Taming Model Incongruities in Networked Systems

Nirav Atre

CMU-CS-25-140

September 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Justine Sherry (Chair)
Vyas Sekar
Weina Wang
Brighten Godfrey (UIUC)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was funded by National Science Foundation (NSF) Awards 1700521 and 2007733, the CONIX Research Center (one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA), Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, a VMWare Systems Research Award, a Cylab Presidential Fellowship, and a Google Research Gift.

Keywords: model incongruity, caching with delayed hits, Belatedly, algorithmic complexity attacks, adversarial scheduling, SurgeProtector, hardware packet scheduling, BBQ

For my father, Rajesh Atre, who taught me to pour everything I have into anything I do; and my mother, Vasundhra Atre, from whom I learned that when something is necessary, it cannot be impossible.

Abstract

We reason about computer systems via models of their behavior — whether implicit *mental* models, or explicit *mathematical* models. These models are the linchpins of our decision-making ability, *e.g.*, in formulating service-level agreements (SLAs), or tendering performance claims. Unfortunately, a growing disconnect between how systems are modeled and how they are actually deployed has engendered a class of problems that we call **model incongruity**: circumstances where a model's prediction deviates significantly from real-world behavior. Model incongruities are highly pervasive in modern systems, resulting in expensive performance anomalies, scalability bottlenecks, and security vulnerabilities.

In this thesis, we argue that many incongruities observed in practice today are not a fundamental limitation of our modeling capabilities, but rather artifacts of using the *wrong* models. We show that: (a) assumptions centrally underpinning contemporary models of network subsystems have drifted far from deployment realities; (b) these assumptions are frequently violated in the field, subverting the operator's expectations about key metrics in highly unexpected ways; and, (c) making modest model refinements not only yields designs with state-of-the-art performance, attack resilience, and scalability, but also enables us to make rigorous mathematical guarantees about the resulting system's behavior.

We exemplify this point using case studies of three ubiquitous network subsystems. First, we will describe *delayed hits*, an incongruity arising in high-performance caching systems which breaks the textbook caching principle that maximizing cache hit-rate also minimizes latency, and causes *every* existing caching algorithm to make latency-suboptimal decisions; in this context, we will introduce Minimum-AggregateDelay (MAD), a turnkey augmentation to existing algorithms that makes them aware of delayed hits, yielding 5–35% lower request latencies. Second, we will describe *algorithmic complexity attacks* (ACAs), a highly potent class of Denial-of-Service attacks arising from transient workload incongruity; in this context, we will introduce SurgeProtector, an adversarial scheduling framework that *provably* protects network dataplanes against ACAs, resulting in 90–99% reduction in harm for the same volume of attack traffic. Finally, we will describe BBQ, a system borne out of addressing design incongruity in hardware packet schedulers. BBQ, for the first time, makes it *feasible* to deploy packet scheduling at line-rate on modern network switches and SmartNICs, bridging a long-standing gap between the concept and execution of programmable hardware packet scheduling.

Acknowledgments

Anyone who had the misfortune of asking me "how I was doing" between 2018 and 2025 has likely endured my monologue on why life as a Ph.D. student is, in my opinion, the ideal state of being. I gave three arguments in support of this controversial claim: (1) one has infinite intellectual freedom coupled with little-to-no external accountability; (2) one gets paid every two weeks, no questions asked; and, (3) everyone answers one's emails (at least the first time). It took me several years to realize that these were not, in fact, universal features of a Ph.D. program, but rather almost purely a function of one's advisor. So, thank you, Justine, for your unwavering (often inexplicably so) faith in me, for shielding me from the riptides of academic bureaucracy, and for allowing me to spend freely the vast amount of goodwill and professional capital that you have so effortlessly cultivated over the years. You are the absolute best.

I owe an immense debt of gratitude to an incredible set of mentors, starting with my thesis committee: Weina Wang, Vyas Sekar, and Brighten Godfrey. Thank you, Weina, for taking me under your wing (despite my penchant for flipping inequalities), and for being such a gentle, patient, and magnanimous guide through the beautiful world of theory; I am (and always will) be delighted to be mistaken for your student. Thank you, Vyas, for being my North Star in uncertain times; your knack for casually dispensing life-changing and research-defining advice has kept me from going astray more times than I can count; to this day, I give floundering students the exact same advice I did 5 years ago: stop talking to me, and go find Vyas. Thank you, Brighten, for being so deeply invested and championing this work, and for your unwavering support throughout my job search (including entertaining my preposterously last-minute letter requests). I am also deeply grateful to Aditi Raghunathan, Anirudh Sivaraman, Anupam Gupta, Aurojit Panda, Balaji Prabhakar, Daniel Berger, Giulia Fanti, James Hoe, Jonathan Rose, Keith Winstein, Mahmoud Elhaddad, Mor Harchol-Balter, Nathan Beckmann, Nick McKeown, Peter Steenkiste, Phil Levis, Rashmi Vinayak, Riccardo Paccagnella, Roy Want, Ruben Martins, Srini Seshan, and Yuvraj Agarwal for their sage advice and generous mentorship through various pivotal moments in my academic career.

I started graduate school with the preconceived notion that the Ph.D. journey was a solitary one; thankfully, I was grossly mistaken. I am incredibly fortunate to have met some truly remarkable people who have made Pittsbugh feel like home, starting with Justine's extended research group and the SNAP Lab: Adithya Abraham Philip, Antonis Manousis, Anup Agarwal, Benjamin Carleton, Chris Canel, Erica Chiang, Francisco Pereira, Hugo Sadok (special thanks for being my closest collaborator, patiently listening to my unhinged rants about hardware synthesis tools, and tolerating 6 years' worth of questionable humor), Ioannis Anagnostides, Isabel Suizo, Kshitij Rana, Margarida Ferreira, Matt Butrovich, Miguel Ferreira, Ranysha Ware, Sagar Bharadwaj, Shawn Chen, Yiran Lei, Zhipeng Zhao, and Zili Meng.

Beyond my research group, I am fortunate to have met a truly wonderful set of souls in the broader CSD, CMU, UPitt, and academic communities. My housemates over the years: Abhiram Kothapalli, Ben Koby, Izzy Grosof, Pete Gungel, and Roger Iyengar. The brunch/tennis group: Adithya Pratapa, Aditi Kabra, Ankush Jain, Arjun Lakshmipathy, Jovina Vaswani, Mansi Goyal, Mara Aragones, Mihir Bala, Mugdha Deokar, Pratiksha Thaker, Sai Sandeep, Sakshi Satyanand, Sanjith Athlur, Shilpa Anna George, Sitoshna Jatty, and Surabhi Gupta. The best CSD cohort ever (plus honorary members): Arjun Teh, David Kahn, Giulio Zhou, Jalani Williams, Joshua Williams, Mark Gillespie, Mikhail (Misha) Khodak, Pallavi Koppol, Shir Maimon, and Siva Somayyajula. And dear friends I made through the most fortuitous of circumstances: Akshay Narayan, Ananya Joshi, Angela Jiang, Anuj Kalia, Carlos Martin, Daiyaan Arfeen, Deepanjali Mishra, Deepti Raghavan, Devdeep Ray, Dorian Chan, Francisco Maturana, Gabriele Farina, Gaurav Manek, Jack Kosaian, Jason Choi, Karan Ahuja, Long Pham, Pranav Khadpe, Prateesh Goyal, Rishabh Iyer, Sam Westrick, Sara McAllister, Siddharth Jayashankar, Sol Boucher, Venkat Arun, and Vibhaalakshmi Sivaraman. I am also grateful to the Parallel Data Lab (PDL) for broadening my horizons, both in terms of research, and social/industry connections.

After 7 years of scheduling conference rooms at the very last minute, requesting reimbursements several weeks past due, and scrambling for spare office keys, I am convinced beyond a shadow of doubt that CSD and PDL have *the best* administrative and support staff in existence. I am deeply grateful to Angy Malloy, Catherine Copetas, Deb Cavlovich, Emily Spencer, Joan Digney, Karen Lindenfelser, Matthew Stewart, Olivia Zane, and Sara Kuntz for all their help throughout the years. This place would not function without any one of you!

Finally, none of this would have been possible without my parents (Rajesh and Vasundhra Atre), and grandparents (Padmakar and Jayashree Atre, and Trilok and Kananbala Kohli), who have made ineffable sacrifices to get me here. This thesis is dedicated to them.

Contents

1 Introduction **1**
- 1.1 Why Now? . . . 3
- 1.2 Summary of Results . . . 4
- 1.3 Thesis Organization . . . 5

2 Caching with Delayed Hits **7**
- 2.1 Incongruity in the Classical Model of Caching . . . 8
- 2.2 Impact of Delayed Hits . . . 10
  - 2.2.1 Misleading Operators . . . 10
  - 2.2.2 Misleading Algorithms . . . 12
- 2.3 Latency Minimization in the Delayed Hits Model . . . 13
  - 2.3.1 A Case for Principled Analysis . . . 13
  - 2.3.2 Problem Formulation . . . 14
- 2.4 Belatedly: Offline, Latency-Optimal Caching . . . 16
  - 2.4.1 Network Flow Formulation . . . 17
  - 2.4.2 Delayed Hits and Empirical Latencies . . . 20
  - 2.4.3 Optimizations to Reduce Complexity . . . 22
  - 2.4.4 Performance Evaluation . . . 26
  - 2.4.5 Proof of Optimality . . . 27
- 2.5 MAD: Online, Low-Latency Caching . . . 34
  - 2.5.1 Proxy Oracle: Bélády-AggregateDelay . . . 35
  - 2.5.2 Online Algorithm: Minimum-AggregateDelay (MAD) . . . 36
- 2.6 Evaluating MAD . . . 38
  - 2.6.1 Experimental Setup . . . 38
  - 2.6.2 Prototype Evaluation on CDN Trace . . . 39
  - 2.6.3 Simulation Results: Systems . . . 41
  - 2.6.4 Simulation Results: Analysis . . . 43
- 2.7 Related Work . . . 45
- 2.8 Broader Impact, Open Questions, and Conclusion . . . 46

3 Mitigating Algorithmic Complexity Attacks on Network Subsystems **47**
- 3.1 Workload Incongruity . . . 48
- 3.2 Background and Motivation . . . 49

3.2.1 High Pervasiveness and Potency of ACAs . . . 50
3.2.2 Existing Defenses are Insufficient . . . 51
3.3 Incongruity-Aware Workload Model . . . 52
3.3.1 System Model . . . 52
3.3.2 Threat Model . . . 53
3.3.3 Quantifying Vulnerability . . . 53
3.4 Mitigating ACAs using Scheduling . . . 54
3.4.1 First-Come First-Serve (FCFS) . . . 54
3.4.2 Fair Queueing (FQ) . . . 55
3.4.3 Shortest Job First (SJF) . . . 57
3.4.4 Weighted Shortest Job First (WSJF) . . . 59
3.5 SurgeProtector Implementation . . . 63
3.5.1 Overview of Vulnerable Components . . . 64
3.5.2 Predicting Job Sizes . . . 65
3.5.3 Keeping (TCP) Flows In-Order . . . 66
3.5.4 Designing Adversary-Proof Schedulers . . . 67
3.6 Evaluation . . . 68
3.6.1 SurgeProtector + Pigasus . . . 68
3.6.2 SurgeProtector in Simulation . . . 70
3.6.3 SurgeProtector Scheduler . . . 74
3.7 Limitations and Open Questions . . . 75
3.8 Related Work . . . 76
3.9 Conclusion . . . 77

4 Deployable Hardware Packet Scheduling 79
4.1 Background and Motivation . . . 80
4.1.1 Design Incongruity . . . 80
4.1.2 Exploring a Different Tradeoff . . . 83
4.2 BBQ Overview . . . 84
4.2.1 Data Structure . . . 84
4.2.2 BBQ Primitive . . . 85
4.2.3 Goals and Challenges . . . 86
4.3 BBQ Architecture . . . 88
4.3.1 Hardware Pipeline . . . 89
4.3.2 A Fully-Pipelined Architecture . . . 89
4.3.3 Implementing Subtree Occupancy Counters (StOC) . . . 93
4.4 Logical Partitioning in Practice . . . 95
4.4.1 Existing Designs . . . 96
4.4.2 Logical Partitioning in BBQ . . . 98
4.5 Extensions to the BBQ Primitive . . . 99
4.5.1 BBQ$_{\odot}$: A Latency-Free BBQ . . . 99
4.5.2 Dynamic Priority Ranges . . . 103
4.6 Evaluation . . . 104
4.6.1 Setup and Methodology . . . 104

4.6.2 FPGA . . . 104
4.6.3 ASIC . . . 107
4.7 Applications . . . 108
4.7.1 Packet Scheduling on Switches . . . 108
4.7.2 Packet Scheduling on Cloud SmartNICs . . . 108
4.8 Limitations and Open Questions . . . 109
4.9 Related Work . . . 110
4.10 Conclusion . . . 111

5 Conclusion **113**

Bibliography **115**

List of Figures

2.1 Two requests for object X arrive within Z timesteps of each other. 8
2.2 Example cumulative density function (CDF) of request latencies. Delayed hits account for the gap between the *true* hit-rate (HR) and the miss-rate (MR). . . . 9
2.3 Normalized latency predictions with (*Actual*) and without (*Idealized*) accounting for delayed hits. 11
2.4 Pairwise comparisons between four online caching algorithms. 11
2.5 For the given trace, with a cache of size 2 and a Z value of 3, Bélády's algorithm chooses a latency-suboptimal schedule. 12
2.6 The BELATEDLY pipeline for computing bounds on latency-optimal cache schedule using MCMCF reduction. 16
2.7 Latency gap between Bélády and BELATEDLY for different application scenarios (Network, CDN, Storage) today. Top to bottom: 1%, 5%, and 10% cache sizes. . . 21
2.8 %Relative latency difference between Bélády and BELATEDLY as a function of Z. Using cache size, $c = 5\%$ (expressed as a percentage of the maximum number of concurrent flows). 21
2.9 %Relative latency difference between Bélády and BELATEDLY as a function of cache size. Using $Z = 500$. 21
2.10 Relative latency improvement *vs* burstiness (for Network traffic). Bursty flows suffer less under BELATEDLY. 22
2.11 A fragment of a request sequence highlighting nodes and edges corresponding to object a (colored red), with $Z = 3$. 23
2.12 A fragment of a request sequence highlighting ingress and egress edges for node $V_{\text{mem},t}$, with $Z = 3$. 24
2.13 The optimized representation with backing store nodes removed. 25
2.14 Number of decision variables in the naive MCMCF formulation versus BELATEDLY for different application scenarios. 26
2.15 Ranking objects *solely* based on aggregate delay may lead to poor utilization of cache space. 35
2.16 BÉLÁDY-AD closely trails BELATEDLY. 36
2.17 Architecture of our experimental prototype. 38
2.18 Prototype results for different origin locations. 39
2.19 CDF of latencies in simulation versus real experiments. 40
2.20 MAD simulations for the CDN Trace. 41

2.21 MAD simulations for the Network Trace. . . . 42
2.22 MAD simulations for the Storage Trace. . . . 42
2.23 Relative latency difference between LRU-MAD and LRU as a function of the cache size. Using $Z = 100K$. . . . 43
2.24 Like BELATEDLY, MAD prioritizes bursty objects. . . . 44
2.25 Percent relative change in miss-rate between MAD and various baseline caching algorithms for Network, CDN, and Storage. . . . 44

3.1 The classical model posits a false dichotomy between workload characteristics. 48
3.2 TCP reassembly implemented using a linked list [162]. Each node in the list represents a range of packet sequence numbers. . . . 50
3.3 Refined model treating the workload as a superposition of *purely stochastic* innocent traffic (purple) and a sliver of *adversarial* traffic (orange). . . . 52
3.4 FCFS fails to protect against ACAs. Using minimum-sized packets encoding maximum-sized jobs allows the attacker to induce significant amounts of work in the system. . . . 55
3.5 FQ fails to protect against ACAs. Since FQ allocates service time *per flow*, the attacker can trick the FQ scheduler into giving them a large fraction of service time using multiple flows. In steady-state, the attacker receives 75% of the total service time despite using a small attack bandwidth. . . . 56
3.6 In order to exploit SJF, the attacker uses minimum-sized packets with a job size (i.e., CPU time) between that of packets 1 and 3. The attack packets (i.e., 2, 5, 6) are scheduled before more expensive ones (3, 4), pushing the system into overload and displacing packet 4. . . . 57
3.7 Optimal choice of adversarial job size J_A. . . . 58
3.8 WSJF service order in steady state. . . . 60
3.9 The simplified Pigasus IDS pipeline. . . . 64
3.10 Linked-list state for an out-of-order flow. . . . 65
3.11 Pigasus Full Matching pipeline. . . . 65
3.12 A Hierarchical FFS Queue implemented using 2-bit bitmaps and height of $h = 3$. A '1' in any bitmap indicates a non-empty priority bucket in the subtree rooted at that node. In order to find the *min* (or *max*) priority bucket, we recursively follow the leftmost (or rightmost) set bit. . . . 68
3.13 Goodput of Pigasus' TCP Reassembler under FCFS and SURGEPROTECTOR. . . . 69
3.14 Goodput of Pigasus' TCP Reassembler for different degrees of out-of-orderness of attack flows. . . . 70
3.15 Goodput of Pigasus' Full Matcher under FCFS and SURGEPROTECTOR. . . . 70
3.16 Simulator pipeline. . . . 71
3.17 Goodput and Displacement Factor (DF) for TCP Reassembly. Left to right: Increasing innocent input rate, r_I, from 1 Gbps to 10 Gbps. . . . 72
3.18 Goodput and Displacement Factor (DF) for Pigasus Full Matching. Left to right: Increasing innocent input rate (r_I) from 0.1 Gbps to 0.9 Gbps. . . . 72

3.19 Absolute change in DF achieved by WSJF-Inorder due to the heuristic. Portions highlighted in red indicate regions where the heuristic does worse than the baseline. . . . 73
3.20 Goodput for different heap implementations. . . . 74

4.1 Scheduler architecture using priority queues without (left), and with (right) support for logical partitioning. In a k-port switch using priority queues without logical partitioning, we need $k^2 + k$ priority queues in order to both implement output queueing *and* absorb incast traffic from all the ports. Support for logical partitioning reduces the number to $2k$. . . . 81
4.2 2-level BBQ with $w = 3$ bit bitmaps. To dequeue the highest-priority element, we recursively perform FFS at each level starting with the root, following the most-significant set bit (MSSb) until we arrive at the required priority bucket. . . . 85
4.3 Non-atomic read-modify-write accesses to bitmaps cause OP_B (the second of two consecutive DEQUEUE–MAX operations) to be incorrectly routed to the left-hand subtree. . . . 91
4.4 If there are conflicting operations in the L_i PHRs, operations issued later compute bit indices *speculating* that earlier operations will change the bitmap. . . . 92
4.5 Dependency chain involving 16-bit counter logic for a DEQUEUE operation. The critical path comprises of (a) a 15-bit Reduce-OR (to determine whether the StOC becomes 0), chained with (b) more combinational logic (which uses (a) as a predicate). . . . 94
4.6 The waterlevel bit (WLb) replaces (a), improving $f_{\max}$ by removing the counter operations from the critical path. . . . 95
4.7 A single physical PIFO partitioned into 2 logical PIFOs: q_0 consisting of 4 elements, and q_1 consisting of 3 elements. Available queue memory is fully multiplexed among the logical PIFOs, resulting in zero fragmentation. . . . 96
4.8 A 2-level 2-way BMW-Tree instance partitioned into 2 logical queues. Each logical queue must be mapped to a *physical* subtree, resulting in external fragmentation. Once a subtree becomes full, enqueues into the corresponding logical queue are impossible even if there are available QEs in other subtrees. . . . 97
4.9 Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 2 logical BBQs, q_0 and q_1. Each logical BBQ is allocated disjoint ranges of 8 priorities. To DEQUEUE from a logical BBQ, we first apply the corresponding *mask* to the L_1 bitmap before performing FFS on it. . . . 98
4.10 Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 8 logical BBQs. The root (L_1) bitmap no longer adds any value, so we replace it with a *steering stage* that simply routes operations on logical BBQs to the corresponding subtree. 99
4.11 Issuing concurrent DEQUEUE requests, in combination with BBQ's pipeline latency, incorrectly causes a lower-priority flow, B, to be extracted (at $t = 3$) and scheduled (at $t = 6$). . . . 100
4.12 BBQ$_{\odot}$ design. . . . 101
4.13 Clock frequency as we scale the queue capacity. . . . 104
4.14 Throughput as we scale the queue capacity. . . . 105

4.15 ALM utilization as we scale the queue capacity. 105
4.16 SRAM block utilization as we scale the queue capacity. PIFO is not included in the plot as it does not use SRAM. 106
4.17 BBQ throughput as we increase the number of priorities when using different bitmap widths (W). 107
4.18 BBQ throughput as we increase the number of priorities when using different number of levels (D). 107
4.19 A two-level hierarchical NIC scheduler using BBQ. The first BBQ schedules traffic *across* tenants. The second BBQ (which is logically partitioned) is used to schedule traffic *within* each tenant. 109

List of Tables

2.1 Average inter-request times (IRT), typical latencies, and Z values seen today. . . 10
2.2 Empirical bounds on BELATEDLY's error (calculated by comparing the integer upper-bound to the relaxed lower-bound). 27
2.3 Overview of implemented online caching algorithms. 41

3.1 Summary of notations used in the model. 53

4.1 Priority Queue operations supported by BBQ. 86
4.2 11-stage hardware pipeline for a 2-level BBQ (without operation coloring) highlighting independent pipeline hazard regions (PHRs). $\hookrightarrow$ and $\rightsquigarrow$ indicate same-stage dependencies, which result in more complex combinational logic. In general, a BBQ with $D > 1$ levels entails a pipeline depth of $p = 7 + 4 \times (D - 1)$ stages. 88
4.3 Updated stages $7 - 10$ showing operation coloring. 93
4.4 Chip area for the different designs. BBQ uses little logic, causing its area to be primarily determined by SRAM. 107

Chapter 1

Introduction

"All models are wrong, but some are useful."

