f be one such flow, removed because of some link $z \in L_i^k$. We know that $A[f] = R_i^k[z]$ and from lines 12–13 that $R_i^k[z] \leq R_i^k[l]$ because link l is a first-degree neighbor of z. Hence, for all flows f carried by l that are removed in iteration i, $A[f] \leq R_i^k[l]$. Link l's allocation in the next iteration is:

$$
\begin{aligned}
R_{i+1}^k[l] &= \frac{C_{i+1}^k[l]}{N_{i+1}^k[l]} \\
&= \frac{C_i^k[l] - \sum_{f \in F} A[f]}{N_i^k[l] - |F|} \\
&\geq \frac{C_i^k[l] - \sum_{f \in F} R_i[l]}{N_i^k[l] - |F|} && \text{(using } A[f] \leq R[l]) \\
&= \frac{N_i^k[l] R_i[l] - \sum_{f \in F} R_i[l]}{N_i^k[l] - |F|} && \text{(replacing } R_i[l] = C_i^k[l]/N_i^k[l]) \\
&= R_i^k[l]
\end{aligned}
$$

□

This result will allow us to prove that the algorithm does compute max-min fair rates.

Theorem 3.4.2. *The* WF^{k} *algorithm (Algorithm 7) terminates after a finite number of iterations W^2, and all flows in the network are allocated their max-min fair rates.*

Corollary 3.4.3. *If a link is removed in some iteration n, the WF rate computed for the link in iteration n is the max-min rate of its bottleneck flows.*

Proof of Theorem 3.4.2. We will prove that every flow is bottlenecked at some link. This implies that the rate allocation of the flows is max-min fair because in order to increase the rate of a flow f, one must decrease the rate of another flow e that crosses f's bottleneck link and gets a lower or equal rate [58].

In each iteration, a link is removed if it has the lowest rate of its neighbors (as defined in lines 11–13). So at least one link (e.g., the link with the minimum rate of L) is removed in each iteration. The algorithm proceeds until there are no more links with active flows. So every flow gets a rate.

Consider a flow f that is removed in iteration n and allocated $A[f] = R_n^k[l]$ because of some link l (line 17). We will show that flow f is bottlenecked at link l. In other words, link l is fully used and the flow f gets

the maximum allocation of all flows in Q_l. Link l is fully used because when it is removed, its capacity is shared equally among all flows that it carries in iteration n.

To show that flow f gets the maximum allocation, we note that any flow f' that is removed from link in an iteration $m < n$ because of some link z is allocated $A[f'] = R^k_m[z] \leq R^k_m[l] \leq R^k_n[l]$. The first inequality follows from lines 12–13 of the algorithm since l and z share f', and the second inequality follows from the Monotonic Rate Behavior of link l (Lemma 3.4.1). □

Hence, a WF$^{\text{k}}$ algorithm partitions the set of bottleneck links in the network into $L^k_1, ..., L^k_{W^k}$, where L^k_i is the set of links removed in iteration i and W^k is the number of iterations until the algorithm terminates. The links that remain in the network when the algorithm terminates are not bottlenecked, and we use $L^k_{W^k+1}$ to refer to them. We use the convention that L^k_0 is the empty set. Analogous to LG^k_n and FG^k_n, we use LL^k_n and FL^k_n to refer to the set of links and flows that were removed in iterations up to (and including) n (and the convention that LL^k_0 is the empty set):

$$\begin{aligned}
LG^k_n &= L^k_{n+1}, ..., L^k_{W^k+1} \\
LL^k_n &= L^k_0, ..., L^k_n \\
FG^k_n &= Q_{LG^k_n} \setminus Q_{LL^k_n} \\
FL^k_n &= Q_{LL^k_n}
\end{aligned}$$

Different WF$^{\text{k}}$ algorithms generate *different partitions* of the bottleneck links. For example, water-filling or WF$^\infty$ generates the partition $L^\infty_1, ..., L^\infty_{W^\infty}$. On the other hand, CPG or WF^1 generates the partition $L^1_1, ..., L^1_{W^1}$. In this section, we will introduce a new WF$^{\text{k}}$ algorithm called WF^2 and focus on its partition $L^2_1, .., L^2_{W^2}$. The WF^2 algorithm removes a link if it has the lowest WF rate of its first- and second-degree neighbors (hence $k = 2$ in the superscript.)

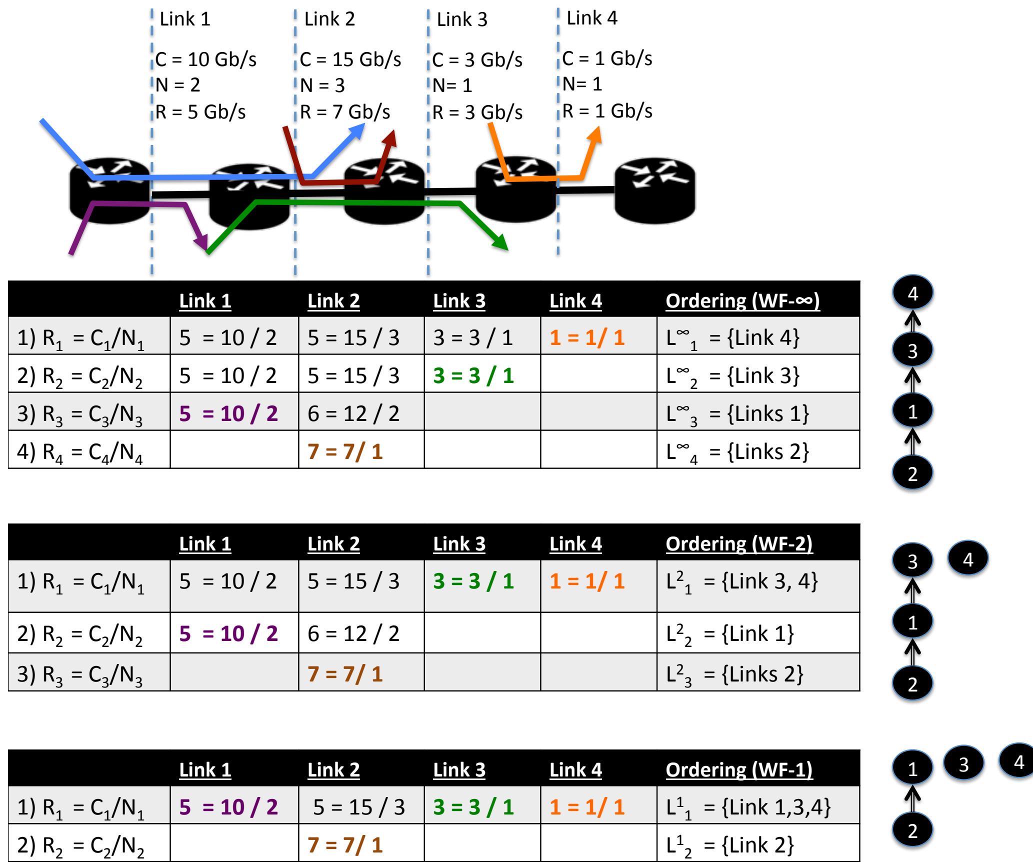
Because a WF$^{\text{k}}$ algorithm computes max-min fair rates, the WF rate computed for a link l in iteration $n+1$ is exactly the remaining capacity per flow that the link has after removing the flows carried by $L^k_1, ..., L^k_n$ and subtracting their max-min rates from the link capacity. Using $FL^k_n(l)$ to denote the subset of flows in FL^k_n that cross link l, we have:

$$R^k_{n+1}[l] = \frac{C - \sum_{f \in FL^k_n(l)} A(f)}{|FG^k_n|} \tag{3.8}$$

This is useful in analyzing the dynamic state of the s-PERC algorithm. Consider, for example, the proof of Lemma 3.2.2, where we used the partition from the water-filling algorithm and showed that once links in $L^\infty_1, .., L^\infty_n$ have converged by time T, the rate propagated by any neighbor of link $l \in L^\infty_{n+1}$ is at least $\mathrm{R}(l)$

by time $T + 1$ *round*. We used the fact that the "remaining capacity per flow" of any neighbor x of link l is at least $\mathrm{R}(l)$, after removing the max-min rates of flows carried by $L_1^\infty, ..., L_n^\infty$. This fact follows directly from the water-filling algorithm (WF^∞) because the criterion for removing link l in iteration $n+1$ of WF^∞ is exactly that it has the lowest rate of *all* links that remain in iteration $n+1$ (line 11).

3.4.3 Why Do We Need Different WF^k Algorithms?



	Link 1	Link 2	Link 3	Link 4	Ordering (WF-∞)
1) $R_1 = C_1/N_1$	5 = 10 / 2	5 = 15 / 3	3 = 3 / 1	**1 = 1/ 1**	L^∞_1 = {Link 4}
2) $R_2 = C_2/N_2$	5 = 10 / 2	5 = 15 / 3	**3 = 3 / 1**		L^∞_2 = {Link 3}
3) $R_3 = C_3/N_3$	**5 = 10 / 2**	6 = 12 / 2			L^∞_3 = {Links 1}
4) $R_4 = C_4/N_4$		**7 = 7/ 1**			L^∞_4 = {Links 2}

	Link 1	Link 2	Link 3	Link 4	Ordering (WF-2)
1) $R_1 = C_1/N_1$	5 = 10 / 2	5 = 15 / 3	**3 = 3 / 1**	**1 = 1/ 1**	L^2_1 = {Link 3, 4}
2) $R_2 = C_2/N_2$	**5 = 10 / 2**	6 = 12 / 2			L^2_2 = {Link 1}
3) $R_3 = C_3/N_3$		**7 = 7/ 1**			L^2_3 = {Links 2}

	Link 1	Link 2	Link 3	Link 4	Ordering (WF-1)
1) $R_1 = C_1/N_1$	**5 = 10 / 2**	5 = 15 / 3	**3 = 3 / 1**	**1 = 1/ 1**	L^1_1 = {Link 1,3,4}
2) $R_2 = C_2/N_2$		**7 = 7/ 1**			L^1_2 = {Link 2}

Figure 3.6: Three different WF^k algorithms, their respective partitions (last column of each table), and their dependency graphs.

As we saw, different WF^k algorithm have different criteria for removing a link l in some iteration $n+1$ (lines 12–13). The criteria naturally establish different relations between the remaining capacity per flow of the link and of its neighbors, after removing the max-min rates of flows carried by links in $L_1^k, .., L_n^k$. Figure 3.6 shows the WF rates computed by three different WF^k algorithms for a toy example. Consider link 1. When it is removed in WF^1 (third table, first row), its WF rate is lower than or equal to its first-degree

neighbor link 2 but higher than its second-degree neighbor link 3. It turns out that it is enough for neighbors of link 1 to have a remaining capacity per flow of at least 5 Gb/s in order for link 1's (bottleneck) flows to be correctly updated at link 1 in the s-PERC algorithm. As we saw in the proof of Lemma 3.2.2, this ensures that they propagate bottleneck rates that are at least $\mathrm{R}(l)$, which would allow link l to recognize its bottleneck flows.

However, in order for link 1's flow to be correctly updated at link 2, which has a higher max-min rate, we also need the property that link 1's WF rate when removed is lower than or equal to any second-degree neighbor's WF rate. We will explain the reason later (see Lemmas 3.4.15 and 3.4.17 in §3.4.6). This is a property we get from WF^2 and WF^∞ but not WF^1. In WF^1, link 1's WF rate when removed is higher than the rate of link 3, which is also active in the same iteration. Notice, that when link 1 is removed in WF^2 (second table, second row), link 3 has already been removed in a previous iteration and is no longer a second-degree neighbor. What this means for the s-PERC algorithm is that link 3 must converge before link 1's flows are correctly updated at link 2. We say that link 1 depends on link 3. We will define *precedence relationships* (or dependencies) for WF^2 (and s-PERC) in §3.4.10. In the figure, we have shown precedence relationships for the WF^k algorithms on the right. As is evident, the WF^1 algorithm, where a link need only have rates lower than or equal to its first-degree neighbors, has shorter dependency chains and needs the fewest iterations to converge.

3.4.4 Using WF^2 to Reason About s-PERC

In the next section, §3.4.5, we describe the properties of the WF rates in the WF^2 algorithm. In addition to the tautological property that a link has the smallest rate of its first- and second-degree neighbors when it is removed, we show that when an E flow of a link is removed in some iteration from the network, the link's WF rate for the same iteration strictly exceeds the max-min rate of the flow. We also show that the WF rate of a link l in iteration $n+1$ of the WF^2 algorithm is equal to the remaining capacity per flow of the link after removing the allocations of $\mathrm{E}(l)$ flows carried by links $L_1^2, ..., L_n^2$.

In §3.4.6 we state some invariants about updates at a link $l \in L_{n+1}^2$ and its neighbors in the s-PERC algorithm, following the convergence of links in $L_0^2, ..., L_n^2$. Note that L_0^2 is the empty set of links. The invariants rely on properties of the WF rates and we prove the invariants in Sections 3.4.7–3.4.9.

This enables us to prove, by induction on the sets $L_0^2, ..., L_{W^2}^2$, that s-PERC converges in a known bounded time:

Theorem 3.4.4. *Once the set of flows and links stabilize*[1] *at some time, the s-PERC algorithm converges to*

[1] That is, all the links have started their resets, and following this, the set of flows has been the same for at least one $round$ and is

the max-min fair rates for all flows within $6W^2$ rounds, where W^2 is the number of iterations that the WF^2 algorithm takes for the given set of flows and links.

Like the CPG algorithm (WF^1) provides a tighter convergence bound for the $Fair$ algorithm, the WF^2 algorithm provides a tighter convergence bound for the s-PERC algorithm. It shows that in order for a link l with the highest max-min rate to converge in the s-PERC algorithm, if l happens to be in L^2_{n+1} it is sufficient for links in $L^2_0, ..., L^2_n$ to converge, rather than all links with smaller max-min rates than l, that is $L^\infty_0, ..., L^\infty_{W^\infty-1}$. Moreover, it is necessary for at least one link in L^2_n to converge before l can converge (§3.4.10). This leads to a new definition of Precedent Link Relationship. The new definition is relevant for algorithms like s-PERC, where a link depends on properties of both first- and second-degree neighbors to converge.

3.4.5 Properties of the WF^2 Algorithm

Definition 3.4.5. *Consider the WF^2 algorithm. Let link $l \in L^2_n$, that is l is removed from L in iteration n. There exists at least one iteration from 1 to n when link l has the smallest rate among its first-degree neighbors (e.g., iteration n). We define the "freeze-iteration" m of link l as the first iteration when link l has the smallest rate among its first-degree neighbors:*

$$m = min(1 \leq i \leq n\colon R^2_i[l] = \min_{x \in X1_i[l]} R^2_i[x]) \tag{3.9}$$

Corollary 3.4.6. *Suppose link $l \in L^2_{n+1}$, then link has freeze-iteration m for some $1 \leq m \leq n$.*

Consider the freeze-iteration m. Since the rate of link l in this iteration is no more than the rate of its neighbors, and because the rates of the neighbors are non-decreasing (Lemma 3.4.1), we can show that link l's rate plateaus or is "frozen" from iteration m onward:

Lemma 3.4.7. [Plateau Rate Behavior] *Let link $l \in L^2_n$ have freeze-iteration m.*

1. *Flows carried by l that are removed in iterations i where $1 \leq i < m$ are removed and allocated $A[f] < R_i[l]$ and cause link l's rate to increase in the next iteration $R_i[l] < R_{i+1}[l]$.*

2. *Flows carried by l that are removed in iterations i where $m \leq i < n$ are removed at allocated $A[f] = R_i[l]$ and cause link l's rate to remain the same in the next iteration $R_i[l] = R_{i+1}[l]$.*

Corollary 3.4.8. *The rate computed for link l in iterations m through n is exactly its max-min rate $\mathrm{R}(l)$, while the rate computed before iteration m (when $m > 1$) is strictly smaller than the max-min rate $\mathrm{R}(l)$.*

---

guaranteed to stay the same in the future.

Hence, any flow $f \in \mathrm{E}(l)$ is removed in iteration i where $1 \leq i < m$ and allocated $\mathrm{A}(f) < R_i^2(l)$, while any $\mathrm{B}(l)$ flow is removed in iteration i where $m \leq i < n$ and allocated $\mathrm{A}(f) = R_i^2(l) = \mathrm{R}(l)$.

In other words, a link's rate "freezes" or "plateaus" at the max-min rate from the freeze-iteration until it is finally removed.

Proof of Lemma 3.4.7. Consider an iteration $i < m$. Let the shorthand $F = FG_{i-1}^k(l) \cap Q_{L_i^k}$ denote the set of flows carried by l that are removed in iteration i. Let f be one such flow removed because of some link $z \in L_i^k$. We know that $A[f] = R_i^2[z]$, and for any first-degree neighbor of l, let's call it x, $R_i^2[z] \leq R_i^2[x]$ because x is a second-degree neighbor of link z (lines 12–13 in Algorithm 7). But we also know by the definition of the freeze-iteration m (Definition 3.4.5), that for $i < m$, l has a first-degree neighbor x' with a smaller rate than l $R_i^2[x'] < R_i^2[l]$. Hence, $A[f] \leq R_i^2[x'] < R_i^2[l]$. Link l's allocation in the next iteration is:

$$\begin{aligned}
R_{i+1}^2[l] &= \frac{C_{i+1}^2[l]}{N_{i+1}^2[l]} \\
&= \frac{C_i^2[l] - \sum_{f \in F} A[f]}{N_i^2[l] - |F|} \\
&> \frac{C_i^2[l] - \sum_{f \in F} R_i^2[l]}{N_i^2[l] - |F|} && \text{(using } A[f] < R[l]\text{)} \\
&= \frac{N_i^2[l] R_i^2[l] - \sum_{f \in F} R_i^2[l]}{N_i^2[l] - |F|} && \text{(replacing } C_i^2[l] = N_i^2[l] R_i^2[l]\text{)} \\
&= R_i^2[l]
\end{aligned}$$

Consider iteration $i = m$. If no flows carried by link l are removed, then link l's rate is unchanged. Otherwise, again let the shorthand $F = FG_{i-1}^k(l) \cap Q_{L_i^k}$ be the set of flows carried by l that are removed in iteration i. Let f be one such flow removed because of some link $z \in L_i^k$. We know from lines 12–13 that $A[f] = R_i^2[z] \leq R_i^2[l]$ since l and z share a flow. We also know by definition of m (Definition 3.4.5) that any first-degree neighbor of l must satisfy $R_i^2[z] \geq R_i^2[l]$. Hence, $A[f] = R_i^2[z] = R_i^2[l]$. Link l's allocation in

the next iteration is

$$\begin{aligned}
R^2_{i+1}[l] &= \frac{C^2_{i+1}[l]}{N^2_{i+1}[l]} \\
&= \frac{C^2_i[l] - \sum_{f \in F} A[f]}{N^2_i[l] - |F|} \\
&= \frac{C^2_i[l] - \sum_{f \in F} R^2_i[l]}{N^2_i[l] - |F|} && \text{(using } A[f] = R[l]\text{)} \\
&= \frac{N^2_i[l] R^2_i[l] - \sum_{f \in F} R^2_i[l]}{N^2_i[l] - |F|} && \text{(replacing } R^2_i[l] = C^2_i[l]/N^2_i[l]\text{)} \\
&= R^2_i[l]
\end{aligned}$$

In iteration $i+1$, link l still has rate $R^2_i[l]$, while any link x that shares a flow with l has rate $R^2_{i+1}[x] \geq R^2_i[x] \geq R^2_i[l]$. The first inequality follows from Lemma 3.4.1, which says that the rate computed for a link is non-decreasing. So we can repeat the same argument for iterations i, where $m+1 \leq i < n$. □

We can think of links with freeze-iteration i as the set of links whose rates first froze in iteration i. For such a link, any flow removed before round i is in $\mathrm{E}(l)$, while any flow removed in round i or later is in $\mathrm{B}(l)$. Consider the example in Figure 3.6 (second table): link 1 has freeze-iteration 1, since it had a rate of 5 Gb/s, which equals the rate of its first-degree neighbor (link 1) in iteration 1. However, it was not removed in iteration 1, because link 3, which shares a flow with link 1's neighbor, had a rate of 3 Gb/s, which was still smaller.

While Equation 3.8 holds for WF^2, it will also be useful to look at the remaining capacity per flow after removing allocations of $\mathrm{E}(l)$ flows only.

Lemma 3.4.9. *Suppose link $l \in L^2_{n+1}$ has freeze-iteration $m+1$ for some $m \leq n < W^2$.*

When $1 \leq k \leq m$, the remaining capacity per flow of link l, after removing the max-min fair rates of $\mathrm{E}(l)$ flows carried by $L^2_1, ..., L^2_k$, is the WF rate of l in iteration $k+1$. When $k > m$, it is the WF rate in iteration $m+1$.

$$\begin{aligned}
\frac{C - \sum_{f \in \mathrm{E}(l) \cap FL^2_k(l)} A(f)}{|FG^2_k \cup \mathrm{B}(l)|} &= R^2_{k+1}[l] \quad \textit{when } k \leq m \\
&= R^2_{m+1}[l] = \mathrm{R}(l) \quad \textit{otherwise}
\end{aligned}$$

Proof. Corollary 3.4.8 says that any flow removed before the freeze-iteration must be in $\mathrm{E}(l)$ and the WF rate

is frozen at $\mathrm{R}(l)$ thereafter. We have the first result when $1 \leq k \leq m$ because any flow removed in iteration k must be in $\mathrm{E}(l)$. We have the second result when $m+1 \leq k$, because all E flows have been removed by iteration m, at which point the remaining capacity per flow, after removing the $\mathrm{E}(l)$ flows, is just the max-min rate $\mathrm{R}(l)$. □

We summarize the properties of the WF^2 algorithm that we shall use in our proof of convergence for s-PERC. Suppose link $l \in L^2_{n+1}$ has freeze-iteration m for some $m \leq n$. We will state some properties of the intermediates rates in the WF^2 algorithm and use Lemma 3.4.9 to infer properties of the remaining capacity per flow at link l and its neighbors with respect to some subset of links in the WF^2 order :

(1) For any first-degree neighbor in iteration $m+1$, $x \in \cup_{f \in FG^2_m(l)} P_f$, the rate of link x in iteration $m+1$ of the WF^2 algorithm is at least $\mathrm{R}(l)$. Hence, the remaining capacity per flow of link x, after removing the max-min fair rates of $\mathrm{E}(x)$ flows carried by $\cup_{i=1}^m L^2_i$, is at least $\mathrm{R}(l)$.

(2) The rate of link l in round $m+1$ is exactly $\mathrm{R}(l)$. Hence, the remaining capacity per flow of link l, after removing the max-min fair rates of $\mathrm{E}(l)$ flows carried by $\cup_{i=1}^m L^2_i$, is exactly $\mathrm{R}(l)$.

(3) For any second-degree neighbor in iteration $n+1$, $y \in \cup_{f \in FG^2_n(x)} P_f$, where the link $x \in \cup_{f \in FG^2_n(l)} P_f$, the rate of link y in round $n+1$ is at least $\mathrm{R}(l)$. Hence, the remaining capacity per flow of link y, after removing the max-min fair rates of $\mathrm{E}(y)$ flows carried by $\cup_{i=1}^n L^2_i$, is at least $\mathrm{R}(l)$.

(4) For any flow $f \in FG^2_n(l)$ and any link $x \in P_f$, if flow $f \in \mathrm{E}(x)$, then the rate of link x when f is removed in round $n+1$ is strictly greater than $\mathrm{A}(f) = \mathrm{R}(l)$ and less than its max-min rate. Hence, all flows of x carried by $\cup_{i=1}^n L^2_i$ are in $\mathrm{E}(x)$, and the remaining capacity per flow of link x, after removing the max-min fair rates of these flows, is strictly greater than $\mathrm{R}(l)$.

Property (1) is true because link l has the smallest rate out of all its first-degree neighbors in its freeze-iteration, by definition. Property (2) is true because the rate computed for link l starting from its freeze-iteration is exactly its max-min rate. Property (3) is true because link l has the smallest rate out of all its neighbors (both first- and second-degree) when it is removed. Property (4) is true because we know that flow $f \in \mathrm{E}(x)$ is removed in iteration n and we can use Corollary 3.4.8 to conclude that x freezes after iteration n; we can further use Lemma 3.4.7 to see that x's rate in iteration n must exceed flow f's allocation $\mathrm{A}[f]$, which equals $\mathrm{R}(l)$. Note that for all these properties, we use Lemma 3.4.9 to relate a link's remaining capacity per flow to the link's WF rate computed in an iteration of the WF^2 algorithm.

3.4.6 Invariants of s-PERC Based on the WF^2 Order

In this section, we will consider a link $l \in L^2_{n+1}$ and state invariants about l and its neighbors that hold after links $\cup_{i=0}^{m} L_i^2$ (or $\cup_{i=0}^{n} L_i^2$) have converged. We will also show how we can tie the invariants together to show that the flow allocations in the s-PERC algorithm converge to max-min fair rates within $6W^2$ $rounds$ of when the flows and links stabilize, where W^2 is the number of iterations that the WF^2 algorithm takes. In the three sections that follow this section (§3.4.7–§3.4.9) we will show how to prove the invariants using properties of the WF^2 ordering.

The first set of invariants are about link l, which should recognize its $FG_n^2(l)$ flows as $\hat{B}(l)$. The first step is that limit rates of the flows at link l must be high enough, that is at least $\mathrm{R}(l)$.

Lemma 3.4.10. *Let l have freeze-iteration $m+1$ and f be any flow in $FG_m^2(l)$. Suppose $L_1^2, ..., L_m^2$ have converged at time T. The bottleneck rate for f from a link $x \in P_f$ is either not propagated or at least $\mathrm{R}(l)$ from time $T+1$ $round$.*

Corollary 3.4.11. *The limit rate of f at link l is at least $\mathrm{R}(l)$ from time $T+2$ $rounds$.*

Note that Lemma 3.2.2 shows a similar result but is weaker because it assumes that for l's neighbors to propagate high enough rates, we need all links with smaller max-min rates than $\mathrm{R}(l)$ to have converged. We state the proof in §3.4.7. Next, we define what it means for a flow to be updated correctly at its bottleneck link.

Definition 3.4.12. *We say that a flow $f \in \mathrm{B}(l)$ is updated correctly at l from time T, if for all updates after T, the limit rate is at least $R(l)$, the bottleneck rate is exactly $R(l)$, and $MaxE$ is smaller than the bottleneck rate.*

Corollary 3.4.11 allows link l to recognize its own bottleneck flows:

Lemma 3.4.13. *Consider any link l with freeze-iteration $m+1$. If $L_1^2, ..., L_m^2$ have converged by time T, then any flow $f \in FG_m^2(l)$ is updated correctly at l from time $T+5$ $rounds$.*

We state the proof in §3.4.8. It uses Property (2), which says that the remaining capacity per flow at link l is exactly the max-min rate $\mathrm{R}(l)$, when l is frozen.

We outline why the worst case scenario needs five rounds: it takes two rounds from T for the limit rates of $\mathrm{B}(l)$ flows at l to be at least $\mathrm{R}(l)$ (Lemma 3.4.10), a third round for the value of $C - SumE/NumB$ at l to drop below (or equal) $\mathrm{R}(l)$, a fourth round for all $\mathrm{B}(l)$ to be permanently moved to $\hat{B}$ at l, and finally, as late as some time during the fifth round, $MaxE$ at l drops below $\mathrm{R}(l)$ so that the bottleneck rate is correctly propagated. Hence, from time $T+5$ $rounds$ all updates of $\mathrm{B}(l)$ flows at l are correct.

The next set of invariants are about link l's neighbors, with max-min rates greater than $\mathrm{R}(l)$, which should recognize link l's bottleneck flows as $\hat{\mathrm{E}(x)}$. We first define what it means for a flow to be updated correctly at a non-bottleneck link.

Definition 3.4.14. *We say that a flow $f \in \mathrm{E}(x)$ is updated correctly at x from time T, if for all updates after T, the limit rate is exactly $\mathrm{A}(f)$ and the bottleneck rate exceeds $\mathrm{A}(f)$.*

To show that flows in $FG_n^2(l)$ are updated correctly at non-bottleneck links x, we need an analog of Lemma 3.4.10 and its corollary, which says that the limit rates of flows at the neighbor link x are high enough.

Lemma 3.4.15. *Let l be any link in L_{n+1}^2, x be any link in $\cup_{f \in FG_n^2(l)} P_f$, and f be any flow in $FG_n^2(x)$. If $L_1^2, ..., L_n^2$ have converged by time T, the bottleneck rate for f from a link $y \in P_f$ is either not propagated or at least $\mathrm{R}(l)$ from time $T + 1\ round$.*

Corollary 3.4.16. *The limit rate of f at link x is at least $\mathrm{R}(l)$ from time $T + 2\ rounds$.*

We include the proof in the Appendix. It is identical to that of Lemma 3.4.10 and its corollary but uses Property (3), which says that neighbors of link l's neighbors have at least $\mathrm{R}(l)$, when l is removed[2], whereas Lemma 3.4.10 uses Property (1), which says that the remaining capacity per flow at neighbors of link l is at least $\mathrm{R}(l)$, when l is frozen. Property (3) is unique to the WF^2 algorithm and is not true in the context of the WF^1 algorithm.

Then we can show that the bottleneck rate at links x where $f \in \mathrm{E}(x)$ does exceed the flow's max-min rate $\mathrm{A}(f) = \mathrm{R}(l)$.

Lemma 3.4.17. *Consider any link $l \in L_{n+1}^2$. If $L_1^2, ..., L_n^2$ have converged by time T, then for any link x that is a non-bottleneck link for a bottleneck flow of l, that is, $|\mathrm{E}(x) \cap FG_n^2(l)| \geq 1$, at any time $t > T + 4\ rounds$, either $NumB(t) = 0$ or $(\mathrm{C} - SumE(t))/NumB(t) > \mathrm{R}(l)$. In the first case, there are no flows in $\hat{B}(x)$, while in the second case, the flows in $\hat{B}(x)$ have strictly more than $\mathrm{R}(l)$ each, after removing the allocations of the $\hat{E}$ flows.*

Corollary 3.4.18. *The bottleneck rate computed for $f \in FG_n^2(x)$, which includes any flow shared with link l and in $FG_n^2(l)$, is strictly greater than $\mathrm{R}(l)$ from time $T + 4\ rounds$.*

We outline why the worst case scenario needs four rounds: it takes two rounds from T for the limit rates of $FG_n^2(x)$ flows to be at least $\mathrm{R}(l)$ at x (Lemma 3.4.15), which is required to show that the value of

[2] i.e., after removing the max-min allocations of flows carried by links in $L_1^2, ..., L_n^2$, where n is link l's rank in the WF^2 order.

$C - SumE/NumB$ at x exceeds $\mathrm{R}(l)$ during some update; a third round is needed to ensure that the previous allocation any $FG_n^2(x)$ flow was at least $\mathrm{R}(l)$, which is required to show that once $C - SumE/NumB$ at x exceeds $\mathrm{R}(l)$, it remains higher. Hence, from time $T + 4\,rounds$ all updates of $FG_n^2(x)$ flows at x see bottleneck rates that exceed $\mathrm{R}(l)$.

We state the proof in §3.4.9. It relies on Property (4), which says that the remaining capacity per flow at first-degree neighbor x, which has a greater max-min rate than l, is strictly greater than $\mathrm{R}(l)$ when l is removed.

We can put the above results together to prove Theorem 3.4.4.

Proof of 3.4.4. We prove by induction on the WF^2 remove ordering $L_0^2, ..., L_{W^2+1}^2$.
Base case: The base case is trivial since L_0^2 is the empty set of links and has converged as soon as the set of flows and links stabilize.
Induction hypothesis: The induction hypothesis is that the $L_0^2, ..., L_n^2$ have converged by time T.
Induction step: Consider any link $l \in L_{n+1}^2$. Given the induction hypothesis, we will show that l will converge by time $T + 6\,rounds$. We need to show that any flow $f \in FG_n^2(l)$ is updated correctly at all links $x \in P_f$ from time $T + 6\,rounds$. Any flow f carried by $L_0^2, ..., L_n^2$ is already updated correctly at all its links, because of the induction hypothesis.

Consider any flow $f \in FG_n^2(l)$. We will consider each kind of link $x \in P_f$ that f might traverse and show that f is updated correctly at the link from time $T + 6\,rounds$.

1. Consider some link $x \in P_f$ where $f \in \mathrm{B}(x)$. By definition, since $f \in FG_n^2(l)$ and $l \in L_{n+1}^2$, both l and f are removed in iteration $n+1$ of the WF^2 algorithm. Since $f \in \mathrm{B}(x)$, we can use Lemma 3.4.7 to see that x is already frozen by round $n + 1$, that is, the freeze-iteration is $m + 1$ for some $m \leq n$. Since links in $L_0^2, ..., L_m^2$ have converged by time T, we can use Lemma 3.4.13 to prove that any flow in $FG_m^2(x)$ is updated correctly at x from time $T + 5\,rounds$, including $f \in FG_n^2(x) \subset FG_m^2(x)$.

2. Consider some link $x \in P_f$ where $f \in \mathrm{E}(x)$. Since $f \in \mathrm{E}(x)$, we can use Corollary 3.4.18 to see that the the bottleneck rate computed for f by x is strictly greater than $\mathrm{R}(l)$ from time $T + 4\,rounds$. We have already explained that f is updated correctly at its bottleneck links from $T + 5\,rounds$. Hence, the limit rate of f at link x from time $T + 6\,rounds$ is exactly $\mathrm{R}(l)$. So f is updated correctly at x from time $T + 6\,rounds$.

Hence, $f \in FG_n^2(l)$ is updated correctly at all $x \in P_f$ from time $T + 6\,rounds$. □

3.4.7 Why Limit Rates of Flows Are at Least $\mathrm{R}(l)$ at Link l and Neighbors

We present here why the limit rates of flows in $FG_m^2(l)$ are at least $\mathrm{R}(l)$ at any link l with freeze-iteration $m+1$, after links in $L_0^2, ..., L_m^2$ have converged.

Proof of Lemma 3.4.10. Let l be any link with freeze-iteration $m+1$ and f be any flow in $FG_m^2(l)$. Suppose $L_0^2, ..., L_m^2$ have converged at time T. The bottleneck rate for f from a link $x \in P_f$ is either not propagated or at least $\mathrm{R}(l)$ from time $T+1$ $round$.

Consider an update for a flow $f \in FG_m^2(l)$ at a link $x \in P_f$ at some time after $t+1$ $round$. Suppose $b \geq MaxE$ and the rate is propagated (line 15 of Algorithm 6). If $MaxE \geq \mathrm{R}(l)$, the proof is done. So let us consider the case when $MaxE < \mathrm{R}(l)$.

Let $SumE$, $NumB$ and $MaxE$ represent the link state of link x at line 8 of Algorithm 6 just before it computes a bottleneck rate for f. Note that because of lines 5–7, we can take for granted that there is at least one flow in $\hat{B}$, namely f. We will break down the aggregate link state in terms of the contribution of $\mathrm{E}(x)$ flows carried by $L_0^2, ..., L_m^2$, that is, $\mathrm{E}(x) \cap FL_m^2(x)$, and the remaining flows. The remaining flows, which we represent as $\mathrm{B}(x) \cup FG_m^2(x)$, include both $\mathrm{B}(x)$ flows carried by $L_0^2, ..., L_m^2$ as well as flows carried by the remaining links. The bottleneck rate computed for the flow f satisfies the following (omitting x for simplicity):

$$\begin{aligned} b &= \frac{C - SumE}{NumB} \\ &= \frac{C - \sum_{f \in \hat{\mathrm{E}}} a_f}{|\hat{\mathrm{B}}|} \\ &= \frac{C - \sum_{f \in \mathrm{E} \cap FL_m^2} \mathrm{A}(f) - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B} \cup FG_m^2)} a_f}{|\hat{\mathrm{B}}|} \end{aligned}$$

In the last step, we used the induction hypothesis that $L_0^2, ..., L_m^2$ have converged by time T, which implies that their flows are updated correctly at x from time T. In particular, the subset of flows in $\mathrm{E}(x)$ are all correctly classified as belonging to $\hat{E}(x)$ and allocated their max-min rate from time $T+1$ $round$ (hence the second term of the numerator). The remaining flows may also be classified as belonging to $\hat{E}(x)$ (hence the third term of the numerator). Since flows in $\mathrm{E}(x) \cap FL_m^2(x)$ are classified correctly, flows classified as

$\hat{B}(x)$ must be a subset of $\mathrm{B}(x) \cup FG^2_m(x)$. Rearranging the terms, we get:

$$
\begin{aligned}
b \cdot |\hat{\mathrm{B}}| &= C - \sum_{f \in \mathrm{E} \cap FL^2_m} \mathrm{A}(f) - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B} \cup FG^2_m)} a_f \\
&\geq \mathrm{R}(l) \cdot |\mathrm{B} \cup FG^2_m| - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B} \cup FG^2_m)} a_f && \text{(using Property (1))} \\
&\geq \mathrm{R}(l) \cdot |\mathrm{B} \cup FG^2_m| - MaxE \cdot |\hat{\mathrm{E}} \cap (\mathrm{B} \cup FG^2_m)| && \text{(using Lemma 3.2.4)} \\
&\geq \mathrm{R}(l) \cdot |\mathrm{B} \cup FG^2_m| - \mathrm{R}(l) \cdot |\hat{\mathrm{E}} \cap (\mathrm{B} \cup FG^2_m)| && \text{(we are considering the case } MaxE < \mathrm{R}(l)) \\
&\geq \mathrm{R}(l) \cdot |\hat{\mathrm{B}} \cap (\mathrm{B} \cup FG^2_m)| \\
&= \mathrm{R}(l) \cdot |\hat{\mathrm{B}}| && \text{(since } \hat{\mathrm{B}}(x) \subset \mathrm{B}(x) \cup FG^2_m(x)) \\
b &\geq \mathrm{R}(l) && \text{(since } |\hat{\mathrm{B}}| > 0)
\end{aligned}
$$

□

Proof of Corollary 3.4.11. The limit rate of f at link l is at least $\mathrm{R}(l)$ from time $T + 2\, rounds$.

Flow f is seen at every link in P_f at least once between times $T + 1\, round$ and $T + 2\, rounds$ and picks up a bottleneck rate that is either not propagated or $\mathrm{R}(l)$. Flow f's limit rate at link l is the smallest propagated bottleneck rate from any link in $P_f \setminus l$ and hence is at least $\mathrm{R}(l)$ from time $T + 2\, rounds$. □

3.4.8 Why Flows Are Updated Correctly at Bottleneck Links

Consider a link $l \in L^2_{n+1}$, which has freeze-iteration $m + 1$ for some $m \leq n$. We present here the proof for why l's bottleneck flows $f \in FG^2_m(l)$ are updated correctly at link l. The only precondition is that links in $L^2_0, ..., L^2_{\mathbf{m}}$ have converged.

Proof of Lemma 3.4.13. Consider any link l with freeze-iteration $m + 1$. If $L^2_0, ..., L^2_m$ have converged by time T, then any flow $f \in FG^2_m(l)$ is updated correctly at l from time $T + 5\, rounds$.

During any update after $T + 3\, rounds$, the bottleneck rate for flow f at link l is at most $\mathrm{R}(l)$. We can explain this by looking at how the $\mathrm{E}(l)$ flows carried by $L^2_0, ..., L^2_m$ contribute to the aggregate state. We shall use Corollary 3.4.8, which says that for link l with freeze-iteration $m + 1$, the set of $\mathrm{E}(l)$ flows carried by $L^2_0, ..., L^2_m$ is exactly $\mathrm{E}(l)$, while the set of remaining flows is exactly $\mathrm{B}(l)$.

Let $SumE$, $NumB$ and $\hat{E}$ represent the link state at line 8 of Algorithm 6, as they are used to compute

the bottleneck rate. For all updates at time t after $T + 3\ rounds$ (omitting l for simplicity):

$$
\begin{aligned}
b &= \frac{\mathrm{C} - SumE}{NumB} \\
&= \frac{\mathrm{C} - \sum_{f \in \mathrm{E}} A(f) - \sum_{f \in \mathrm{B} \cap \hat{E}} a_f}{|\mathrm{B} \cap \hat{B}|} && \text{(induction hypothesis)} \\
&= \frac{\mathrm{R}(l) \cdot \mathrm{B} - \sum_{f \in \mathrm{B} \cap \hat{E}} a_f}{|\mathrm{B} \cap \hat{B}|} && \text{(using Property (2))} \\
&\leq \frac{\mathrm{R}(l) \cdot |\mathrm{B}| - \mathrm{R}(l) \cdot |\mathrm{B} \cap \hat{E}|}{|\mathrm{B} \cap \hat{B}|} && \text{(using Corollary 3.4.11)} \\
&= \mathrm{R}(l)
\end{aligned}
$$

In the second-to-last step we used Corollary 3.4.11, which says that limit rates of $FG^2_m(l)$ flows (equal to $\mathrm{B}(l)$) at l are at least $\mathrm{R}(l)$ from $T + 2\ rounds$. Since every flow is seen once between $T + 3\ rounds$ and $T + 3\ rounds$, we conclude that the allocation of the $FG^2_m(l)$ flows are at least $\mathrm{R}(l)$ after T + 3 rounds.

Hence, every flow in $\mathrm{B}(l)$, including f, is moved to $\hat{B}(l)$ once and for all, and the $\hat{B}$ and $\hat{E}$ sets stabilize to the correct values latest by $T + 4\ rounds$. During any update after $T + 4\ rounds$, the bottleneck rate for any flow $f \in \mathrm{B}(l)$ at link l is exactly $\mathrm{R}(l)$. But the bottleneck rate may not be propagated since the set of $\hat{E}$ allocations could have stabilized to E as late as $T + 4\ rounds$ and the last inconsistent allocation could have been made as late as $T + 3\ rounds$. $MaxE$ can take up to two additional rounds to reflect the maximum $\hat{E}$ allocation and drop below $\mathrm{R}(l)$ (Lemma 3.2.5). Hence, the bottleneck rate must be propagated correctly after $T + 5\ rounds$. So flow $f \in FG^2_m(l)$ is "correctly updated" at link l from time $T + 5\ rounds$. □

3.4.9 Why Flows are Updated Correctly at Non-Bottleneck Links

Consider a link $l \in L^2_{n+1}$ with freeze-iteration $m + 1$ for some $m \leq n$. We present here why the bottleneck rate computed for l's $FG^2_n(l)$ flows at their non-bottleneck links exceeds $\mathrm{R}(l)$, their max-min rate. The precondition here is that links in $L^2_0, ..., L^2_{\mathbf{n}}$ have converged.

Proof of Lemma 3.4.17. Consider any link $l \in L^2_{n+1}$. *If* $L^2_0, ..., L^2_n$ *have converged by time* T, *then for any link* x *that is a non-bottleneck link for a bottleneck flow of l, that is,* $|\mathrm{E}(x) \cap FG^2_n(l)| \geq 1$, *at any* $t > T + 4\,rounds$, *either* $NumB(t) = 0$ *or* $(\mathrm{C} - SumE(t))/NumB(t) > \mathrm{R}(l)$. *In the first case, there are no flows in* $\hat{B}(x)$, *while in the second case, the flows in* $\hat{B}(x)$ *have strictly more than* $\mathrm{R}(l)$ *each, after removing the allocations of the* $\hat{E}$ *flows.*

First, we show that the condition must hold following some update between $T + 3\ rounds$ and $T + 4\ rounds$. Consider all the updates during this interval. We shall use Property (4), which says that all flows

of x carried by $L_1^2, ..., L_n^2$ are in $\mathrm{E}(x)$ so we can replace $\mathrm{E}(x) \cap FL_n^2(x) = FL_n^2(x)$ and express the set of remaining flows as $FG_n^2(x) \cup \mathrm{B}(x) = FG_n^2(x)$. It also says that the rate of x when link l's flow $f \in \mathrm{E}(x)$ is removed, exceeds $\mathrm{R}(l)$.

If any of the flows in $FG_n^2(x)$ is classified as $\hat{E}$, the condition holds after the update because of Corollary 3.4.11, which says that the flow's limit rate is at least $\mathrm{R}(l)$, and Lemma 3.2.1, which says that following an update when a flow is moved to $\hat{E}$, either $NumB = 0$ or $\mathrm{C} - SumE/NumB$ exceeds the flow's limit rate. If, on the other hand, all of the flows in $FG_n^2(x)$ are classified into $\hat{B}$, then after the last of these updates we have (omitting x for simplicity):

$$
\begin{aligned}
\frac{\mathrm{C} - SumE(t)}{NumB(t)} &= \frac{\mathrm{C} - \sum_{f \in FL_n^2} \mathrm{A}(f) - \sum_{f \in \hat{E} \cap FG_n^2} a_f}{|\hat{B} \cap FG_n^2|} && \text{(induction hypothesis)} \\
&= \frac{\mathrm{C} - \sum_{f \in FL_n^2} \mathrm{A}(f)}{|FG_n^2|} && \text{(considering case where } FG_n^2(x) \subset \hat{\mathrm{B}}\text{)} \\
&> \mathrm{R}(l) && \text{(using Property (4))}
\end{aligned}
$$

Next, we show that once this condition becomes true at some time between $T + 3\,rounds$ and $T + 4\,rounds$, it remains true thereafter. If a flow in $FG_n^2(x)$ is marked $\hat{E}$, then we have already explained why the condition holds after the update. Otherwise, if a flow in $FG_n^2(x)$ moves from $\hat{E}$ to $\hat{B}$, the intuition is that it makes its old allocation, which was at least $\mathrm{R}(l)$, available to the $\hat{B}$ flows. Since the $\hat{B}$ flows already had more than $\mathrm{R}(l)$ each, the new value of $\frac{C - SumE}{NumB}$ remains higher. We formalize this intuition below:

Let $SumE(t), NumB(t)$ represent the link state at line 5 in the Algorithm, *before* they are adjusted to compute b for some flow $f \in FG_n^2(x)$. We know that $C - SumE(t) > NumB(t)\,\mathrm{R}(l)$, since we assume that the condition holds before the update. Now we show why the condition would hold after the update as well. After the flow moves to $\hat{B}$ there is at least one flow marked in $\hat{B}$, and the value of $(\mathrm{C} - SumE(t))/NumB(t)$ is:

$$
\begin{aligned}
\frac{C - (SumE(t) - e)}{(NumB(t) + 1)} &= \frac{(C - SumE(t)) + e}{(NumB(t) + 1)} \\
&> \frac{NumB(t) \cdot \mathrm{R}(l) + e}{(NumB(t) + 1)} && \text{(condition holds before update)} \\
&\geq \frac{NumB(t) \cdot \mathrm{R}(l) + \mathrm{R}(l)}{(NumB(t) + 1)} && \text{(using Corollary 3.4.16)} \\
&= \mathrm{R}(l)
\end{aligned}
$$

□

Proof of Corollary 3.4.18. The bottleneck rate computed for $f \in FG_n^2(x)$*, which includes any flow shared with link* l *and in* $FG_n^2(l)$*, is strictly greater than* $\mathrm{R}(l)$ *from time* $T + 4\ rounds$*.*

If the flow was previously in $\hat{B}$, the bottleneck rate computed is exactly $b = \frac{C - SumE}{NumB} > \mathrm{R}(l)$. If the flow was previously in $\hat{E}$, the bottleneck rate computed is $b = \frac{C-(SumE(t)-a_f^l(t^-))}{NumB+1} > \frac{\mathrm{R}(l)NumB+a_f^l(t^-)}{NumB+1} \geq \mathrm{R}(l)$. The first inequality follows from Lemma 3.4.17 above. The second inequality follows from Corollary 3.4.16, which says that that for any flow $f \in FG_n^2(x)$, the limit rate at x is at least $\mathrm{R}(l)$ from $T + 2\ rounds$, which implies that the allocation at x is at least $\mathrm{R}(l)$ from $T + 3\ rounds$, in particular $a_f^l(t^-) \geq \mathrm{R}(l)$. □

3.4.10 Dependencies from the WF^2 Algorithm

It follows from the WF^2 algorithm that in order for a link $l \in L_{n+1}^2$ to "converge" (in a distributed algorithm like s-PERC), it is sufficient for links $L_0^2, ..., L_n^2$, rather than all links with lower max-min rates than l. The following lemma shows that it is in fact necessary for at least one link in L_n^2 to converge. This is analogous to the *precedent link relationship* for the CPG algorithm defined in [75].

Theorem 3.4.19. (WF^2 Precedent Link Relationship) *Let* l *be a link removed in iteration* $n+1$ *of the* WF^2 *algorithm, that is,* $l \in L_{n+1}^2$*. Then one of the following statements must be true about iteration* n*:*

1. *There exists a link* x *that shared a flow with link* l *in iteration* n *and had a lower WF rate than link* l*, and*
   - *(a) either* x *was removed in iteration* n
   - *(b) or* x *was not removed but a link* z *that shared a flow with link* x *in iteration* n *and had a lower WF rate than link* x *was removed in iteration* n*.*
2. *All links that share a flow with link* l *in iteration* n *have rates that are at least link* l*'s WF rate in iteration* n*. However, there exists a link* y *that shared a flow with a neighbor* x *of link* l *in iteration* n*, where* y *had a lower WF rate than link* l*, and*
   - *(a) either* y *was removed in iteration* n
   - *(b) or* y *was not removed but a link* z *that shared a flow with link* y *in iteration* n *and had a lower WF rate than link* y *was removed in iteration* n*.*

Proof. Since link l was removed in iteration $n+1$ of the WF^2 algorithm, it had the lowest rate among its neighbors in the iteration (lines 12–13 of Algorithm 7), that is:

$$R^2_{n+1}[l] \le R^2_{n+1}[x], \quad \text{for all } x \in X1_{n+1}[l]$$
$$R^2_{n+1}[l] \le R^2_{n+1}[y], \quad \text{for all } y \in X2_{n+1}[l]$$

This means that in iteration n, link l did not have the lowest rate among its neighbors; otherwise it would have been removed in iteration n. Hence, either a first-degree neighbor $x \in X1_n(l)$ or a second-degree neighbor $y \in X2_n(l)$ had a lower rate than $R^2_n(l)$, the rate of link l in iteration n, that is:

$$R^2_n[l] < R^2_n[x], \quad \text{for some } x \in X1_n[l] \quad \text{or}$$
$$R^2_n[l] < R^2_n[y], \quad \text{for some } y \in X2_n[l]$$

Suppose $x \in X1_n(l)$, which had $R^2_n(x) < R^2_n(l)$, was not removed in iteration n—that is, suppose 1(a) does not hold. We will show then that 1(b) must hold. Since $R^2_{n+1}(x) \ge R^2_{n+1}(l)$ and the rates are non-decreasing, we know that the rate of x must have increased from iteration n to $n+1$. From Lemma 3.4.7, we know that for the rate of link x to increase, a flow carried by x must have been removed in iteration n with allocation $A[f] < R^2_n(x)$. The flow must have been removed because it was carried by a neighbor z of link x that was removed with rate $R^2_n[z] = A[f] < R^2_n(x)$. Hence, we see that if there exists $x \in X1_n(l)$, which had $R^2_n(x) < R^2_n(l)$, either x was removed in iteration n or a neighbor z of x with a lower rate than link l was removed $R^2_n[z] < R^2_n[x] < R^2_n[l]$.

If there is no first-degree neighbor with a lower rate than l, then we can use a similar argument to show that there exists a second-degree neighbor $y \in X2_n(l)$, which had $R^2_n(y) < R^2_n(l)$, either y was removed in iteration n (i.e., 2(a) holds) or a neighbor z of y with a lower rate than link l was removed $R^2_n[z] < R^2_n[y] < R^2_n[l]$ (i.e., 2(b) holds).