George E. P. Box

Since the early days of the ENIAC, our ability to engineer complex computer systems has evolved at an astounding pace. Where previously instantiating a service at Internet scale was a matter of innovation, today it is one of simple, formulaic composition: integrating the application with a few elemental *subsystems* (*e.g.*, caches, firewalls, and schedulers) that imbue it with properties such as performance, security, and scalability. But the system itself is only part of the equation; a primary bottleneck in the systems-building process — and one where human ingenuity continues to shine — lies in making the key design and business decisions governing deployment. *Which caching algorithms should we use? How much attack mitigation capacity should we pay for? How many elements must the scheduler support?*

While these questions appear in different contexts, they share a common epistemological basis: answering them requires us to invoke **models** of how the underlying components behave. Sometimes, the model is *explicit*, based on formal or mathematical reasoning; an archetypal example of this is the *Average Memory Access Time (AMAT)*, an analytical expression we teach in undergraduate computer architecture which posits that improving a cache's hit-rate also reduces its average request latency. Other times, it is *implicit*, *e.g.*, encoded as a practitioner's heuristic or belief that deploying a firewall with higher nominal throughput will improve user experience. Regardless of how they manifest, these models endow us with all of the predictive power that we have and need to make real-world decisions.

However, unsurprisingly, models are not *perfect*. As abstractions, they are necessarily limited in precision both by their form (distilled representations of reality), and their function (being analytically tractable and fitting within the cognitive limits of their designers). Given this fundamental tension between precision and simplicity, the usefulness of a model ultimately depends on *how much* and *how frequently* it is wrong about the details that matter. When the magnitude and frequency of errors grows too large, it results in a problem I call **model incongruity**: material dissonance between the model and the system it is *supposed* to represent.

Model incongruities matter because they disrupt our ability to make the "right" design and business decisions. They might cause a caching algorithm with the highest hit-rate to unexpectedly deliver *worse* latency in the field, or a throughput-optimizing firewall to altogether *starve* traffic under certain workloads. The problem is compounded by their subtlety; unlike traditional bugs which violate easily-checkable correctness criteria, incongruities distort higher-order metrics (*e.g.*, performance, scalability) that are beyond the purview of established testing methodologies and verification tools. Consequently, incongruities remain latent in production systems until they are triggered by a subtle shift in workload, a scaling inflection point, or an adversary who learns how to exploit them.

Over the years, the computer systems community has evolved several adaptations to manage the perennial risk posed by incongruities, from over-provisioning resources to extensive benchmarking as a way of compensating for theoretical blind spots. And while these strategies are effective (perhaps even necessary) in the short term, they only provide temporary, symptomatic relief; as deployment circumstances evolve (*e.g.*, network line-rates go up, or the number of users double), each stopgap must be re-tuned or replaced. In effect, system design has converged to a precarious equilibrium: relying on system models to guide billion-dollar deployment decisions, while quietly dreading the reality that the system is more brittle than it appears. This raises the central question: *how can we begin to unequivocally trust our models of computer systems?*

In this thesis, we argue that many incongruities observed in practice today are not a fundamental limitation of modeling, but rather artifacts of using the *wrong* models. We exemplify this point using case studies of three ubiquitous network subsystems: caches, packet processors (*e.g.*, firewalls), and packet schedulers. In each case, we show that: (a) fundamental assumptions underpinning contemporary models of network subsystems have drifted far from deployment realities; (b) these assumptions are frequently violated in the field, subverting the operator's expectations about key metrics in highly unexpected ways; and, (c) making modest model refinements not only yields designs with state-of-the-art performance, attack resilience, and scalability, but also enables us to make rigorous mathematical guarantees about the resulting system's behavior which, importantly, also hold in practice.

Thesis: *Refining classical models of network subsystems (e.g., caches, packet processors, schedulers) to reflect modern deployment contexts yields designs with provably higher performance, greater attack resilience, and improved scalability.*

Overall, our findings suggest that a concerted effort towards revamping system models *en masse* will not only improve our short-term understanding of today's deployments, but will also pay long-term dividends in our ability to ultimately *build* better systems in the future.

1.1 Why Now?

While model incongruities are as old as modeling itself, three recent trends have exacerbated their prevalence and impact in modern networked systems.

Growing disparity between system complexity and model sophistication. While systems today can be deployed at scales and levels of complexity unimaginable a few decades ago, the models we use to reason about them have not advanced at the same pace. For instance, the analytical and simulation caching models used today remain anchored in abstractions dating back to the 1960s: Bélády's paging model of caching and offline optimal algorithm (1966) continue to serve as the *de facto* benchmark for evaluating modern caching algorithms, and the AMAT expression (1990), which posits that maximizing cache hit-rate is mathematically equivalent to minimizing request latency, still guides deployment decisions in Internet-scale caching. And while the theoretical foundations due to these models have faithfully served many generations of systems, as we will see in §2.1, they are now misaligned with deployment realities. This misalignment is especially problematic in the context of *networked* systems, which are routinely subject to interactions with all classes of Internet-capable users: inquisitive humans, algorithmic bots, and, increasingly, autonomous AI agents. In such settings, it is all the more imperative that our models of system behavior be commensurate with the system itself.

Increasing cost of model predictions being wrong. The early Internet had a single goal: communication. As a result, service providers' concerns were largely centered around whether or not users' data was correctly delivered across the network. In contrast, modern services are evaluated on a much broader set of metrics, many of which are now enshrined in service-level agreements (SLAs). Thanks to SLAs, expectations are commodified, and properties once treated as "best-effort" (*e.g.*, throughput, latency, and availability) have now become contractual obligations. On the one hand, if model predictions are *too optimistic* and service providers fall short of delivering on them, they risk SLA violations, financial penalties, and reputational damage. Conversely, predictions that are *too pessimistic* entail wasteful over-provisioning of resources, opportunity costs (losing customers to competitors), or both. In either case, model mispredictions entail high collateral, and the cost only seems to be going up.

Rise of high-stakes failure domains. Finally, the promise of low latency and high throughput has catapulted many "high-stakes" services into the virtual realm (*e.g.*, robotic surgery, autonomous vehicle control); in these domains, even small deviations between predicted and actual behavior can have disproportionate effects. As these applications slowly but surely become mainstream, we can expect the tolerance for modeling error to drop precipitously.

Together, these trends have created conditions in which the limitations of existing models have become both more apparent and costly. We believe that closing the gap between system complexity and model fidelity is therefore an essential prerequisite for building the next generation of reliable, efficient, and secure networked systems.

1.2 Summary of Results

This thesis presents three case studies, each corresponding to a real, documented failure mode in the model of a ubiquitous network subsystem. In each case, we demonstrate the implications of incongruity, how modest refinements to the model translate into actionable insights and, ultimately, state-of-the-art system-level improvements realized by leveraging these insights.

Delayed Hits in Latency-Minimizing Caches [12]. Textbooks tell us that there are exactly two possible outcomes when an object is requested: a low-latency cache *hit*, or a higher latency cache *miss*. In reality, there is a third potential outcome that the classical caching model missed: a *delayed hit*. Delayed hits occur in high-throughput systems when multiple requests for the same object occur before an outstanding cache miss is resolved. We demonstrated:

- Delayed hits cause existing caching models and simulators to underestimate average request latencies by up to 63%, and lead practitioners to inadvertently deploy the wrong caching algorithm — problems that are only going to get worse with time.
- In the presence of delayed hits, maximizing hit-rate and minimizing latency are *fundamentally* different optimization problems, implying that (a) the hit-rate maximizing oracle (Bélády's algorithm) is *not* latency-optimal, and (b) existing caching algorithm are chasing the wrong metric.
- The design of a truly latency-optimal *offline* caching algorithm called Belatedly which yields 9–37% lower average latency than Bélády across a broad range of network settings.
- The design and implementation of Minimum Aggregate-Delay (MAD), a simple turnkey augmentation (based on model insights from Belatedly) to make existing *online* caching algorithms aware of delayed hits, yielding 12–35% latency improvements on production caching workloads running on a real system.

Algorithmic Complexity Attacks (ACAs) on Network Packet Processors [13]. ACAs are a class of denial-of-service (DoS) attacks targeting variance in the run-time complexity of algorithms underlying packet processors (*e.g.*, firewalls). By using carefully-crafted attack traffic, ACAs allow an attacker to produce harm inordinately higher than the attacker's own resource investment (up to 2,000,000×). Prior to our work, there was no *general* fix to ACAs; every "patch" would have to be designed and engineered on a case-by-case basis, necessitating intrusive design changes and sacrificing some system property or another (*e.g.*, memory efficiency, or average-case performance) in exchange for ACA resilience. We demonstrated:

- ACAs are fundamentally artifacts of incongruity between the choice of algorithm, which is fixed at deployment time, and drift in workload characteristics (from purely stochastic to partially adversarial) at run-time.
- A mathematical model of ACAs targeting single-server packet processors, and a novel metric (called the "displacement factor") to quantify their impact.
- That a simple augmentation to an extensively studied, decades-old scheduling algorithm (Shortest Job First) guarantees a theoretical upper-bound on the harm induced by ACAs.

- The design and implementation of SurgeProtector, a drop-in scheduling framework that leverages this augmented scheduling algorithm (called Weighted Shortest Job First, or WSJF) to *provably* protect packet processors from ACAs. SurgeProtector necessitates no changes to the underlying algorithm, providing, for the first time, a *general-purpose* approach to mitigating ACAs on networked systems. In the context of a real system, SurgeProtector yields a 90–99% reduction in harm induced by ACAs.

Deployment Barriers for Hardware Packet Schedulers [14]. Packet schedulers decide the order in which network packets ought to be served or transmitted, for which they rely on a *priority queue* data-structure. Existing hardware priority queue designs used comparison-based sorting, which entails a *fundamental* trade-off between performance and scalability. This trade-off meant that schedulers could not be deployed on modern switches and SmartNICs because they either did not scale to realistic connection counts or failed to deliver good throughput. Conversely, the *approximate* versions of these scheduling algorithms introduce priority inversions (i.e., scheduling errors) that are at odds with the requirements of safety-critical applications. We demonstrated:

- A full quantitative analysis of the minimum requirements imposed on the scheduler dataplane by modern switches and SmartNICs (spanning packet throughput, scalability, and logical multiplexing).
- That the lack of suitable scheduler deployment candidates is, in fact, the result of incongruity between the prescribed hardware design objectives and minimum operational requirements (i.e., support for 100K connections at 100 Gbps line-rate is not merely an objective, but a *constraint*).
- How an existing priority queue design with constant worst-case time complexity (previously used in software) can be ported to a fully-pipelined hardware architecture, thereby circumventing the performance/scalability barrier imposed by comparison-based sorting.
- The design and implementation of the Bitmapped Bucket Queue (BBQ), a hardware priority queue design that, for the first time, makes it feasible to deploy priority packet scheduling on modern switches and SmartNICs. BBQ supports 100K+ flows, 32K priorities, and 128-way logical multiplexing at 100 Gbps line-rate on a commodity FPGA, 3× the packet rate of existing hardware priority queues designs.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we present Caching with Delayed Hits, describing a model incongruity caused by network effects in latency-minimizing caches. In Chapter 3, we present a case study on Algorithmic Complexity Attacks (ACAs), a security vulnerability caused by transient mismatch between dynamic workloads and Pareto-optimal design decisions. In Chapter 4, we present BBQ, a system borne out of addressing design incongruity in hardware packet schedulers. Finally, in Chapter 5, we discuss future research directions and conclude.

Chapter 2

Caching with Delayed Hits

"...physics should represent a reality in time and space, free from spooky action at a distance."

Albert Einstein

Caches are key components of the computer systems toolkit: they reduce bandwidth demand to a bottlenecked backing store, they improve throughput for memory-intensive services, and they reduce read delays for latency-sensitive applications. Consequently, caches appear across seemingly every class of computer system: *e.g.*, in microprocessors [72], in distributed file systems [119], in CDN proxies [26, 53], and in software switches [113].

In this chapter, we focus on a surprisingly overlooked aspect of caching and latency. Although caches fundamentally aim to mask the *high delay* of communicating with a remote backing store, **caching models and simulators irrationally assume that data, once requested from the backing store, appears instantaneously in the cache with *zero delay***. In reality, the interplay of high request throughputs and non-zero delay in communicating with the backing store gives rise to a phenomenon called **delayed hits** [63, 142]. As we will see later in this chapter, delayed hits have significant implications for both the theory and practice of caching, from undercutting practitioners' ability to accurately predict request latencies (affecting large-scale cache deployment outcomes), to causing *all* existing cache replacement algorithms to make latency-suboptimal decisions.

The remainder of this chapter is organized as follows. We begin with an exposition of the incongruity in the classical model of caching in §2.1. In §2.2, we describe the impact of delayed hits, both in the context of *our* decision-making abilities, as well those of our *algorithms*; for instance, we show that Bélády's algorithm (the offline oracle that guides the decision-making of every caching algorithm designed and deployed today) is *itself latency-suboptimal* in the presence of delayed hits. We demonstrate throughout these two sections that our classical model of caching is outdated, omitting a fundamental temporal dimension of system behavior; we argue, therefore, that we need a new model to characterize the behavior of latency-sensitive caching systems.

In §2.3, we present our incongruity-aware caching model, a refinement of the classical abstraction that accounts for delayed hits. In §2.4, we present BELATEDLY, a truly latency-optimal caching oracle that yields up to 33% lower latencies compared to Bélády's algorithm. Based on model insights derived from BELATEDLY's decision-making, in §2.5 we present the design and implementation of MINIMUM AGGREGATE-DELAY (MAD), a simple, turnkey augmentation to existing caching algorithms that makes them aware of delayed hits. We evaluate MAD in §2.6, demonstrating 12–18% latency improvements in a real system on production caching workloads. Finally, we discuss related work in §2.7, and conclude in §2.8.

2.1 Incongruity in the Classical Model of Caching

Classical Model. As per textbook caching, every cache request experiences one of two possible outcomes: a high-latency *cache miss*, which occurs when the requested object is not present in the cache, in which case it must first be fetched from a backing store before the request can be satisfied; and a low-latency *cache hit*, which occurs when a subsequent request to the same object can be served directly from the cache. For a given "trace" (sequence of cache requests), the model then annotates each request as either a *miss* or *hit* depending on whether or not the requested object was present in the cache at the time. It also defines two dual metrics to capture caching efficacy: the *hit-rate (HR)*, defined as the ratio of hits to total number of requests; and the *miss-rate (MR)*, defined analogously as the proportion of misses, and equal to $(1-\text{HR})$.

To understand the incongruity in this model, consider the example depicted in Figure 2.1 below. ① A request arrives for object X (which not stored in the cache at the time), incurring a cache miss. ② This triggers a fetch to retrieve this object from the backing store, a process which takes Z timesteps to complete.[1] ③ After the object is requested, but before Z timesteps have completed, another request arrives for the same object. For this very simple trace, we can ask: what outcomes do the two requests experience? As per the classical model, the first request would result in a *miss*, while the second request (which happens immediately after the miss) would result in a *hit*.

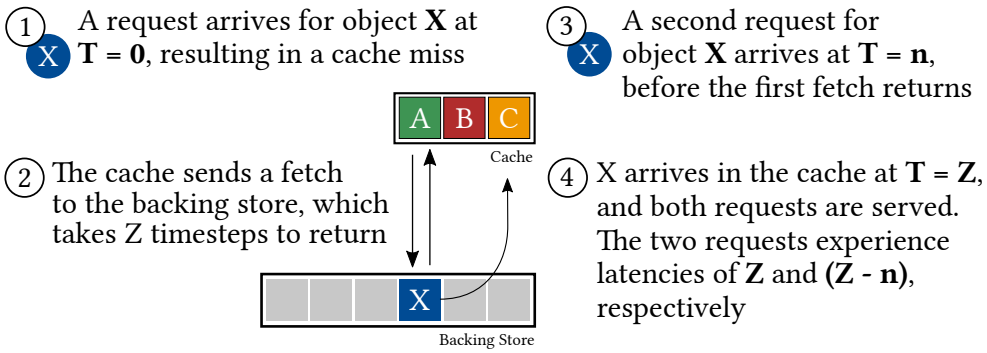


Figure 2.1: Two requests for object X arrive within Z timesteps of each other.

[1] In general, the retrieval might take some non-zero amount of time, L seconds, and the average inter-request arrival time might be R seconds. For simplicity we say that R seconds is one *timestep*, and that the amount of time to fetch the object is $Z = \frac{L}{R}$ timesteps.

However, a closer look at the example reveals that this is not the case!

> **Incongruity**: From the instant the cache miss happens at $T = 0$, it takes Z timesteps to fetch object X into the cache. As such, it is *impossible* for the second request for X (which appears just $n < Z$ timesteps after the original miss) to result in a cache hit, because the object could not physically be present in the cache until $T = Z$.

In fact, as shown in ④, the second request for X will experience *neither* a cache hit (because it cannot be served from the cache), *nor* a full cache miss (because a fetch for that object is already in progress). Instead, it will incur a **delayed hit**, with a latency in-between that of a hit and miss. To concretize this notion, consider a cache where $Z = 10$. At timestep $T = 3$, a request for object X arrives, resulting in a cache miss; this triggers a fetch to the backing store for object X, which will complete at time $T = 13$; thus, the first request will be served after a total latency of 10 timesteps. If additional requests for X arrive at $T = 5$ and $T = 11$, then they too will complete at $T = 13$, and will experience latencies of 8 and 2, respectively.[2] Figure 2.2 depicts the physical interpretation of delayed hits, and the relationship between the *true* hit-rate, the *idealized* hit-rate, and the miss-rate. The classical caching model ignores the latency contribution of delayed hits, engendering a host of problems (§2.2).

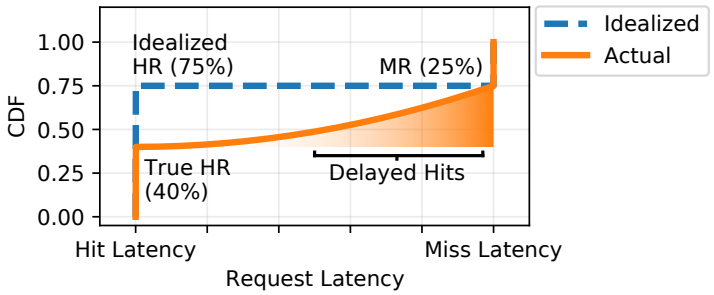


Figure 2.2: Example cumulative density function (CDF) of request latencies. Delayed hits account for the gap between the *true* hit-rate (HR) and the miss-rate (MR).

Why now? Why hasn't anyone noticed before that delayed hits play an important role in cache latencies? Delayed hits are noted in passing in several places in the literature [63, 142], and anyone who has ever *implemented* a cache has had to consider delayed hits as well [1, 21, 87, 105, 132, 145].

We conjecture that the problem has only recently become perceptible from a performance perspective due to an evolving ratio between system throughputs and latencies. If throughput is low relative to latency, it may only be possible for 1–2 requests to arrive during a fetch. However, if throughput is higher relative to latency, we can expect more outstanding requests per fetch. The occurrence of delayed hits is ultimately governed by the ratio between the object fetch time and mean request inter-request time (the parameter we called Z), and to provide some context for what Z values one might find in practical systems today, we describe a few examples in Table 2.1. Inter-request times (IRTs) are based on three timestamped datasets that we will refer to throughout this chapter: a large content distribution network (CDN) [28], the CAIDA Equinix 10G Packet dataset [147], and a networked file system at Microsoft [79].

[2]For the purposes of modeling, we assume that processing time for each request is 0 — that is, as soon as the data arrives, all requests are served instantly. In many systems this is not true, and each request must be processed serially, *e.g.*, reading, modifying, and writing updates to the cached object. Non-zero processing times therefore

Trace	Use Case	Latency	Z
CDN	Intra-datacenter proxy	$1ms$	1K
	Forward proxy, nearby datacenter [86]	$10ms$	10K
IRT = 1μs	Forward proxy, remote datacenter [86]	$200ms$	200K
Network	Single cache line DRAM lookup [72]	$100ns$	<1
	Traversing a DRAM datastructure [72]	$500ns$	<1
	RDMA Access in GEM-switch [85]	$5\mu s$	2
IRT = 3μs	IDS with reverse DNS lookups [110]	$200ms$	67K
Storage	1MB SSD Disk Read [50]	$50\mu s$	2
	Hard Disk Seek & Read [50]	$3ms$	100
IRT = 30μs	Cross Datacenter Filesystem Read [50, 153]	$150ms$	5K

Table 2.1: Average inter-request times (IRT), typical latencies, and Z values seen today.

In recent years, Z has grown across a wide range of systems. For example, DRAM latencies are only marginally improving, while newer memory technologies (*e.g.*, High Bandwidth Memory, or HBM) boast order-of-magnitude improvements in bandwidth over current DDR standards [91]. Similarly, the latency between a CDN forward proxy and a central data center is defined by wide-area latencies; meanwhile, throughputs are rapidly growing, *e.g.*, with network links moving from 10 Gbps to 100 Gbps and 400 Gbps [59]. The fundamental problem is that latencies are edging marginally closer and closer to limits imposed by the speed of light, while throughputs keep growing unhindered. Hence, we believe that the impact of delayed hits on latency-minimizing caching systems will grow with time.

2.2 Impact of Delayed Hits

In this section, we highlight how delayed hits subvert both *our* ability to make sound, high-level deployment decisions, and our *algorithms'* capacity to realize the low-level caching outcomes they were designed for.

2.2.1 Misleading Operators

We observe that, in the presence of delayed hits, the classical caching model can mislead operators managing caching systems in two ways: (a) systematically underestimating request latencies, and (b) deploying the wrong caching algorithm.

Underestimating latency. Measuring cache hit-rate (HR), the efficacy metric prescribed by the classical model, provides operators useful insight into their deployments. For example, if a cache is deployed to reduce bandwidth consumption to a backing store (*e.g.*, a forward proxy in a bandwidth-limited network), then the miss rate, MR $= (1 - \text{HR})$ is proportional to the bandwidth consumption on the path to the backing store. Other caches are deployed to minimize latency. When assuming delayed hits do not exist (and that backing store latencies are uniform [56, 81, 94]), the average latency can be computed using the *Average Memory Access Time (AMAT)* [72] expression: HR $\times$ hit latency $+$ MR $\times$ miss latency (2.1)

introduce an additional queueing delay which further increases the latency due to delayed hits.

However, in the presence of delayed hits, the latency estimates derived from traditional hit-rate based models *underestimate true latency*. Some so-called "hits" will in practice experience latencies closer to the high latency of a miss than the low latency of a true hit. If delayed hits happened infrequently, the gap between the predicted latency derived from hit-rates using Equation 2.1 and true latencies would be marginal. But, in Figure 2.3, we show how the normalized average latency reported by a simulator that models delayed hits differs from one that does not. In our simulations, we scale up the latency to the backing store; on the X-axis we plot Z, the ratio of the backing store latency to average inter-request time (IRT).

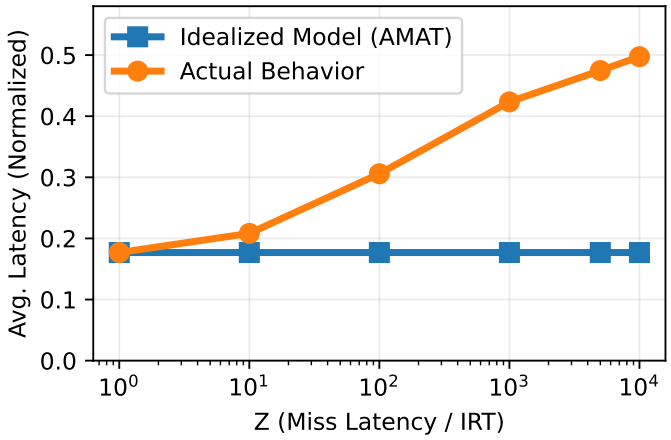


Figure 2.3: Normalized latency predictions with (*Actual*) and without (*Idealized*) accounting for delayed hits.

We see that, as the latency to the backing store increases (and, correspondingly, Z), so do the frequency and magnitude of delayed hits. At the rightmost point in the graph, there is a 63% error in the classical model's prediction relative to the actual latency cost incurred by the system. This is problematic in several contexts, but perhaps most so in content delivery networks (CDNs) where incorrect latency projections can significantly hurt the bottom-lines of their clients (*e.g.*, e-commerce platforms).

Deploying the wrong caching algorithm. A *caching algorithm* is an algorithm to decide, given a cache and a sequence of object requests, when and which objects to store in the cache, and when and which objects to evict. The choice of caching algorithm is an important (perhaps *the* most important) deployment decision for operators because it tangibly affects the metrics that they care about (*e.g.*, request latency, or bandwidth consumption).

	LFU	ARC	LHD
LRU	✗	✓	✓
LFU		✗	✓
ARC			✗

(a) CAIDA Chicago ($Z = 2K$)

	LFU	ARC	LHD
LRU	✗	✗	✓
LFU		✗	✓
ARC			✓

(b) CAIDA NYC ($Z = 2K$)

	LFU	ARC	LHD
LRU	✓	✓	✓
LFU		✗	✗
ARC			✓

(c) CDN ($Z = 100K$)

Figure 2.4: Pairwise comparisons between four online caching algorithms.

However, in the presence of delayed hits, the gap between a hit-rate derived estimate of latency and the true latency varies by trace and by algorithm. This means that comparisons of caching policies — even using real, not simulated systems — based on hit-rate measurements and Equation 2.1 rather than true measurements of latency may lead to incorrect conclusions about which caching algorithm is "better" for the system under consideration. Figure 2.4 depicts pairwise comparisons between four caching algorithms in three different scenarios.

✗s denote situations where choosing an algorithm on the basis of hit-rate would result in *worse* average latency. We find that in a third of the comparisons, not incorporating delayed hits into the system evaluation would lead one to make suboptimal decisions about the "right" caching algorithm, which in turn leads to higher average latency in practice.

2.2.2 Misleading Algorithms

We saw in §2.2.1 that, by discounting delayed hits, the classical caching model leads operators to mispredict latency and make suboptimal deployment decisions. However, there is a third, even more insidious problem lurking in our methodology for designing latency-minimizing caching algorithms today: *we are using the wrong oracle.*

Bélády's algorithm is not latency-optimal. One way to understand fundamental trade-offs in caching is by studying offline-optimal algorithms ("oracles"), which we assume to have perfect knowledge of future requests. Despite the fact that they cannot be deployed in practice, offline algorithms provide bounds and guidance for practical systems. For instance, if an offline, hit-rate optimal algorithm achieves a $k\%$ hit-rate, then we can be certain that any online algorithm will achieve *at most* $k\%$ hit-rate, and that we ought to provision our system accordingly. They also serve as a "North Star" in the design of online algorithms [25, 80, 138].

Bélády's algorithm [20] is a classic example of this. It is provably optimal at maximizing hit-rate in the basic setting where objects are all the same size [27] and backing store latencies are uniform. Further, since the AMAT expression in Equation 2.1 tells us that cache hit-rate is the only metric we ought to care about (latency is but a derived quantity), Bélády serves as the *de facto* caching oracle, regardless of whether the overarching caching objective is to reduce bandwidth demand to the backing store, or to minimize latency. The algorithm itself is simple: in choosing which object to evict from the cache, pick the one with the next request that is the *farthest in the future*; in other words, Bélády ranks cached objects in increasing order of their *Time to Next Access (TTNA)*, and evicts the object with the largest TTNA.

Thanks to its elegance and simplicity, most (if not all) caching algorithms designed and deployed today seek to emulate Bélády's decisions *as closely as possible*. For instance, the ubiquitous online caching algorithm, Least-Recently Used (LRU), assumes that recency (i.e., the time *since* the last access) is a good proxy for the future; more sophisticated algorithms such as Least Hit Density (LHD) approximate it by explicitly learning each object's reuse distance distribution and computing its conditional expectation. In either case, the expectation is that operating close to Bélády improves hit-rate, which in turn reduces latency.

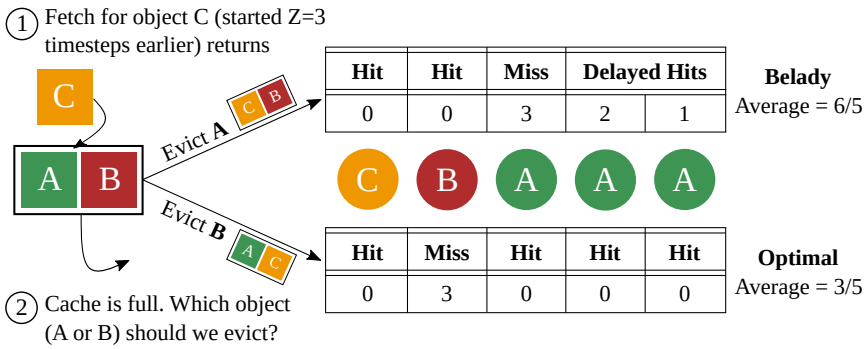


Figure 2.5: For the given trace, with a cache of size 2 and a Z value of 3, Bélády's algorithm chooses a latency-suboptimal schedule.

The problem, however, is that **in the presence of delayed hits, maximizing cache hit-rate is a fundamentally different problem than minimizing latency**; as a result, Bélády is *not* latency-optimal in the context of caching with delayed hits. We illustrate this point in Figure 2.5. In the example, the cache currently contains objects A and B, and ① a fetch for object C (initiated $Z = 3$ timesteps earlier) has just completed. Now, ② the cache must evict either A or B to accommodate C. Since B is accessed earlier than A, Belady's algorithm would choose to evict A. However, in our example, we see that there is a *burst* of requests to A, resulting in a series of delayed hits with several requests to A experiencing higher latencies. An algorithm that evicts B instead of A experiences a single miss corresponding to B, but all of the subsequent requests to A would have been true hits, resulting in lower average latency.

2.3 Latency Minimization in the Delayed Hits Model

The findings in §2.2 indicate that, when our ultimate goal is to minimize latency, hit-rate is in fact the **wrong** metric to optimize for, and Bélády's algorithm is **not** the right oracle to follow. This raises the question: how *do* we minimize latency in the presence of delayed hits? Answering this question turns out to be more challenging that one might think. In this section, we first present an example illustrating why *ad hoc* methods for dealing with delayed hits do not work (§2.3.1). This motivates the need for more systematic approach, which we subsequently present in §2.3.2.

2.3.1 A Case for Principled Analysis

To illustrate the challenge presented by delayed hits, we present an example where the right decision highly depends on Z. Intriguingly, we find that, as Z increases, the right caching schedule (i.e., sequence of decisions) can change entirely. The example consists of the following sequence of requests to objects A and B, which is repeated indefinitely. Requests in yellow (labelled X) denote empty time slots.

We assume a cache of size 1 which either caches A or B. We consider four different Z values corresponding to the following fetch delays (L): 1 ms, 5 ms, 17 ms, and 22 ms (assuming the inter-request time, R, is 1 ms). For each Z value, we calculate the latency achieved by three algorithms: (a) always caching the "bursty" object, A, (b) always caching the paced object, B, and (c) LRU. A green box denotes the lowest latency for each value of Z.

Algorithm	$Z = 1$	$Z = 5$	$Z = 17$	$Z = 22$
Cache Bursty, A	0.5 ms	1.9 ms	4.3 ms	6.0 ms
Cache Paced, B	0.5 ms	1.5 ms	7.5 ms	5.5 ms
LRU	0.2 ms	2.2 ms	5.9 ms	6.6 ms

We find that, while LRU is latency-optimal for $Z = 1$, the paced algorithm is optimal for $Z = 5$. For $Z = 17$, the bursty algorithm becomes latency-minimizing (albeit not optimal), and for $Z = 22$, the paced algorithm is latency-optimal once again. The difference in latencies is significant (between $1.1\times$ and $2.5\times$), even for this simple example. We conclude that any traditional algorithm (or *ad-hoc* design approach thereof), which only considers the sequence of requests but does not systematically model delayed hits, cannot expect to achieve good latencies in practice. In fact, even an educated guess, *e.g.*, preferring bursty flows — which suffer especially under delayed hits — does not consistently lead to the right strategy.

To further complicate matters, parallel work in our group [97] shows that the latency objective for the delayed hits caching problem is *not* antimonotone.[3] Consequently, a caching algorithm that *improves* average latency under delayed hits might actually *lower* the true hit-rate. In fact, it might even *increase* the miss-rate (i.e., inflate the fraction of requests relegated to the backing store). This result further corroborates the point that optimizing for latency is a fundamentally different problem than optimizing for hit- or miss-rates. It also has implications for the bandwidth consumption of latency-minimizing caching algorithms, which we discuss further in §2.6.4.

2.3.2 Problem Formulation

In this section we give a formal definition of the latency minimization problem for caching with delayed hits. We consider a cache of size C and M objects indexed by $i \in [M]$. We are given a sequence object requests, where $\sigma(T)$ denotes the object requested at timestep T with $T = 0, 1, \ldots, N$. We use the following quantities to describe the state of the system at the beginning of each timestep T. For each object i, let

$$x_0^{(i)}(T) = \mathbb{1}_{\{\text{object } i \text{ is in the cache at } T\}}, \tag{2.2}$$

$$x_\tau^{(i)}(T) = \mathbb{1}_{\substack{\{\text{object } i \text{ was requested at } T-(Z+1-\tau) \\ \text{and the request has not been resolved}\}}} \quad \tau = 1, \ldots, Z. \tag{2.3}$$

Here, when an object i is requested but cannot be resolved immediately, we say that we put it in a queue. So (2.3) describes the state of the queue for i.

We specify a cache schedule using the following decision variables. Let $a_i(T)$ be defined by

$$a_i(T) = \begin{cases} 1 & \text{if object } i \text{ is admitted to cache at } T, \\ -1 & \text{if object } i \text{ is evicted from cache at } T, \\ 0 & \text{if no action is taken on object } i \text{ at } T. \end{cases} \tag{2.4}$$

[3] For a request sequence of size T, we can encode a cache schedule as a *hit vector* of boolean values, $b \in \{0,1\}^T$, where $b_i = 1$ if the i'th request experienced a *true* hit, and $b_i = 0$ otherwise (i.e. delayed hit, or miss). Then, we can define a latency function, $l : \{0,1\}^T \to \mathbb{R}$, such that $l(b)$ represents the total latency for schedule b. We say that l is antimonotone if, for every pair of schedules $b, b' \in \{0,1\}^T$, where $b'_i \geq b_i \forall i$, it holds that $l(b') \leq l(b)$. Perhaps surprisingly, [97] shows that this is *not* the case, implying that it is sometimes preferable to forgo a true cache hit in order to achieve lower latency.

To make sure $a_i(T)$ with $i \in [M], T = 0, 1, \ldots, N$ form a valid cache schedule, we enforce the following constraints for each object $i \in [M]$ and timestep $T = 0, 1, \ldots, N$:

- An object can be admitted only when its data arrives:

$$\mathbb{1}_{\{a_i(T)=1\}} \leq x_1^{(i)}(T) \tag{2.5}$$

- An object can be evicted only when it is already in the cache:

$$\mathbb{1}_{\{a_i(T)=-1\}} \leq x_0^{(i)}(T) \tag{2.6}$$

- The schedule should guarantee that the number of objects in the cache is no larger than the cache size C:

$$\sum_{i \in [M]} x_0^{(i)}(T) \leq C \tag{2.7}$$

  Although it seems that this is a constraint on the state, it is in fact a constraint on the cache schedule since the state at the current timestep is determined by the past decisions. This will become clear after we describe the relation between the state and the schedule next.

We can then write out how the system state evolves over time as follows:

- The data that just arrived resolves the requests for the same object in the queue, and other requests move forward in queue:

$$x_\tau^{(i)}(T+1) = x_{\tau+1}^{(i)}(T) \cdot (1 - x_1^{(i)}(T)), \quad i \in [M], \tau = 1, \ldots, Z-1, T = 0, 1, \ldots, N-2 \tag{2.8}$$

- The admission or eviction of an object changes the state in the cache:

$$x_0^{(i)}(T+1) = x_0^{(i)}(T) + a_i(T), \qquad i \in [M], T = 0, 1, \ldots, N-2 \tag{2.9}$$

- The new request comes in and is added to the queue if the requested object is not in the cache:

$$x_Z^{(i)}(T+1) = \mathbb{1}_{\{\sigma(T)=i\}} \cdot (1 - x_0^{(i)}(T+1)), \qquad i \in [M], T = 0, 1, \ldots, N \tag{2.10}$$

It can be proven that the state that evolves according to the dynamics above satisfies that for any $T = 0, 1, \ldots, N-2$,

$$x_0^{(i)}(T) + \mathbb{1}_{\{\sum_{\tau=1}^{Z} x_\tau^{(i)}(T) > 0\}} \leq 1 \tag{2.11}$$

This inequality states the fact that if object i is in the cache, then there will not be requests for i in the queue, and if there are requests of object i in the queue, then i is not in the cache.

At timestep T, object $\sigma(T)$ is requested. If it is not in the cache nor requested during the past Z timesteps, it will trigger a sequence of delayed hits when $\sigma(T)$ is requested again during the next Z timesteps. Therefore, the total latency can be written as:

$$\sum_{T=0}^{N-2} x_Z^{(\sigma(T))}(T+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x_\tau^{(\sigma(T))}(T+1)\right) \cdot \sum_{t=0}^{Z-1} \mathbb{1}_{\{\sigma(T+t)=\sigma(T)\}} \cdot (Z-t) \tag{2.12}$$

Then the *latency minimization problem* is to find the cache schedule subject to the constraints (2.5)–(2.7) such that the resulting states minimize the total latency in (2.12).

2.4 BELATEDLY: Offline, Latency-Optimal Caching

As we saw in §2.2.2, Bélády, the offline hit-rate maximizing caching algorithm, fails to minimize latency in the presence of delayed hits. In this section, we find the answer to the latency-minimization question by reducing it to a Minimum-Cost Multi-Commodity Flow (MCMCF) problem. We present BELATEDLY, an offline caching algorithm we designed to minimize latency given delayed hits. With this new oracle, we can measure the gap between Bélády and true latency-optimality. Furthermore, BELATEDLY generates a latency-optimal cache schedule which we will later use to guide the design of a practical, online algorithm (MAD).

A latency-minimizing cache schedule minimizes the mean latency of all requests, where latency $= 0$ upon a true cache hit, latency $\in (0, Z)$ upon a delayed hit, and latency $= Z$ upon a miss. In §2.4.1, we show that the latency-minimization problem is equivalent to an MCMCF problem. However, computing integer solutions to MCMCF problems is known to be NP-Complete, and naively implementing the algorithm involves a significant number of decision variables.

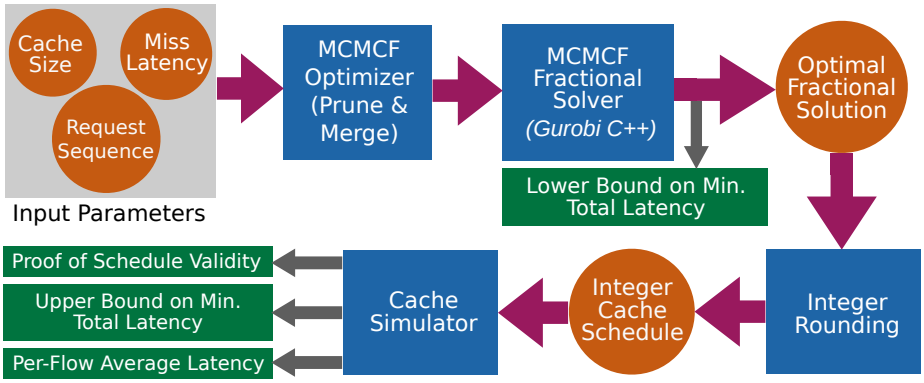


Figure 2.6: The BELATEDLY pipeline for computing bounds on latency-optimal cache schedule using MCMCF reduction.