□

The definitions of *precedent links* follow directly:

Definition 3.4.20. *For a link $l \in L^2_{n+1}$,*

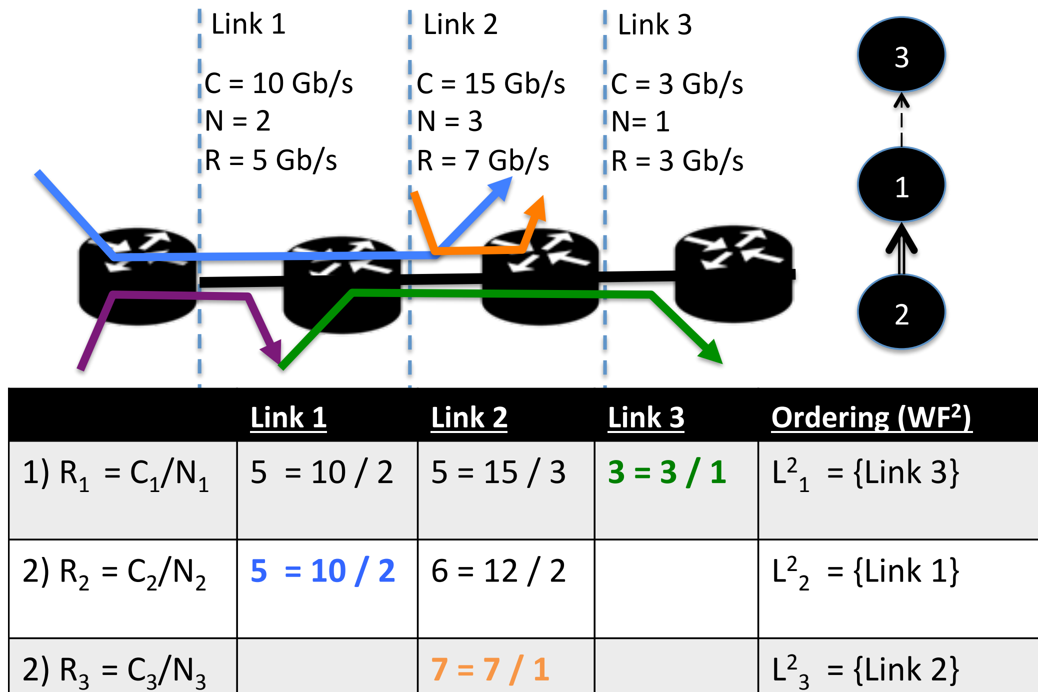
1. *we say that a link x is a* direct precedent *of l if x, l share a flow in iteration n, such that $R^2_n(x) < R^2_n(l)$ and x is removed in iteration n;*

2. *a link z is an* indirect precedent *of l via* medium link x *if z, x and x, l share a flow in iteration n, such that $R^2_n(z) < R^2_n(x) < R^2_n(l)$ and x is not removed but z is;*

3. *a link y is a* second-degree direct precedent *of l if for all x where y, x and x, l share a flow in iteration n, $R_n^2(x) \geq R_n^2(l)$, and $R_n^2(y) < R_n^2(l)$ and y is removed in iteration n; and*

4. *a link z is a* second-degree indirect precedent *of l via* medium link *y if for all x where z, y, y, x and x, l share a flow in iteration n, $R_n^2(x) \geq R_n^2(l)$, and $R_n^2(z) < R_n^2(y) < R_n^2(l)$ and y is not removed but z is.*

Figure 3.7 shows an example of direct precedent links based on the WF^2 algorithm. There are three links and four flows. The green flow is bottlenecked to 3 Gb/s at link 3, the purple and blue flows are bottlenecked to 5 Gb/s at link 1, and the orange flow is bottlenecked to 7 Gb/s at link 2. In the WF^2 algorithm, link 3 is removed in the first iteration because it has the smallest WF rate of its first- and second-degree neighbors. Links 1 and 2 are removed in the second and third iterations, respectively.



	Link 1	Link 2	Link 3	Ordering (WF^2)
1) $R_1 = C_1/N_1$	5 = 10 / 2	5 = 15 / 3	3 = 3 / 1	L^2_1 = {Link 3}
2) $R_2 = C_2/N_2$	5 = 10 / 2	6 = 12 / 2		L^2_2 = {Link 1}
2) $R_3 = C_3/N_3$		7 = 7 / 1		L^2_3 = {Link 2}

Figure 3.7: Examples of *direct* precedent links based on the WF^2 Algorithm. Link 1 is a *direct precedent* of link 2, while link 3 is a *second-degree direct precedent* of link 1. The WF^2 precedence graph is in the top-right corner (double-solid: direct, dashed: second-direct).

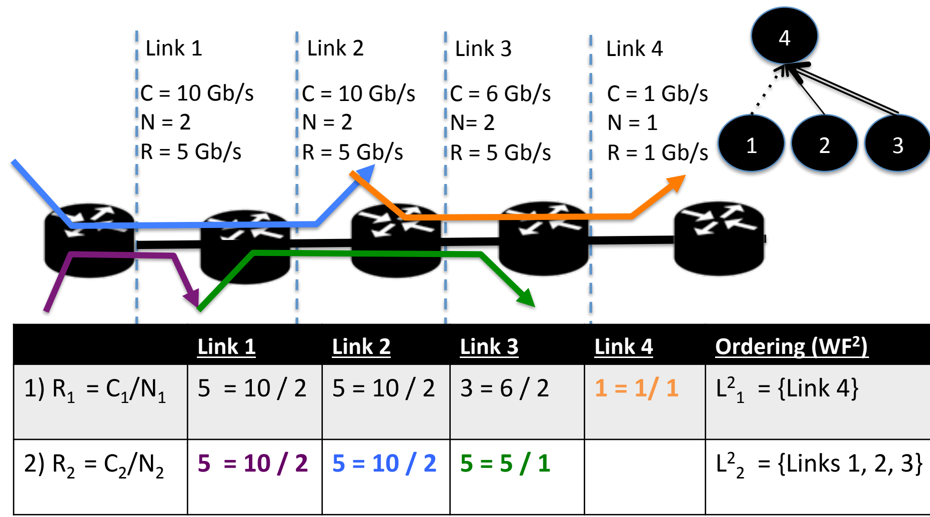
We say that link 3 is a second-degree direct-precedent of link 1, because in the first iteration, link 1's only first-degree neighbor, link 2, has a rate equal to that of link 1 (5 Gb/s), whereas link 3, a second-degree neighbor, has a lower rate 3 Gb/s and is removed by the end of the iteration. In s-PERC, it is only after

link 3 converges and the green flow is correctly classified as $\hat{E}$ and allocated 3 Gb/s at link 2, that link 2's bottleneck rate for link 1's (blue) bottleneck flow can be shown to exceed link 1's max-min rate of 5 Gb/s (to at least 6 Gb/s). This is necessary so that link 1's (blue) bottleneck flow can be correctly classified as $\hat{E}$ at (its non-bottleneck) link 2. Hence, link 1 depends on link 3 to converge.

We say that link 1 is a direct-precedent of link 2, because in the second iteration, it shares a (blue) flow with link 2, has a smaller WF rate than link 2, and is removed by the end of the iteration. In s-PERC, it is only after link 1 converges and its (blue) bottleneck flow is correctly classified as $\hat{E}$ and allocated 5 Gb/s at link 2, that link 2 can update its bottleneck rate for its own (orange) bottleneck flow from 6 Gb/s to exactly 7 Gb/s. Hence, link 2 depends on link 1 to converge.

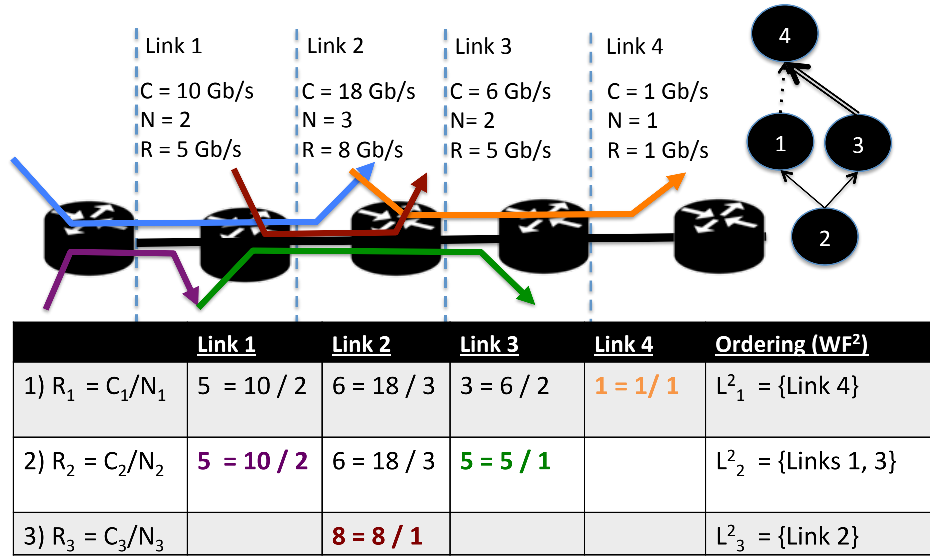
Figure 3.8a shows an example of indirect precedent links based on the WF^2 algorithm. There are four links and four flows. The orange flow is bottlenecked to 1 Gb/s at link 4, the green flow to 5 Gb/s at links 2 and 3, the blue flow to 5 Gb/s at links 2 and 1, and the purple flow to 5 Gb/s at link 1. In the WF^2 algorithm, link 4 is removed in the first iteration because it has the smallest WF rate of its first- and second-degree neighbors. Note that link 1 is not removed in the first iteration because a second-degree neighbor—that is, link 3 has a lower WF rate (3 Gb/s) in iteration 1 than link 1, which has 5 Gb/s. Rather, links 1–3 are all removed in the second iteration, when they each have a WF rate of 5 Gb/s.

We say that link 4 is an indirect precedent of link 2 via medium link 3, because in the first iteration, link 4 shares a (orange) flow with link 3, which shares a (green) flow with link 2, and link 4 has a smaller rate than link 3, which has a smaller rate than link 2, and lastly link 3 is not removed but link 4 is. In s-PERC, it is only after link 4 converges and its (orange) flow is correctly classified as $\hat{E}$ and allocated 1 Gb/s at link 3, that the bottleneck rate propagated from link 3 to link 2 (via the green flow) can be guaranteed to be at least link 2's max-min rate 5 Gb/s (which in this case happens to be same as link 3). This allows link 2 to recognize its (green) bottleneck flow (which in this case also happens to be a bottleneck flow of link 3). Hence, link 2 depends on link 4 to converge.



	Link 1	Link 2	Link 3	Link 4	Ordering (WF²)
1) $R_1 = C_1/N_1$	5 = 10 / 2	5 = 10 / 2	3 = 6 / 2	1 = 1/ 1	L^2_1 = {Link 4}
2) $R_2 = C_2/N_2$	5 = 10 / 2	5 = 10 / 2	5 = 5 / 1		L^2_2 = {Links 1, 2, 3}

(a) Link 4 is an *indirect precedent* of link 2 (via medium link 3) and a *second-degree indirect precedent* of link 1 (via medium link 3 again). Note that link 4 is a *direct precedent* of link 3.



	Link 1	Link 2	Link 3	Link 4	Ordering (WF²)
1) $R_1 = C_1/N_1$	5 = 10 / 2	6 = 18 / 3	3 = 6 / 2	1 = 1/ 1	L^2_1 = {Link 4}
2) $R_2 = C_2/N_2$	5 = 10 / 2	6 = 18 / 3	5 = 5 / 1		L^2_2 = {Links 1, 3}
3) $R_3 = C_3/N_3$		8 = 8 / 1			L^2_3 = {Link 2}

(b) Link 4 is a *second-degree indirect precedent* of link 1 via medium link 3. Note that link 4 is also *direct precedent* of link 3, while links 1 and 3 are *direct precedents* of link 2.

Figure 3.8: Examples of *indirect* precedent links based on the WF^2 Algorithm. The WF^2 precedence graph is in the top-right corner of each example (double-solid: direct, dashed: second-direct, single-solid: indirect, dotted: second-indirect).

Figure 3.8b shows another example of an indirect precedent link based on the WF^2 algorithm. We describe the setup first. There are four links and five flows. The orange flow is bottlenecked to 1 Gb/s at link

4, the green flow to 5 Gb/s at link 3, the blue and purple flows to 5 Gb/s at link 1, and the brown flow to 8 Gb/s at link 2. In the WF^2 algorithm, link 4 is removed in the first iteration because it has the smallest WF rate of its first- and second-degree neighbors. Note that link 1 is not removed in the first iteration because a second-degree neighbor, that is, link 3 has a lower WF rate (3 Gb/s) in iteration 1 than link 1, which has 5 Gb/s. Rather, links 1 and 3 are removed in the second iteration, when they each have a WF rate of 5 Gb/s, while link 2 is removed in the third iteration when it has a WF rate of 8 Gb/s.

We say that link 4 is a second-degree indirect-precedent of link 1 via medium link 3, because in the first iteration, link 1's only first-degree neighbor, link 2, had a rate greater than (or equal) to that of link 1, whereas link 3, a second-degree neighbor had a lower rate 3 Gb/s, and link 3 was not removed but link 3's first-degree neighbor link 4 was removed. In s-PERC, it is only after link 4 converges, and its (orange) flow is correctly classified as $\hat{E}$ and allocated 1 Gb/s at link 3, that the bottleneck rate propagated from link 3 to link 2 (via the green flow) can be guaranteed to be at least link 1's max-min rate 5 Gb/s. This allows one to guarantee that when link 2 computes a bottleneck rate for link 1's bottleneck flows, the rate exceeds link 1's max-min rate. This is necessary so that link 1's (blue) bottleneck flow can be correctly classified as $\hat{E}$ at (its non-bottleneck) link 2. Hence, link 1 depends on link 4 to converge.

The precedent link relationship implies that for a link $j \in L^2_{n+1}$, there is at least one precedent link in L^2_n. Hence, the precedent link relationships form a directed acyclic graph of depth W^2. Following the example of [75], we call this the WF^2 Precedence Graph.

The *precedent link relationship* is an example of a dependency where we define a dependent link of link l as any link that must converge before link l can converge. For example, notice that in Figure 3.7, link 3 is a direct precedent of link 1, and link 1 depends on link 3 to converge, because it is only after link 3 is removed in iteration 1 that link 1 has the lowest rates of its first- and second-degree neighbors. The WF^2 algorithm implies that any dependent link of link $j \in L^2_{n+1}$ must be among links in $L^2_1, .., L^2_n$ so that the dependencies, which are a superset of the precedent link relationships, also form a directed acyclic graph of depth W^2.

Chapter 4

Evaluating s-PERC for Data Centers

In this chapter, we will describe how s-PERC can be practically deployed. We focus on the data-center setting, though many design considerations are relevant to other settings like enterprise networks or WANs. We will use s-PERC* to distinguish the implementation of the s-PERC algorithm from the basic algorithm itself. We will describe how we deployed s-PERC in a 4x10 Gb/s NetFPGA test bed, which validates that it is easy to implement in hardware. We will use simulations to evaluate benefits that s-PERC* can provide for data-center workloads, especially relative to reactive algorithms that converge fast (represented by RCP) and scheduling algorithms that prioritize short flows (represented by p-Fabric). We will also describe real measurements of s-PERC, TCP, and DCTCP from our 4 x 10 Gb/s NetFPGA test bed, where we compare s-PERC's convergence times relative to TCP and DCTCP for small incasts and traffic matrices with multiple bottlenecks.

4.1 Practical Design Considerations

We made several design and implementation choices to make s-PERC* simple, practical, and robust for data centers.

Dynamic flow arrivals and departures: In real networks, flows come and go. But so far, the s-PERC algorithm has been described as if the number of flows is constant. Let us see how s-PERC* works when flows are arriving and departing. A new flow's control packet carries $s[l] = E$ and $a[l] = 0$ (bottlenecked elsewhere and allocated 0 Gb/s) for every link l so that the link will increase $NumB$ during the update (see Table 3.2). The last control packet of a flow that is terminating will be tagged with a FIN so that the links that carry the flow can remove its contribution to the aggregate state at the link ($NumB$ or $SumE$).

Limiting control overhead: We want control packets to zip through the network in order to calculate rates quickly. However, we do not want the control packets to take too much bandwidth away from the data packets. Hence, control packets are prioritized at every link, but rate-limited to some fraction (2%-5%) of the link capacity. For a given fraction of link capacity, say $C_{ctrl} = 2$ Gb/s, and an observed number of flows at a link, say $N = 100$, each control packet is limited to the rate $C_{ctrl}/N = 20$ Mb/s. This means, for example, that a host can send a 64 byte control packet no more frequently than one every $Tx_{ctrl} = 64$ bytes$/(C_{ctrl}/N) = 25.6\mu$s. Notice that Tx_{ctrl} can exceed the unloaded RTT in a network. To implement this simple rate-limiting scheme, s-PERC* maintains an additional variable N at the switch for each link (similar to $NumB$) to track the number of flows traversing the link in either direction, and calculate Tx_{ctrl}. The s-PERC* control packet carries the smallest value of Tx_{ctrl} seen at any link in the flow's path. We do not expect the control packets to see queues since they are strictly prioritized over data packets and limited to a small fraction of the link capacity. Also, we do not expect the control packets to starve data-packets because they are rate-limited.[1]

Adjusting $round$ automatically: We want $MaxE$ and $MaxE'$ to be reset only after every flow has been seen at the link at least once. This ensures that when $MaxE$ is reset to $MaxE'$, it is at least the allocation of any flow in $\hat{E}$ at the time. Otherwise, the link may wrongly propagate a bottleneck rate that is inconsistent with the $\hat{E}$ allocations and delay convergence. As mentioned above, a control packet's period Tx_{ctrl} can exceed the unloaded RTT in a network, and moreover it can change as the number of flows changes. Hence, in s-PERC*, $MaxE$ and $MaxE'$ are reset once every $round$, rather than every RTT, where $round$ is adjusted dynamically to be at least $\max(Tx_{ctrl}, RTT)$, which is the time it takes to see a control packet from every flow at least once. At each link, the variable $round$ starts with an initial value of 2 RTTs. Thereafter, it is multiplicatively adjusted based on the current number of flows. Any time when $\max(Tx_{ctrl}, RTT)$ exceeds the current value, $round$ is doubled. When $\max(Tx_{ctrl}, RTT)$ has been smaller than the current value for at least $MaxRound$ seconds, $round$ is halved (but to no less than 1 RTT). We fixed the value of $MaxRound$ to about 10 RTTs in our simulations and (passively) checked that it never exceeded the period of a control packet.

Handling dropped control packets: If a flow's control packet gets dropped, the end-host will not received any more rate updates, and the s-PERC algorithm is stalled because the flow's bottleneck rate information is no longer carried between its links. Moreover, the links on the path of the flows would continue to allocate the same bandwidth to the flow forever. In s-PERC*, the source end-host detects the dropped control

[1]The only exception is when a large number of flows start at once; the control traffic could exceed the link capacity in the unlikely event that first control packet of 2000 different flows arrive at a link in precisely the same RTT (assuming a control packet size of 64 bytes, a link capacity of 100 Gb/s and an unloaded RTT of 10.2 μs).

packet using a timeout RTO_{ctrl}, stops sending any new data packets and restarts the flow. The timeout is long enough for the links to have cleared any state corresponding to the flow (as described below). When the source end-host restarts the flow, it sends an initial control packet to request a new rate. It would resume transmitting data-packets starting from the last unacknowledged packet.

In order to allow the s-PERC algorithm to make progress, the per-link state in the active control packets must be consistent with the aggregate $NumB$ and $SumE$ at the links. For example, a link on the path of the affected flow should remove the flow's contribution to $NumB$ or $SumE$ since the flow's control packet is no longer active. In s-PERC*, the links use shadow variables $NumB'$ and $SumE'$ to re-compute the latest values of $NumB$ and $SumE$ on the side (lines 11–13 and 30–33 in Algorithm 8) and sync up periodically (Algorithm 9) every $round$, right after $MaxE$ and $MaxE'$ are reset. To implement this consistency scheme, in addition to the shadow variables, the links also keep count of their local $roundNumber$ and copy this to a per-link field $roundNumber[l]$ in the control packet so that they can identify if they have already accounted for the flow in a particular $round$ (Table 4.1). A similar consistency scheme for a different max-min algorithm was first described in [33], with a proof of correctness. For an example of why this works, consider a flow f whose control packet was dropped at some time T right after it had been classified as $\hat{E}$ at some link l. The shadow $SumE'$ at the link will not include flow f's allocation two $rounds$ from T, since the flow's control packet was dropped and not retransmitted until later. However, after the source detects the packet drop at some time following $T + 2\, rounds$, it will retransmit a control packet, initialized as for a new flow, with $roundNumber[l] = 0$. This new control packet will be in sync with the aggregate and shadow link state at l.

We have verified using numerical simulations that this mechanism allows links to converge to the right allocation for a static set of flows, despite random packet drops (e.g., packet drop probability 1e-4, see §4.3.3 for details). The proof of correctness is left to future work.

Algorithm 8 Control Packet Processing at link l for s-PERC*. We show the steps that make s-PERC* robust to control packet drops (Lines 5–7, 11–16, 27 and 33–36). For simplicity, we do not show the steps that calculate rate-limits for control packets and automatically adjust the value of $round$. See also Algorithm 9.

1: $SumE = 0$: sum of allocations of flows in $\hat{E}$ in this round ▷ Initial state at link l
2: $NumB = 0$: number of flows in $\hat{B}$ in this round
3: $MaxE = 0$: maximum allocation of flows moved to $\hat{E}$ since last round
4: $MaxE' = 0$: maximum allocation of flows moved to $\hat{E}$ in this round
5: $SumE' = 0$: sum of allocations of flows in $\hat{E}$ since last round ▷ Shadow state
6: $NumB' = 0$: number of flows in $\hat{B}$ since last round ▷ Shadow state
7: $roundNumber = 1$ ▷ Interval for resetting aggregate variables
8: **if** $s[l] == \text{E}$ **then** ▷ Flow was last bottlenecked elsewhere
9: $\quad SumE \leftarrow SumE - a[f]$ ▷ Update link state to assume flow is going to be bottlenecked
10: $\quad NumB \leftarrow NumB + 1$
11: **if** $roundNumber[l] == roundNumber$ **then** ▷ Flow has been seen in this $round$
12: $\quad$ **if** $s[l] == \text{E}$ **then** ▷ Update shadow link state to assume flow is going to be bottlenecked
13: $\quad\quad SumE' \leftarrow SumE' - a[f]$ ▷ SumE' includes previous allocation of flow
14: $\quad\quad NumB' \leftarrow NumB' + 1$
15: **else** ▷ Flow has not been seen in this $round$
16: $\quad NumB' \leftarrow NumB' + 1$ ▷ Update shadow link state to assume flow is going to be bottlenecked
17: $b \leftarrow \frac{C - SumE}{NumB}$
18: $e \leftarrow min_{\substack{m \in P_f \setminus l, \\ \text{i}[m]=0\}}} b[m]$ (or ∞ if there is no other link in P_f with ignore bit unset) ▷ Find flow's new limit rate
19: $a \leftarrow \min(b, e)$ ▷ Find flow's new allocation
20: **if** $b <= e$ **then** $s \leftarrow \text{B}$ ▷ Flow is now bottlenecked here, classify as $\hat{B}$
21: **else** $s \leftarrow \text{E}$ ▷ Flow is now bottlenecked elsewhere, classify as $\hat{E}$
22: $b[l] \leftarrow b, \quad a[l] \leftarrow a, \quad s[l] \leftarrow s$ ▷ Update control packet
23: **if** $b < MaxE$ **then** $i[l] \leftarrow 1$ ▷ Bottleneck rate is low; do *not* propagate it
24: **else** $i[l] \leftarrow 0$ ▷ Bottleneck rate is high enough; propagate it
25: **if** flow is leaving **then** ▷ Update link state to remove flow
26: $\quad NumB \leftarrow NumB - 1$
27: $\quad NumB' \leftarrow NumB' - 1$ ▷ Update shadow link state
28: **else if** $s == \text{E}$ **then** ▷ Update link state to reflect flow is in $\hat{E}$
29: $\quad NumB \leftarrow NumB - 1$
30: $\quad SumE \leftarrow SumE + a$
31: $\quad MaxE \leftarrow \max(MaxE, a)$
32: $\quad MaxE' \leftarrow \max(MaxE', a)$
33: $\quad NumB' \leftarrow NumB' - 1$ ▷ Update shadow link state
34: $\quad SumE' \leftarrow SumE' + a$ ▷ Update shadow link state
35: **if** $roundNumber[l] < roundNumber$ **then** ▷ Flow has not been seen in this $round$
36: $\quad roundNumber[l] \leftarrow roundNumber[l] + 1$ ▷ Update control packet

Table 4.1: Initial Control Packet for Flow f_W of size 10 KB in s-PERC*, after it has left the end-host. The control packet carries information for each link that the flow crosses. The first row corresponds to the virtual link l_0, which carries size information that is relevant for short flows. Note that the bottleneck state, allocation and $roundNumber$ for this virtual link are optional and can simply be stored at the end-host, or even ignored.

Link	Bottleneck State (s) Allocation (a)	Bottleneck rate (b)	Ignore bit	roundNumber
l_0	(E @ 0)	10 MB / 1 RTT	0	(0)
l_{30}	E @ 0	0	1	0
l_{12}	E @ 0	0	1	0
		$Tx_{ctrl} : \infty$		
		$FIN : 0$		

Algorithm 9 Timeout action at link l for s-PERC*, every round starting from time $T_0(l)$.

1: $MaxE \leftarrow MaxE', MaxE' \leftarrow 0$
2: $SumE \leftarrow SumE', SumE' \leftarrow 0$
3: $NumB \leftarrow NumB', NumB' \leftarrow 0$
4: $roundNumber \leftarrow roundNumber + 1$

Reverse-path symmetry of control packets: A control packet must traverse the same links as the data packets in both the forward and reverse directions. In a fat-tree or spine-leaf topology (and in our data-center simulations), s-PERC* enforces path-symmetry of the forward and reverse control packets using a non-standard ECMP (as in [29]) where the hashing is done on the sorted src, dst tuple, and the same hash functions are used at all switches that are in the same layer of the spine-leaf topology. This way, they will hash to the same switch in the next layer of the topology, for packets between a given pair of end hosts and there is no packet overhead. Note that such modifications should be relatively easy with programmable switches. In an arbitrary topology (and in our WAN simulations), s-PERC* allows links to update control packets only when the packets traverse in their forward direction, from source to destination end-host. This requires no changes to routing but doubles the convergence time in the worst case. Alternatively, one can log the path of the control packet in the forward direction and then use source routing for the reverse direction.

Accounting for bandwidth used by short flows: Short flows may not have enough bytes to send at the allocated rate for a full RTT. To avoid allocating more bandwidth than necessary, the s-PERC* control packet carries an additional field for a virtual link l_0 (unique to each flow) on the flow's path, which has a capacity equal to remaining bytes$/RTT$ (similar to [50]) (row 1 of Table 4.1). Concretely, this means that the packet must carry an additional bottleneck rate $b[l_0] =$ remaining bytes$/RTT$ and ignore bit $i[l_0] = 0$, which the other links consider when calculating a limit rate for the flow.