To make the problem tractable enough to compute over our empirical datasets, we apply two optimizations: (a) we "prune" and "merge" states in the MCMCF graph using *a priori* insights about caching, and (2) we configure our MCMCF solver (Gurobi [109]) to solve for a *fractional* solution, which can be found in polynomial time, and then perform randomized integer rounding [27, 117] to recover a valid caching schedule. We present the details of these optimizations in §2.4.3, and summarize their impact on BELATEDLY's performance in §2.4.4. The BELATEDLY pipeline is depicted in Figure 2.6.

2.4.1 Network Flow Formulation

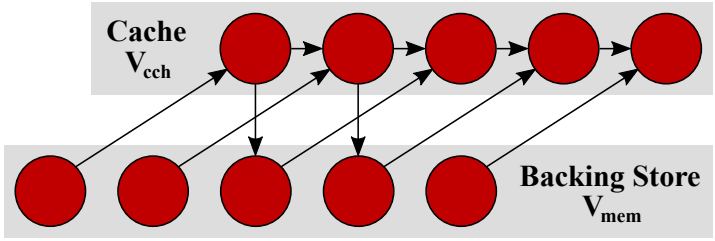
We first describe our MCMCF formulation, BELATEDLY. For the sake of readability, we defer the proof of equivalence between latency-minimizing caching and BELATEDLY to the end of this section (§2.4.5).

Overview: MCMCF is a classic network flow problem and a generalization of Min-Cost Flow (MCF) [4]. Min-Cost Flow involves a set of *sources* and *sinks* embedded in a larger graph; every edge in the graph has a *capacity* representing the maximum amount of flow which may traverse that edge. A solution to MCF must route *flow* from the sources to the sinks without exceeding any individual edge capacity. Furthermore, each edge is also associated with a *cost*. The ultimate goal of Min-Cost Flow is to route flow across the edges *such that the total cost of all traversed edges is minimized*. MCMCF adds an additional twist to the problem: flows are associated with a *commodity*, and edges may have different costs for different commodities.

Our reduction from minimum latency caching to MCMCF constructs a commodity for each object requested from the cache. Vertices in the graph represent either that the object is in the cache, or that it is in the backing store; edges between vertices represent the object entering the cache, remaining in the cache, or being evicted from the cache. Weights along edges represent the latency cost of misses and delayed hits. By minimizing the weights of traversed edges, MCMCF equivalently computes a cache schedule with a minimal latency cost.

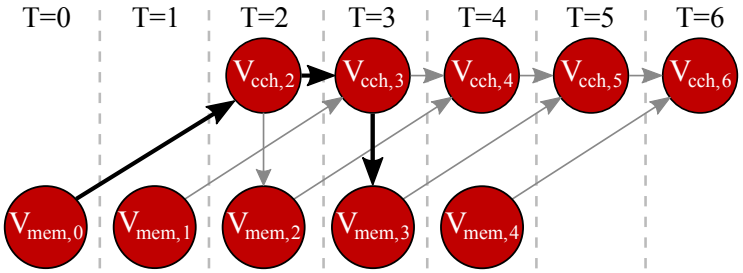
A key component in this formulation is the *costs* we assign to edges in the flow network, which reflect the *true* latency costs of misses. Our main finding is that the right costs to assign are the "aggregate delays". Specifically, the aggregate delay of a miss is the total delay of the miss and all the delayed hits within a time window of Z of the miss (see Equation 2.13 for the mathematical definition). *This notion of aggregate delay influences the design of our online algorithms, discussed in §2.5.*

Construction of the Flow Network: BELATEDLY operates on a *flow network*, a directed graph consisting of a set of vertices and edges. In our formulation, the vertex set, V, consists of two types of vertices, which we draw as two rows. The bottom and top rows represent the backing store and the cache, respectively. We refer to the set of *backing store* nodes as V_{mem}, and the set of *cache* nodes as V_{cch}.



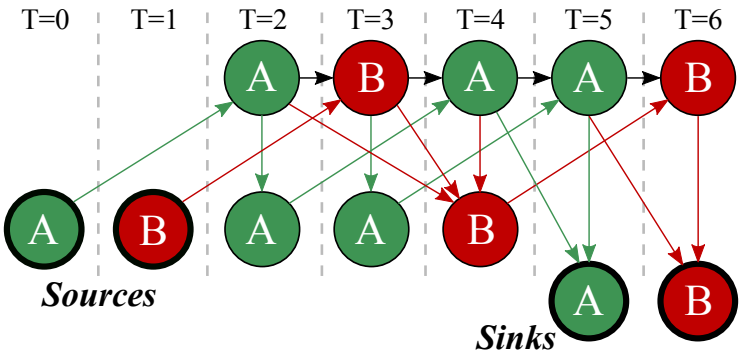
Note that the rows are slightly *offset*. This is because we index each row by time, and have vertices for each timestep. For the bottom row, we have one vertex for each timestep $T = 0, 1, \ldots, N-1$. We denote these vertices as $V_{\text{mem},T}, T = 0, 1, \ldots, N-1$. For the top row, we duplicate the vertices in the bottom row, but shift them to the right by Z timesteps as shown in the figure below. We denote the vertices in the top row by $V_{\text{cch},T}, T = Z, 1+Z, \ldots, N-1+Z$. In the figure below, $Z = 2$.

Flow moving along an edge represents an object moving in and out of the cache. In the following figure, an object is requested at time $T = 0$, arrives in cache at time $T = 2$, and is evicted at time $T = 3$.



Since there are multiple objects, we view each object as a commodity and index them by $i \in [M]$, where M is the number of objects and $[M] = \{1, 2, \ldots, M\}$. We also say $\sigma(x)$ is the object requested at time x. We have 1 unit of demand for each object. The source vertex for each object, i, is the vertex V_{mem,T_i} where T_i is the first timestep at which i is requested. We also add a sink vertex for each object i in the bottom row, denoted by $V_{\text{sink}}^{(i)}$.

At a high level, each node in the bottom layer represents the time of request to exactly one object; we construct an edge from $V_{\text{mem},t}$ to $V_{\text{cch},t+Z}$ to allow the flow for that object to move from the backing store to the cache. In the top layer, each node $V_{\text{cch},t+Z}$ represents the request from time t being served. Objects may stay in the cache by following edges from some $V_{\text{cch},n}$ to the next $V_{\text{cch},n+1}$ — all nodes in $V_{\text{cch},}$ have an edge to the subsequent cache node. To leave the cache, an object follows an edge from some $V_{\text{cch},n}$ to some $V_{\text{mem},x}$ for x, the next time ($\geq n$) the same object is requested — hence all nodes in $V_{\text{cch},}$ have M edges back to $V_{\text{mem},}$ nodes, one for each object that *could* be evicted at this point. If there is no further request to an object, the edge points to the *sink* node for that object rather than $V_{\text{mem},x}$. We illustrate the request sequence $\{A,\ B,\ A,\ A,\ B\}$ for objects A and B:

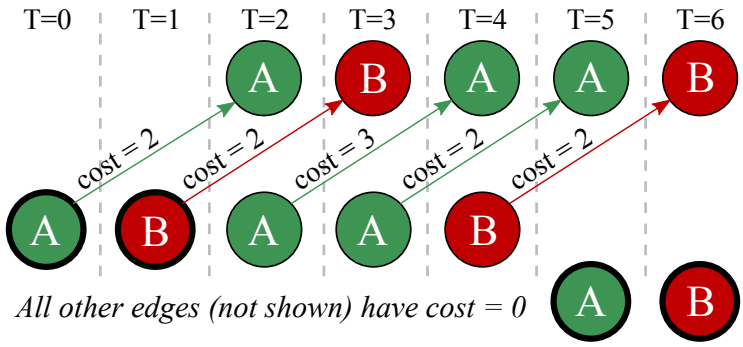


Looking at the above figure, it is obvious that some edges will never be taken (*e.g.*, $V_{\text{cch},2}$ has an edge to $V_{\text{mem},4}$ despite the fact that it is impossible for flow corresponding to object B to have reached $V_{\text{cch},2}$). We discuss pruning superfluous edges and merging nodes to improve performance in §2.4.3.

The last features to add to our construction are *capacities* and *costs* along edges to ensure that each object's flow obeys a valid caching schedule that minimizes latency. For example, we want to prevent all objects simply following the edges $(V_{\text{cch},n}, V_{\text{cch},n+1})$ for the entire duration and exceeding the cache capacity. No more than *capacity* flows may traverse an edge, and our solver will try to minimize the *total cost* of routing flow across these edges. We assign capacity and cost to edges as follows:

- $(V_{\text{cch},n}, V_{\text{cch},n+1})$ edges (which represent staying in the cache) are assigned capacity C, and the cost of routing flow across them is 0. This models the fact that staying in the cache does not increase latency, but the cache can only hold C objects at the same time.
- $(V_{\text{cch},n}, V_{\text{mem},x})$ edges (which represent evicting an object whose next request is at $T = x$) are assigned capacity 1, and the cost is ∞ for all commodities *except* $\sigma(x)$ (the object requested at time x), for which the cost is 0. This prevents objects from exiting the cache along edges for a different object. Intuitively, the action of eviction itself does not incur a latency cost. But it forces the object out of the cache so the next request for the object and the corresponding delayed hits will experience non-zero latencies.
- $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ are the edges that represent bringing an object into the cache, which happens when there is a miss. It is here that we encode delayed hit latency into the cost. The capacity of $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is 1, and the cost is ∞ for all objects other than $\sigma(T)$. The cost of routing $\sigma(T)$ along $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is the *aggregate delay* for requests of object $\sigma(T)$ while the data is being fetched; i.e., it is the *total latency* for the miss plus all requests that arrive during the delayed hits. The miss experiences a latency of Z, and a delayed hit that arrives t timesteps after the miss experiences a latency of $Z - t$. Therefore, the cost is:

$$Z + \sum_{t=1}^{Z-1} \mathbb{1}_{\{\sigma(T+t)=\sigma(T)\}} \cdot (Z - t) \tag{2.13}$$



In the above figure, the cost for all edges is 2 (the latency Z to the backing store) except for the edge $(V_{\text{mem},2}, V_{\text{cch},4})$. Because A is also requested at $T = 3$, it will be queued and later served by the request being fetched; as such, we need to account for *both* the cost of serving the request at $T = 4$ (which is 2) and the request at $T = 3$ (which is 1).

Routing Flows: The MCMCF problem is to then find routes for the objects such that the total routing cost is minimized. Specifically, the routes are represented by flow variables, where each flow variable represents whether an object/commodity is routed along an edge or not. Here flow variables need to satisfy link capacity constraints and flow conservation constraints, which will guarantee that the flow variables can be converted to a valid cache schedule.

Equivalence to Latency-Minimizing Caching.

Theorem 1. *BELATEDLY's underlying MCMCF formulation is equivalent to the latency minimization problem (§2.3.2).*

The detailed proof of Theorem 1 can be found in §2.4.5. Both the MCMCF problem and the latency minimization problems are optimization problems. To show that these are equivalent,[4] we first show in Lemma 1 that the feasible set of flow variables is "equivalent" to the feasible set of caching schedules (i.e., from any feasible cache schedule, we can define a set of flow variables that are also feasible for the MCMCF problem, and vice versa).

Lemma 1. *Given a sequence of object requests, there is a bijection between the set of feasible flow variables and the set of feasible cache schedules.*

Once we have this bijection, we can show that the objective functions of these two problems are the same. With equivalent feasible sets and objective functions, the MCMCF problem and the latency minimization problem are thus equivalent.

2.4.2 Delayed Hits and Empirical Latencies

We now evaluate BELATEDLY's latency estimates relative to Bélády for a range of scenarios.

BELATEDLY provides significantly better average latency than Bélády for today's highest latency systems. In Figure 2.7, we plot Bélády's percent error relative to the optimal upper-bound provided by BELATEDLY.[5] For the highest latencies — referring to Table 2.1, those with Z values in the thousands — Bélády deviates from the optimal by 9–37%. However, for more modest latencies to the backing store, BELATEDLY does not have noticeably lower latencies than Bélády. Even in the original FPGA-based switching scenario which caused us to detect delayed hits, the gap between Bélády and BELATEDLY is less than 1%.

Z is correlated with an increasing gap between Bélády and BELATEDLY. In Figure 2.8 we see that for all three datasets, Bélády performs progressively worse with respect to true latency optimality as Z increases — until Z moves past $10K$. The growth correlation follows intuition: as Z grows, there are more chances for delayed hits to occur, and hence more opportunities for Bélády to err. We find that narrowing of the gap between Bélády and BELATEDLY beyond

[4] At this juncture, one might ask: why bother with MCMCF instead of solving the latency minimization ILP directly? The answer is two-fold. First, it is the network flow formulation that allows us to implement the optimizations described in §2.4.3; without these, even modest LP instances of the problem are too compute- or memory-intensive for today's solvers. And second, formulating the problem as an MCMCF naturally leads to the notion of aggregate delay; as we show in §2.5, this is a key component of our online algorithm.

[5] $\frac{(\text{Bélády} - \text{BELATEDLY})}{\text{BELATEDLY}} \times 100\%$

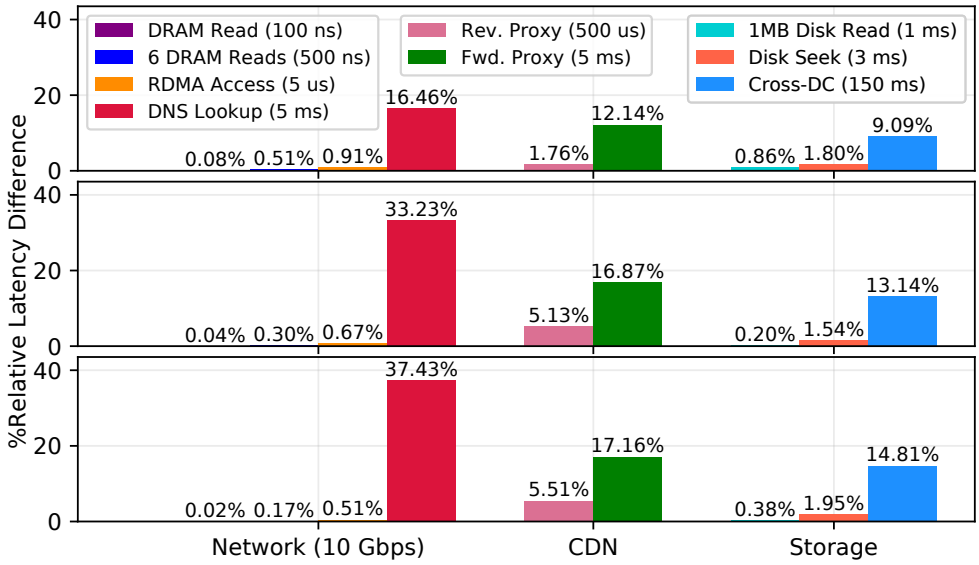


Figure 2.7: Latency gap between Bélády and BELATEDLY for different application scenarios (Network, CDN, Storage) today. Top to bottom: 1%, 5%, and 10% cache sizes.

$Z = 10K$ is an artifact of our simulation duration; since Z is large relative to the size of the trace ($250K$ requests), it also exceeds the duration of most flows. As a result, most requests experience 'forced' cache misses, raising the latency baseline and giving BELATEDLY fewer opportunities to make meaningful caching decisions.

The gap between Bélády and BELATEDLY varies with cache size. In Figure 2.9, we see that the latency difference first rises, then falls as the cache size increases. When the cache is extremely small, neither BELATEDLY nor Bélády's caching decisions have significant impact on latency (since most requests experience cache misses, the average latency is close to the full latency of a cache miss); similarly, as the cache capacity becomes very large, both strategies can afford to simply cache all or almost all objects (the extreme case being a cache large enough to fit all concurrent flows or active objects). In between, however, all three datasets "peak" at different points. In particular, the Network trace has a sharp spike at 10%, while the CDN and Storage traces have more gradual curves.

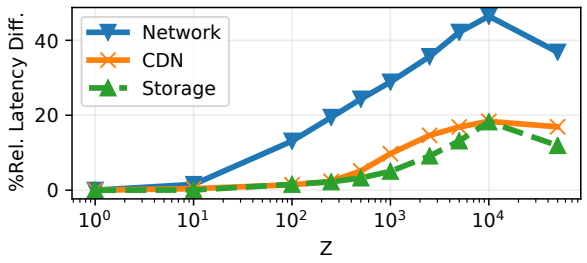


Figure 2.8: %Relative latency difference between Bélády and BELATEDLY as a function of Z. Using cache size, $c = 5\%$ (expressed as a percentage of the maximum number of concurrent flows).

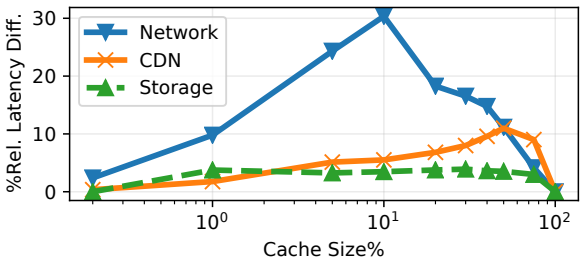


Figure 2.9: %Relative latency difference between Bélády and BELATEDLY as a function of cache size. Using $Z = 500$.

BELATEDLY's caching decisions are correlated with the burstiness of requests. The Goh-Barabasi Score [66] is a statistical measure of *burstiness* in a sequence of events. A score of "1" reflects many arrivals in a short period of time (a *request train*) followed by longer periods with no requests. A score of "-1" represents a perfectly paced stream of arrivals with one request every fixed number of timesteps. In Figure 2.10, we see that bursty traffic (with a high Goh-Barabasi score) incurs a lower percent latency relative to Bélády. This suggests that burstiness may be a worthwhile candidate for consideration in the design of *online* algorithms that optimize for latency in the context of delayed hits. It is this observation that guides us in the development of our online strategy, and we discuss it in more detail in the following section.

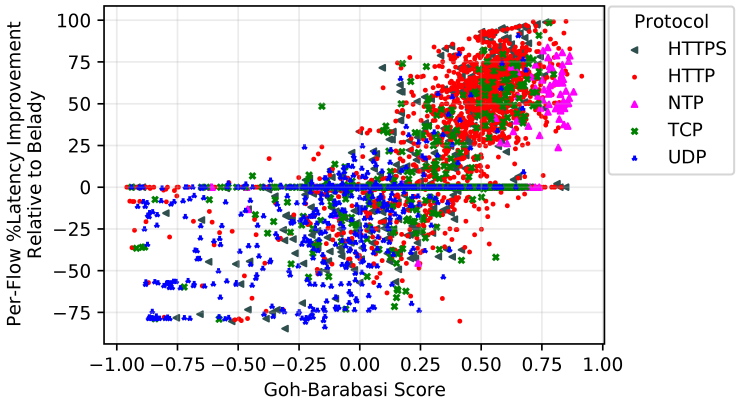


Figure 2.10: Relative latency improvement *vs* burstiness (for Network traffic). Bursty flows suffer less under BELATEDLY.

2.4.3 Optimizations to Reduce Complexity

In this section, we provide a few key implementation details for BELATEDLY.

Pruning and Merging

While the MCMCF formulation is conceptually simple, a naive implementation of the algorithm has serious practical limitations. Observe that the number of flow variables in the MCMCF formulation is $O(N \cdot M)$. For a request sequence of size $N = 250,000$ containing $M = 20,000$ objects, the number of decision variables alone would be on the order of 10^{10}. Further, the total number of flow conservation constraints is $O(N \cdot M)$ (see (2.18)–(2.21)). In Gurobi, where decision variables are encoded as 64-bit floating-point values, and constraint expressions as vectors of 64-bit pointers to the relevant decision variables, simply encoding the model would require well over 400 GB of memory.

Here, we describe two optimizations to the above formulation that allow us to significantly tighten the resource requirements (memory and execution time) for solving the MCMCF problem and to make it more tractable. Our goal is to be able to compute BELATEDLY on a 32-core x86 server with 128 GB of RAM, for request sequences containing $N \approx 250,000$ requests, $M \approx 50,000$ objects, and any combination of z and C.

Caching Intervals. Since the majority of decision variables stem from either $(V_{\text{cch},n}, V_{\text{cch},n+1})$ (cache-to-cache) or $(V_{\text{cch},n}, V_{\text{mem},x})$ (cache-to-memory) edges, we first attempt to reduce the number of elements in these sets. The key idea here is that, for each object, the request sequence can be partitioned into disjoint intervals (composed of one or more consecutive timesteps) where BELATEDLY is never incentivized to change its caching decision for that object; we call these *caching intervals*.

To concretize this notion, consider the subproblem depicted in Figure 2.11. Per the original MCMCF formulation, there are four distinct decision variables on edges between cache vertices corresponding to a (denoted by x_1, x_2, x_3, and x_4). Now, consider the possibility of routing flow along edges labeled **1**, **2**, and **3**. All three edges have the same capacity, cost-per-unit-flow, and destination node. Effectively, the latency cost incurred by evicting a using any of these edges is identical. However, observe that routing a's flow along edge **2** involves keeping a in the cache for one timestep longer than routing it along edge **1**. Similarly, routing a's flow along edge **3** involves keeping it in the cache for two additional timesteps. Since deferring the eviction consumes valuable cache space (but yields no tangible benefit in terms of latency cost), it is *strictly better* to evict a using edge **1** (at timestep $t + 4$) than using edges **2** or **3**.

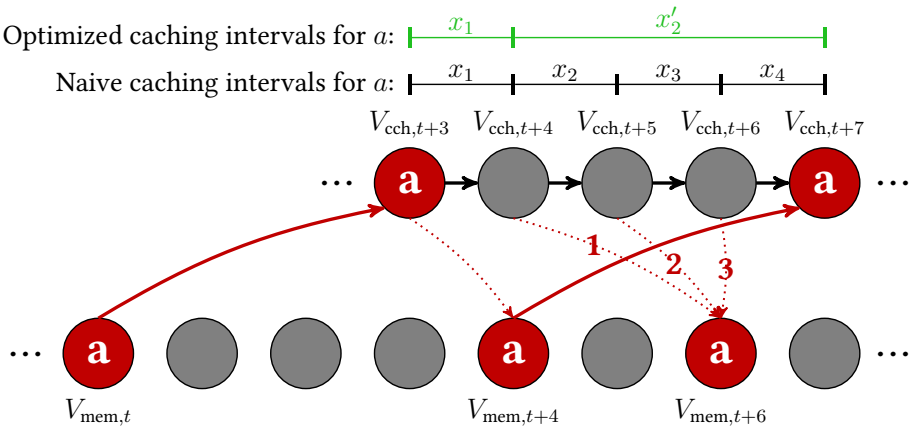


Figure 2.11: A fragment of a request sequence highlighting nodes and edges corresponding to object a (colored red), with $Z = 3$.

This simple observation gives us three major optimization opportunities, enabling us to:

- Eliminate the redundant edges **2** and **3** (along with the corresponding decision variables).
- Replace x_2, x_3, and x_4 with a single decision variable, x'_2. Since edges **2** and **3** no longer exist, any flow entering $cch_{(t+4)}$ must remain in the cache until $cch_{(t+7)}$; in other words, BELATEDLY's caching decision remains the same for the entire duration of the interval $[(t + 4), (t + 7))$.
- Eliminate flow conservation constraints involving object a for nodes $cch_{(t+5)}$ and $cch_{(t+6)}$. In the new representation, for each object, i, we only need flow conservation constraints for V_{cache} nodes corresponding to the end-points of i's caching intervals.

Lastly, this representation also allows us to bound the total number of caching intervals for any request sequence. Let n_i denote the number of requests to object i in a given request sequence of size N. Observe that an endpoint of object i's caching intervals is a V_{cch} node that either corresponds to i being admitted into the cache, i being evicted from it, or both. Since there are exactly n_i admission edges corresponding to object i, there must be *at least* n_i endpoints (or, equivalently, $n_i - 1$ intervals) corresponding to i. Conversely, in the worst case, there are $n_i - 1$ additional V_{cch} nodes which have eviction edges corresponding to object a. Thus, there may as many as $2n_i - 1$ unique endpoints (or, equivalently, $2n_i - 2$ caching intervals) corresponding to i. The total number of caching intervals (for all objects), K, can then be bounded as follows:

$$\sum_{i \in [M]} (n_i - 1) \leq K \leq \sum_{i \in [M]} 2(n_i - 1)$$
$$\Rightarrow N - M \leq K \leq 2(N - M).$$

For a fragment of an empirical trace (CAIDA Chicago, 2014) containing $N = 250,000$ packets and $M = 37,725$ objects (unique connections), the total number of caching intervals is on the order of $400,000$. Compared to the naive formulation, this optimization reduces the number of decision variables from 18×10^9 to 10^6, and model constraints from 9×10^9 to 10^6.

Optimizing Away Backing Store Nodes. Partitioning the global set of nodes into cache nodes and backing store nodes is a convenient abstraction since it allows us to reason about cache evictions and admissions independently of one another. Unfortunately, this representation also adds considerable overhead: excluding sink nodes, there are N backing store nodes, each of which contributes one decision variable on an edge $(V_{\text{mem},T}, V_{\text{cch},T+Z})$, as well as one flow conservation constraint. However, observe that, in our MCMCF formulation, any flow entering a $V_{\text{mem},T}$ node *must* be routed to the corresponding cache node, $V_{\text{cch},T+Z}$. This leads us to our next optimization: replacing pairs of cache eviction and admission edges of the form $(V_{\text{cch},T}, V_{\text{mem},x})$ and $(V_{\text{mem},x}, V_{\text{cch},x+Z})$ with a single edge $(V_{\text{cch},T}, V_{\text{cch},x+Z})$ with unit capacity and cost $c^{(i)}(V_{\text{cch},T}, V_{\text{cch},x+Z}) = c^{(i)}(V_{\text{mem},x}, V_{\text{cch},x+Z})$ for object i.

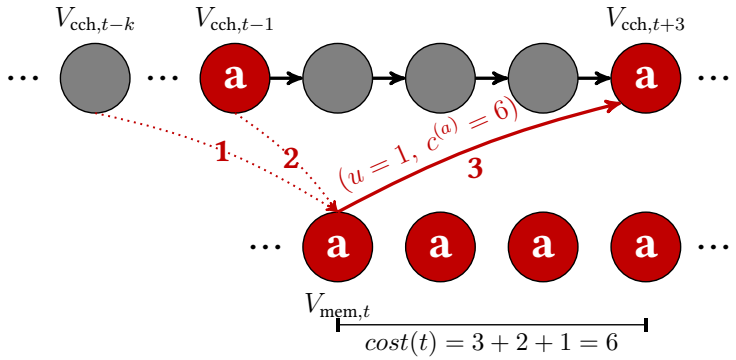


Figure 2.12: A fragment of a request sequence highlighting ingress and egress edges for node $V_{\text{mem},t}$, with $Z = 3$.

As an example, consider the subproblem depicted in Figure 2.12. Here, $V_{\text{mem},t}$ has two in-edges, labelled **1** and **2**, and one out-edge, labeled **3**. Using the optimization strategy discussed above, we can coalesce edges **1** and **3** into a single edge, **1'**, with a capacity of 1 and a cost-per-unit-flow of $c^{(a)} = 6$. Similarly, we can coalesce edges **2** and **3** into a single edge, **2'**. This effectively disconnects node $V_{\text{mem},t}$ from the remainder of the flow graph, and we can safely remove it from V. A visual representation of the optimized flow graph is depicted is Figure 2.13. Consequently, this optimization:

- Eliminates N decision variables corresponding to all N backing store to cache edges.
- Eliminates N flow conservation constraints corresponding to backing store nodes (*excluding* sink nodes).

For the aforementioned empirical trace, this optimization reduces the total number of decision variables and model constraints by another 25% (down to $750,000$ each). Overall, the optimized MCMCF formulation (expressed in Gurobi C++ format) occupies under 25 GB of memory.[6]

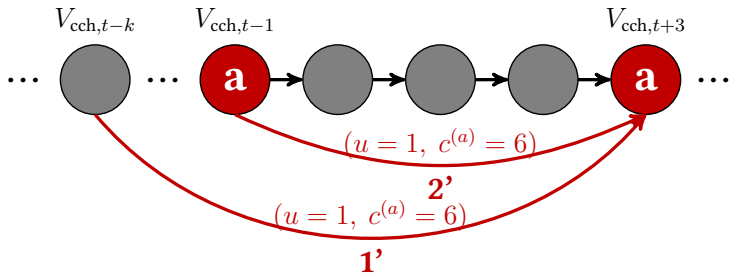


Figure 2.13: The optimized representation with backing store nodes removed.

Rounding to Approximate Integer Solutions

Recall that, since the integer version of MCMCF is NP-Complete, we instead opt to solve a fractional (or *relaxed*) version of the problem by removing the integrality constraints. However, this often results in solutions that do not map on to realistic caching strategies.[7] In this section, we describe our methodology for extracting an implementable caching schedule from a fractional solution.

A naive, yet intuitive, strategy is to simply round any non-zero fractions of evicted flows to 1, thereby always creating enough space in the cache for the next object to be admitted; unfortunately, this greedy rounding strategy does not generally work. It is easy to construct request sequences where evicting *too much flow* results in a violation of the cache capacity constraint several timesteps later. Further, attempting to satisfy the constraint by randomly evicting objects causes the upper-bound on the latency cost to diverge significantly from the true optimum.

[6]Includes overheads due to Gurobi's internal data-structures; the raw model itself is significantly more compact.
[7]The optimal fractional solution may involve caching *half* an object, which is not particularly meaningful.

BELATEDLY addresses this problem in two ways:

1. Instead of rounding *all* non-zero evicted flow fractions to 1, rounding is done with a probability corresponding to the fraction itself (a form of *randomized* rounding). In other words, if the fraction of flow for object i evicted at timestep T is $f^{(i)}_{evict}(T) \in [0, 1]$, then we perform eviction with probability $f^{(i)}_{evict}(T)$. This ensures that, in expectation, the cache occupancy at any timestep is equal to the total flow routed along the corresponding edge in the MCMCF solution.

2. While randomized rounding works well in theory, it does not *guarantee* that the cache capacity constraint is satisfied. In order to enforce this, we introduce the notion of *flow balance*. The idea is to track the *expected* amount of cached flow for each object (according to the fractional solution); then, at any timestep, if the cache occupancy would exceed the cache size, we evict the flow that is most *unbalanced* (deviates the most from its expected cached fraction). In practice, this is implemented using a priority queue.

2.4.4 Performance Evaluation

In the previous subsection (§2.4.3), we described two optimizations to make the MCMCF problem tractable: pruning and merging states in the flow graph to reduce the number of decision variables, and solving a "relaxed" version of the problem, followed by integer rounding, to ensure that the algorithm terminates in polynomial time. In this section, we evaluate the benefits of these optimizations (using the naive MCMCF formulation as a baseline), as well the impact of rounding on BELATEDLY's latency upper-bound.

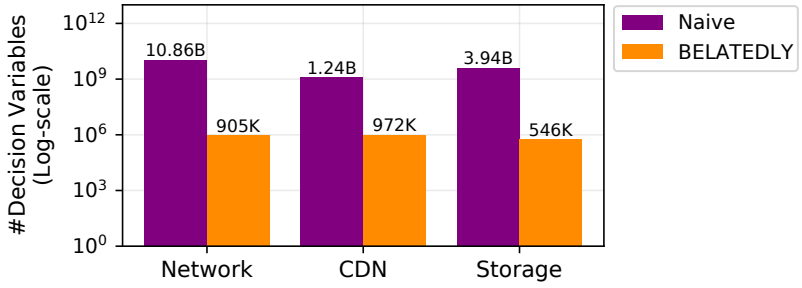


Figure 2.14: Number of decision variables in the naive MCMCF formulation versus BELATEDLY for different application scenarios.

Our optimizations to the original MCMCF formulation reduces BELATEDLY's memory and compute requirements by orders of magnitude. In Figure 2.14 we count the number of decision variables in the MCMCF formulation given our naive construction (§2.4.1) and our pruned version (§2.4.3). For all three application scenarios, the number of decision variables is reduced by three to four orders of magnitude.

Empirically, the formulation provides tight bounds. While solving a 'relaxed' version of the problem only gives us a lower-bound on the total latency (and not an implementable schedule), our randomized rounding strategy and flow balance heuristics work well in practice. For

each application scenario, we perform 20 runs of BELATEDLY sweeping different Z values and cache sizes. Across all three scenarios, we see a median error of at most 0.05% and a maximum error of 1.71%. Table 2.2 lists the relative error between the upper- and lower-bounds of the solution generated by BELATEDLY.

	Mean Error%	Median Error%	Max. Error%
Network	0.017	0.004	0.124
CDN	0.325	0.051	1.707
Storage	0.015	0.007	0.072

Table 2.2: Empirical bounds on BELATEDLY's error (calculated by comparing the integer upper-bound to the relaxed lower-bound).

2.4.5 Proof of Optimality

In this subsection, we show that BELATEDLY is latency-optimal[8] by proving equivalence between its underlying MCMCF formulation and the latency-minimization problem described in §2.3.2. We start with some notation for the flow variables used in the MCMCF formulation.

Notation

We define a flow variable for each object on each edge, which takes values from $\{0, 1\}$ and represents the fraction of flow for the object routed along that edge. In particular, we define the flow variables below:

$$\begin{aligned} f^{(i)}_{\text{mem}}(T)&: \text{ object } i \text{ along edge } (V_{\text{mem},T}, V_{\text{cch},T+Z}),\\ &\quad T = 0, 1, \ldots, N-1-Z;\\ f^{(i)}_{\text{cch}}(T)&: \text{ object } i \text{ along edge } (V_{\text{cch},T}, V_{\text{cch},T+1}),\\ &\quad T = Z, 1+Z, \ldots, N-2+Z;\\ f^{(i)}_{\text{evict}}(T)&: \text{ object } i \text{ along edge } (V_{\text{cch},T}, V^{(T)}_{\text{next},i}),\\ &\quad T = Z, 1+Z, \ldots, N-1+Z. \end{aligned}$$

Note that $f^{(i)}_{\text{mem}}(T)$ is always 0 if $i \neq \sigma(T)$ due to the infinite cost. Similarly, the flow variable for object j along edge $(V_{\text{cch},T}, V^{(T)}_{\text{next},i})$ with $j \neq i$ is also always 0. Here our formulation is a so-called 'single-path routing' formulation, i.e., the flow variables are either 0 or 1 and they together represent a path for each object. Additionally, for convenience, for each vertex $V_{\text{mem},T}$, we use $\mathcal{P}^{(j)}(V_{\text{mem},T})$ to denote the set of vertices in the top row that have outgoing edges to $V_{\text{mem},T}$ associated with object j. Our goal is to minimize the following objective function:

$$\sum_{T=0}^{N-1} c^{(\sigma(T))}(V_{\text{mem},T}, V_{\text{cch},T+Z}) \cdot f^{(\sigma(T))}_{\text{mem}}(T), \tag{2.14}$$

[8]The proof corresponds to the *integral* version of BELATEDLY (i.e., before applying the fractional relaxation described in §2.4.3). Empirically, the approximation appears to be within a small error margin of the integral solution (§2.4.4), but a formal characterization of the integrality gap is left to future work.

where $c^{(\sigma(T))}(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is the latency cost in (2.13). The minimization problem is subject to the following constraints for each object i:

- **Link capacity:**

$$f_{\text{mem}}^{(\sigma(T))}(T) \leq 1, T = 0, 1, \ldots, N-1, \tag{2.15}$$

$$\sum_{i \in [M]} f_{\text{cch}}^{(i)}(T) \leq C, T = Z, 1+Z, \ldots, N-2+Z, \tag{2.16}$$

$$f_{\text{evict}}^{(i)}(T) \leq 1, T = Z, 1+Z, \ldots, N-1+Z. \tag{2.17}$$

  Here (2.16) models the constraint that we can have at most C objects in the cache. The constraints (2.15) and (2.17) are automatically satisfied.

- **Flow conservation:**

$$f_{\text{mem}}^{(i)}(T_i) = 1, \text{ where } V_{\text{mem},T_i} \text{ is the source of } i, \tag{2.18}$$

$$\text{total incoming flow to } V_{\text{sink}}^{(i)} \text{ is } 1, \tag{2.19}$$

$$\sum_{t:\, V_{\text{cch},t} \in \mathcal{P}^{(i)}(V_{\text{mem},T})} f_{\text{evict}}^{(i)}(t) = f_{\text{mem}}^{(i)}(T), i = \sigma(T), T > T_i, \tag{2.20}$$

$$f_{\text{cch}}^{(i)}(T-1) + f_{\text{mem}}^{(i)}(T-Z) = f_{\text{cch}}^{(i)}(T) + f_{\text{evict}}^{(i)}(T), \quad T = Z, 1+Z, \ldots, N-1+Z. \tag{2.21}$$

  Here the constraints (2.18) and (2.19) at sources and sinks are straightforward. The constraint (2.20) is a flow conservation constraint at vertex $V_{\text{mem},T}$. It implies that if object i was evicted from the cache before T and has not been requested since, then its data will be fetched from the backing store to the cache when it is requested at T. The constraint (2.21) is a flow conservation constraint at vertex $V_{\text{cch},T}$. Let us set $f_{\text{cch}}^{(i)}(Z-1) = f_{\text{cch}}^{(i)}(N-1+Z) = 0$ so (2.21) is valid at $T = Z$ and $T = N-1+Z$. This constraint guarantees the obvious requirement that an object is either in the cache or not in the cache.

Proof of Optimality

Lemma. *Given a sequence of object requests, there is a bijection between the set of feasible flow variables and the set of feasible cache schedules.*

Proof of Lemma 1. We first prove that any feasible cache schedule defines a set of feasible flow variables. Let $a_i(T), i \in [M], T = 0, 1, \ldots, N$ be a feasible cache schedule. We show that the flow variables defined below are feasible:

$$f_{\text{mem}}^{(i)}(T) = x_Z^{(i)}(T+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x_\tau^{(i)}(T+1)\right), \tag{2.22}$$

$$f_{\text{cch}}^{(i)}(T) = x_0^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=0\}} + x_1^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=1\}}, \tag{2.23}$$

$$f_{\text{evict}}^{(i)}(T) = x_0^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=-1\}} + x_1^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=0\}}. \tag{2.24}$$

Let us first consider the capacity constraints (2.15)–(2.17). It is easy to check that the constraints (2.15) and (2.17) are satisfied. Now we check the constraint (2.16). By the definition of $f^{(i)}_{\text{cch}}(T)$ in (2.23),

$$\begin{aligned}&\sum_{i\in[M]} f^{(i)}_{\text{cch}}(T)\\ =&\sum_{i\in[M]}\left(x^{(i)}_0(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}+x^{(i)}_1(T)\cdot\mathbb{1}_{\{a_i(T)=1\}}\right)\end{aligned}\tag{2.25}$$

When $a_i(T)=-1$, the summand in (2.25) is 0; when $a_i(T)=0$, the summand equals $x^{(i)}_0(T)=x^{(i)}_0(T+1)$; otherwise, when $a_i(T)=1$ and $x^{(i)}_1(T)=1$, object i will be admitted to the cache so $x^{(i)}_0(T+1)=1$. Combining these cases, we have that the summand in (2.25) is always no larger than $x^{(i)}_0(T+1)$. Thus,

$$\sum_{i\in[M]} f^{(i)}_{\text{cch}}(T)\le\sum_{i\in[M]} x^{(i)}_0(T+1)\le C,\tag{2.26}$$

and the constraint (2.16) is satisfied.

Next let us consider the flow conservation constraints (2.18)–(2.21). It is easy to check that the constraints (2.18) and (2.19) are satisfied. For the constraint (2.20), let $i=\sigma(T)$. Let $t^*=\max\{t\colon t<T,\sigma(t)=i\}$. Then one can check that $\{t\colon V_{\text{cch},t}\in\mathcal{P}^{(i)}(V_{\text{mem},T})\}=\{t^*+1,t^*+2,\dots,T\}$. So it suffices to show that

$$\begin{aligned}&\sum_{t=t^*+1}^{T}\left(x^{(i)}_0(t)\cdot\mathbb{1}_{\{a_i(t)=-1\}}+x^{(i)}_1(t)\cdot\mathbb{1}_{\{a_i(t)=0\}}\right)\\ &\qquad=x^{(i)}_Z(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x^{(i)}_\tau(T+1)\right).\end{aligned}\tag{2.27}$$

First, consider the case where $x^{(i)}_1(t)=1$ for some $t^*<t\le T$. Then we must have $t\le t^*+Z$ since there is no request for object i after t^* and before T. This arrival at t will resolve all the requests for i in the queue (if there exist any). We observe that $x^{(i)}_0(u)=0$ for $t^*<u\le t$ by (2.11) and $x^{(i)}_{t+1-u}(u)=x^{(i)}_1(t)=1$. Also $x^{(i)}_1(u)=0$ for $t^*<u<t$ since otherwise it would have resolved the request and thus results in no data arrival at t. If $a_i(t)=0$, then the data is not admitted to the cache. Also there is no request for object i on or after t (before T). So $x^{(i)}_1(u)=x^{(i)}_0(u)=0$ for $t<u\le T$. Then when the request for i comes in at T, it sees nothing in the cache nor the queue. So by the dynamics in (2.10), we have $x^{(i)}_Z(T+1)=1$. Therefore, the right-hand-side (RHS) of (2.27) is equal to 1, which is equal to the left-hand-side (LHS). For the case that $a_i(t)=1$, the data is admitted to the cache at t. There can be at most one eviction after t and no later than T (two evictions require data arrival in between). If there is no eviction, then the LHS is 0. The RHS is also 0 since the request for i at T will not be put in the queue and thus $x^{(i)}_Z(T+1)=0$. If there is an eviction at some u with $t<u\le T$, then all the summands

except $x_0^{(i)}(u) \cdot \mathbb{1}_{a_i(u)=-1}$ on the LHS are 0. So the LHS is equal to 1. The RHS is also equal to 1 since the request for i at T sees nothing in the cache nor the queue. In summary, (2.27) holds when $x_1^{(i)}(t) = 1$ for some $t^* < t \le T$.