Optimizing latency for short flows: s-PERC* starts short flows at line rate and prioritizes them at every

link (with priority second to control packets, for a total of three priority levels for all traffic). This optimization provides minimum possible latencies for short flows in heavy tail workloads (e.g., web search, data mining) where they contribute a small fraction of the total bytes (see §4.3.2).

Control packet size: The s-PERC* control packet carries five pieces of per-link information: the allocation a (4 bytes) and bottleneck state s (1 bit), bottleneck rate b (4 bytes) and $ignore$ bit (1 bit), and the $roundNumber$ (1 byte) (Table4.1). Together with the flow size information (4 bytes for $b[l_0]$ and 1 bit for $i[l_0]$), the FIN tag (1 bit) and the control-packet sending rate Tx_{ctrl} (4 bytes), this requires 46 bytes for a two-level data-center with four hops. The convergence bound of s-PERC is unchanged if we replace the per-link bottleneck rate with the minimum and additionally carry the identifier of the link with the minimum rate. This requires a total of 30 bytes and fits within a minimum-size packet. We believe it is possible to optimize the control packet state further to carry only the bottleneck state as a per-link variable, similar to [63]. The global control packet state would include the allocation, common to all links where the flow is $\hat{E}$ in each direction, and the bottleneck rate, which carries the minimum bottleneck rate at any link. We leave this optimization to future work.

Headroom: We want transient queues to drain quickly when a link is over-subscribed for extended periods of time. A transient queue can build up, when a flow's end-host is asked to increase its rate by a bottleneck link, before other flows that share the link have decreased their rates. Consider the example of two flows sharing a single 100 Gb/s bottleneck link. The first flow to start gets an initial rate of 100 Gb/s, while the second flow gets an initial rate of 50 Gb/s. At the instant when the second flow's source sees the initial rate of 50 Gb/s, the first flow's source may still be sending at 100 Gb/s. The first flow would get a new rate of 50 Gb/s within a round-trip but a transient queue can build up while the second flow sends traffic at 50 Gb/s and the first flow is yet to decrease its rate. The s-PERC* algorithm allows links to leave some fraction (e.g., 1%-2%) of the capacity unallocated as headroom, to allow transient queues to drain quickly.

Implementing s-PERC in hardware: The next two points describe how to approximate the s-PERC algorithm to implement it in hardware. These techniques were used in our 40 Gb/s NetFPGA prototype and are relevant for programmable switches.

Atomic read-modify-write updates: Control packets need to be processed at line-rate (i.e., less than the arrival time of a minimum-size packet) in case they arrive back to back. The most demanding computation is to check whether $b \leq e$, line 8 of Algorithm 6, which we simplify to $(C - SumE) <= e * NumB$. This requires an integer multiplication, a subtraction, and a comparison, and because the state $(NumB, SumE)$ is updated as a read-modify-write, the computation must finish in one cycle (one minimum-packet time). While multiplication is not typically supported natively in fixed-function switch ASICs, newer programmable

switches include integer ALUs for read-modify-write state updates [1] running at faster than 1 GHz.

Approximating division: In order to classify a flow as B or E, the switch needs to calculate $b = (C - SumE)/NumB$, which requires a division, which is harder to do in one cycle. Fortunately, the division can be pipelined (i.e., can take more than one clock cycle) as it does not need to modify the switch state atomically. Following [79] we can use look-up tables to approximate the division. Switch ASICs have hundreds of megabytes of look-up tables already, and so it is reasonable to assume we could use a small fraction (say, fewer than 1%) of the tables to perform a division. In our micro-benchmark results (see §4.3.3) we explore practical tables sizes that work well for s-PERC.

4.2 Hardware s-PERC Prototype

We implemented the s-PERC switch data plane in hardware using the 4x10 Gb/s NetFPGA SUME platform [85]. The end host is implemented using the MoonGen DPDK packet processing library [37]. The prototype demonstrates that s-PERC can be deployed at high link speeds. We use it to evaluate s-PERC's performance.

4.2.1 NetFPGA s-PERC Switch

We programmed the switch's packet processing logic in P4 [23] and compiled it to the NetFPGA platform using the P4→NetFPGA workflow [6], built on top of the Xilinx SDNet [9] compiler. The NetFPGA switch has a core clock frequency of 200 MHz and a 256-bit datapath; fast enough to support more than 40 Gb/s.

The P4 pipeline is as follows: after the standard L2 forwarding logic, the switch processes control and data packets differently. Data packets are marked as low priority, whereas control packets pass through a series of processing steps in order to implement the computations described in Algorithm 6. Two steps perform atomic operations on state variables stored in the switches ($SumE, NumB$, and $MaxE$). They are not expressed in P4, but implemented in Verilog and included in the processing pipeline using $P4_{16}$'s extern interface.

The most challenging part of the switch prototype design is implementing the 32-bit division at line rate. We use the lookup technique described in §4.1. We required fewer than 2048 TCAM entries (32-bit key, 10-bit result), and about 32 Kb of exact-match memory. These requirements fit comfortably within an FPGA and are tiny compared to the memories in line-rate switch ASICs.

Each output port of our NetFPGA switch maintains two 128 Kb queues: a high-priority queue for control packets and a low-priority queue for data traffic.

4.2.2 MoonGen s-PERC End Host

The s-PERC end host performs two main tasks: (1) receives and resends s-PERC control packets for each flow, and (2) uses the bandwidth demands stamped in received control packets to perform per-flow rate limiting. Our prototype uses a timing wheel to limit flow rates, based on Carousel [77]. For our prototype, we reserve 1 Gb/s of bandwidth for control traffic, leaving 9 Gb/s for data traffic.

4.3 Evaluating s-PERC* for Data Centers

First, we compare **convergence times** of s-PERC* with (the reactive) RCP algorithm [36] in §4.3.1. Second, we compare **flow completion times** (FCTs) of s-PERC* with reactive schemes RCP, a scheduling-based scheme p-Fabric and an ideal max-min fair allocator in §4.3.2. Both convergence time and FCT results in this section use the ns-2 [4] simulator. Third, we perform micro-benchmarks to verify that s-PERC* is robust to dropped control packets and to imprecise calculations that may be done in switch hardware. These micro-benchmarks use numerical simulations of s-PERC* in Python and are reported in §4.3.3. Finally, in §4.4 we evaluate our 4 x 10 Gb/s NetFPGA hardware prototype, comparing s-PERC* convergence times with TCP and DCTCP.

Setup for convergence Time and FCT experiments: We use a three-tier topology with 100 Gb/s edge links from 144 servers to nine TORs, with 400 Gb/s uplinks to four aggregation switches. The propagation delay of each link is 0.2 μs, end-host delay is 2.5 μs, and the longest RTT between any two hosts is 11.6 μs for four hops (4 x 0.2 μs + 4 x 2.5 μs). The switch buffer size is set to 128 KB per port for RCP and s-PERC* and 2.25 MB per port for p-Fabric. To load balance traffic across different paths, the switches use a modified ECMP as described in §4.1 (for s-PERC* and RCP[2]) and standard ECMP for p-Fabric.

4.3.1 Convergence Times

Our first experiments evaluate how the algorithms converge after a fraction of the flows are replaced with new ones; something we refer to as "churn."

Workload: We start flows between random pairs of hosts, with 20 flows per server on average. After the rates have converged, we replace 40% of the flows. We replace all flows at once to evaluate convergence times of the scheme, as well as robustness to sudden changes in traffic matrix. Our graphs show results for ten changes per run, over 100 runs, for a total of 1000 convergence tests. The number of flows per server is the same order of magnitude as the number of simultaneously active flows we observed on congested links

[2] s-PERC* requires symmetric forward and reverse paths though RCP does not.

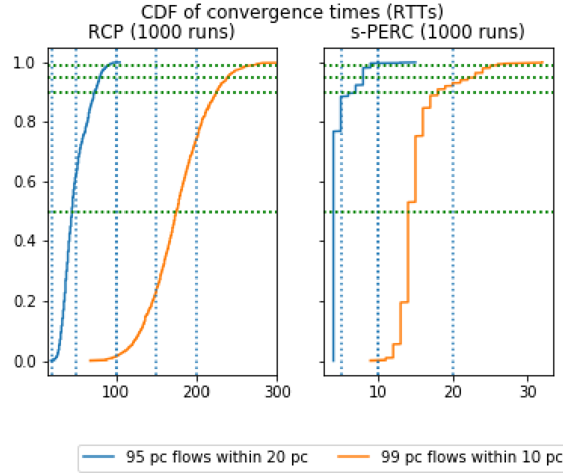


Figure 4.1: CDF of convergence times for 40% churn for RCP and s-PERC*. Note difference in x-axis scales. Settings- RCP α : 0.4, β : 0.2. s-PERC* initial timeout : 20 us, control overhead : 4%, headroom : 2%.

in simulations of realistic workloads (such as search and data mining) using Poisson arrivals at 80% load (for FCT experiments in §4.3.2).

Metrics: We say that the flow rates have converged when most of the flows remain within a small distance of the ideal max-min values for at least 100 consecutive RTTs. Figure 4.1 compares the convergence times for s-PERC* with RCP when we change 40% of the flows each time the flows converge during the run. We look at different degrees of accuracy, from when 95% of flows are within 20% of ideal to when 99% of flows are within 10% of ideal.[3]

Results: As Figure 4.1 shows, s-PERC* converges ten times faster than RCP. In order to get within 10% of the ideal rates for at least 99% of flows s-PERC* takes 14 RTTs at the median while RCP needs 174 RTTs. In fact, it only takes s-PERC* 4 RTTs (median) in order to get within 20% of the ideal rates for at least 95% of flows; RCP, on the other hand, needs 45 RTTs. Regardless of the metric, s-PERC* is at least ten times faster than RCP.

4.3.2 Flow Completion Times

Workload: We use flow size distributions from two realistic workloads from data centers running search and data mining applications with loads of 60% and 80% ([13]) and assume that flow arrival times are Poisson.

Metrics: We look at the normalized flow Completion times (FCTs) of different groups of flows. The actual

[3]For s-PERC*, we typically only allocate 95% of the capacity for data packets, but we still require it to converge to within 10% of the ideal, which uses 100% of the capacity for data packets.

flow completion times are normalized by the time it would take to transmit the flow on an unloaded network. We group flows based on their size and the fraction of total bytes they carry. We sort the flows in increasing order of size. The first bin contains the smallest flows that contribute 1% of the total bytes. The remaining bins contain equal fractions (by bytes contributed) of the remaining flows.

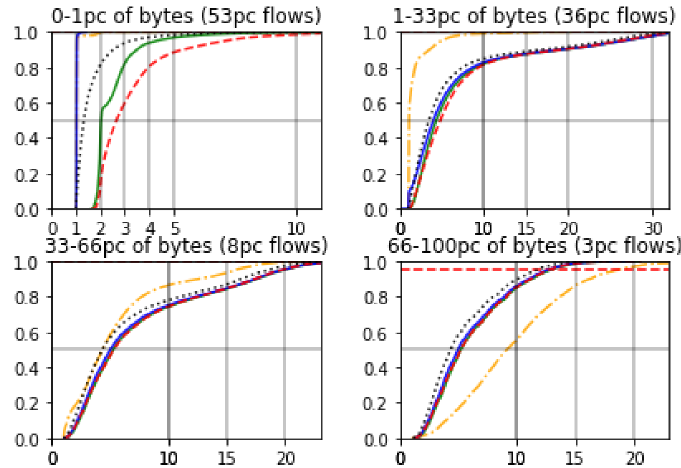
Both workloads we consider are heavy-tailed, where a few flows are extremely large and carry most of the bytes. Most flows are small and end up in the first bin. For the search workload, the first bin contains 53% of flows (max flow size: 79 KB, which is roughly half the BDP); for the data-mining workload, the first bin contains 94% of flows (max flow size: 2381 KB, which is roughly 16 times the BDP). For search, we divide the remaining flows into three equal groups, each contributing 33% of total bytes. For data-mining, we have one group that contributes 99.5% of total bytes and carries 8% of the flows.[4]

Schemes compared: We compare RCP, s-PERC*, p-Fabric, and an ideal max-min allocator. RCP and s-PERC* are fast rate-allocation schemes that target max-min fairness, and the ideal max-min allocator serves as reference for both. On the other hand, p-Fabric is a state-of-the-art congestion control scheme that seeks to emulate SRPT (shortest remaining processing time)– switches schedule packets in increasing order of remaining flow size, while end hosts use a minimal rate-control scheme (a TCP variant). We evaluate two versions of s-PERC*. The first version, s-PERC* basic, does not optimize latency for short flows— all flows must wait to get a rate from the control packets before sending any packets and all flows get the same priority at all links. In the second version s-PERC* short, we start short flows at line rate and prioritize them at all links. We define $short$ as flows that are < 1 BDP[5].

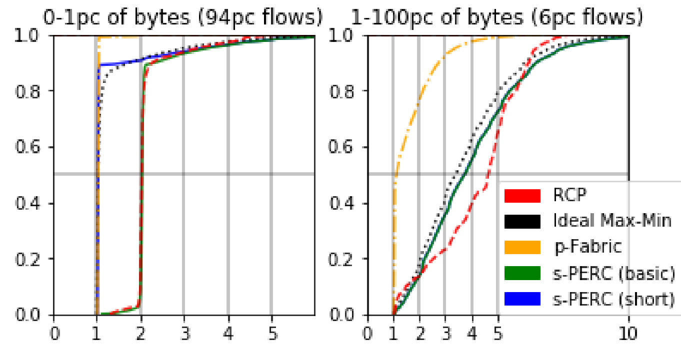
Results: Consider the search workload at 60% load (Figure 4.2a). First we compare the max-min fair schemes. For the smallest flows (bin 1), basic s-PERC* and RCP both start at $x = 2$, because of the extra RTT it takes to get a starting rate. For the smallest flows, s-PERC* basic is 25-50% better than RCP because of shorter queues. For the remaining flows that carry 99% of the bytes, both are close to the ideal max-min allocator and to each other. The optimized version, s-PERC* short, however, is 400% better than RCP for the smallest flows. The smallest flows start at line rate and see no queues with s-PERC*; hence, they can get the smallest flow completion time possible. Because these flows finish in less than 1 RTT, we do not send any control packets to schedule bandwidth for them. As a result, the s-PERC* algorithm allocates all the link capacity to the remaining larger flows. The additional "schedulable" bandwidth (about 2%) for the large flows makes s-PERC* short slightly more efficient than s-PERC* basic for the large flows. This effect is more evident at 80% load.

Next, we compare s-PERC* short with p-Fabric. Note that they try to achieve different global objectives,

[4]So that the CDF plot is statistically significant given the total number of flows in each experiment (100,000.)
[5]A BDP is defined as the product of the server's up-link capacity and the longest RTT in an unloaded network.



(a) 60% load running web-search workload. Note that there are not enough flows in the fourth bin for the empirical CDF to be accurate above the horizontal dotted-line (fewer then 30 samples for the 95th percentile.)



(b) 60% load running data-mining workload.

Figure 4.2: FCT results for RCP, s-PERC* (with and without optimizing for short flows), ideal max-min and p-Fabric at 60% load for search, and data-mining workloads. Normalized FCT on x-axis. Settings– RCP $\alpha : 0.4, \beta : 0.2$. s-PERC* initial timeout : 20 μs, control overhead : 2%, headroom : 2%.

that is, max-min vs SRPT. p-Fabric aims to emulate a greedy SRPT-based algorithm, which is known to achieve close to optimal *average* FCTs [13], [19]. This may come at the expense of hurting the largest flows. As a result, while the middle 66% of flows get better FCTs under p-Fabric, the largest flows get almost 200% worse throughput. For the smallest flows, optimized s-PERC* is better than p-Fabric at the tail, and both are close to the minimum possible at the median.

Results at 80% load (Figure 4.3) are qualitatively similar except for two differences. First, the difference between ideal (and s-PERC*) and p-Fabric for flows in the third bin is larger. Second, for the search workload, while optimized s-PERC* matches RCP for the large flows, basic s-PERC* is about 5% worse (at the median). This is because it becomes more important at high loads to have additional “schedulable” bandwidth, which

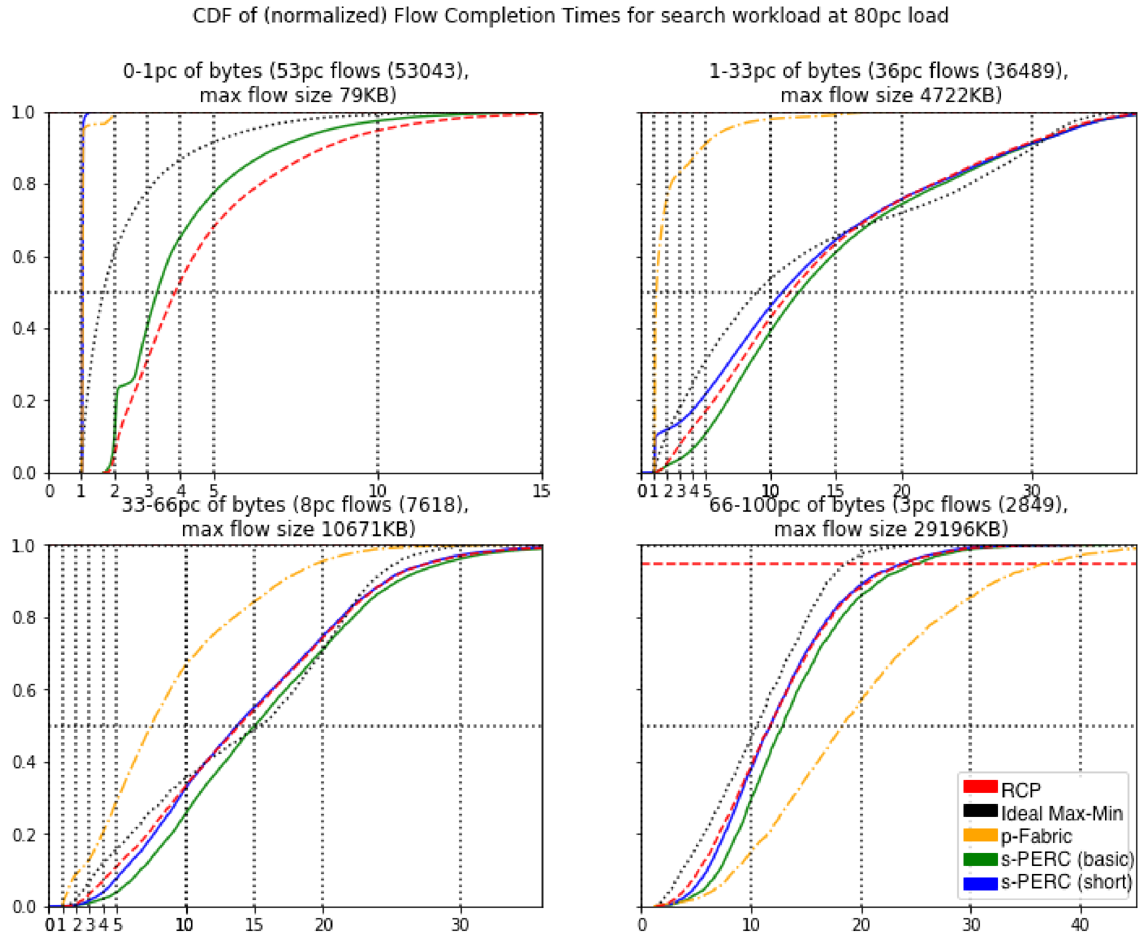


Figure 4.3: FCT results for RCP, s-PERC* (with and without optimizing for short flows), ideal max-min and p-Fabric at 80% load for search workload. Settings– RCP $\alpha : 0.4, \beta : 0.2$. Normalized FCT on x-axis. s-PERC* initial timeout : 20 μs, control overhead : 2%, headroom : 2%. Note that there are not enough flows in the fourth bin for the empirical CDF to be accurate above the horizontal dotted-line (fewer then 30 samples for the 95th percentile.)

the large flows use efficiently.

Next, consider the data-mining workload at 60% load (Figure 4.2b). Flows that are less than 1 BDP are part of the first bin and make up less 1% of the total bytes. Hence, optimized s-PERC* can easily send the short flows at line rate and high priority to obtain the minimum possible flow completion times. The remaining flows get a rate from the switches so that their FCT curve closely tracks the ideal max-min curve. This explains the sharp bend for optimized s-PERC* at the 90th percentile for the 1% bin. Results at 80% load are similar (Figure 4.4).

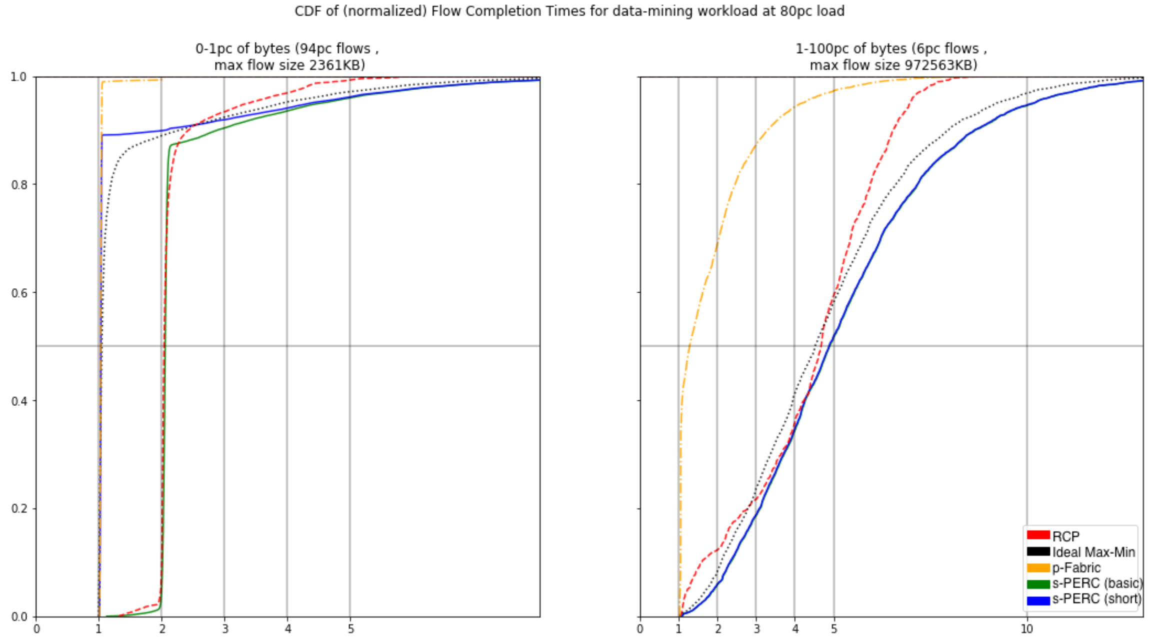


Figure 4.4: FCT results for RCP, s-PERC* (with and without optimizing for short flows), ideal max-min and p-Fabric at 80% load for data-mining workload. Normalized FCT on x-axis. Settings– RCP $\alpha : 0.4, \beta : 0.2$. s-PERC* initial timeout : 20 μs, control overhead : 2%, headroom : 2%.

4.3.3 Micro-Benchmarks

We used numerical simulations of s-PERC* (in Python) to verify that it is robust to control packet drops and to imprecise rate calculations in hardware. Our simulations use a fully connected network with K links, where each link has a uniform delay of 10 us (with some random jitter on the order of a nanosecond to allow reordering of packets). We use static workloads where for each workload, we randomly generate a set of long-lived flows, where each flow traverses the same number of links (for simplicity.) Note that this is not a packet-level simulation, and we only perform the calculations involved in control packet processing at intervals dictated by the link delays and configured timeouts (e.g., every round) using an event-queue. Moreover, we use a static value of $round$ instead of dynamically adjusting it based on the number of flows.

Robustness to control packet drops

Control packets in s-PERC* are limited to a small fraction (2%-5%) of the link capacity and prioritized so we expect packet drops to be rare. However, as described in §4.1, s-PERC* can recover from packet drops by recomputing the aggregate state using shadow variables $NumB'$ and $SumE'$ on the side, which are synced with $NumB$ and $SumE$ every round. We used a fully connected network of ten switches ($K = 100$ links).

We fixed the values of $round = 200\mu s$ (roughly 1.2 RTTs) at every link and $RTO_{ctrl} = 360\mu s$ (almost 2 rounds) at every source. In each run, we started with a randomly generated set of a 400 long lived flows and let the algorithm run for 200 RTTs with the control packet drop probability at all links set to 0.01% for the first half of the run (on average, there are at least a few tens of drops in each run.) We observed that in the time following the last packet drop until the end of the run, s-PERC* converged to the max-min allocation for all flows in every run.

Calculation precision

We cannot implement infinite precision floating point division at line rate in the switches. However, it is possible to implement approximate division using lookup tables with mean errors of less than 1% at the cost of hundreds of Kilobytes of SRAM and tens of entries in the TCAM [34]. We present results from numerical simulations (in Python) to understand the errors in flow rates due to approximate divisions. We approximated division by using look-up tables of different sizes. We used a fully connected network of eight to twelve switches ($K = 56$ to $K = 132$ links) and simulated a total of 1500 different static workloads where each workload has an average of 20 flows per link (140–264 flows). We ran each simulation for 1000 RTTs and tried three versions of s-PERC* where division is implemented using 32-bit integer look-up tables of different sizes (13 KB, 52 KB, and 340 KB). We define convergence as the first time when each of the flow rates has been within T% of the optimal value for at least 200 consecutive RTTs and until the end of the run. We observed that as the table sizes get smaller the flows stabilize to values that are farther from the optimal or have wider oscillations around the optimal. Hence we looked at different thresholds for convergence, ranging from $T = 10\%$ to $T = 40\%$.

We found that a table size of at least 52 KB was needed in order for all the workloads to converge to within 30% of the optimal rates. As shown in Figure 4.5, there isnâĂŹt a significant difference in the convergence times as we switched from a table of size 52 KB to a table of size 340 KB. This result indicates that s-PERCâĂŹs precision requirements are easily met in todayâĂŹs switch hardware, which typically have hundreds of megabytes of look-up tables.