Next, consider the case where $x_1^{(i)}(t) = 0$ for all t with $t^* < t \le T$. In this case there is no data arrival for object i during the whole time period. Then again there can be at most one eviction. Suppose there is no eviction for all t with $t^* < t \le T$. Then the LHS of (2.27) is 0. In this case, object i is either in the cache for all timestep t with $t^* < t \le T$ or it is not in the cache for all t with $t^* < t \le T$. If it is in the cache all the time, then the RHS is also 0 since $x_Z^{(i)}(T+1) = 0$. If it is always not in the cache, then $T < t^* + Z$ since the request at t^* is put in the queue and arrive at $t^* + Z$, but we have assumed that $x_1^{(i)}(t) = 0$ for all t with $t^* < t \le T$. However, $T < t^* + Z$ implies that $x_{t^*+Z-T}^{(i)}(T+1) = x_Z^{(i)}(t^*+1) = 1$, which implies that the RHS of (2.27) is 0. Therefore, for the case of no eviction, LHS and RHS are equal. Suppose there is an eviction at some t with $t^* < t \le T$. Then the LHS of (2.27) is equal to 1. Since we have assumed that $x_1^{(i)}(t) = 0$ for all t with $t^* < t \le T$, object i cannot reenter the cache after the eviction. So $x_\tau^{(i)}(T) = 0$ for $0 \le \tau \le Z$. Then the request for i at T will be added to the queue, so the RHS of (2.27) is equal to 1. Therefore, LHS and RHS are also equal in this case.

Combining the arguments above, we have shown that the flow conservation constraint (2.20) is satisfied. Now let us check the constraint (2.21), i.e., we want to show that

$$\begin{aligned} &\underbrace{x_0^{(i)}(T-1) \cdot \mathbb{1}_{\{a_i(T-1)=0\}}}_{\text{Term (L1)}} + \underbrace{x_1^{(i)}(T-1) \cdot \mathbb{1}_{\{a_i(T-1)=1\}}}_{\text{Term (L2)}} \\ &\quad \underbrace{+ x_Z^{(i)}(T+1-Z) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x_\tau^{(i)}(T+1-Z)\right)}_{\text{Term (L3)}} \\ &= \underbrace{x_0^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=0\}}}_{\text{Term (R1)}} + \underbrace{x_1^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=1\}}}_{\text{Term (R2)}} \\ &\quad + \underbrace{x_0^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=-1\}}}_{\text{Term (R3)}} + \underbrace{x_1^{(i)}(T) \cdot \mathbb{1}_{\{a_i(T)=0\}}}_{\text{Term (R4)}}. \end{aligned} \tag{2.28}$$

We start by discussing different cases of Term (L3). Suppose (L3) $= 1$. Then

$$x_Z^{(i)}(T+1-Z) = 1, x_\tau^{(i)}(T+1-Z) = 0, \tau = 1, 2, \ldots, Z-1. \tag{2.29}$$

In this case, $x_0^{(i)}(T+1-Z) = 0$ by (2.11) and object i will not arrive until timestep T. So $x_0^{(i)}(t) = 0$ for $t = T+1-Z, T+2-Z, \ldots, T$, and $x_1^{(i)}(T) = 1$. Then (L1) $=$ (L2) $=$ (R1) $=$ (R3) $= 0$ and (R2) $+$ (R4) $= 1$. Therefore (2.28) holds.

Now suppose (L3) $= 0$. Then either $x_\tau^{(i)}(T+1-Z) = 1$ for some $1 \le \tau \le Z-1$ or $x_\tau^{(i)}(T+1-Z) = 0$ for all $1 \le \tau \le Z-1$ and $x_Z^{(i)}(T+1-Z) = 0$.

- Suppose it is the former case. Then let t^* be the earliest time with $T+1-Z \le t^* \le T-1$ such that $x_1^{(i)}(t^*) = 1$. In fact, since all the requests in queue will be resolved when the

data arrives, t^* is also the only time between $T-Z+1$ and T such that $x_1^{(i)}(t^*)=1$. So (R2) = (R4) = 0. Also, $x_0^{(i)}(t)=0$ for all t with $T-Z\le t\le t^*$. If $t^*=T-1$, then (L1) = 0 and $(L2)$ = (R1) + (R3). If $t^*<T-1$, then (L2) = 0. Since $x_0^{(i)}(T)=x_0^{(i)}(T-1)+a_i(T-1)$, we have (L1) = (R1) + (R3). So (2.28) holds.

- Suppose it is the latter case, i.e., $x_\tau^{(i)}(T+1-Z)=0$ for all $1\le\tau\le Z-1$ and $x_Z^{(i)}(T+1-Z)=0$. Then $x_1^{(i)}(t)=0$ for $T+1-Z\le t\le T$. So (L2) = (R2) = (R4) = 0. Similar to the former case, it can be shown that (L1) = (R1) + (R3).

Combining the arguments above, we have shown that (2.28) always holds and thus the flow conservation constraint (2.21) is satisfied.

Now we prove the other direction of the lemma, i.e., we prove that any feasible set of flow variables define a feasible cache schedule. Let $f_{\text{mem}}^{(i)}(T), f_{\text{cch}}^{(i)}(T), f_{\text{evict}}^{(i)}(T)$ be a set of feasible flow variables. We show that the cache schedule defined below is feasible. For each timestep $T\ge Z$,

$$a_i(T)=\begin{cases}1 & \text{when } f_{\text{mem}}^{(i)}(T-Z)=1,\ f_{\text{cch}}^{(i)}(T-1)=0,\\ & \quad\text{and } f_{\text{evict}}^{(i)}(T)=0,\\ -1 & \text{when } f_{\text{cch}}^{(i)}(T-1)=1 \text{ and } f_{\text{evict}}^{(i)}(T)=1,\\ 0 & \text{otherwise.}\end{cases} \tag{2.30}$$

For T with $0\le T<Z$, let $a_i(T)=0$, which is always feasible. Let $x_\tau^{(i)}(T)$ with $i\in[M], \tau=0,\dots,Z$ be the state of the system as defined in (2.2) and (2.3) under this cache schedule in (2.30). To show that this schedule is feasible, we first prove the following claims.

Claim 1. *For any object i and any timestep $T\ge Z$,*

$$x_0^{(i)}(T)=f_{cch}^{(i)}(T-1). \tag{2.31}$$

Claim 2. *For any object i and any $T\ge 0$,*

$$f_{mem}^{(i)}(T)=x_Z^{(i)}(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(i)}(T+1)\right). \tag{2.32}$$

We note that in Claim 2,

$$\begin{aligned}&x_Z^{(i)}(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(i)}(T+1)\right)=1\\ \Leftrightarrow\ &x_Z^{(i)}(T+1)=1, x_\tau^{(i)}(T+1)=0 \text{ for all } \tau=0,1,\dots,Z-1\\ \Leftrightarrow\ &x_1^{(i)}(T+Z)=1.\end{aligned}$$

Therefore, it is equivalent to $f_{\text{mem}}^{(i)}(T)=x_1^{(i)}(T+Z)$.

We prove both claims by induction.

Proof of Claim 1. **Base case.** When $T = Z$, $x_0^{(i)}(T) = 0$ for all i since we start from an empty cache and $a_i(u) = 0$ for $0 \le u < Z$. We have also defined $f_{\text{cch}}^{(i)}(Z-1)$ to be 0 as an custom. So $x_0^{(i)}(Z) = f_{\text{cch}}^{(i)}(Z-1)$.

Induction step. Assume that for some $T \ge Z$, $x_0^{(i)}(T) = f_{\text{cch}}^{(i)}(T-1)$. We want to show that $x_0^{(i)}(T+1) = f_{\text{cch}}^{(i)}(T)$. Note that by the system dynamics in (2.9), we have that $x_0^{(i)}(T+1) = x_0^{(i)}(T) + a_i(T)$.

We consider the different cases of $a_i(T)$.

- If $a_i(T) = 1$, then by (2.30), $f_{\text{cch}}^{(i)}(T-1) = 0$, $f_{\text{mem}}^{(i)}(T-Z) = 1$ and $f_{\text{evict}}^{(i)}(T) = 0$. By the flow conservation at $V_{\text{cch},T}$, we have that $f_{\text{cch}}^{(i)}(T) = 1$. By the induction assumption, $x_0^{(i)}(T) = f_{\text{cch}}^{(i)}(T-1)$. Then $x_0^{(i)}(T+1) = x_0^{(i)}(T) + a_i(T) = 1$. So $x_0^{(i)}(T+1) = f_{\text{cch}}^{(i)}(T)$.
- If $a_i(T) = -1$, then by (2.30), $f_{\text{cch}}^{(i)}(T-1) = 1$ and $f_{\text{evict}}^{(i)}(T) = 1$. Due to the unit demand of each object, it is not hard to show that the total incoming flow an object to a vertex is at most 1. Specifically, consider the vertex $V_{\text{cch},T}$. Then $f_{\text{cch}}^{(i)}(T-1) + f_{\text{mem}}^{(i)}(T-Z) \le 1$. So $f_{\text{mem}}^{(i)}(T-Z) = 0$. By the flow conservation at $V_{\text{cch},T}$, $f_{\text{cch}}^{(i)}(T) = 0$. Since $x_0^{(i)}(T+1) = x_0^{(i)}(T) + a_i(T) = 0$, we have $x_0^{(i)}(T+1) = f_{\text{cch}}^{(T)}$.
- If $a_i(T) = 0$, by (2.30), we have the following possibilities:

$$f_{\text{cch}}^{(i)}(T-1) = 0, f_{\text{mem}}^{(i)}(T-Z) = 1, \tag{2.33}$$
$$f_{\text{cch}}^{(i)}(T) = 0, f_{\text{evict}}^{(i)}(T) = 1; \tag{2.34}$$
$$\text{or } f_{\text{cch}}^{(i)}(T-1) = 1, f_{\text{mem}}^{(i)}(T-Z) = 0, \tag{2.35}$$
$$f_{\text{cch}}^{(i)}(T) = 1, f_{\text{evict}}^{(i)}(T) = 0; \tag{2.36}$$
$$\text{or } f_{\text{cch}}^{(i)}(T-1) = 0, f_{\text{mem}}^{(i)}(T-Z) = 0, \tag{2.37}$$
$$f_{\text{cch}}^{(i)}(T) = 0, f_{\text{evict}}^{(i)}(T) = 0. \tag{2.38}$$

  For all the possibilities, $f_{\text{cch}}^{(i)}(T) = f_{\text{cch}}^{(i)}(T-1)$. Since $x_0^{(i)}(T+1) = f_{\text{cch}}^{(i)}(T-1) + a_i(T)$, we have $x_0^{(i)}(T+1) = f_{\text{cch}}^{(i)}(T)$.

This completes the proof of Claim 1.

Proof of Claim 2. **Base case.** When $T = 0$, by flow conservation, $f_{\text{mem}}^{(i)}(0) = 1$ if and only if $\sigma(0) = i$. Since we start from an empty cache and $a_i(u) = 0$ for $0 \le u < Z$, by the state dynamics (2.8)–(2.10), $x_\tau^{(i)}(1) = 0$ for all $i \in [M]$ and $\tau = 0, 1, \ldots, Z$, and $x_Z^{(i)}(1) = 1$ for $i = \sigma(0)$ and 0 for other objects. So (2.32) holds for $T = 0$.

Induction step. Assume that for each timestep u with $0 \le u \le T$,

$$f_{\text{mem}}^{(i)}(u) = x_Z^{(i)}(u+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x_\tau^{(i)}(u+1)\right). \tag{2.39}$$

We want to show

$$f^{(i)}_{\text{mem}}(T+1) = x^{(i)}_Z(T+2) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x^{(i)}_\tau(T+2)\right). \tag{2.40}$$

First, it is not hard to see that

$$\begin{aligned}
& x^{(i)}_Z(u+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x^{(i)}_\tau(u+1)\right) = 1 \\
\Leftrightarrow\ & x^{(i)}_Z(u+1) = 1, x^{(i)}_\tau(u+1) = 0 \text{ for all } \tau = 0, 1, \ldots, Z-1 \\
\Leftrightarrow\ & x^{(i)}_1(u+Z) = 1.
\end{aligned}$$

Therefore, the induction assumption (2.39) is equivalent to $f^{(i)}_{\text{mem}}(u) = x^{(i)}_1(u+Z)$.

Observe that the RHS of (2.40) is equal to 1 if and only if $\sigma(T+1) = i$ and $x^{(i)}_\tau(T+2) = 0$ for all $\tau = 1, 2, \ldots, Z-1$. Then (2.40) is trivially true for $i \neq \sigma(T+1)$. So it suffices to focus on the case where $i = \sigma(T+1)$.

If $V_{\text{mem},T+1}$ is a source vertex of object i, then $f^{(i)}_{\text{mem}}(T+1) = 1$. By flow conservation, $f^{(i)}_{\text{mem}}(u) = f^{(i)}_{\text{cch}}(u) = f^{(i)}_{\text{evict}}(u) = 0$ for $0 \le u \le T$. Then $a_i(u) = 0$ for all $0 \le u \le T$. So by the dynamics in the system, at $T+1$ the request will see that i is not in the cache and the queue for i is also empty. Then the RHS of (2.40) is equal to 1.

When $V_{\text{mem},T+1}$ is not a source vertex, let t^* be the last time object i was requested, i.e.,

$$t^* = \max\{t \colon t < T+1, \sigma(t) = i\}. \tag{2.41}$$

Suppose $f^{(i)}_{\text{mem}}(T+1) = 1$. Then by flow conservation at $V_{\text{mem},T+1}$, $f^{(i)}_{\text{evict}}(t) = 1$ for some t with $t^* < t \le T+1$. Let t' be the latest timestep with $t' \le t$ such that $f^{(i)}_{\text{mem}}(t'-Z) = 1$. By the induction assumption, $x^{(i)}_Z(t'-Z+1) = 1$ and $x^{(i)}_\tau(t'-Z+1) = 0$ for all $\tau = 1, 2, \ldots, Z-1$. If $t' \le t-1$, then by flow conservation $f^{(i)}_{\text{cch}}(t'-1) = 0$ and $f^{(i)}_{\text{evict}}(t') = 0$. So $a_i(t') = 1$ and $x^{(i)}_0(t'+1) = 1$. Then this enforces $x^{(i)}_\tau(t'+1) = 0$ for all $\tau = 1, 2, \ldots, Z$. For all u with $t' < u < t$, we can verify that $f^{(i)}_{\text{evict}}(u) = 0$. Then by the construction of the cache schedule, $a_i(u) = 0$. Therefore, the queue stays empty, i.e., $x^{(i)}_\tau(t) = 0$ for $\tau = 1, 2, \ldots, Z$. At t, since $f^{(t-1)}_{\text{cch}} = 1$ and $f^{(t)}_{\text{evict}} = 1$, we have $a_i(t) = -1$, and thus $x^{(i)}_0(t+1) = 0$. For any u with $t < u \le T+1$, we can show that $f^{(i)}_{\text{mem}}(u) = f^{(i)}_{\text{cch}}(u) = f^{(i)}_{\text{evict}}(u) = 0$, so $a_i(u) = 0$. We also know that $\sigma(u-1) \neq i$. So the queue for i stays empty at $T+1$ and i is not in the cache at $T+1$. Combing these, we can see that $x^{(i)}_\tau(T+1) = 0$ for $\tau = 1, 2, \ldots, Z-1$ and $x^{(i)}_Z(T+2) = 1$. So $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$. If $t' = t$, then we have $f^{(i)}_{\text{mem}}(t-Z) = f^{(i)}_{\text{evict}}(t) = 1$ and $f^{(i)}_{\text{cch}}(t-1) = f^{(i)}_{\text{cch}}(t) = 0$, and thus $a_i(t) = 0$. Using similar arguments as above, we can show that the queue for i stays empty and i is not in the cache at $T+1$. Then the RHS is 1 and thus $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$.

Now consider the case where $f^{(i)}_{\text{mem}}(T+1) = 0$. Then $f^{(i)}_{\text{evict}}(t) = 0$ for all t with $t^* < t \leq T+1$. If $f^{(i)}_{\text{cch}}(T) = 1$, then by flow conservation, $f^{(i)}_{\text{cch}}(T+1) = 1$. Then by Claim 1, $x^{(i)}_0(T+2) = f^{(i)}_{\text{cch}}(T+1) = 1$. Then $x^{(i)}_Z(T+2) = 0$ and thus $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$. If $f^{(i)}_{\text{cch}}(T) = 0$, then again, by flow conservation, we have that $f^{(i)}_{\text{cch}}(t-1) = 0$ and $f^{(i)}_{\text{mem}}(t-Z) = 0$ for all t with $t^* < t \leq T+1$. By Claim 1, $x^{(i)}_0(t^*+1) = f^{(i)}_{\text{cch}}(t^*) = 0$. If $f^{(i)}_{\text{mem}}(t^*) = 1$, then we must have $T+1-t^* < Z$. Therefore, $x^{(i)}_{t^*+Z-T-1}(T+2) = x^{(i)}_Z(t^*+1) = 1$ and thus the RHS of (2.40) is equal to 0. If $f^{(i)}_{\text{mem}}(t^*) = 0$, then $x^{(i)}_Z(t^*+1) = 0$ or $x^{(i)}_\tau(t^*+1) = 1$ for some $\tau = 1, 2, \ldots, Z-1$. Since $x^{(i)}_0(t^*+1) = 0$, there must exists a $\tau = 1, 2, \ldots, Z-1$ such that $x^{(i)}_\tau(t^*+1) = 1$. Let τ^* be the smallest τ such that $x^{(i)}_\tau(t^*+1) = 1$. Then $x^{(i)}_1(t^*+\tau^*) = 1$. Since $1 \leq \tau^* \leq Z-1$, we have $t^*+\tau^*-Z \leq T$ and thus by the induction assumption $f^{(i)}_{\text{mem}}(t^*+\tau^*-Z) = x^{(i)}_1(t^*+\tau^*) = 1$. We must have $t^*+\tau^* > T+1$ since $f^{(i)}_{\text{mem}}(t-Z) = 0$ for all t with $t^* < t \leq T+1$. Then $1 \leq t^*+\tau^*-T-1 \leq Z-1$ and $x^{(i)}_{t^*+\tau^*-T-1}(T+2) = x^{(i)}_1(t^*+\tau^*) = 1$. Thus $0 = f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$.

This completes the proof of Claim 2.

From Claims 1 and 2, it is easy to see that

$$\mathbb{1}_{a_i(T)=1} \leq f^{(i)}_{\text{mem}}(T-Z) = x^{(i)}_1(T+Z) \tag{2.42}$$

$$\mathbb{1}_{a_i(T)=-1} \leq f^{(i)}_{\text{cch}}(T-1) = x^{(i)}_0(T) \tag{2.43}$$

$$\sum_{i\in[M]} x^{(i)}_0(T) \leq C = \sum_{i\in[M]} f^{(i)}_{\text{cch}}(T-1) \leq C. \tag{2.44}$$

This verifies the constraints (2.5)–(2.7) and proves that the cache schedule defined in (2.30) is feasible. □

Once we have Lemma 1, the only thing left is to show that the MCMCF problem and the latency minimization problem have the same objective function. This is easy to see once we compare the objective functions (2.14) and (2.12) and apply Claim 2 from the proof of Lemma 1.

2.5 MAD: Online, Low-Latency Caching

BELATEDLY provides two principal lessons for the design of improved low-latency caching algorithms. First, BELATEDLY demonstrates that the opportunity for latency improvement is high: the gap between latency-optimal and hit-rate optimal can be as much as 45%. Second, BELATEDLY provides us with a caching schedule that *achieves* optimal latency for a given trace and Z value. Unfortunately, BELATEDLY is *slow* — taking up to 8 hours to compute an optimal schedule for a trace with 250,000 requests — and *requires knowledge of the future.* Both of these properties mean that BELATEDLY itself cannot serve as a caching algorithm for practical systems.

In this section, we learn from BELATEDLY's optimal schedule how to achieve better latencies in practical implementations. In §2.5.1 we first explore heuristics in the offline setting. In this setting, we still assume an oracle with perfect knowledge of future requests, but we target a

computationally tractable algorithm. In §2.5.2 we then move to a fully online setting where the algorithm both needs to be efficient *and* operate without knowledge of future requests.

2.5.1 Proxy Oracle: Bélády-AggregateDelay

We seek a heuristic *ranking function* which quickly tells us the priority of an object for our goal to minimize latency. To derive a ranking function, we look to BELATEDLY. While we cannot simply emulate BELATEDLY's behavior (unfortunately, flow algorithms like BELATEDLY don't reveal *how* they make decisions), we can search for easily measurable metrics correlated with BELATEDLY's caching decisions. As we saw in §2.4.2, BELATEDLY prioritizes caching *bursty* objects (i.e., those objects with a high Goh-Barabasi score [66]), and we experimented with ranking functions based on this score. While these functions had excellent runtime performance (the Goh-Barabasi score is a function of mean and variance, both of which can be measured cheaply with online algorithms), they delivered poor latency results. Therefore, burstiness on its own is not a good ranking function, which confirms the intuition we derived in §2.3.1.

Instead, we turn to another metric that is directly associated with the latency cost of bursty flows: *aggregate delay*, which is computed in Equation 2.13. To compute the rank of an object, we assume that the object's next access in the future is a miss. Its aggregate delay is the sum of the delay due to the miss and any delayed hits which occur during the next Z timesteps while the object would be fetched. Intuitively, an object with a higher delay cost — with a burst of requests during that Z window — increases average latency more than an object with a lower delay cost, and hence should be prioritized.

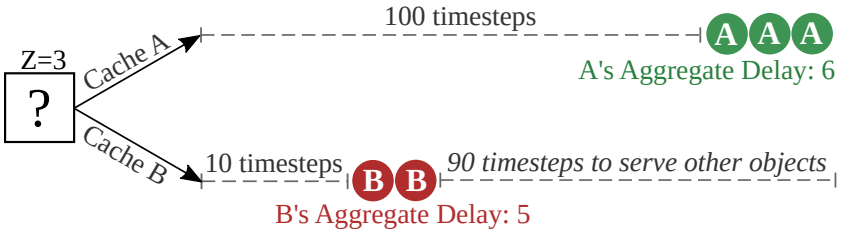


Figure 2.15: Ranking objects *solely* based on aggregate delay may lead to poor utilization of cache space.

Nevertheless, aggregate delay by itself is still not an effective ranking function. Consider the ranking of two objects A and B in a cache where $Z = 3$ as shown in Figure 2.15. A has an aggregate delay of 6 and will not be accessed for another 100 timesteps. B has an aggregate delay of 5 and will be accessed only 10 timesteps in the future. Should the rank function prefer A or B? Assuming we keep the cached object until its next access, keeping A utilizes one cache line (which cannot be used for other objects) for a very long interval. On average, each timestep we keep A in the cache will "save" an average of $\frac{6}{100}$ units of delay. On the other hand, for each timestep we keep B in the cache, we save an average of $\frac{5}{10}$ units of delay, with the opportunity to cache other objects in the remaining 90 timesteps. Hence, B appears to be, on average, a more efficient use of cache space.[9]

[9]This intuition does not necessarily lead to optimal decisions! For example, if we were to prefer B and evict A, but in the 90 timesteps after B *no other requests arrived* then it would have been better to prefer A.

Following this intuition, our offline ranking function, Bélády-AggregateDelay (Bélády-AD), computes two values for each object. AggregateDelay(x) is the aggregate delay for the next access to object x, and TTNA(x) is the number of timesteps until the next access to x.[10] The rank is then:

$$Rank(x) = \frac{\text{AggregateDelay}(x)}{\text{TTNA}(x)} \tag{2.45}$$

We find that, across all Z values, the average request latency provided by Bélády-AD is within 0.1–12% of BELATEDLY. In Figure 2.16, we show the average latency for Bélády-AD and BELATEDLY (normalized against the performance of Bélády's algorithm) for a range of Z values for the CAIDA Chicago network trace; Bélády-AD closely trails BELATEDLY, although the gap between the two widens as Z grows. Furthermore, Bélády-AD runs several orders of magnitude faster than BELATEDLY, computing a cache schedule in under 3 seconds for a trace containing 250,000 requests, where BELATEDLY would take up to 8 hours.

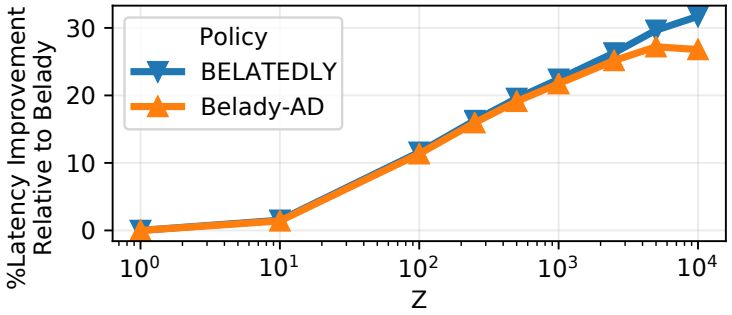


Figure 2.16: Bélády-AD closely trails BELATEDLY.

2.5.2 Online Algorithm: Minimum-AggregateDelay (MAD)

Finally, we turn to the true online setting, where we both need to use simple heuristics to rank objects and do not have knowledge of the future. Fortunately, we can use the past to make predictions about the future. Just as LRU uses recency as a ranking function (§2.2.2) — a "mirrored" version of Bélády's algorithm — we need to "mirror" our measures of AggregateDelay(x) and TTNA(x) to use data from past requests rather than future ones.

Luckily, we already have a large literature of estimators for TTNA(x), as almost all algorithms are essentially predictors of the next access to an object. Recall that Bélády's algorithm ranks objects by TTNA(x) alone, and is optimal in the absence of delayed hits. Hit-rate optimizing algorithms aim to operate as close to Bélády as possible [138], and so the closer their ranking function is to $\frac{1}{\text{TTNA}(x)}$, the better they perform. Hence, in §2.6.3 we experiment with using LRU [154], ARC [98], and LHD [18] as estimators of TTNA(x).

This leaves us with estimating AggregateDelay(x). Recall that we measure the *true* aggregate delay by assuming that the next request to object x will be a miss, and computing the sum of delays for the miss to x and any subsequent delayed hits for x. In the online setting,

[10] Note that Bélády's algorithm uses the ranking function $\frac{1}{\text{TTNA}(x)}$ alone.

we instead assume that all *past* requests to x were misses and then calculating the average aggregate delay per miss; we illustrate this in Algorithm 1. We find that this approximates the true AggregateDelay(x) well, *e.g.* with a Pearson Correlation Coefficient of 0.7 for the network trace. Finally, to create MAD, we combine the code[11] from the algorithm below with a known estimator for TTNA(x). We can now compute the rank using Equation 2.45.

Algorithm 1 Estimating AggregateDelay

```
1:  struct ObjectMetadata
2:      NumWindows = 0
3:      CumulativeDelay = 0
4:      WindowStartIdx = −∞
5:
6:  function EstimateAggregateDelay(X: ObjectMetadata)
7:      return X.CumulativeDelay / X.NumWindows
8:  end function
9:
10: function OnAccess(TimeIdx, X: ObjectMetadata)
11:     // Time since start of the previous miss window
12:     TSSW = (TimeIdx - X.WindowStartIdx)
13:
14:     // New miss window
15:     if TSSW ≥ Z then
16:         X.NumWindows += 1
17:         X.CumulativeDelay += Z
18:         X.WindowStartIdx = TimeIdx
19:     else
20:         X.CumulativeDelay += (Z - TSSW)
21:     end if
22: end function
```

We note that parallel work [97] in our group has shown that any deterministic online algorithm for the delayed hits problem has a competitive ratio[12] of $\Omega(kZ)$, where k is the size of the cache. Despite falling in that category, our empirical evaluations show that MAD yields considerable latency improvements over traditional caching algorithms, and its simplicity lends itself well to implementation. We leave to future work to find a randomized caching strategy which improves upon MAD's worst-case performance.

[11] For the sake of brevity, the provided pseudocode assumes discrete timesteps and prior knowledge of Z. Both of these assumptions are easily dispensable.

[12] The *competitive ratio* of an online algorithm, α, is the worst-case ratio between the costs of the solution computed by α to that of the optimal, offline solution for the same problem instance. Knowledge of the competitive ratio allows us to impose bounds on its worst-case performance (i.e., under the most pessimal workload) [144].

2.6 Evaluating MAD

We evaluate the effectiveness and the overhead of MAD in the context of a real system (§2.6.2), then turn to simulation to explore a wider range of applications and parameters (§2.6.3).

2.6.1 Experimental Setup

Prototype. We emulate a CDN deployment with clients and backends in geographically different locations. For rapid prototyping, we implement our own asynchronous caching system in 1500 lines of C++ code, using Boost.ASIO [10, 126]. Our architecture uses sharding and a single thread per cache shard [18, 26, 55]. An overview of the system architecture is depicted in Figure 2.17.

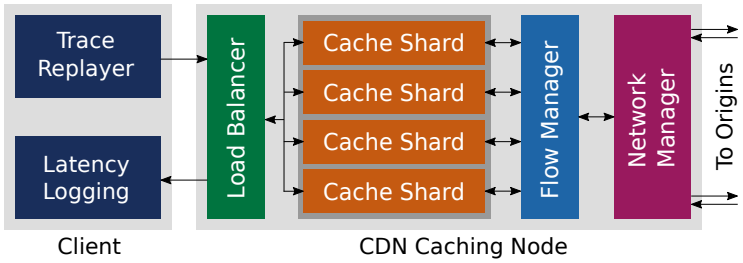


Figure 2.17: Architecture of our experimental prototype.

The client sends requests as 16B object IDs to the *Load Balancer*, which forwards it to the *Cache Shard* corresponding to the object ID. The shard's thread performs a cache look-up. If the object is cached, the request is resolved immediately by relaying a response back to the client (a *true hit*). Else, the request is forwarded to the *Flow Manager*, which maintains queues of outstanding requests separately for each unique object ID.[13] On receiving a request, if the object ID is not mapped to an existing queue, the Flow Manager allocates a new queue for the object and forwards the request to the *Network Manager* (a *miss*). Else, the request is simply inserted at the tail of the queue (a *delayed hit*). The Network Manager use a pool of threads with long-running TCP connections to the backing stores. These threads perform the actual fetch operation and relay the response to the Flow Manager. The Flow Manager buffers the response, flushes the request queue for the corresponding object ID, and issues a write request to the appropriate cache shard. The cache is updated (based on the specified caching policy), and the responses are sent to resolve all queued client requests.

To achieve low latency and high concurrency, the system components communicate using lock-free, single-producer single-consumer queues. The system is capable of sustaining a throughput of 1.2M requests/sec using 12 threads on an x86 server with 16 GB of DRAM.

Cache configuration and policies. We use a 64-way set-associative cache, with the total cache size set to 5% of the maximum number of active concurrent objects (e.g, 67K cache entries overall for the CDN trace from Table 2.1). For the purpose of our experiments, we fix the object size to $1KB$. We implement two policies: *LRU*, and *LRU-MAD*, which combines LRU's TTNA(x) estimator and our AggregateDelay(x) estimator.

[13] We use separate request queues to avoid head-of-line blocking.

Traces. We use a busy period from the CDN trace in Table 2.1 which contains 243M requests, 7.7M unique object IDs, a maximum of 1.3M active concurrent objects, and an average inter-request time of 1 μs.

Setup. To emulate different Z values, we set up backing stores (using GCP VMs) in three different locations around the world: The U.S. West Coast (Los Angeles), Western Europe (the Netherlands), and East Asia (Singapore). For simplicity, we refer to these as Origin A, with an RTT of *68 ms* ($Z = 68$K), Origin B, with an RTT of *103 ms* ($Z = 103$K), and Origin C, with an RTT of *226 ms* ($Z = 226$K), respectively.[14] We deploy our CDN caching node on a server at CMU in Pittsburgh.

2.6.2 Prototype Evaluation on CDN Trace

What latency improvements does LRU-MAD provide for our wide area cache? To answer this question, we consider each of the three backing stores independently, and measure the *average request latency* provided by the two caching policies for the given workload. Figure 2.18 shows the average latencies achieved using LRU-MAD versus LRU. Overall, using LRU-MAD, we see a 12.4%, 14.7%, and 18.3% reduction in average latency for Origins A, B, and C, respectively. As expected, LRU-MAD's benefit increases with Z.

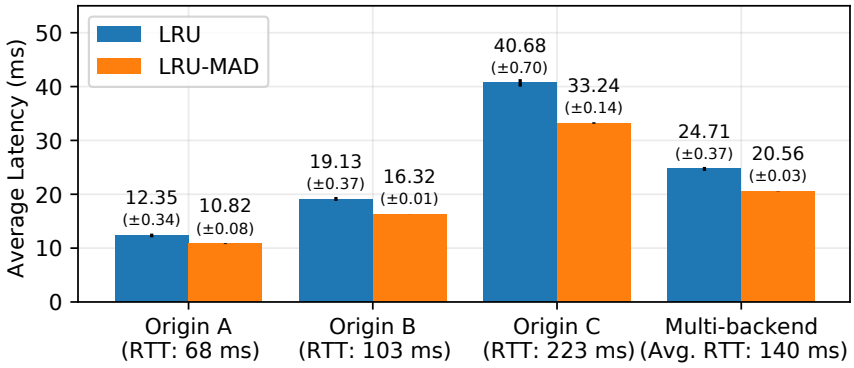


Figure 2.18: Prototype results for different origin locations.

Does the MAD caching strategy still work if multiple, non-uniform backing store latencies[15] are involved? This differs significantly from our offline formulation which only considered uniform latencies (i.e., a single Z value). We find that MAD indeed works well in the multi-backend scenario. Figure 2.18 shows a 16.8% reduction in average latency for this case. This result suggests that maintaining per-object estimates of the backing store latency (instead of a single, global average) is an important feature of the online strategy, since it gives MAD a higher degree of freedom in computing ranks.

[14] We remark that, although the backing store latencies are known a priori, we do *not* explicitly provide this information to MAD; instead, it automatically computes per-object estimates of backing store latencies at run-time.

[15] We map each object ID to a randomly-generated *origin location*, which places a third of object IDs on each origin server. The distribution of *requests* is: 29% to Origin A, 39% to Origin B, and 32% to Origin C.

What are the overheads of using MAD? We discuss two kinds of overheads associated with MAD: *memory* and *request latency*. We evaluate the memory overhead of two different implementations of MAD. Both implementations maintain 4 counters per object. Our *strawman* implementation faithfully implements MAD by persisting these counters for both cached and uncached objects. However, in a long-running caching system, this would require an unbounded amount of memory. Our *efficient* implementation only stores the counters for currently cached objects. Fortunately, we find that the average latency provided by the efficient implementation is within 6% of the strawman over the entire range of Z values, across all traces. In fact, all results presented so far have been using the efficient implementation. Our counters are 8B; so, the overall overhead is 32B per cached object, which is comparable to existing key value stores [55]. Our efficient implementation thus has a memory overhead of just over 3% for small 1KB objects and under 0.003% for objects in the MB range (e.g., video caching [103]).

We compare MAD's request latency to LRU, where eviction is a constant-time operation (the entry to evict is always at the head of a linked list). Evictions in MAD require computing the rank(X) function from §2.5.2 over all objects in the corresponding cache set. While each computation is cheap, its complexity scales linearly with the set-associativity of the cache in our naive implementation. This leads to several microseconds of overhead, which is orders of magnitude lower than the latency of the backing store. We remark that this small overhead can be further reduced using existing techniques. For instance, large-scale production systems achieve constant-time evictions using sampling techniques [18], which can be immediately applied to an implementation of MAD.

How accurately do our simulations reflect results in the wide area? We use simulated results to motivate the problem in §2.2.1, and in the following evaluation sections. While our simulator models the effects of delayed hits, it makes several simplifications. For example, it assumes that arrivals neatly fall into discrete time slots, that cache management operations are instantaneous, and that network latencies are deterministic. We validate these simulation results by comparing the latency distribution (CDF) measured with our prototype to simulations based on averaged estimates of Z for Origin B (results for other origins are the same). Figure 2.19 shows that the simulated latencies indeed closely match the empirical measurements.

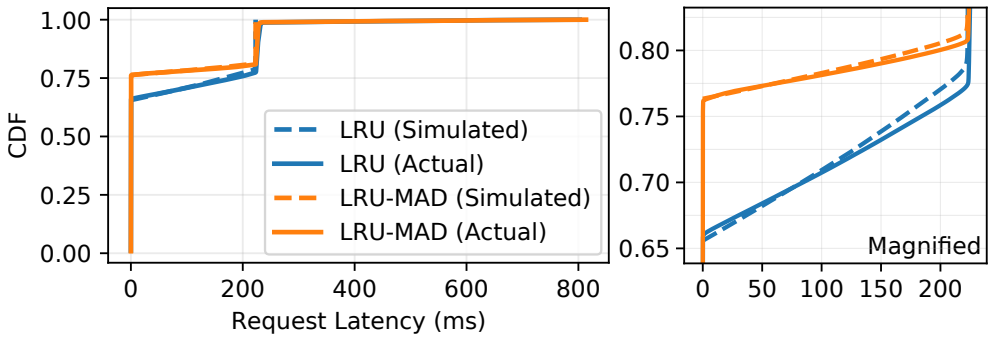


Figure 2.19: CDF of latencies in simulation versus real experiments.

2.6.3 Simulation Results: Systems

Our prototype experiments focus on the CDN setting with a small set of backing latencies and a single algorithm. We turn to a delayed hits-aware simulator we implemented to explore more sophisticated caching algorithms (enumerated in Table 2.3) and their MAD variants in the context of CDNs, network traces, and storage traces.

	Algorithm Description
LRU	Recency-based heuristic, evicts the *least recently used* item from the cache [154]
ARC	Balances frequency and recency [98]
LHD	Learns hit and lifetime distributions, evicts the object with the lowest *hit density* [18]

Table 2.3: Overview of implemented online caching algorithms.

How does MAD help CDNs with other base algorithms and a wider range of latencies? Figure 2.20 illustrates the performance gains from combining AggregateDelay(x) with LRU, LHD, and ARC. The y-axis measures the relative improvement in latency between LRU-MAD and LRU, LHD-MAD and LHD, and ARC-MAD and ARC. MAD *always* performs better than the baseline algorithm, suggesting that there is no downside, from a latency minimization perspective, to adopting MAD — regardless of what ranking algorithm was used initially. As with our LRU prototype, we see gains of 5–20% when latencies are in the tens of milliseconds.

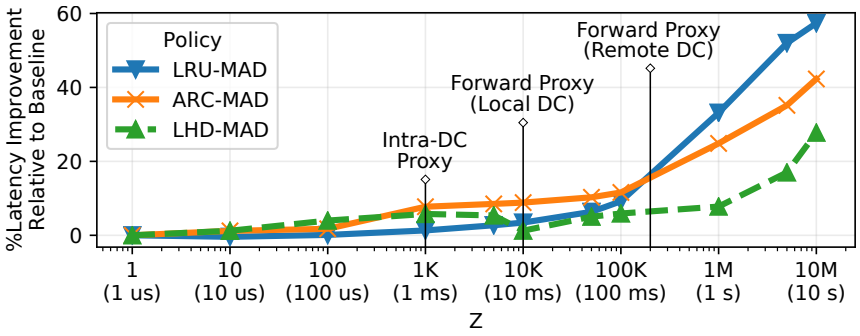


Figure 2.20: MAD simulations for the CDN Trace.

We also see that as Z reaches some extreme values — 1M or even 10M — the gains from MAD increase dramatically. Today, these examples are only useful for an imaginary web user with a CDN cache on the moon. However, they *may* serve as an estimate for the impact of delayed hits on future workloads. Recall that Z does not represent latency itself, but the *ratio* between latency to the backing store and request inter-arrival time (§2.1). Hence, as link and request rates grow by 10×, a Z value of 1M would only represent a 100 ms latency for the CDN. Nonetheless, these extreme values remain flawed estimators; we expect that request arrival rates, their burstiness, and the number of requested objects may all change in this time; these data points are hence little more than an educated guess towards the future.