4.4 NetFPGA Hardware Evaluation

We compared TCP Reno, DCTCP, and s-PERC on a test bed with three NetFPGA switches, connecting four servers using 10 Gb/s links. For each congestion control algorithm, we ran two experiments: (1) an *incast experiment* with three senders transmitting to one receiver; (2) a *dependency chain experiment* in which

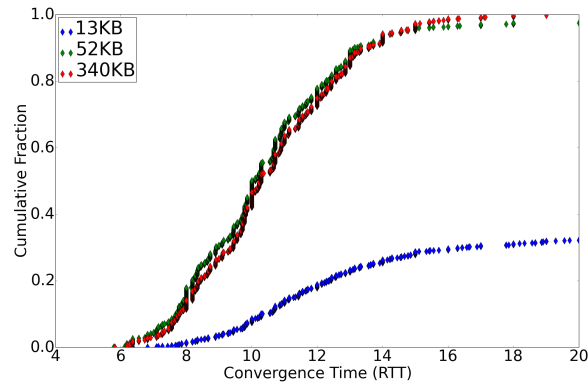


Figure 4.5: CDF of convergence times (to within 30% of ideal) for approximate division with different lookup table sizes.

multiple bottleneck links depend on each other in order to converge.

The goal of the evaluation is twofold: first, to confirm that s-PERC can be implemented in hardware at 40 Gb/s using the NetFPGA SUME platform (4 x 10 Gb/s); second, to evaluate how much faster a practical implementation of s-PERC converges compared to existing reactive algorithms.

Setup: Figure 4.6 shows the topology used for both experiments with 10 Gb/s links. For the incast experiment, hosts H1, H2, and H3 send some number of flows to host H4, all starting at approximately the same time. For the dependency chain experiment, we create a chain of length two: The green and blue flows start first and then the red flows are added. Before the red flows start, the bottleneck is link B1, which changes to link B2 when the red flows are added, which in turn allows the green flow to increase its rate.

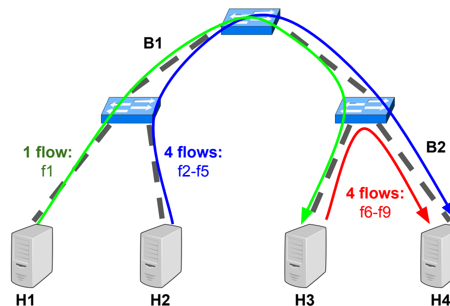


Figure 4.6: The topology and traffic pattern used for the two-level dependency chain experiment. Once the red flows are added, the bottleneck rate of f1 (the green flow) increases as the bottleneck link for f2-f5 (the blue flows) moves from B1 to B2.

Each of the servers has one dual port 10 GE network interface card. Three 10 GE ports of the NetFPGA SUME are used for data and control packets, and one 10 GE port is used to monitor traffic through the

switch. The monitor receives a copy of all packets that pass through the switch, trimmed to about 80 bytes and appended with a 5 ns resolution timestamp. The servers are synchronized using PTP (to within ten microseconds) so that we can accurately coordinate the flows across the servers. We use *iperf* to generate TCP and DCTCP flows and the DPDK-based application to generate flows for s-PERC.

Metrics: For TCP and DCTCP, convergence is defined as the time from the last flow change until the beginning of a 50 ms window during which an exponentially weighted moving average of each flow rate has stabilized to within 20% of their respective stable value. For s-PERC, we reserve 1 Gb/s of each link's bandwidth for control traffic and 9 Gb/s for data traffic. Hence, we define convergence relative to max-min rates, assuming link capacities of 9 Gb/s. Specifically, we measure convergence time as the time from the last flow change until the control packets of all flows specify a bandwidth demand that is within 20% of their max-min rates for at least 50 ms.

Results: Table 4.2 shows that s-PERC converges at least 500 times faster than TCP and at least 150 times faster than DCTCP in small incast scenarios. Furthermore, as the size of the incast increases, the convergence time for TCP and DCTCP becomes less consistent and in some cases, they fail to converge. On the other hand, the convergence time of s-PERC in this scenario is solely dictated by the round trip time of the control packets. Hence it is not significantly affected by the size of the incast as long as there is sufficient bandwidth for control packets.

Table 4.2: Results for incast experiments using a NetFPGA test bed. Average and standard deviation of convergence times, where each result is a summary of 10 runs.

	Convergence Time (ms)		
# flows	TCP	DCTCP	s-PERC
2	137 ± 165	45 ± 12	0.27 ± 0.36
3	483 ± 496	44 ± 14	0.23 ± 0.55
10	N/A	N/A	0.21 ± 0.29

In the dependency chain experiment, s-PERC achieves a convergence time of 1.56 ± 0.23 ms, and DCTCP achieves a convergence time of 65 ± 30 ms. TCP, on the other hand, failed to converge for this experiment. For the workload applied in this experiment, the performance of s-PERC is largely dictated by the timeout used in the switch to reset *MaxE*. This timeout must be chosen to be as long as the maximum possible control packet RTT, which in the case of these experiments was 1 ms. According to Theorem 3.4.4, when the longest dependency chain is of length two and the maximum RTT is 1 ms, s-PERC's convergence time is upper-bounded by 14 ms. Note that this is still almost five times faster than DCTCP's average convergence time.

We expect that s-PERC would converge even faster if control packets were prioritized at the NIC.

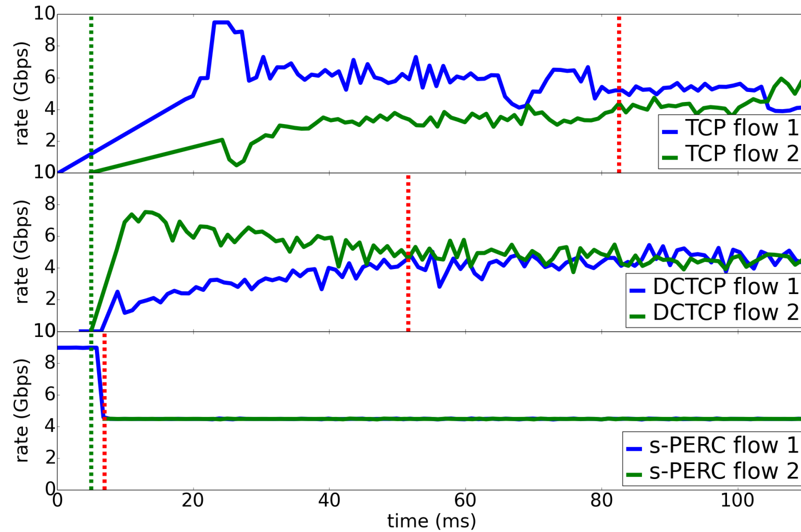


Figure 4.7: Example convergence behavior for TCP, DCTCP, and s-PERC running on a NetFPGA test bed. Two flows share a 10 Gb/s link. The first vertical dashed line is the time the second flow is added to the bottleneck link; the second dashed line is when the flow rates are declared converged.

Figure 4.7 shows a comparison of the convergence behavior for TCP, DCTCP, and s-PERC during a two-flow incast experiment. TCP generally takes longer than DCTCP for the flow rates to stabilize, and both are about an order of magnitude slower than s-PERC for this example run. Note that the s-PERC plot shows convergence of the measured flow rates, which is necessarily longer than the convergence of the bandwidth demands within the control packets, as listed in Table 4.2.

Chapter 5

Future Work on PERC Algorithms

In this chapter we discuss some future avenues of research that build on s-PERC and PERC algorithms.

5.1 Future Avenues for Research

While our experiments in the previous chapter focused on the data center setting, the s-PERC algorithm is general enough to work in arbitrary networks. Preliminary simulations of s-PERC for a wide-area network (WAN) suggests that it can converge 10–15 times faster than reactive alternatives. Table 5.1 shows convergence-time statistics for s-PERC and RCP in a topology similar to Google's inter–data-center WAN called B4 [56]. There are 18 different data-center sites, and links connecting different sites have uniform capacity but heterogeneous propagation delays. Unlike the intra–data-center setting, RTTs of different flows in a WAN can be very different, ranging from one to hundreds of milliseconds, depending on the flow's path. Additionally, the bandwidth-delay–product (BDP) for a 100 ms long path in a 10 Gb/s WAN network is three orders of magnitude more than the BDP for a 10 μs long path in a 100 Gb/s data-center network. Our experiment is as follows. In each run, we start 100 long-lived flows from every site (toward a randomly selected destination site) and measure the flow rates (as configured at the source end-host) every millisecond. We run the simulation for a preconfigured number of seconds. We say that the flow rates have converged when at least x% of the flows remain within y% of the ideal max-min values for at least ten consecutive RTTs and until the end of the simulation for a given value of x and y. In Table 5.1, we report the average convergence times over all runs that converged, for different degrees of convergence, from when 80% of flows are within 30% of ideal to when 99% of flows are within 10% of ideal. The convergence time of each run is normalized by the average RTT (2 x propagation delay) of flows in the run (for reference, the average RTT of flows over

Table 5.1: Convergence times for RCP and s-PERC in a WAN topology for different link rates and convergence metrics. The first table reports results for a WAN with 1 Gb/s links. The first entry of the first table indicates that when using RCP, in all 100 out of 100 runs, at least 80% of the flows converged to rates that were within 30% of their max-min fair allocations before the end of the run. The second entry indicates that the mean convergence time was 78.9 RTTs. Settings—RCP $\alpha : 0.4, \beta : 0.2$, header-size: 40 B, simulated time : 30 and 60 s for 1 and 10 Gb/s respectively. s-PERC initial $round$: 100 ms, control overhead : 4%, headroom : 2%, control-packet size: 64 B[1], simulated time : 4 s. Common settings—buffer size: 125 KB, 100 flows per server (site).

Convergence times for WAN with 1 Gb/s links, 100 flows/site (100 runs)

	converged runs (80 pc flows to 30 pc)	avg. convergence-time (RTTs)	converged runs (95 pc flows to 20 pc)	avg. convergence-time (RTTs)	converged runs (99 pc flows to 10 pc)	avg. convergence-time (RTTs)
rcp	100	78.9	100	141.1	99	233.0
sperc	100	5.1	100	10.3	100	15.7

Convergence times for WAN with 10 Gb/s links, 100 flows/site (100 runs)

	converged runs (80 pc flows to 30 pc)	avg. convergence-time (RTTs)	converged runs (95 pc flows to 20 pc)	avg. convergence-time (RTTs)	converged runs (99 pc flows to 10 pc)	avg. convergence-time (RTTs)
rcp	85	318.3	25	403.8	9	448.7
sperc	100	5.1	100	11.0	100	17.5

all 100 runs is 76 ms.) The convergence times of s-PERC are similar at 1 Gb/s and 10 Gb/s whereas that of RCP gets worse from 1 Gb/s to 10 Gb/s. Note that as the link capacity increases, congestion can build up more quickly, whereas the time it takes to react to congestion is fixed to RTT time-scale. Further experiments are needed to understand how s-PERC can improve flow completion times in the WAN.

We note some avenues for further optimization of s-PERC.

- The per-link state in the s-PERC control packet can become a cause for concern in networks with long paths. The s-PERC control packet (Figure 4.1) carries five pieces of state (roughly eight bytes) for each link on the path of the flow. It is an open question if one can re-factor the control packet state (following the example of SLBN [63]) to carry only the bottleneck state (one bit) as per-link state (plus a constant number of other fields), while maintaining the convergence guarantees of s-PERC.
- The s-PERC algorithm does not take software bottlenecks into account and assumes that servers will enforce the rates calculated by the network. However, it is possible that because of software bottlenecks, the server may not actually be able to send at the allocated rate. One possible solution is to model the bottleneck as an additional link (unique to each flow) on the flow's path, which has a capacity that varies over time to reflect the software bottleneck rate. We have not implemented this optimization in our simulations or prototype and leave this to future work.
- When RTTs are not homogeneous, such as in the WAN setting, some flows may be asked to increase

[1]The maximum path length has seven hops. This would require a 64 B s-PERC header, but we use a minimum-size packet, assuming that one can optimize the control packet state further.

their rates long before other flows (with longer RTTs) that share the same links have decreased their rates. This can result in queuing at the shared links. In s-PERC*, the end hosts change their rates as soon as they receive the control packets. In the data-center setting, it was sufficient to allow a 1%–2% headroom when allocating the link capacity to allow transient queues to drain. However, the end-host strategy may need to be modified for settings where RTTs are not homogeneous to ensure that the sending rates at the end-hosts are feasible all times. Charny et al. [28] describe a strategy that guarantees feasibility of transmission rates for a PERC algorithm, wherein the end host responds immediately when it is asked to decrease the sending rate but delays any rate increases by a certain time. Further experiments are needed to understand the benefits of this optimization for s-PERC.

PERC algorithms as presented in this thesis have separate control and data packets with bandwidth reserved for each. It would be interesting to consider alternate frameworks such as those where control information is piggybacked on data packets to understand how they trade off link utilization with convergence speed.

PERC algorithms can benefit from advances in other areas of networking. We noted earlier that programmable switches allow an easier path to deployment. New abstractions in the software stack can also be useful. For example, recent work [77] on new abstractions for rate-limiting individual flows at high speeds makes it easier to deploy rate-based PERC algorithms in a world where many congestion control algorithms are window-based. Alternatives to socket-based interfaces can provide PERC algorithms with richer information (such as flow sizes or estimate of software bottlenecks) to make better rate allocations.

Another avenue for future research is an incremental deployment strategy for PERC algorithms. One option to incrementally deploy a PERC algorithm would be to partition the network such that one subset uses PERC for finding rates for all flows, whereas the other partition uses a legacy congestion control algorithm. An alternative is a hybrid deployment, where a PERC algorithm and a legacy reactive algorithm coexist in the same network and serve different sets of applications. For the latter, it is an open question how to dynamically share bandwidth between the two kinds of algorithms in the same network.

PERC algorithms can also converge quickly to objectives beyond max-min fairness. For example, by using s-PERC as an underlying weighted max-min fabric for NUMFabric [66], one can converge quickly to general alpha-fair objectives. It is an open question if there is a PERC algorithm, like s-PERC, that can achieve the SRPT objective without keeping any individual flow state. Another open question is how to extend s-PERC to the multi-path setting.

5.2 Summary

The majority of existing congestion control algorithms are essentially reactive control systems, which must figure out rates purely by reacting to congestion signals typically at RTT time scales, then taking small steps over many iterations toward convergence.

As networks get faster and more data can fit into each RTT, buffers can fill up quickly even before there is time to react. Hence, in this thesis, we focused on a different class of algorithms: PERC algorithms, which do not rely on congestion signals but instead use explicit global information (like the number of flows crossing a link) to proactively compute rates for all flows.

We believe that congestion control should converge in a time limited only by fundamental dependency chains, which are a property of the traffic matrix and the network topology. Prior attempts to proactively calculate the fair-share rates in the network were not successful, because they required per-flow state, and the algorithms were not proved to converge.

With s-PERC, we have introduced a PERC algorithm that is practical (it does not require per-flow state, and the calculations are possible at line rate in relatively simple hardware) and guaranteed to converge; our results from simulations and a hardware prototype show that s-PERC is robust to churn and converges several times faster than other algorithms.

We have evaluated s-PERC in a data-center setting to validate that s-PERC achieves flow completion times that are closer to an ideal max-min scheme than a reactive algorithm, because of its faster convergence. For realistic workloads, s-PERC competes favorably with schemes that favor short flows, such as p-Fabric, yielding low latencies for short flows and high throughput for large flows. Preliminary simulations indicate that fast-converging PERC algorithms like s-PERC would be very well suited to long-haul networks where there is a need for fair bandwidth allocation between flows that that is predictable and avoids congestion. As we have mentioned in this chapter, there is more work to be done, and so we believe that this is only the first word about PERC algorithms in high-speed networks, not the last.

Chapter 6

Enabling Other Algorithms in Programmable Switches

In this chapter, we take a step back from congestion control and explore how to enable other algorithms like s-PERC to run in programmable switches. We look at the problem of compiling the logical look-up tables in a P4 program to physical resources in a target programmable switch pipeline while meeting data and control dependencies in the program. We study the interplay between Integer Linear Programming (ILP) and greedy algorithms to generate solutions optimized for latency, pipeline occupancy, or power consumption.

6.1 Introduction

The Internet pioneers called for "dumb, minimal and streamlined" packet forwarding [32]. However, over time, switches have grown complex with the addition of access control, tunneling, overlay formats, etc., specified in over 7,000 RFCs. Programmable switch hardware called NPUs [31], [10] were an initial attempt to address changes. Yet NPUs, while flexible, are too slow: the fastest fixed-function switch chips today operate at over 2.5Tb/s, an order of magnitude faster than the fastest NPU, and two orders faster than a CPU.

As a consequence, almost all switching today is done by chips like Broadcom's Trident [25]; arriving packets are processed by a fast sequence of pipeline stages, each dedicated to a fixed function. While these chips have adjustable parameters, they fundamentally cannot be reprogrammed to recognize or modify new header fields. Fixed-function processing chips have two major disadvantages: first, it can take 2–3 years before new protocols are supported in hardware. For example, the VxLAN field [59]—a simple encapsulation

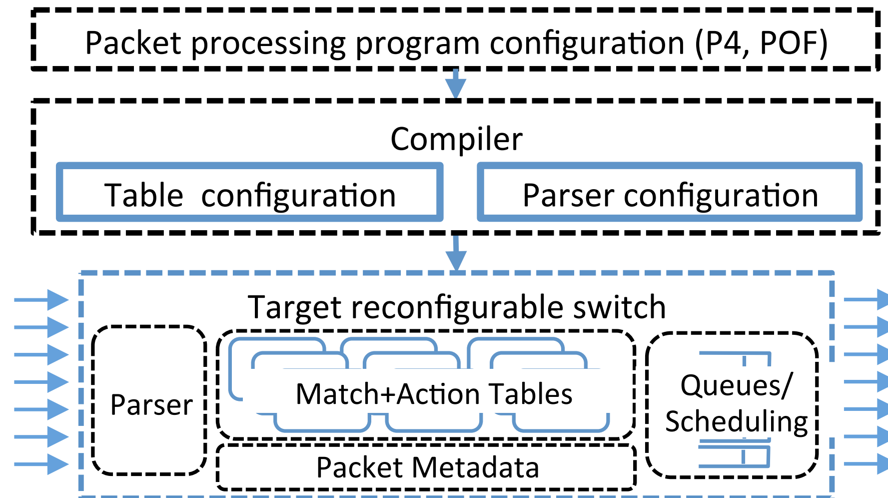


Figure 6.1: A top-down switch design.

header for network virtualization—was not available as a chip feature until three years after its introduction. Second, if the switch pipeline stages are dedicated to specific functions but only a few are needed in a given network, many of the switch table and processing resources are wasted.

A subtler consequence of fixed-function hardware is that networking equipment today is designed *bottom-up* rather than *top-down*. The designer of a new router must find a chip datasheet conforming to her requirements before squeezing her design bottom-up into a predetermined use of internal resources. By contrast, a top-down design approach would enable a network engineer to describe how packets are processed and adjust the sizes of various forwarding tables, oblivious to the the underlying hardware capabilities (Figure 6.1). Further, if the engineer changes the switch mid-deployment, she can simply install the existing program onto the new switch. The bottom-up design style is also at odds with other areas of high technology: for example, in graphics, the fastest DSPs and GPU chips [68], [84] provide primitive operations for a variety of applications. Fortunately, three trends suggest the imminent arrival of top-down networking design:

1. Software-Defined Networking (SDNs): SDNs [49], [69] are transforming network equipment from a vertically integrated model towards a programmable software platform where network owners and operators decide network behavior once deployed.

2. Reconfigurable Chips: Emerging switch chip architectures are enabling programmers to reconfigure the packet processing pipeline at runtime. For example, the Intel FlexPipe [3], the RMT [24], and the Cavium XPA [2] follow a flexible *match+action* processing model that maintains performance comparable to

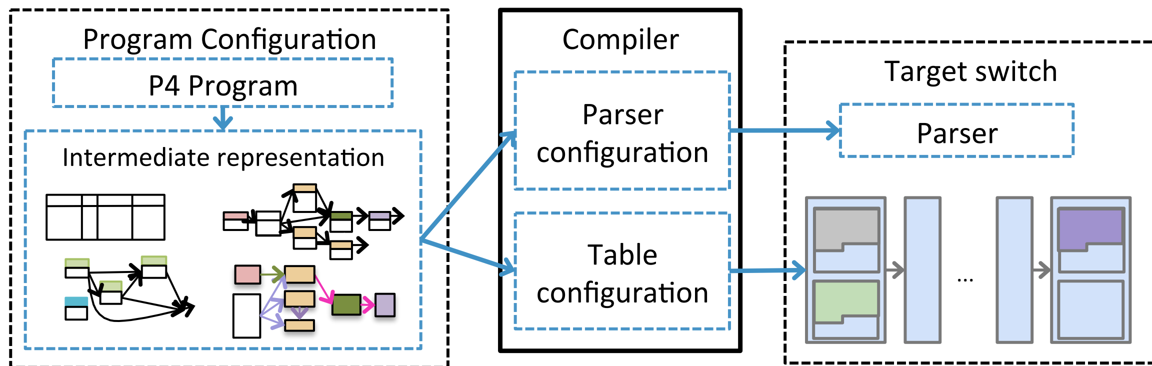


Figure 6.2: Compiler input and output.

fixed-function chips. Yet to accommodate flexibility, the switches have complex constraints on their programmability.

3. Packet Processing Languages: Recently, new languages have been proposed to express packet processing, like Huawei's Protocol Oblivious Forwarding [8] and P4 [23], [7], [5]. Both POF and P4 describe how packets are to be processed abstractly—in terms of match+action processing—without referencing hardware details. P4 can be thought of as specifying control flow between a series of logical match+action tables.

With the advent of programmable switches and high-level switch languages, we are close to programming networking behavior top-down. However, a top-down approach is impossible without a *compiler* to map the high-level program down to the target switch hardware. In this chapter, we the design of such a compiler, which maps a given program configuration—using an intermediate representation called a Table Dependency Graph, or TDG (§6.2.1)—to a target switch. The compiler should create two items: a *parser configuration*, which specifies the order of headers in a packet, and a *table configuration*, which is a mapping that assigns the match+action tables to memory in a specific target switch pipeline (Figure 6.2). Previous research has shown how to generate parsing configurations [42]; the second aspect, table configuration, is our focus.

To understand the compilation problem, we first need to understand what a high-level packet processing language specifies and how an actual switch constrains feasible table configurations.

6.1.1 Packet Processing Languages

High-level packet processing languages such as P4 [23] must describe four things:

- **Abstract Switch Model:** P4 uses the abstract switch model in Figure 6.1 with a programmable parser followed by a set of match+action tables (in parallel and/or in series) and a packet buffer.

- **Headers**: P4 declares names for each header field, so the switch can turn incoming bit fields into typed data the programmer can reference. Headers are expressed using a parse graph, which can be compiled into a state machine using the methods of [55], [42].
- **Tables**: P4 describes the *logical tables* of a program, which are match+action tables with a maximum size; examples are a 48-bit Ethernet address exact match table with at most 32,000 entries, or an 8K-entry table of 256-bit wide ACL matches.
- **Control Flow**: P4 specifies the control flow that dictates how each packet header is to be processed, read, and modified. A compiler must map the program while preserving control flow; we give a more detailed example of this requirement in §6.2.1.

6.1.2 Characteristics of Switches

Once we have a high-level specification in a language, the compiler must work within the constraints of a target switch, which include the following:

- **Table sizes**: Hardware switches contain memories that can be accessed in parallel and whose number and granularity are constrained.
- **Header field sizes**: The width of the bus carrying the headers limits the size and number of headers that the switch can process.
- **Matching headers**: There are constraints on the width, format, and number of lookup keys to match against in each match+action stage.
- **Stage Diversity**: A stage might have limited functionality; for example, one stage may be designed for matching IP prefixes, and another for ACL matching.
- **Concurrency**: The biggest constraints often come from concurrency options. The three recent flexible switch ASICs (FlexPipe, RMT, XPA) are built from a sequential pipeline of match+action stages, with concurrency possible within each stage. A compiler must analyze the high-level program to find dependencies that limit concurrency; for example, a data dependency occurs when a piece of data in a header cannot be processed until a previous stage has finished modifying it. We follow the lead of Bosshart, et al. [24] and express dependencies using a TDG (§6.2.1).

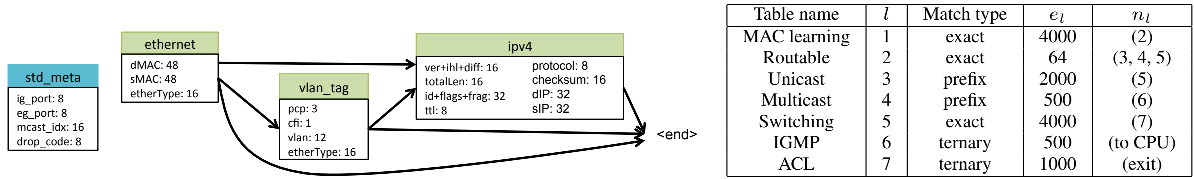
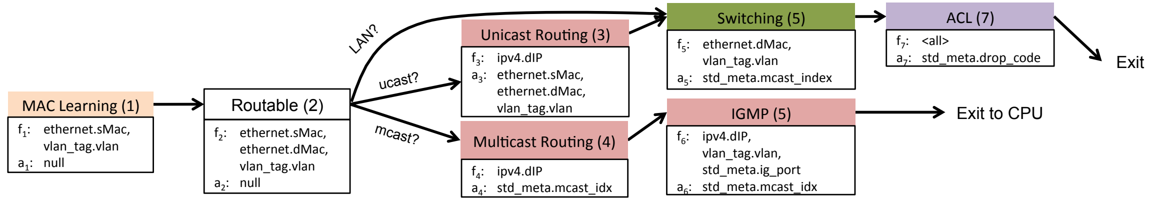


Table name	l	Match type	e_l	n_l
MAC learning	1	exact	4000	(2)
Routable	2	exact	64	(3, 4, 5)
Unicast	3	prefix	2000	(5)
Multicast	4	prefix	500	(6)
Switching	5	exact	4000	(7)
IGMP	6	ternary	500	(to CPU)
ACL	7	ternary	1000	(exit)

(a) The parse graph specifies the order of the packet headers (green); the metadata (blue) is separate. All field names and lengths (in bits) are also specified.

(b) Each logical table l has a table name, maximum entry count e_l, and next table addresses n_l.



(c) The control flow program. Each table l has match fields f_l and modified fields a_l.

Figure 6.3: A packet processing program named L2L3 describing a simple L2/L3 IPv4 switch.

6.1.3 Approach and Contributions

We define and systematically explore how to build a switch compiler by using abstractions to hide hardware details while capturing the essence required for mapping (§6.2 and §6.3). Ideally we would like a switch-dependent front-end preprocessor, and a switch-independent back-end; we show how to relegate some switch-specific features to a preprocessor. We identify key issues for any switch compiler: table sizes, program control flow, and switch memory restrictions. In a sense, we are adapting instruction reordering [57], a standard compilation mechanism, to efficiently configure a packet-processing pipeline. We reinterpret traditional control and data dependencies [57] in a match+action context using a Table Dependency Graph (TDG).

A second contribution is to compare greedy heuristic designs to Integer Linear Programming (ILP) ones; ILP is a more general approach that lets us optimize across a variety of objective functions (e.g., minimizing latency or power). We analyze four greedy heuristics and several ILP solutions on two switch designs, FlexPipe and RMT. For the smaller FlexPipe architecture, we show that ILP can often find a solution when greedy fails. For RMT, the best greedy solutions can require 38% more stages, 42% more cycles, or 45% more power than ILP. We argue that with more constrained architectures and more complex programs (§6.6), ILP approaches will be needed.

A third contribution is exploring the interplay between ILP and greedy, given ILP's optimal mappings despite its longer runtime. For each switch architecture, we design a tailored greedy algorithm to use when a quick fit suffices. Further, by analyzing the ILP for the "tightest" constraints, we find we can improve the

greedy heuristics. Finally, a sensitivity analysis shows that the most important chip constraints that limit mapping for our benchmarks are the *degree of parallelism* and *per-stage memory*.

We proceed as follows. §6.2 defines the mapping problem and TDG, §6.3 abstracts FlexPipe and RMT architectures, §6.4 presents our ILP formulation, and §6.5 describes greedy heuristics. §6.6 presents experimental results, and §6.7.3 describes sensitivity analysis to determine critical constraints.

6.2 Problem Statement

Our objective is to solve the table configuration problem in Figure 6.2. We focus on mapping P4 programs to FlexPipe and RMT, while respecting hardware constraints and program control flow. Since the abstract switch model in Figure 6.1 does not model realistic constraints such as concurrency limits, finite table space, and finite processing stages, the compiler needs two more pieces of information. First, the compiler creates a table dependency graph (TDG) from the P4 program to deduce opportunities for concurrency, described below. Second, the compiler must be given the physical constraints of the target switch; we consider constraints for specific chips in §6.3.

6.2.1 Table Dependency Graph

We describe program control flow using an example P4 program called L2L3. Figure 6.3 describes the program by showing three of the four items described in §6.1.1: headers, tables, and control flow. The fourth item, the abstract switch model, is described in §6.3.

Our L2L3 program supports unicast and multicast routing, Layer 2 forwarding and learning, IGMP snooping, and a small access control list (ACL) check. Figure 6.3a is a parse graph declaring three different header fields (Ethernet, IPv4, and VLAN) and the metadata used during processing. The features and control flow of the six logical tables in L2L3 are shown in Figure 6.3b and 6.3c.

Table l has attributes (f_l, e_l, a_l, n_l) that determine how a program should be allocated onto a target switch. A set of *match fields* f_l, from the packet header or metadata, are matched against e_l table entries. For example, the IPv4 unicast routing table in L2L3 matches a 32-bit IPv4 destination address and holds up to 2,000 entries. In practice, table l may have much fewer than e_l entries, but the programmer provides e_l as an upper bound. Tables can have different *match types*: exact, prefix (longest prefix match), or ternary (wildcard). If the match type is ternary or prefix, the set f_l also specifies a bit mask. Based on the match result, the table performs actions on *modified fields* a_l and jumps to one of the tables specified in the set of *next table addresses*, n_l.

Figure 6.3c illustrates how header fields are processed by logical tables in an imperative control flow

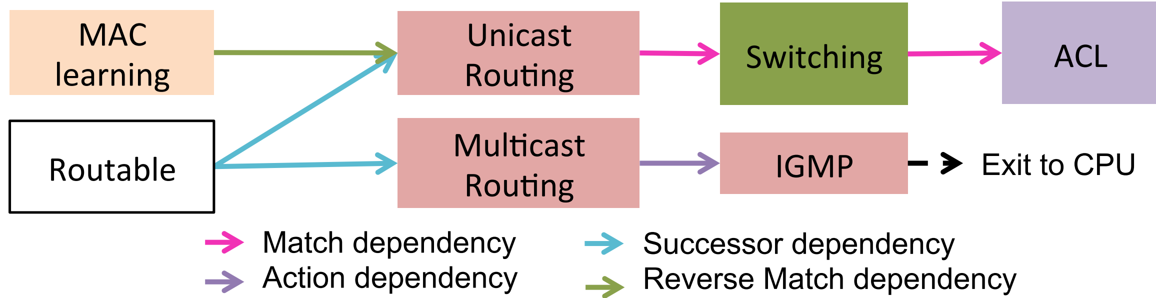


Figure 6.4: Table dependency graph for the L2L3 program.

program. For example, the unicast routing table sets a new destination MAC address and VLAN tag before visiting the switching table, which sets the egress port, and so on. The compiler must ensure that the matched and modified headers in each table correctly implement the control flow program.

We define a table dependency graph (TDG) as a directed acyclic graph (DAG) of the dependencies (edges) between the N logical tables (vertices) in the control flow. Dependencies arise between logical tables that lie on a common path through the control flow, where table outcomes can affect the same packet.

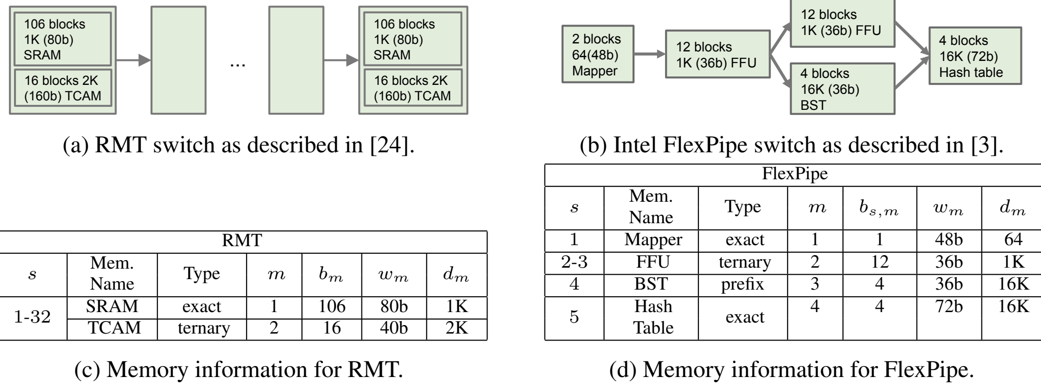
Figure 6.4 shows the TDG for our L2L3 program, which is generated directly from the P4 control flow and table description in Figure 6.3. From the next table addresses it is evident that some tables precede others in an execution pipeline; more precisely, Table A would precede Table B in an execution pipeline if there is a chain of tables $l_1, l_2, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$. If the result of Table A affects the outcome of Table B, we say that Table B has a *dependency* on Table A. In this case, there is an edge from Table A to Table B in the table dependency graph.

Different types of dependencies affect both the arrangement of tables in a pipeline and the pipeline latency.We present the three dependencies described in [24] and introduce a fourth below.

1. Match dependency: Table A modifies a field that a subsequent Table B matches.

2. Action dependency: Tables A and B both change the same field, but the end result should be that of Table B, which modifies later.

3. Successor dependency: Table A's match result determines whether Table B should be executed or not. More formally, there is a chain of tables $l_1, \ldots, l_k$ from A to B, where $l_1 \in n_A$, $l_2 \in n_{l_1}$, etc., and $B \in n_{l_k}$, such that every table $l_i \neq A$ in this chain is followed by B in each possible execution path. Additionally, there is a chain of next table addresses from A that does not go through B. For example, the routable table's outcome determines whether multicast routing and IGMP will be executed. Thus, both have successor dependencies on routable. On the other hand, IGMP does not have a successor dependency on



(a) RMT switch as described in [24].

(b) Intel FlexPipe switch as described in [3].

RMT						
s	Mem. Name	Type	m	b_m	w_m	d_m
1-32	SRAM	exact	1	106	80b	1K
	TCAM	ternary	2	16	40b	2K

(c) Memory information for RMT.

FlexPipe						
s	Mem. Name	Type	m	$b_{s,m}$	w_m	d_m
1	Mapper	exact	1	1	48b	64
2-3	FFU	ternary	2	12	36b	1K
4	BST	prefix	3	4	36b	16K
5	Hash Table	exact	4	4	72b	16K

(d) Memory information for FlexPipe.

Figure 6.5: Switch configurations for RMT and FlexPipe. The tuple (s, m) refers to memory type m ($m \in \{1, ..., K\}$) in the stage indexed by $s \in \{1, ..., M\}$. Each (s, m) has attributes $(b_{s,m}, w_m, d_m)$, where $b_{s,m}$ is the number of blocks of the m-th memory type, and each of these blocks can match d_m words (the "depth" of each block) of maximum width w_m bits.

multicast routing, or vice versa.

4. Reverse match dependency: Table A matches on a field that Table B modifies, and Table A must finish matching before Table B changes the field. This often occurs, as in our example, where source MAC learning is an item that occurs early on, but the later Unicast table modifies the source MAC for packet exit.

Note that these dependencies roughly map to control and data dependencies in traditional compiler literature [16], where a match on a packet header field (or metadata) corresponds to a read and an action that updates a packet header (or metadata) corresponds to a write (Table 6.1).

Switch compiler	Traditional compiler
Match dependency	Read-After-Write
Action dependency	Write-After-Write
Successor dependency	Control dependence
Reverse-match dependency	Write-After-Read

Table 6.1: Mapping switch compiler dependencies to traditional compiler dependencies.

While the TDG is strictly a multigraph, as there can be multiple dependencies between nodes, the mapping problem only depends on the strictest dependency that affects pipeline layout; the other dependencies can be removed to leave a graph. In summary, a TDG is a DAG of N logical tables (vertices) and dependencies (edges), where table $l \in \{1, \ldots, N\}$ has match fields, maximum match entries, modified fields, and next table addresses, denoted by f_l, e_l, a_l, and n_l, respectively.

6.3 Target Switches

The two backends we use—RMT [24] and Intel's FlexPipe [3]—represent real high-performance, programmable switch ASICs. Both conform to our abstract forwarding model (Figure 6.1) by implementing a pipeline of match+action stages and can run the L2L3 program in Figure 6.3. While both switches have different constraints, we can define hardware abstractions common to both chips: a pipeline DAG, memory types, and assignment overhead. We describe these abstractions and switch-specific features, and highlight how our compiler represents each chip's constraints.

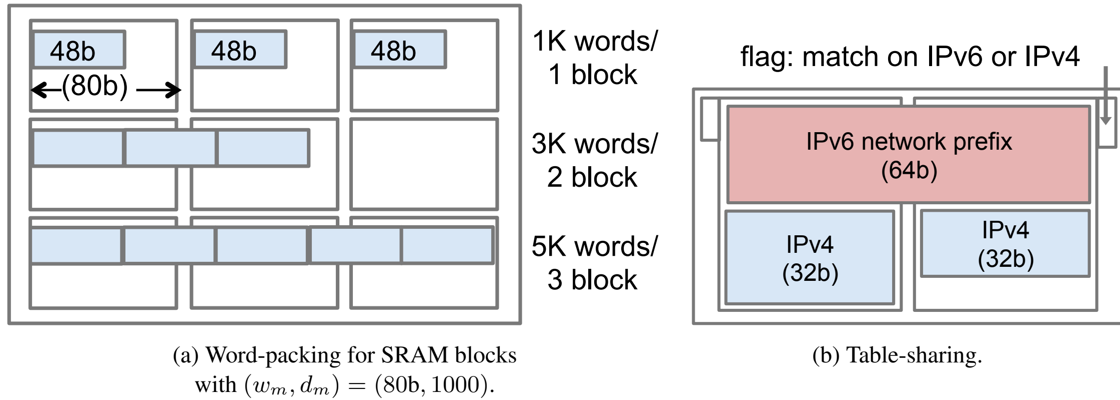
Pipeline Concurrency: We model the physical pipeline of each switch using a DAG of *stages* as shown in Figures 6.5a and 6.5b; a path from the i-th stage to the j-th stage implies that stage i starts execution before stage j. In the FlexPipe model (Figure 6.5b), the second Frame Forwarding Unit (FFU) stage and the Binary Search Tree (BST) stage can execute in parallel because there is no path between them.

Memory types: Switch designers decide in advance the allocation of different *memory blocks* based on programs they anticipate supporting. We abstract each memory block as having a *memory type* that supports various logical *match types* (§6.2.1). For example, in RMT, the TCAM allows ternary match type tables, while SRAM supports exact match only; in FlexPipe, FFU, hash tables, and BST memory types support ternary, exact, and prefix match, respectively.

Memory information for RMT and FlexPipe are in Tables 6.5c and 6.5d. We annotate the DAG to show the number, type and size of the memory blocks in each stage.

Assignment overhead: A table may execute actions or record statistics based on match results; these actions and statistics are also stored in the stage they are referenced. The number of blocks for action and statistics memory, collectively referred to as *assignment overhead*, is linearly dependent on the amount *match memory* a table has in a stage. In RMT, both TCAM and SRAM match memory store their overhead memory in SRAM; we ignore action and statistics memory in FlexPipe.

Combining entries: RMT allows a field to efficiently match against multiple words in the same memory block at a time, a feature we call *word-packing*. Different *packing formats* allow match entries to be efficiently stored in memory; for example, a packing format of 3 creates a *packing unit* that strings together two memory blocks and allows a 48b MAC address field to match against three MAC entries simultaneously in a 144b word (Figure 6.6a). FlexPipe only supports stringing together the minimum number of blocks required to match against one word, but does allow *table-sharing* in which multiple logical tables share the same block of SRAM or BST memory, provided the two tables are not on the same execution path. Table-sharing is shown in Figure 6.6b: since routing tables make decisions on either IPv4 or IPv6 prefixes, both sets of prefixes can share memory.



(a) Word-packing for SRAM blocks with $(w_m, d_m) = (80\text{b}, 1000)$.

(b) Table-sharing.

Figure 6.6: Block layout features in different switches.

Per-stage resources: RMT uses three crossbars per stage to connect subsets of the packet header vector to the match and action logic. Matches in SRAM and TCAM, and actions all require crossbars composed of eight 80b-wide subunits for a total of 640 bits. A stage can match on at most 8 tables and modify at most 8 fields. There appears to be no analogous constraints for FlexPipe.

Latency: Generally, processing will begin in each pipeline stage as soon as data is ready, allowing for overlapping execution. However, logical dependencies restrict the overlap (Figure 6.7). In RMT, a match dependency means no overlap is possible, and the delay between two stages will be the latency of match and action in a stage: 12 cycles. Action dependent stages can have their match phases overlap, and so the minimum delay is 3 cycles between the stages. Successor and reverse-match dependencies can share stages, provided that tables can be run speculatively [24]. Note that even if there are no dependencies there is a one cycle delay between successive stages.

While RMT's architecture requires that match or action dependent tables be in strictly separate stages, FlexPipe's architecture resolves action dependencies at the end of each stage, and thus only match dependencies require separate stages. In summary, the compiler models specific switch designs abstractly using a DAG, multiple memory blocks per stage, constraints on packing, per-stage resources and latency characteristics. While we have described how to model RMT and FlexPipe, newer switches can be described using the same model if they use some form of physical match+action pipeline.

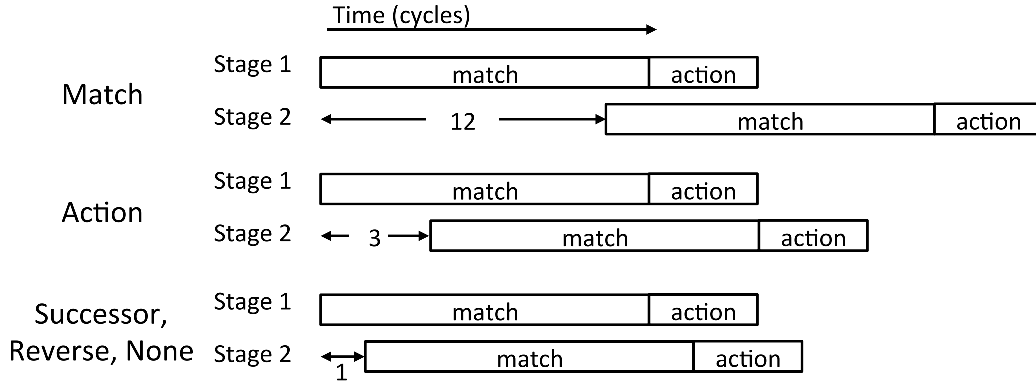


Figure 6.7: Dependency types and latency delays in RMT. In this figure, Table B in Stage 2 depends on Table A in Stage 1.

6.4 Integer Linear Programming

To build a compiler, we must map programs (parse graphs, table declarations, and control flow) to target switches (modeled by a DAG of stages with per-stage resources) while maximizing concurrency and respecting all switch constraints. Because constraints are integer-valued (table sizes, crossbar widths, header vectors), it is natural to use integer linear programming (ILP). If all constraints are linear constraints on integer variables and we specify an objective function (e.g., "use the least number of stages" or "minimize latency"), then fast ILP solvers (like CPLEX [47]) can find an optimal mapping.

We now explain how to encode switch and program constraints and specify objective functions. We divide the ILP-based compiler into a switch-specific preprocessor (for switch-specific resource calculation) and a switch-dependent compiler. We start with switch-independent common constraints.

6.4.1 Common Constraints

The following constraints are common to both switches:

Assignment Constraint: All logical tables must be assigned somewhere in the pipeline. For example, if a table l has $e_l = 5000$ entries, the total number of entries assigned to that logical table, or $W_{s,l,m}$ over all memory types m and stages s, should be at least 5000. Hence, we require:

$$\forall l : \sum_{s,m} W_{s,l,m} \geq e_l. \tag{6.1}$$

Capacity Constraint: For each memory type m, the aggregate memory assignment of table l to stage s,

$U_{s,l,m}$, must not exceed the physical capacity of that stage, $b_{s,m}$:

$$\forall s, m : \sum_{l} U_{s,l,m} \leq b_{s,m}. \tag{6.2}$$

We define the assignment overhead as $\lambda_{m,l}$, which denotes the necessary number of action or statistics blocks required for assigning one match block of table l in memory type m. Thus the aggregate memory assignment is the sum of match memory blocks $\mu_{s,l,m}$ and assignment overhead blocks:

$$U_{s,l,m} = \mu_{s,l,m}\big(1 + \lambda_{l,m}\big).$$

Dependency Constraint: The solution must respect dependencies between logical tables. We use boolean variable $D_{A,B}$ to indicate whether Table B depends on Table A and the start- and end-stage numbers of any table l are denoted by S_l and E_l, respectively. If table B depends on Table A's results, then the first stage of Table B's entries, S_B, must occur after the match results of Table A are known, which is at the earliest E_A (tables are allowed to span multiple pipeline stages):

$$\forall D_{A,B} > 0 : \quad E_A \leq S_B. \tag{6.3}$$

If A must completely finish executing before B begins (e.g., match dependencies), then the inequality in Equation 6.3 becomes strict.

6.4.2 Objective Functions

A key advantage of ILP is that it can find an optimal solution for an objective function. We focus our attention on three objective functions.

Pipeline stages: To minimize the number of pipeline stages a program uses, σ, we ask ILP to minimize:

$$\min \ \sigma, \tag{6.4}$$

where for all stages s:

$$\text{If} \sum_{l,m} U_{s,l,m} > 0 : \qquad \sigma \geq s.$$

Latency: We can alternatively pick an objective function to minimize the total pipeline latency, which is more involved. Consider RMT, in which match and action dependencies both affect when a pipeline stage can start (whereas successor and reverse-match dependencies do not affect when a stage starts). If a table in

stage s has a match or action dependency on a table in stage s', then s' cannot start until 12 or three clock cycles, respectively, after s. Building on how we expressed dependencies in Equation 6.3, we assign stage s a start time, t_s, where t_s is strictly increasing with s. Now consider two tables A and B, and assume Table B has a match dependency (i.e., 12-cycle wait) on Table A. E_A is the last stage A resides in, and S_B is the first stage B resides in. We constrain S_B as follows:

$$t_{E_A} + 12 \leq t_{S_B}.$$

We write the same constraints for all pairs of tables with action dependencies (three-cycle wait). Then we minimize the start time of the last stage, stage M:

$$\min \ t_M. \tag{6.5}$$

Power: Our third objective function minimizes power consumption by minimizing the number of active memory blocks, and where possible, uses SRAM instead of TCAM. The objective function is therefore as follows:

$$\min \ \sum_m g_m \Bigg(\sum_{s,l} U_{s,l,m} \Bigg), \tag{6.6}$$

where $g_m(\cdot)$ returns the power consumed for memory type m.

6.4.3 Switch-Specific Constraints

Our ILP model requires switch-specific constraints, and we push as many details as possible to our preprocessor.

RMT: We start with RMT's ability to pack memory words together to create wider matches. Recall from §6.3 that a packing format p packs together p words in a single wide match; $B_{l,m,p}$ specifies the number of memory type m blocks required for p words of table l in a single wide match. While $B_{l,m,p}$ is precomputed by the preprocessor from the widths of the table entries and memory blocks, the ILP solver decides the number of packing units $P_{s,l,m,p}$ for each stage. We can thus find the number of match memory blocks $\mu_{s,l,m}$ and number of assigned entries $W_{s,l,m}$ for each stage:

$$\mu_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p} B_{l,m,p}.$$

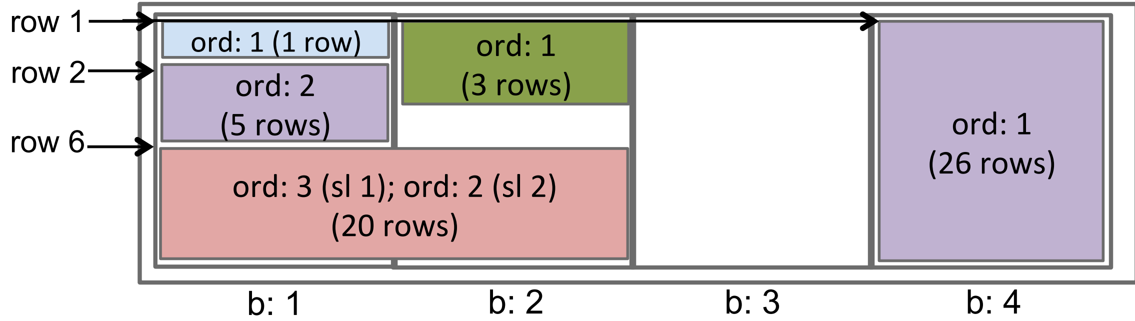


Figure 6.8: FlexPipe table sharing in detail. The pink table occupies the first two memory blocks, but different sets of tables share the first two memory blocks.

$$W_{s,l,m} = \sum_{p=1}^{p_{max}} P_{s,l,m,p}(p \cdot d_m),$$

where $p \cdot d_m$ is the number of table l's entries that can fit in a single packing unit of format p in memory type m.

Per-Stage Resource Constraints: We must incorporate RMT-specific constraints such as the input action and match crossbars. The preprocessor can compute the number of input and action subunits needed for a logical table as a function of the width of the fields on which it matches or modifies, respectively.

FlexPipe: FlexPipe can share memory blocks at a finer granularity than RMT, and so we need to preprocess the constraints differently for FlexPipe.

To support configurations as in Figure 6.8, we need to know which rows within a set of blocks are assigned to each logical table. This is because multiple tables can share a block, and different blocks associated with the same table can have very different arrangements of tables, such as blocks 1 and 2 assigned to the pink table.

Note that this issue does not arise in RMT; all memory blocks that contain a logical table will be uniform, and a solution can be rearranged to group together all memory blocks of a particular table assignment in a stage. We thus index the memory blocks $b \in 1, \ldots, b_{s,m}$, where $b_{s,m}$ is the maximum number of blocks of type m in stage s.

The solver decides how many logical table entries to assign to each block in each stage. For the remainder of this discussion, we differentiate between logical table entries and physical memory block entries by referring to the latter as rows, where row 1 and row e_m are the first and last rows, respectively, of a block of memory type m.

For table l assigned to start in the bth block of memory type m, we use the variable $\hat{r}_{l,m,b}$ to denote the

starting row, and the variable $r_{l,m,b}$ to denote the number of consecutive rows that follow.[1] To make sure rows do not overlap within a block, we constrain their placement by introducing the notion of *order*. Order is defined by the variable $\theta \in \{1, \ldots, \theta_{max}\}$, where θ_{max} is the maximum number of logical tables that can share a given memory block. In Figure 6.8, the light-blue assignment has order $\theta = 1$, because it has the earliest row assignment. We define two additional variables, $\hat{\rho}_{m,b,\theta}$ and $\rho_{m,b,\theta}$, the start row and the number of rows of table with order θ, and we prevent overlaps by constraining the assignment as follows.

If θ-th order is assigned:

$$\hat{\rho}_{m,b,\theta-1} + \rho_{m,b,\theta} \leq \hat{\rho}_{m,b,\theta}$$

To calculate the assignment constraint (Equation 6.1), the total number of words assigned to table l in stage s is:

$$W_{s,l,m} = \sum_{b=1}^{b_{s,m}} r_{l,m,b}.$$

where $r_{l,m,b}$ denotes the number of rows assigned for table l in all orders θ in block b of memory type m.

While the capacity constraint in Equation 6.2 is per stage, in FlexPipe we must also implement a capacity constraint per block. We restrict the number of rows we can assign to a block by checking the last row of the last order, θ_{max}:

$$\hat{\rho}_{m,b,\theta_{max}} + \rho_{m,b,\theta_{max}} \leq d_m.$$

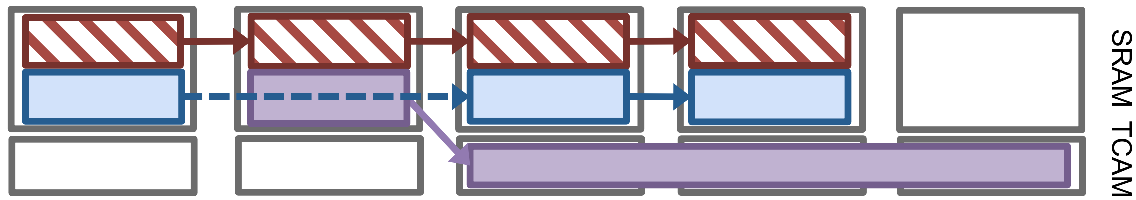
Dependency Constraints: Fortunately, the dependency analysis is similar to RMT in §6.4.3, with the additional feature that only match dependencies require a strict inequality; action, successor, and reverse-match dependencies can be resolved in the same stage.

Objectives: Since FlexPipe has a short pipeline, we minimize the total number of blocks used across all stages.

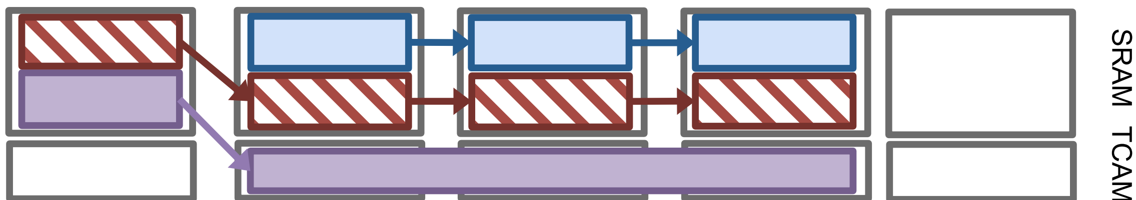
6.5 Greedy Heuristics

Since a full-blown optimal ILP algorithm takes a long time to run, we also explored four simpler greedy heuristics for our compiler: First Fit by Level (FFL), First Fit Decreasing (FFD), First Fit by Level and Size (FFLS), and Most Constrained First (MCF). All four greedy heuristics work as follows: First, sort the logical tables according to a metric. For each logical table in sorted order, pick the first set of memory blocks in

[1] Note that if a second table, l', has entries in an adjacent block b', but the entries are wide and overflow into block b, $\hat{r}_{l',m,b} = 0$ because the starting row for l' was not assigned in this block; similarly, $r_{l,m,b}$ is irrelevant.



(a) FFL: Tables are placed in order of level. This configuration takes five stages and wastes all of the TCAMs in the second stage.



(b) FFLS: The first purple table with a large ternary table following it is placed first, even though the blue table has more match dependencies following it. This configuration uses only four stages.

Figure 6.9: Multiple-metric heuristics. A toy RMT example where a table with a single, dependent large ternary table must be placed before a table with a longer dependency chain.

the first pipeline stage the table can fit in without violating any capacity constraints, dependencies, or switch-specific resources. If it cannot fit, the heuristic finds the next available memory blocks, in the same stage or a subsequent stage. Like ILP, we leave switch-specific resource calculation like crossbar units and packing formats to a preprocessor. A heuristic terminates when all tables have been assigned or when it runs out of resources.

6.5.1 Ordering Tables

The quality of the mapping depends heavily on the sort order. Three sorting metrics seem to matter most in our experiments, described in more detail below.

Dependency: Tables that come early in a *long dependency chain* should be placed first because we need at least as many stages left as there are match or action dependencies. We thus define the *level* of a table to be the number of match or action dependencies in the longest path of the TDG from the table to the end.

Word width: In RMT, tables with wide match or action words use up a large fraction of the *fixed resources* (action/input crossbars) and should be prioritized; they may not have room if smaller tables are assigned first. In FlexPipe, tables with larger match word width should be assigned first because there is less memory per

stage.

Memory types: While blocks with memory types like TCAM, BST, and FFU can also fit exact-match tables, exact memory like SRAM is generally more abundant than flexible memory due to switch costs. Thus, in FlexPipe, heuristics should prioritize the assignment of more *restrictive tables*, or tables that can only go in ternary or prefix memories; otherwise, assigning exact match tables to flexible memories first can quickly lead to memory shortage. In RMT, restrictive tables go into TCAM, available in every stage. But large TCAM tables in a long dependency chain push back tables that follow them. So we should prioritize tables that imply high TCAM usage in their dependency chain.

6.5.2 Single-Metric Heuristics

Two of our greedy heuristics sort on a single metric: FFL is inspired by bin packing with dependencies [40] and sorts on table level, where tables at the head of long dependency chains are placed earlier. FFD is based on the First Fit Decreasing Heuristic for bin packing [40]. In our case, we prioritize tables that have wider action or match words and consequently use more action or input crossbar subunits. This heuristic should work well when fixed switch resources—not program table sizes—are the limiting factor.

6.5.3 Multi-Metric Heuristics

Some programs fit well if we consider only one metric: if there are plenty of resources at each stage, we need only worry about long dependency chains. Our next two heuristics sort on multiple metrics. Sometimes being greedy on just one metric might not work, as shown in Figure 6.9: here, our first multi-metric heuristic FFLS incorporates dependencies and TCAM usage, where tables with larger TCAM tables in their dependency chains are assigned earlier.

Our other multi-metric heuristic, MCF, is motivated by FlexPipe's smaller pipeline with more varied memory types. We pick the "most constrained" table first: a table restricted to a particular memory type and with a high level should have higher priority. Ties are broken by placing the table with wider match words first. FFL and FFD that ignore the memory type do not work well for FlexPipe, which does not have uniform memory layout per stage like RMT; for example, ternary match tables can only go in stages 2 or 3 in FlexPipe.
Variations: Each of the basic four heuristics has two variants: by default, an exact match table spills into TCAMs if it runs out of SRAMs in a stage. Our first variant prevents the spillage to preserve the TCAMs for ternary tables. Second, by default, when we reserve space for a TCAM table, we do not reserve space in SRAM to hold the associated action data, which means we may run out of SRAM and not be able to use the TCAM. Our second variant sets aside SRAM for action memory from yet-to-be allocated ternary tables; in

our experiments we fix the amount to be 16 SRAMs.

There can be cases where the best combination of metrics is unclear, as in Figure 6.10 for RMT and Figure 6.11 for FlexPipe. Our experiments in §6.6 seek the right combination of metrics for an efficient greedy compiler.

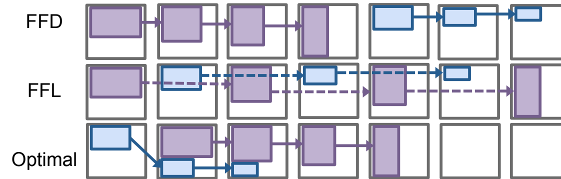
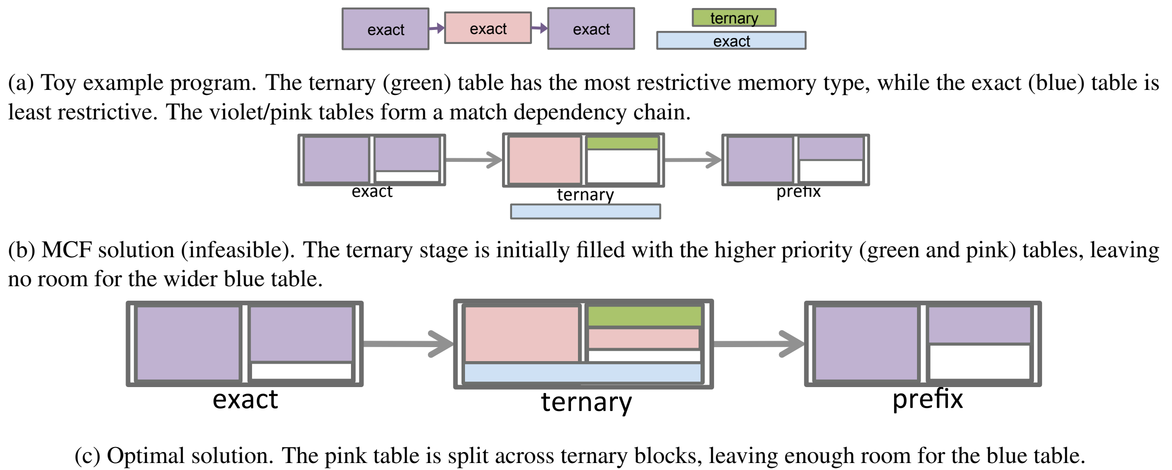


Figure 6.10: Greedy performing much worse than ILP (RMT). In this toy example, the blue and purple tables form separate match dependency chains. The initial (blue) table in the optimal mapping is narrower and has a lower level than the purple table, counterintuitive to both FFD and FFL metrics.



(a) Toy example program. The ternary (green) table has the most restrictive memory type, while the exact (blue) table is least restrictive. The violet/pink tables form a match dependency chain.

(b) MCF solution (infeasible). The ternary stage is initially filled with the higher priority (green and pink) tables, leaving no room for the wider blue table.

(c) Optimal solution. The pink table is split across ternary blocks, leaving enough room for the blue table.

Figure 6.11: Greedy performing much worse than ILP (FlexPipe.)

6.6 Experiments

We tested our algorithms by compiling the four benchmark programs listed in Table 6.2 for the RMT and FlexPipe switches.

Name	Switch	N	Dependencies		
			Match	Action	Other
L2L3 -Complex	RMT	24	23	2	10
L2L3 -Simple	RMT	16	4	0	15
	FlexPipe	13	12	0	4
L2L3 -Mtag	RMT	19	6	1	16
	FlexPipe	11	9	1	3
L3DC	RMT	13	7	3	1

Table 6.2: Logical program benchmarks for RMT and Flexpipe.
N is the number of tables.

The benchmarks are in Table 6.2: L2L3-Simple, a simple L2/L3 program with large tables; L2L3-Mtag, which is L2L3-Simple plus support for the mTag toy example described in [23]; L2L3-Complex, a complex L2/L3 program for an enterprise DC aggregation router with large host routing tables, and L3DC, which is a program for Layer 3 switching in a smaller enterprise DC switch. Differently sized, smaller variations of the L2L3-Simple and L2L3-Mtag programs are used for FlexPipe. L2L3-Complex and L3DC cannot run on Flexpipe because the longest dependency chain for each program needs 9 and 6 stages respectively, exceeding FlexPipe's 5-stage pipeline.

ILP: We used three ILP objective functions for RMT: number of stages (*ILP-Stages*), pipeline latency (*ILP-Latency*), and power consumption (*ILP-Power*). For FlexPipe, since we struggle to fit the program, we simply looked for a feasible solution that fit the switch. All of our ILP experiments were run using IBM's ILP solver, CPLEX. Note that CPLEX has a *gap tolerance* parameter, which sets the acceptable gap between the best integer objective and the current solution's objective. For ILP-Stage, we required zero-gap tolerance. For ILP-Latency and ILP-Power, we set the gap tolerance to be within 70% and 5%, respectively, of the best integer value; we found that lower gaps highly increased runtime with little improvement in objective value.

Greedy heuristics: For each RMT program, we ran all four greedy heuristics (FFD, FFL, FFLS, MCF). We also ran the variant that set aside 16 SRAM blocks for ternary action memory (labeled as FFD-16, etc.) and a combination of the two variants to also avoid spilling exact match tables into TCAM (labeled as FFD-exact16, etc.). For each FlexPipe program, we simply ran the greedy heuristic MCF. The other three heuristics do not combine enough metrics to fit either of our FlexPipe benchmarks.

All of our experiments were run on an Amazon AWS EC2 c3.4xlarge instance with 16 processor cores and 30 GB of memory. For FlexPipe, we generated 20 and 10 versions of the L2L3-Simple and L2L3-Mtag programs, respectively, with varying table sizes and checked how many greedy and ILP mappings fit the switch (Table 6.3). For RMT, we compiled every program 10 times for each of the greedy heuristics

and the ILP objective functions and report the medians of the number of stages used, pipeline latency, and power consumed for each algorithm. We show in detail L2L3-Complex results in Table 6.4. To facilitate presentation, for all other programs we display results for ILP and the "-exact16" greedy variant only (Table 6.5), since this variant generally tended to have better stage and power usage than other greedy heuristics.

Solver	L2L3-Simple	L2L3-Mtag
	% solved	% solved
MCF	75	60
ILP	75	80

Table 6.3: Benchmark results for 5-stage FlexPipe.

Solver	L2L3-Complex			
	St.	Lat.	Pwr	RT.
FFD	22	135	4.98	0.25
FFD-16	21	135	5.51	0.27
FFD-exact16	21	135	4.62	0.27
FFL	19	131	5.61	0.25
FFL-16	19	131	6.09	0.27
FFL-exact16	17	132	4.61	0.24
FFLS	19	130	5.66	0.33
FFLS-16	19	130	6.42	0.35
FFLS-exact16	17	131	4.66	0.32
MCF	20	132	4.67	0.26
MCF-16	19	132	6.43	0.27
MCF-exact16	18	132	4.67	0.25
ILP-Latency	32	104	7.78	233.84
ILP-Stages	16	131	6.66	12.13
ILP-Power	32	131	4.44	147.10

Table 6.4: Benchmark results for RMT for L2L3-Complex. All greedy heuristics and variants are shown (St: number of stages occupied, Lat.: Latency (cycles), Pwr.: Power (watts), RT.: Time to run solver (s).

Solver	L2L3-simple				L2L3-Mtag				L3DC			
	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.	St.	Lat.	Pwr	RT.
FFD-exact16	21	64	7.54	0.18	22	75	7.65	0.21	7	88	2.34	0.08
FFL-exact16	19	55	7.55	0.19	19	66	7.66	0.21	7	88	2.34	0.08
FFLS-exact16	20	64	7.88	0.23	21	75	8.10	0.27	7	88	2.34	0.12
MCF-exact16	19	55	7.54	0.18	19	66	7.65	0.21	7	88	2.34	0.09
ILP-Latency	32	51	9.18	2.22	32	53	9.65	3.62	32	62	3.21	23.16
ILP-Stages	19	55	7.52	2.57	19	72	9.62	3.52	7	88	2.46	1.88
ILP-Power	32	62	7.55	2.27	32	71	7.63	2.53	9	86	2.34	1.87

Table 6.5: Benchmark results for 32-stage RMT for L2L3-simple, L2L3-Mtag, and L3DC. See Table 6.4 for units.

6.7 Analysis of Results

We analyze Tables 6.3, 6.4, and 6.5 for major findings. A salient observation is that the MCF heuristic for FlexPipe fits 16 out of the 20 versions of L2L3-Simple. For some programs where the heuristic could not fit, it was difficult to manually analyze the incomplete solution for feasibility. However, ILP can both detect infeasible programs and find a fitting when feasible (assuming match tables are reasonably large[2], for instance, if they occupy at least 5% of a hash table memory block).

Another important observation for reconfigurable chips where one can optimize for different objectives is that the best greedy heuristic can perform 25% worse on the objectives than ILP; for example, the optimal 104-cycle latency for ILP in the second column of Table 6.4 is far better than the best latency of 130 cycles by FFLS. A detailed comparison follows.

6.7.1 ILP vs Greedy

The following observations can be made after closely comparing ILP and greedy solutions in Figure 6.12.

1. Global versus local optimization: For the L2L3-Complex use case (Figure 6.12a), even the best greedy heuristic FFL-exact16 takes 17 stages, while ILP takes only 16 stages. Figures 6.12c and 6.12d show FFL-16 and ILP solutions, respectively. ILP breaks up tables over stages to pack them more efficiently, whereas greedy tries to assign as many words as possible in each stage per table, eventually wasting some SRAMs in some stages and using up more stages overall.

In switch chips with shorter pipelines than RMT's, this could be the difference between fitting and not fitting. If all features in a program are necessary, then infeasibility is not an option. Unlike register allocation, there is no option to "spill to memory"; on the other hand, the longer runtime for ILP may be acceptable when adding a new router feature. Therefore, it seems very likely that programmers will resort to optimal algorithms, such as ILP, when they really need to squeeze a program in.

2. Greedy poor for pipeline latency: Our greedy heuristics minimize the stages required to fit the program and are good at minimizing power—the best greedy is only 4% worse than optimal (for L2L3-Complex, FFL-exact16 consumes 4.61W, versus ILP's 4.44W); technically, this is true only because the "-exact" variant avoids using power-hungry TCAMs. But greedy heuristics are much worse for pipeline latency; minimizing latency with greedy algorithms will require improved heuristics.

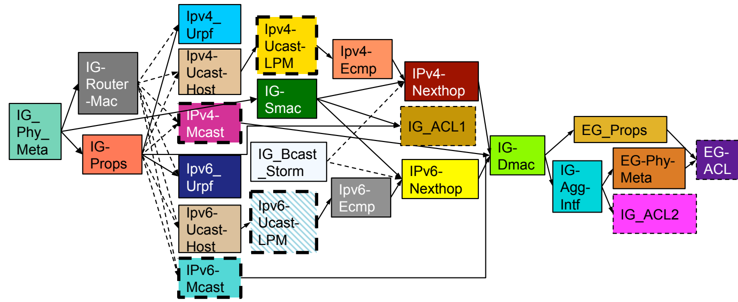
[1]The minimum table size constraint helps us scale the ILP to handle FlexPipe, where a table can be assigned any number of rows in each memory block. Since the size of logical tables and memory blocks are at least on the order of hundreds, it seems reasonable to impose a minimum match table size of at least a hundred in these memory blocks.

6.7.2 Comparing Greedy Heuristics

1. Prioritize dependencies, not table sizes: In L2L3-Mtag, both FFL and FFLS assign the exact-match tables in the first stage, but they differ in how they assign ternary tables. FFLS prioritizes the larger ACL table over the IPv6-Prefix and IPv6-Fwd tables, which are early tables in the long (red) dependency chain in Figure 6.13a. As a result, the IPv6-Prefix and IPv6-Fwd tables cannot start until stages 16 and 17, and FFLS ends up using two more stages than FFL. Although FFLS prioritizes large TCAM tables and avoids the problem discussed in Figure 6.9, it is not sophisticated enough to recognize other opportunities for sharing stages between dependency chains.

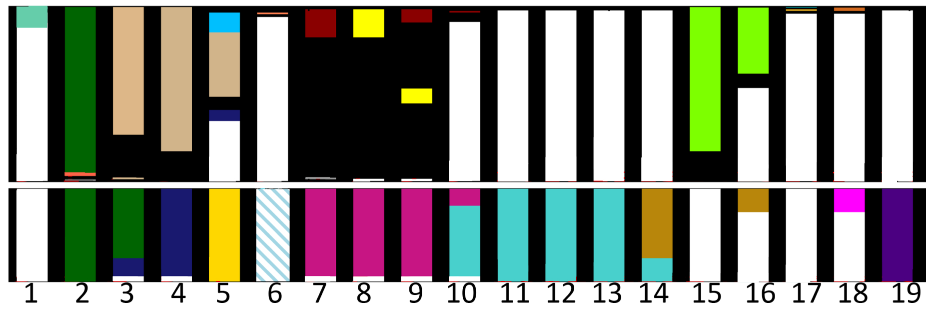
2. Sorting metrics matter: FFD results show that incorrect sorting order can be expensive (22 stages versus the optimal 16 for L2L3-Complex). We predict that FFD will only be useful for use cases with many wide logical tables or more limited per-stage switch resources, neither of which was a limiting factor in our experiments.

3. Set aside SRAM for TCAM actions: The "-16" variation of our greedy heuristics estimates the number of SRAMs needed for ternary tables (for their action memory) in each stage and blocks them off when initially assigning SRAMs to exact match tables. Our experiments show that this local optimization usually avoids having to move a ternary table to a new stage because it does not have enough SRAMs for action memory.

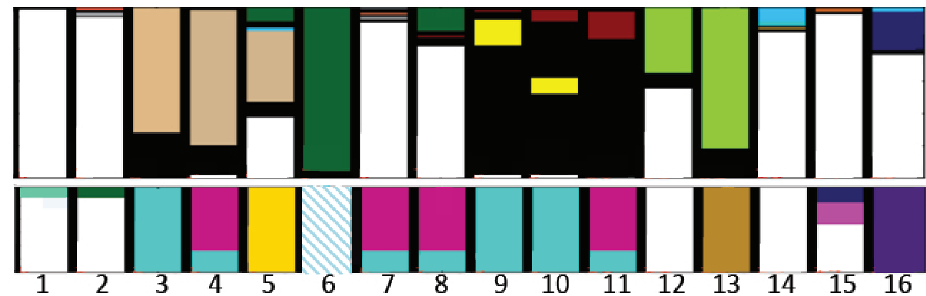


(a) TDG for L2L3-Complex. Solid and dashed arrows indicate match/action and successor dependencies, respectively, while solid and dashed blocks are exact and ternary tables, respectively.

(b) Number of TCAMs required to fit the wide match words of ternary IP routing tables in L2L3Complex with packing factor 1.

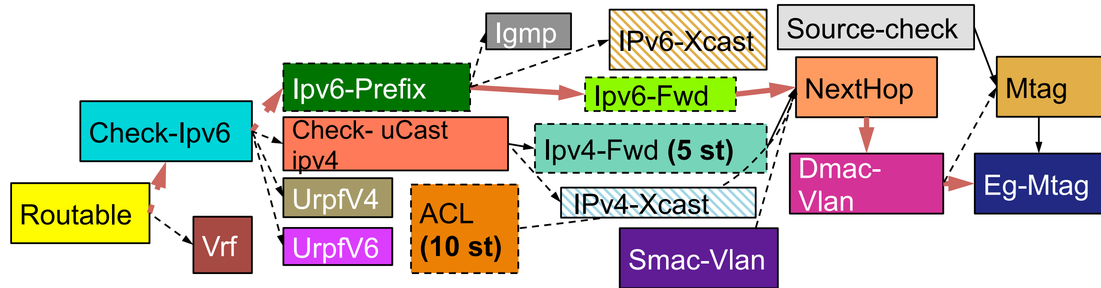


(c) FFL-16 solution (19 stages). FFL-16 uses five 3-wide packing units to assign IPv4-Mcast in stages 7 to 9, leaving one TCAM per stage that cannot be used by any other ternary table. Overall, FFL-16 wastes a total of six TCAMs between stages 3 and 10.

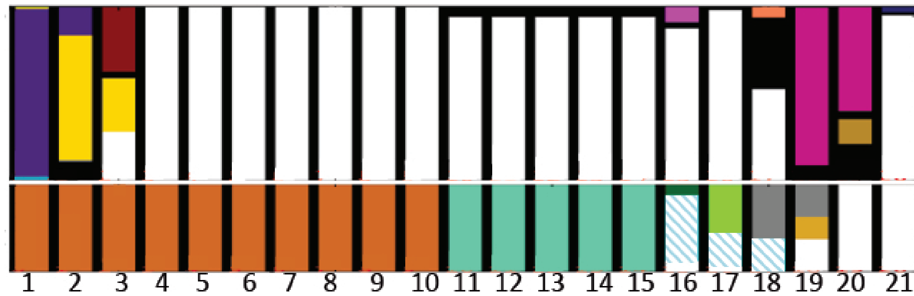


(d) ILP solution (16 stages). ILP utilizes all TCAMs in stages 4, 7, 8, and 11 by sharing the TCAMs between IPv4-Mcast (four 3-wide packing units) and IPv6-Mcast (one 4-wide packing unit).

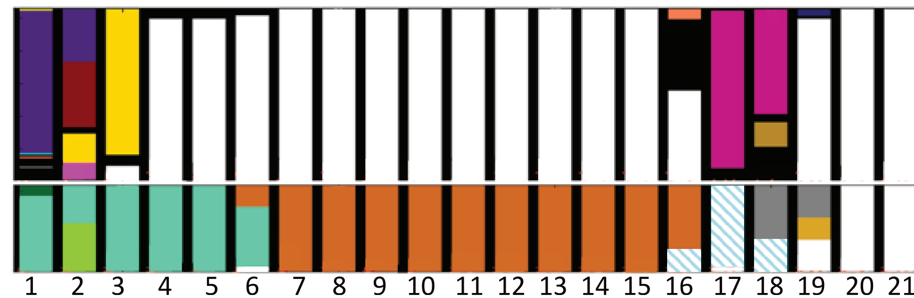
Figure 6.12: FFL-16 and ILP solutions for L2L3-Complex. In assigning packing units to the ternary IP routing tables, FFL-16 locally maximizes the number of words per stage, whereas ILP optimizes over a set of stages. Each stage has 106 SRAMs (top row) and 16 TCAMs (bottom row) and is colored according to the amount of match memory assigned to each logical table in the program TDG; all action memory is colored in black.



(a) TDG for L2L3-Mtag. Red arrows mark a long dependency chain.



(b) FFLS solution (21 stages). FFLS places the ACL and IPv4-Fwd tables in stages 1 through 15, leaving no room for the smaller IPv6-Prefix and IPv6-Fwd tables until stages 16 and 17.



(c) FFL solution (19 stages). FFL prioritizes the IPv6-Prefix and IPv6-Fwd tables and fits them in stages 1 and 2, allowing earlier assignment of Nexthop and other dependent tables. As a result, FFL needs two stages fewer than FFLS.

Figure 6.13: FFLS and FFL solutions for L2L3-Mtag. FFL prioritizes dependencies over table sizes and uses two fewer stages than FFLS.

6.7.3 Sensitivity Experiments

In this section, we analyze ILP solutions by ignoring and relaxing various constraints in order to improve the running time of ILP and the optimality of greedy heuristics. We run these ILP experiments for our most complicated use case (L2L3-Complex) on the RMT chip, for two different objectives: minimum stages and minimum pipeline latency. For reference, the original ILP yields a solution that uses 16 stages in 12.13 s, and a solution that uses 104 cycles in 233.84 s, respectively, for the two objectives.

To improve ILP runtime, we measure how long the ILP solver takes while ignoring or relaxing each

constraint, which is a proxy for how hard it is to fit programs in the switch. This helps us identify constraints that are currently “bottlenecks" in runtime for the ILP solver and also helps us understand how future switches can be designed to expedite ILP-based compilation.

We identify candidate metrics for greedy heuristics to optimize a given objective by ignoring constraints and identifying which have a significant impact on the quality of the solution. Our experiments also help identify the critical resources needed in the chip for typical programs, so chipmakers can design for better performance.

Sensitivity results for optimality of greedy heuristics: We discovered that the dependency constraint for ILP has the largest impact on the minimum stages objective. If we remove dependencies from the TDGs, we can reduce the number of stages used from 16 to 13 and pipeline latency by two cycles. This explains why greedy heuristics focusing on the dependency metric (i.e., FFL and FFLS) do particularly well. Ignoring other constraints (like resource constraints) makes no difference to the number of stages used or latency. In addition, relaxing various resource constraints showed that some resources affect fitting more than others. For example, doubling the number of TCAM blocks per stage reduced the number of stages needed from 16 to 14. But doubling the number (or width) of crossbars made no difference. This explains why our FFD greedy heuristic (which focuses on non-limiting resources in the RMT switch) performs worse than other algorithms.

Lessons for chipmakers: Our results above indicate that chipmakers can improve turnaround for optimal ILP compilers by carefully selecting memory width. Moreover, if flexible memory is a rare resource, then increasing a non-limiting resource like crossbar complexity will not improve performance.

6.8 Related Work on Compilers

Compiling packet programs to reconfigurable switches differs from compiling to FPGAs [72], other spatial architectures such as PLUG [35], or to NPUs [34]. We focus on *packing* match+action tables into memories in *pipelined stages* while satisfying dependencies. Nowatzki et al. [67] developed an ILP scheduler for a spatial architecture that maps instructions entailed by program blocks to hardware, by allocating computational units for instructions and routing data between units. The corresponding problems for reconfigurable switches—assigning action units and routing data among the packet header, memories, and action units—are less challenging once we have a table placement. Sivaraman et al. [80] propose implementing the action unit in a match-action table as a digital circuit called an “atom” such that non-trivial stateful data-plane algorithms can be compiled to run on a sequence of atoms over multiple stages of a target switch pipeline. Because of the focus on the action unit, this work is complementary to our compiler. NPUs such as the IXP network

processor architecture [10] have multi-threaded packet processing engines that can be pipelined. Approaches like that of Dai et al. [34] map a sequential packet processing application into pipelined stages. However, the processing engines have a large shared memory; thus, NPU compilers do not need to address the problem of packing logical tables into physical memories. Schlesinger et al. [78] develop a new packet processing language called Concurrent NetCore (CNC) and a compiler that maps a CNC program to a target switch architecture like RMT. The table fitting phase of their compiler also has the goal of mapping logical to physical tables and implements a dynamic programming algorithm to minimize the number of physical tables with a formal type-theory based proof of correctness. In contrast, we develop and implement greedy and ILP approaches to target different objectives and use the concept of table-dependencies to justify reordering tables in the pipeline.

6.9 Summary

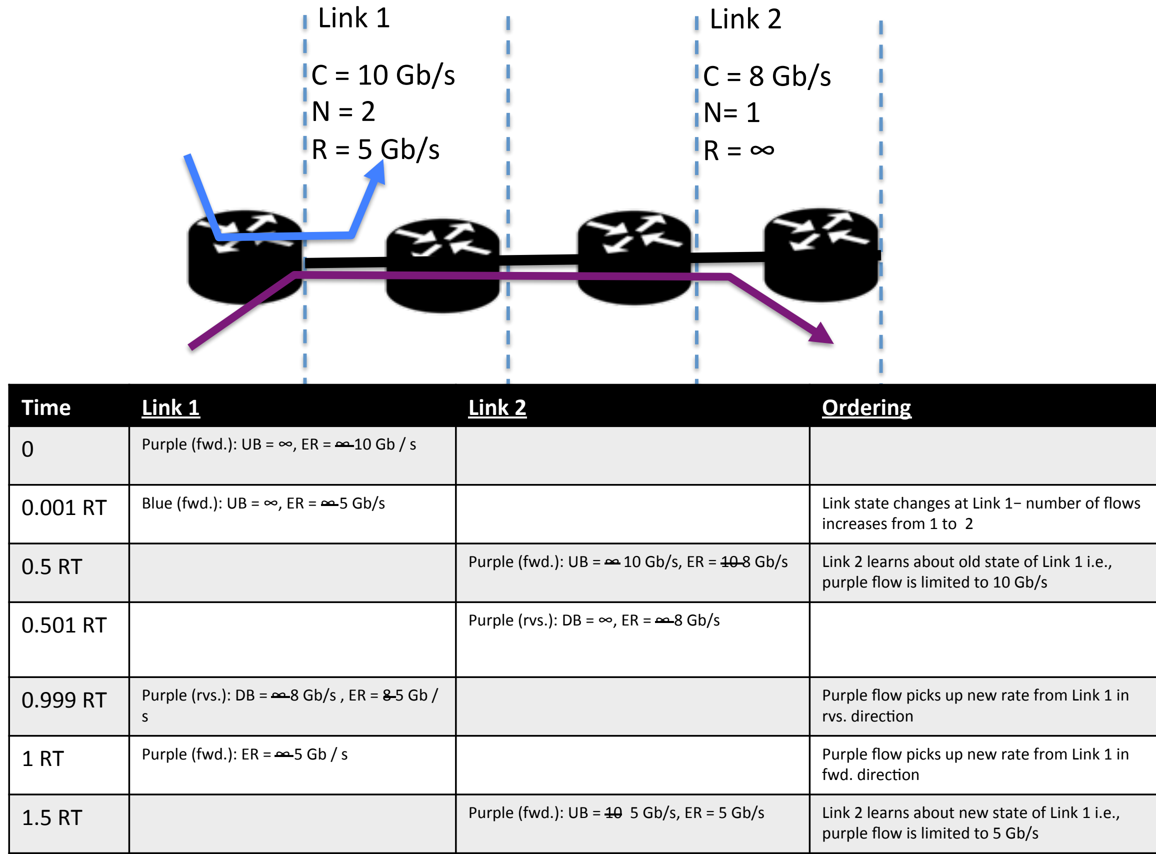
We defined the problem of mapping logical tables in packet processing programs. We evaluated greedy heuristics and ILP approaches for mapping logical tables on realistic benchmarks. While fitting tables is the main criterion, we also computed how well solvers minimize pipeline latency on the long RMT pipeline. We found that for RMT, there are realistic configurations where greedy approaches can fail to fit and need up to 38% more memory resources on the same benchmark. Three situations when ILP outperforms greedy are when there are *multiple conflicting metrics*, *multiple memory types* and *complicated objectives*. We believe future packet programs will get more complicated with more control flows, more different size tables, more dependencies and more complex objectives, arguing for an ILP-based approach. Further, sensitivity analysis of critical ILP constraints provides insight into designing fast tailored greedy approaches for particular targets and programs, marrying compilation speed to optimality.

Appendix A

Supplementary Material for Chapter 2 (*Fair*)

A.1 $Fair$ v/s d-CPG

There are three differences between $Fair$ and the distributed algorithm called d-CPG described by Ros-Giralt et al. (see Chapter IV Figures 1–4 in [74]). First, the "limit rate" calculation in $Fair$ and d-CPG have different implementations albeit the results are the same. Second, the "bottleneck rate" calculation in $Fair$ is slightly different from d-CPG–when a link l computes a bottleneck rate for a flow f, it temporarily assumes that the flow is not limited. Without this change, the bottleneck rate calculation in d-CPG ("ComputeAR()" procedure in Figure 8 of [74]) is undefined when the sum of limit rates is less than the link capacity. Finally, the proof of the CPG algorithm claims that it takes no more than half a round trip for information about a change in the state of one link to propagate to another (Proof of Theorem 4.1 in [74]), whereas we we can easily show a counter-example that suggests it can take up to 1.5 RTTs (see Figure A.1). For the $Fair$ algorithm, we assume that it can take up to 2 RTTs (or $rounds$) for new information to propagate between links rather than half a round trip. Hence, the convergence bound of $Fair$ in Theorem 2.6.1 is four times longer than the convergence bound of d-CPG as stated in Theorem 4.1 in [74].

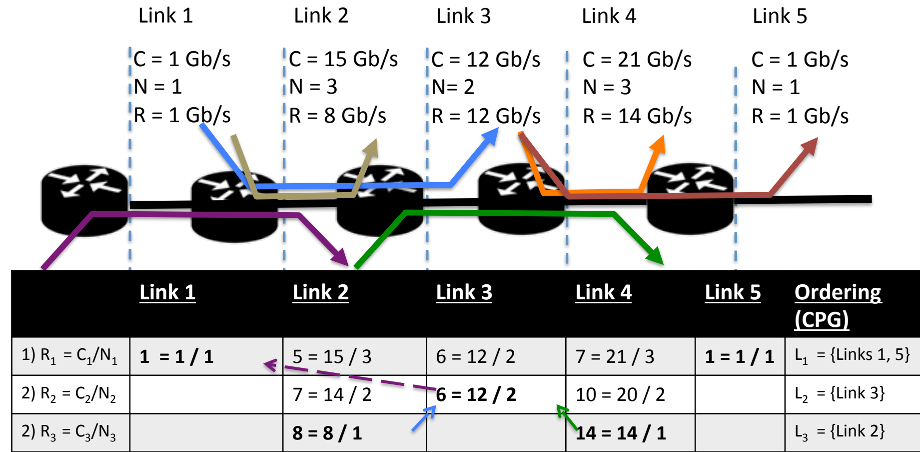


Time	Link 1	Link 2	Ordering
0	Purple (fwd.): UB = ∞, ER = ~~∞~~ 10 Gb / s		
0.001 RT	Blue (fwd.): UB = ∞, ER = ~~∞~~ 5 Gb/s		Link state changes at Link 1– number of flows increases from 1 to 2
0.5 RT		Purple (fwd.): UB = ~~∞~~ 10 Gb/s, ER = ~~10~~ 8 Gb/s	Link 2 learns about old state of Link 1 i.e., purple flow is limited to 10 Gb/s
0.501 RT		Purple (rvs.): DB = ∞, ER = ~~∞~~ 8 Gb/s	
0.999 RT	Purple (rvs.): DB = ~~∞~~ 8 Gb/s , ER = ~~8~~ 5 Gb / s		Purple flow picks up new rate from Link 1 in rvs. direction
1 RT	Purple (fwd.): ER = ~~∞~~ 5 Gb / s		Purple flow picks up new rate from Link 1 in fwd. direction
1.5 RT		Purple (fwd.): UB = ~~10~~ 5 Gb/s, ER = 5 Gb/s	Link 2 learns about new state of Link 1 i.e., purple flow is limited to 5 Gb/s

Figure A.1: An example of a sequence of updates in the CPG algorithm where it takes 1.5 round-trips for new information to propagate from one link to another via the control packet of a shared flow.

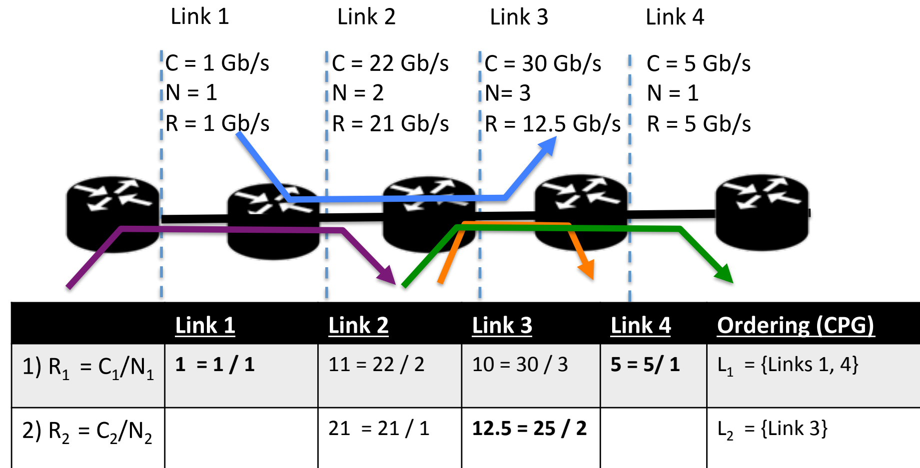
A.2 Dependencies from the CPG Algorithm

The *precedent link relationship* is an example of a dependency, and any path constraint precedence graph is a dependency chain, where we define a dependent link of link l as any link that must converge before link l can converge. We have found that there may be other kinds of dependencies that do not fit the precedent link definition (e.g., a link $j \in L_{n+1}$ may depend on a link removed before round n, that is not in its CPG graph, see Figure A.2; or a link $j \in L_{n+1}$ may depend on a link removed in round n that is not quite an indirect precedent, because the rate of the "Medium" link in round n exceeded the rate of link j in round n, see Figure A.3). However, the CPG algorithm implies that any dependent link of link $j \in L_{n+1}$ must be among links in $L_1, .., L_n$ so that the dependencies, which are a super-set of the precedent link relationships, also form a directed acyclic graph of depth D.



	Link 1	Link 2	Link 3	Link 4	Link 5	Ordering (CPG)
1) $R_1 = C_1/N_1$	**1 = 1 / 1**	5 = 15 / 3	6 = 12 / 2	7 = 21 / 3	**1 = 1 / 1**	L_1 = {Links 1, 5}
2) $R_2 = C_2/N_2$		7 = 14 / 2	**6 = 12 / 2**	10 = 20 / 2		L_2 = {Link 3}
2) $R_3 = C_3/N_3$		**8 = 8 / 1**		**14 = 14 / 1**		L_3 = {Link 2}

Figure A.2: Dependent links that are not precedent links: We show the progress of the CPG algorithm for this setup. A cell in a row is bolded when a link is bottlenecked (i.e., it has the smallest rate among neighbors) in the respective iteration. Link 4 depends on link 5 to calculate the correct rate but link 5 is an ancestor in the precedence graph containing 4. We have drawn solid/ dotted arrows in the table to denote direct/ indirect precedent edges, where the color of the arrow corresponds to the color of the precedent link's bottleneck flow.



	Link 1	Link 2	Link 3	Link 4	Ordering (CPG)
1) $R_1 = C_1/N_1$	**1 = 1 / 1**	11 = 22 / 2	10 = 30 / 3	**5 = 5/ 1**	L_1 = {Links 1, 4}
2) $R_2 = C_2/N_2$		21 = 21 / 1	**12.5 = 25 / 2**		L_2 = {Link 3}

Figure A.3: Dependent links that are not precedent links: We show the progress of the CPG algorithm for this setup. A cell in a row is bolded when a link is bottlenecked (i.e., it has the smallest rate among neighbors) in the respective iteration. Link 3 depends on link 1, because unless link 1 converges, Link 2's rate cannot exceed link 3's max-min rate of 12.5 Gb/s. However, link 1 is not an *indirect precedent link* of link 3 as defined in [75]. Notice that in round 1, link 3 shares a flow with link 2, which has $R_1(3) < R_1(2) \leq \mathrm{R}(3)$ (rather than $R_1(3) > R_1(2)$) and is not bottlenecked in round 1, but link 1 that shares a flow with link 2 is bottlenecked in round 1.

Appendix B

Supplementary Material for Chapter 3 (s-PERC)

Proof of Corollary 3.4.16. Flow f is seen at every link in P_f at least once between times $T + 1\,round$ and $T + 2\,rounds$ and picks up a bottleneck rate that is either not propagated or $\mathrm{R}(l)$. Flow f's limit rate at link x is the smallest propagated bottleneck rate from any link in $P_f \setminus x$ and hence is at least $\mathrm{R}(l)$ from time $T + 2\,rounds$. □

Proof of Lemma 3.4.10. Consider an update for a flow $f \in FG_n^2(x)$ at a link $y \in P_f$ at some time after $t + 1\,round$. Suppose $b \geq MaxE$ and the rate is propagated (Line 15 of Algorithm 6). If $MaxE \geq \mathrm{R}(l)$, we're done. So let us consider the case when $MaxE < \mathrm{R}(l)$

Let $SumE$, $NumB$, $MaxE$ represent the link state of link y at Line 8 of Algorithm 6 just before it computes a bottleneck rate for f. Note that because of Lines 5–7, we can take for granted that there is at least one flow in $\hat{B}$, namely f. We will break down the aggregate link state in terms of the contribution of $\mathrm{E}(y)$ flows carried by $L_0^2, ..., L_n^2$ and the remaining flows. The remaining flows, which we represent as $\mathrm{B}(y) \cup FG_n^2(y)$, include both $\mathrm{B}(y)$ flows carried by $L_0^2, ..., L_n^2$ as well as flows carried by the remaining links

LG^2_n. The bottleneck rate computed for the flow f satisfies the following:

$$\begin{aligned} b &= (C - SumE)/NumB \\ &= (C - \sum_{f \in \hat{\mathrm{E}}} a_f)/|\hat{\mathrm{B}}| \\ &= (C - \sum_{f \in \mathrm{E}(y) \cap FL^2_n(y)} \mathrm{A}(f) - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B}(y) \cup FG^2_n(y))} a_f)/|\hat{\mathrm{B}}| \end{aligned}$$

In the last step, we used the induction hypothesis that links in $L^2_0, ..., L^2_n$ have converged by time T, which implies that their flows FL^2_n are updated correctly at y from time T. In particular, the subset of FL^2_n flows that are in $\mathrm{E}(y)$ are all correctly classified as belonging to $\hat{E}(y)$ and allocated their max-min rate from time $T + 1$ $round$. This also means that $\hat{\mathrm{B}} \subset \mathrm{B}(y) \cup FG^2_n(y)$. Rearranging the terms we get:

$$\begin{aligned} b.|\hat{\mathrm{B}}| &= C - \sum_{f \in \mathrm{E}(y) \cap FL^2_n(y)} \mathrm{A}(f) - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B}(y) \cup FG^2_n(y))} a_f \\ &\geq \mathrm{R}(l).|\mathrm{B}(y) \cup FG^2_n(y)| - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B}(y) \cup FG^2_n(y))} a_f && \text{(using Property (3))} \\ &\geq \mathrm{R}(l).|\mathrm{B}(y) \cup FG^2_n(y)| - \sum_{f \in \hat{\mathrm{E}} \cap (\mathrm{B}(y) \cup FG^2_n(y))} MaxE && \text{(using Lemma 3.2.4)} \\ &\geq \mathrm{R}(l).|\hat{\mathrm{B}} \cap \mathrm{B}(y) \cup FG^2_n(y)| && \text{(assuming } R(l) \geq MaxE\text{)} \\ &= \mathrm{R}(l).|\hat{\mathrm{B}}| && \text{(since } \hat{\mathrm{B}} \subset \mathrm{B}(y) \cup FG^2_n(y)\text{)} \\ b &\geq \mathrm{R}(l) && \text{(since } |\hat{\mathrm{B}}| > 0\text{)} \end{aligned}$$

□

Bibliography

[1] Barefoot Networks. barefootnetworks.com/technology/#tofino. Accessed: 2017-09-19.

[2] Cavium. cavium.com/xpliant-ethernet-switch-CNX680XX-and-CNX780XX-family.html. Accessed: 2018-10-01.

[3] Intel Flexpipe. intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[4] The network simulator (ns-2). https://www.isi.edu/nsnam/ns/. Accessed: 2018-12-1.

[5] P4 language spec version 1.0.0-rc2. http://www.p4.org/spec/p4-latest.pdf. Accessed: 2014-09-22.

[6] P4-NetFPGA Wiki. https://github.com/NetFPGA/P4-NetFPGA-public/wiki. Accessed: 2017-09-19.

[7] P4 website. http://www.p4.org/. Accessed: 2014-09-22.

[8] Protocol oblivious forwarding (pof) website. http://www.poforwarding.org/. Accessed: 2014-09-22.

[9] Xilinx SDNet. xilinx.com/sdnet. Accessed: 2017-09-19.

[10] White paper: IntelÂő processors in industrial control and automation applications. Technical report, 2014.

[11] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Convergence complexity of optimistic rate-based flow-control algorithms. *Journal of Algorithms*, 30(1):106–143, January 1999.

[12] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[13] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[14] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[15] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, February 1999.

[16] Andrew Appel. *Modern compiler implementation in C*. Cambridge University Press, 2004.

[17] B. Awerbuch and Y. Shavitt. Converging to approximated max-min flow fairness in logarithmic time. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1350–1357 vol.3, March 1998.

[18] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking*, 25(4):1954–1967, August 2017.

[19] Amotz Bar-Noy, Magnús M Halldórsson, Guy Kortsarz, Ravit Salman, and Hadas Shachnai. Sum multicoloring of graphs. *Journal of Algorithms*, 37(2):422–450, 2000.

[20] Jeff Barr. In the works âĂŞ aws region in the middle east. `https://aws.amazon.com/blogs/aws/in-the-works-aws-region-in-the-middle-east/`. Accessed: 2018-10-17.

[21] Yair Bartal, Martin Farach-Colton, Shibu Yooseph, and Lisa Zhang. Fast, fair and frugal bandwidth allocation in atm networks. *Algorithmica*, 33(3):272–286, 2002.

[22] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[24] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[25] Broadcom Corporation. *Broadcom BCM56850 StrataXGS® Trident II Switching Technology*. Broadcom, 2013.

[26] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50, 2016.

[27] Anna Charny, David D Clark, and Raj Jain. Congestion control with explicit rate indication. In *Communications, 1995. ICC'95 Seattle,'Gateway to Globalization', 1995 IEEE International Conference on*, volume 3, pages 1954–1963. IEEE, 1995.

[28] Anna Charny, KK Ramakrishnan, and Anthony Lauck. Time scale analysis scalability issues for explicit rate allocation in atm networks. *IEEE/ACM Transactions on Networking*, 4(4):569–581, 1996.

[29] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '17, pages 239–252, New York, NY, USA, 2017. ACM.

[30] Nikos Chrysos and Manolis Katevenis. Transient behavior of a buffered crossbar converging to weighted max-min fairness. 2002.

[31] Cisco Systems. *Deploying Control Plane Policing*. Cisco White Paper, 2005.

[32] David Clark. The design philosophy of the darpa internet protocols. *ACM SIGCOMM Computer Communication Review*, 18(4):106–114, 1988.

[33] Jorge A. Cobb and Mohamed G. Gouda. Stabilization of max-min fair networks without per-flow state. *Theoretical Computer Science*, 412(40):5562–5579, September 2011.

[34] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 237–248, New York, NY, USA, 2005. ACM.

[35] Lorenzo De Carli, Yi Pan, Amit Kumar, Cristian Estan, and Karthikeyan Sankaralingam. Plug: Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *Proceedings of the 2009 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '09, pages 207–218, New York, NY, USA, 2009. ACM.

[36] Nandita Dukkipati. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3292347.

[37] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 275–287. ACM, 2015.

[38] Eli Gafni and Dimitri Bertsekas. Dynamic control of session input rates in communication networks. *IEEE Transactions on Automatic Control*, 29(11):1009–1016, 1984.

[39] Eliezer M Gafni. *The integration of routing and flow-control for voice and data in a computer communication network*. PhD thesis, Massachusetts Institute of Technology, 1982.

[40] Michael R Garey, Ronald L Graham, David S Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A*, 21(3):257–298, 1976.

[41] Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. Accurate congestion control for rdma transfers. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE, 2018.

[42] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 13–24. IEEE, 2013.

[43] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication*, pages 29–42. ACM, 2017.

[44] Howard P Hayden. *Voice flow control in integrated packet networks*. PhD thesis, Massachusetts Institute of Technology, 1981.

[45] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.

[46] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

[47] IBM. *IBM ILOG CPLEX Optimization Studio V12.4*.

[48] Jeffrey Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[49] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[50] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. High speed networks need proactive congestion control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 14:1–14:7, New York, NY, USA, 2015. ACM.

[51] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 103–115, Berkeley, CA, USA, 2015. USENIX Association.

[52] Shivkumar Kalyanaraman, Raj Jain, Sonia Fahmy, Rohit Goyal, and Bobby Vandalore. The erica switch algorithm for abr traffic management in atm networks. *IEEE/ACM Transactions on Networking*, 8(1):87–98, 2000.

[53] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 89–102, New York, NY, USA, 2002. ACM.

[54] Yuseok Kim, Wei Kang Tsai, Mahadevan Iyer, and Jordi Ros-Giralt. Minimum rate guarantee without per-flow information. In *Network Protocols, 1999.(ICNP'99) Proceedings. Seventh International Conference on*, pages 155–162. IEEE, 1999.

[55] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[56] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[57] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. *Compilers: Principles, techniques, and tools*. Addison-Wesley, 2006.

[58] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial, 2000.

[59] D. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Vxlan: A framework for overlaying virtualized Layer 2 networks over Layer 3 networks. Internet-Draft draft-mahalingam-dutt-dcops-vxlan-00, IETF, 2011.

[60] Jelena Marasevic, Cliff Stein, and Gil Zussman. A fast distributed stateless algorithm for alpha-fair packing problems. *arXiv preprint arXiv:1502.03372*, 2015.

[61] Microsoft. Microsoft to deliver cloud services from new datacentres in germany in 2019 to meet evolving customer needs. `https://news.microsoft.com/europe/2018/08/31/microsoft-to-deliver-cloud-services-from-new-datacentres-in-germany-in-2019-to-meet-evolving-customer-needs/`. Accessed: 2018-10-17.

[62] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[63] Alberto Mozo, Jose Luis López-Presa, and Antonio Fern'ndez Anta. Slbn: A scalable max-min fair algorithm for rate-based explicit congestion control. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pages 212–219. IEEE, 2012.

[64] Alberto Mozo, Jose Luis López-Presa, Antonio Fern, et al. B-neck: A distributed and quiescent max-min fair algorithm. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pages 17–24. IEEE, 2011.

[65] Ali Munir, Ghufran Baig, Syed M. Irteza, Ihsan A. Qazi, Alex X. Liu, and Fahad R. Dogar. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '14, pages 491–502, New York, NY, USA, 2014. ACM.

[66] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, pages 188–201. ACM, 2016.

[67] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 495–506, New York, NY, USA, 2013. ACM.

[68] NVIDIA Corporation. *NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*. NVIDIA White Paper, 2009.

[69] Open Networking Foundation. *Software-Defined Networking: The new norm for networks*. Open Networking Foundation White Paper, 2012.

[70] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 421–435, Berkeley, CA, USA, 2017. USENIX Association.

[71] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.

[72] Teemu Rinta-Aho, Mika Karlstedt, and Madhav P. Desai. The click2netfpga toolchain. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[73] Larry Roberts. Enhanced prca (proportional rate-control algorithm). In *ATM Forum*, 1994.

[74] Jordi Ros-Giralt. *A Theory of Lexicographic Optimization for Computer Networks*. PhD thesis, University of California, Irvine, 2003.

[75] Jordi Ros-Giralt and Wei K Tsai. A lexicographic optimization framework to the flow control problem. *IEEE Transactions on Information Theory*, 56(6):2875–2886, 2010.

[76] Jordi Ros-Giralt and Wei Kang Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001 Conference on Computer Communications*, pages 717–726. IEEE, 2001.

[77] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '17, pages 404–417, New York, NY, USA, 2017. ACM.

[78] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 11–24, New York, NY, USA, 2014. ACM.

[79] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 67–82, Berkeley, CA, USA, 2017. USENIX Association.

[80] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM.

[81] Brian Stevens. Google cloud platform sets a course for new horizons. `https://cloud.google.com/blog/products/gcp/google-cloud-platform-sets-a-course-for-new-horizons`. Accessed: 2018-10-17.

[82] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[83] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[84] Xilinx, Inc. *DSP: Designing for Optimal Results*. Xilinx, Inc., 2005.

[85] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014.