Can MAD help network switch caches? We were surprised to see the *lowest* gains with regard to practical caching scenarios in the networking setting (recall Table 2.1).[16] The only application where we would expect to see any gains is an IDS with a reverse-DNS lookup, which we would expect to run in the 10s or 100s of milliseconds; the simulation here predicts latency gains of 10–35%. Nonetheless, most IDSes which perform such lookups are not inline, and hence we would not expect to see these latency gains passed on to Internet users whose traffic is intercepted by the IDS.

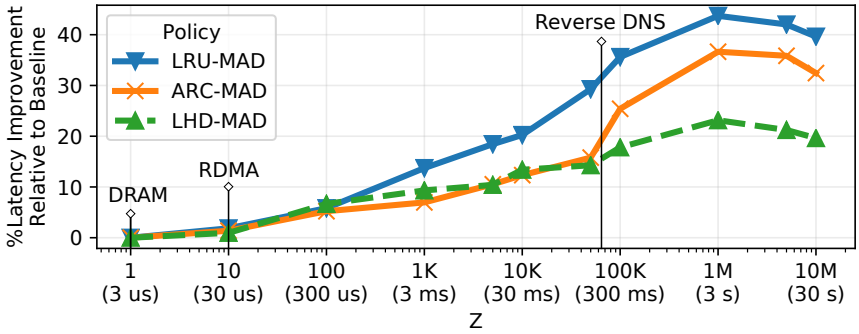


Figure 2.21: MAD simulations for the Network Trace.

Looking to the future and very high Z values, we see a tapering off trend which we do *not* observe in the CDN scenario. As discussed in our BELATEDLY results, this tapering off in the network setting is due to flows beginning and ending during the entire Z window; we do not see this trend in the CDN or storage scenarios because objects are much longer lived than a few milliseconds or even seconds. The simulations are hence flawed for network traffic in this regard — in practice, a switch would hold the first SYN packet until its flow context were fetched and subsequent packets would not arrive at the switch until the SYN completed. We leave to future work a more accurate model of network traffic and Z values where the arrival time of packets is *dependent* on the time it takes to serve the first packet.

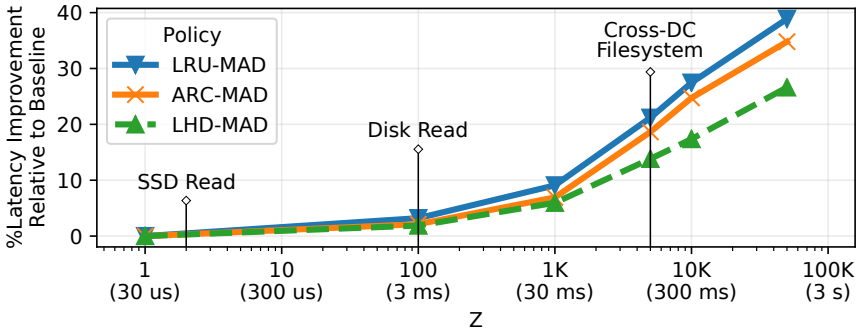


Figure 2.22: MAD simulations for the Storage Trace.

[16]The reason this was surprising is that, ironically, we had first encountered delayed hits on an FPGA-based programmable switch, with incoming packets triggering access to a flow context stored in either an SRAM-based cache (5 ns reads), or a DRAM-based global backing store (100 ns reads). When a flow's packet results in a cache miss, it triggers the 100 ns fetch operation. At high throughput, a second packet of the same flow arrives before 100 ns have passed. However, our simulation suggests essentially no performance gains for this scenario from using MAD.

Can MAD help distributed storage? Our storage trace has similar results to the CDN result; we see that in the millisecond range MAD achieves gains between 3–30%, representing improvements for wide-area or cross-datacenter storage systems. However, when deployed intra-datacenter where network latencies are in the microseconds and system latencies in the low milliseconds, we can expect much more marginal returns.

Summary. Overall, our experiments suggest that the systems that would benefit most from MAD today are CDNs and distributed storage systems with high latencies to the backing store. While switch workloads tend to be more bursty (resulting in higher gains for MAD even at relatively low Z values), few scenarios involve this latency being passed on to end-users.

We note that there are several interesting properties of real systems that are not captured here. For instance, while MAD may only shave off a few *ms* worth of latency on each individual request, some tasks, such as loading web pages, involve *chains* of serialized requests (*e.g.* due to recursive dependencies in HTML or CSS elements [106]); consequently, the overall impact (*e.g.* on page load time) may be more significant. Similarly, fetching large objects from the backing store may require multiple RTTs, exacerbating the effect of delayed hits. Additionally, certain objects *must* be periodically purged from the cache due to TTL expiration (*e.g.* cached DNS entries), introducing an additional layer of complexity in the design of online algorithms. We leave a more detailed investigation of these effects to future work.

2.6.4 Simulation Results: Analysis

We now present findings that are not tied to any particular system.

Impact of cache sizing: We evaluate how cache size impacts MAD's improvements over traditional caching algorithms. Recall that we measure the cache size as a fraction of the peak number of concurrently-active objects.[17] We calculate the latency improvement of MAD relative to LRU for all four scenarios while keeping Z fixed at $Z = 100$K.

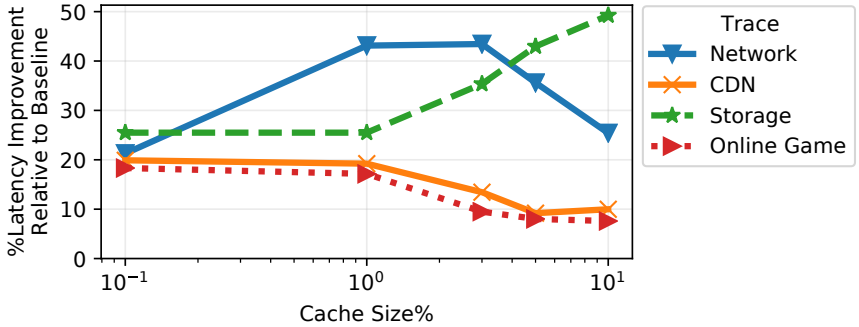


Figure 2.23: Relative latency difference between LRU-MAD and LRU as a function of the cache size. Using $Z = 100K$.

[17]Note that our cache size definition is motivated by networking applications where flow state only needs to be tracked for "active" flows. Caching papers on CDNs and storage systems typically express the cache size as a fraction of the working set [18, 26, 98], which is orders of magnitude larger. The cache size numbers shown in our graphs thus might look comparably large but they are based on a different denominator.

Figure 2.23 shows the results for cache sizes between 0.1% and 10%. We find that MAD's improvement is around 20% for small caches ($< 1\%$) in the CDN and online gaming scenarios. In the networking scenario, MAD's improvement is between 20% and 43% (we fix Z to demonstrate the effect of cache sizing, but we note that the chosen value is higher than one would expect to see in a networked setting). Finally, we see that MAD's improvement is highest in the storage scenario, with a 26% to 50% lower latency than LRU.

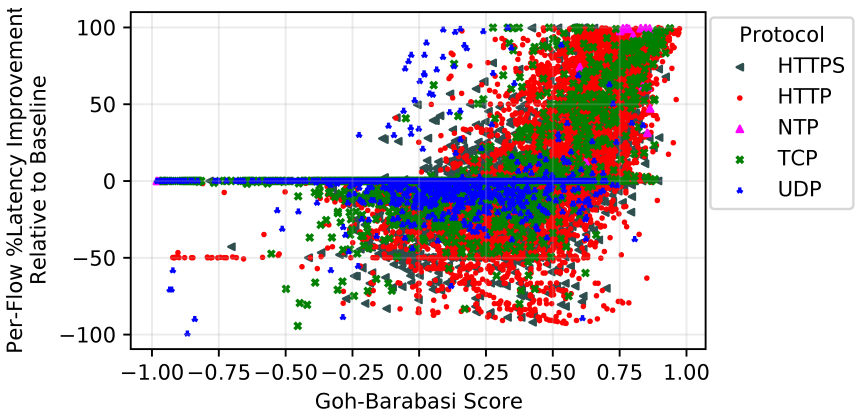


Figure 2.24: Like BELATEDLY, MAD prioritizes bursty objects.

MAD prioritizes bursty objects, just like BELATEDLY. We described the intuition behind MAD as prioritizing bursty objects, just like BELATEDLY. Nonetheless, we use aggregate delay rather than true burstiness (Goh-Barabasi score) and we weigh aggregate delay against time to next access. Hence it is worth asking: does our intuition about burstiness indeed map on to why MAD is doing well? Figure 2.24 shows per-object latency gain (or loss) between LRU and LRU-MAD's caching schedule for the Network trace. Much like Figure 2.10 illustrating BELATEDLY's correlation with burstiness, MAD prioritizes bursty objects as well.

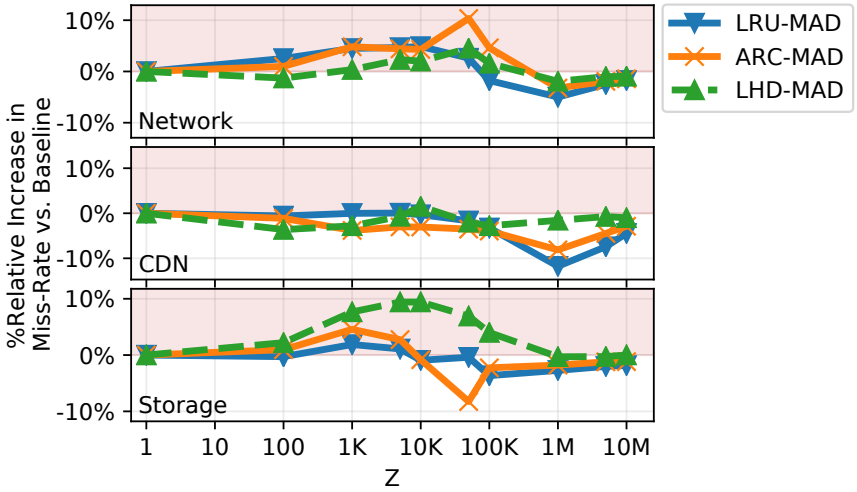


Figure 2.25: Percent relative change in miss-rate between MAD and various baseline caching algorithms for Network, CDN, and Storage.

Impact on cache miss-rate. As described in §2.3.1, latency-minimizing algorithms might in fact *increase* the overall miss-rate. Hence, we quantify the impact of MAD an algorithm designed to minimize latency, on the overall cache miss-rate (which in turn affects the bandwidth consumption on the link to the backing store). Figure 2.25 depicts the relative change in miss-rates[18] between MAD and our three baseline algorithms as a function of Z. Regions where MAD *increases* the miss-rate (i.e. performs worse than the baseline) are highlighted in *red*. We find that, across all Z values and choice of baseline algorithms, MAD increases miss-rates by at most 10% for the Network and Storage settings[19] ($+1.84\%$ and $+1.43\%$ on average), but almost always reduces miss-rates in the CDN setting (-1.89% on average). We conclude that, depending on the workload, there is a tradeoff between optimizing for latency and bandwidth.

2.7 Related Work

Caching algorithms have received a significant amount of research attention, but the aspect of delayed hits is largely disregarded in the literature. We are not aware of any prior work proposing an analytical model for the delayed hits problem, or designing algorithms targeting delayed hits. Most existing caching algorithms focus on maximizing hit ratios, with significant advances in recent work [18, 26, 30, 80, 92, 138] and excellent surveys of older work [114, 149]. There are two groups of prior work that look at maximizing metrics other than hit ratios.

1. **Cost-aware online caching algorithms.** This group of algorithms [37, 81, 82, 83, 90, 159, 160] seeks to minimize the average cost of misses, where an object's cost models differences in retrieval latencies or computation costs. In this setting, if an object is cached, its next request does not contribute to the overall average cost, but no other requests are affected. This is different from the delayed hits settings where a single caching decision may affect many future requests (to the same object). By assuming independence, cost-aware caching assumes that misses are retrieved before another request to the same object arrives.

2. **Weighted, general, and other offline caching theory.** This group of algorithms [5, 19, 27, 35, 39, 41, 42, 150] considers offline caching problems beyond Belady. Weighted caching is like cost-aware caching, but using offline knowledge [41]. Caching for variable object sizes optimizes hit ratios, but considers objects that require a different number of bits to be stored in cache [5, 27]. General caching generalizes both by considering both weighted and variably-sized objects at once [35, 42]. In general, these problems are NP-hard, except for weighted caching which can also be approximated using a flow formulation.

The architecture community has a rich literature on *implementing* non-blocking caches to handle multiple outstanding misses [1, 21, 87, 105, 132, 145] — a prerequisite for the occurrence of delayed hits. In addition, [115] considers the effect of *correlated* cache misses (different from delayed hits, but in a similar vein) on Memory Level Parallelism (MLP) performance in processors. Finally, we are aware of two prior works [63, 142] which report improved accuracy when accounting for delayed hits in simulations of processor caches.

[18] $\frac{\text{MR}_{\text{(MAD)}} - \text{MR}_{\text{(BASELINE)}}}{\text{MR}_{\text{(BASELINE)}}} \times 100\%$

[19] Note that this value represents a *relative* increase in miss-rate compared to the baseline. In our experiments, the *absolute* difference in miss-rates never exceeds 1%.

2.8 Broader Impact, Open Questions, and Conclusion

More than anything, this work opened a Pandora's box by highlighting how little we understand in the face of systemic complexity, even in a domain as old as caching. Since the time this work was published [12], there has been a concerted effort to model and account for delayed hits in high-performance systems ranging in scale from a single server (*e.g.*, CXL interconnect [48]) to the distributed data store for the Large Hadron Collider [43]. Nonetheless, there remain a broad range of practical and theoretical questions left to address.

On the practical side, while the online algorithm we propose in this thesis seems to perform well empirically, we now know that it has poor competitive ratio [97]. Consequently, we don't expect MAD to be the final word on latency-minimizing caching in the presence of delayed hits, and we expect to see better results from more sophisticated (possibly randomized) algorithms. Similarly, in the online setting, prefetching algorithms may also merit a second look with respect to latency and delayed hits.

On the theoretical side, even with our refinement, there are many attributes of practical systems the caching model does not capture; richer and more complex scenarios hence merit additional investigation, both in the offline and online settings. For example, our theoretical model does not account for variable backing store latency (although our evaluation does capture this setting), nor does it account for differing object sizes. Both our theory and simulator assume that, once the data fetch delay has passed, all outstanding delayed hits are immediately processed and released, although many systems may instead operate over each response sequentially leading to additional queuing at the cache. Finally, even after 5 years, the hardness question of the delayed hits optimization problem still remains open.

With continuous advancements in bandwidth and throughput (*e.g.*, in networks, memory, new storage technologies, and CPU-interconnects), access latencies are becoming progressively larger relative to request inter-arrivals, increasing the likelihood of delayed hits. Indeed, we believe that the problem of delayed hits will surface in almost any caching scenario sooner or later. Our work constitutes a first step in recognizing the incongruity and mitigating the increased latencies created by this fundamental trend. Nonetheless, as we described above, there remain many open questions about incorporating delayed hits into practical caching systems, and we look forward to theoretical refinements and practical algorithms that further close the gap between caching models and reality.

"Focus on the common case: in making a design trade-off, favor the frequent case over the infrequent one"

John Hennessy and David Patterson

Chapter 3

Mitigating Algorithmic Complexity Attacks on Network Subsystems

Denial-of-Service (DoS) attacks are the bane of public-facing network deployments, costing enterprises between $120K and $2.3M *per incident* in lost revenue and operational expenses. They also evoke a picture of sheer brute force; after all, many high-profile attacks in recent times have been distributed DoS (DDoS) attacks involving massive botnets that transmit terabits-per-second of network traffic to overload the victim server.

However, there is another, more sophisticated class of DoS attacks, called **Algorithmic Complexity Attacks (ACAs)** that are *highly pervasive*, *far more potent* (requiring only a fraction of the resources for the attacker to produce), and notoriously difficult to even *detect*, let alone defend against. ACAs target a system's underlying algorithms and data-structures, using specially-crafted inputs to trigger its worst-case behavior, thereby driving the victim into overload and ultimately causing it to drop requests from the innocent, intended users of the service. ACAs are especially dangerous when compared to traditional, *volumetric* DoS attacks (*e.g.*, DDoS) because they allow the attacker to do disproportionate harm: a 2019 study showed that an ACA targeting Tuple Space Search (TSS) [139], a popular packet classification algorithm, allowed attackers to drop 8.83 Gbps of innocent traffic by investing just 670 Kbps of their own bandwidth into the attack [46] (13,000× harm, compared to DDoS's 1×).

As we will discuss in §3.2, systems today have evolved awkward adaptations (often implicitly) to account for the perennial threat posed by ACAs. Since ACA vulnerabilities are rooted in the variance between an algorithm's average-case and worst-case performance, existing defenses against ACAs revolve around "patching" algorithms to shrink this variance. Unfortunately, this is not a *general* remedy: every patch needs to be designed and engineered on a case-by-case basis, entailing significant amounts of time and effort. Worse, these patches are no free lunch; they require sacrificing some system property or another (*e.g.*, memory efficiency, or average-case performance) in exchange for ACA resilience. This leaves system designers with two equally unappealing alternatives: risk the debilitating effects of ACAs, or spend resources only to end up with a system that has inferior common-case characteristics.

In this chapter, we take a first-principles approach towards designing a *general* mitigation strategy against ACAs on network packet processors (*e.g.*, intrusion detection systems). We argue that ACAs are fundamentally artifacts of **workload incongruity**: a transient mismatch between the choice of algorithm, which is fixed at *deployment time*, and its input workload, which can change (from purely stochastic to partially adversarial) at *run-time*. We present an analytical framework for modeling ACAs as a superposition of stochastic and rate-limited adversarial traffic, thereby allowing us to systematically reason about the attacker's actions and their collateral on innocent traffic. Finally, we show how insights from this abstraction ultimately enable us to develop an attack mitigation strategy that is *general*, entails *no intrusive algorithmic changes*, and comes with *mathematical guarantees* regarding its efficacy.

The remainder of this chapter is organized as follows. In §3.1, we show how incongruity in the classical model of packet processing workloads engenders ACAs. In §3.2, we provide the requisite background on these attacks and existing mitigation techniques. In §3.3, we characterize our refinement to the workload model, and introduce a novel metric to quantify the impact of ACAs. In §3.4, we describe our key insight: *Weighted Shortest Job First (WSJF)*, a simple augmentation to a decades-old packet scheduling algorithm, enforces a theoretical upper-bound on the harm induced by ACAs; we prove this using a novel adversarial analysis framework, and also use it to demonstrate why existing scheduling algorithms deployed today (*e.g.*, Fair Queueing) *don't* work. In §3.5, we present the design and implementation of SurgeProtector, our drop-in packet scheduler to protect networked systems against ACAs. In §3.6, we evaluate SurgeProtector in the context of Pigasus, an open-source Intrusion Detection System (IDS), where it achieves 90–99% reduction in harm induced by ACAs. We address limitations and open questions in §3.7, describe related work in §3.8, and conclude in §3.9.

3.1 Workload Incongruity

To illustrate how workload incongruity leads to ACAs, let us put ourselves in the shoes of an operator looking to deploy a network firewall to protect our service from attackers. Consider the hypothetical example depicted in Figure 3.1, where ① we need to decide between two candidate designs: *Design A*, which promises 100 Gbps performance on average, but only guarantees 1 Gbps in the worst case; and *Design B*, which always operates at 10 Gbps, regardless of the workload. So, which one of these designs should we deploy?

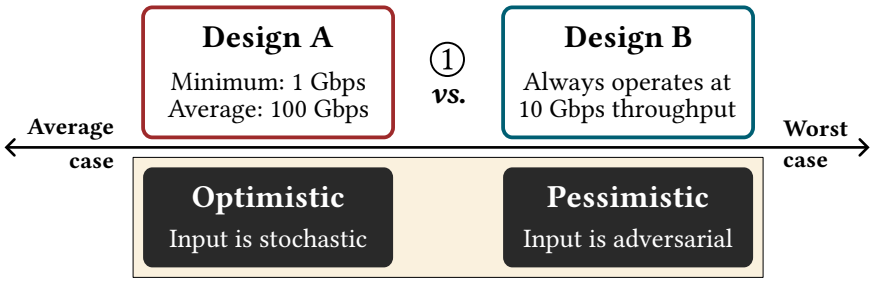


Figure 3.1: The classical model posits a false dichotomy between workload characteristics.

Even in this very contrived setup, the choice turns out to be a difficult one. The difficulty arises from two salient aspects of the problem. First, neither design is strictly better than the other; in other words, both A and B represent Pareto-optimal points in the design space, with the only difference being that they maximize distinct objectives (average-case *vs.* worst-case throughput). Second, committing to a design ② forces us to conform to one of two very different beliefs (i.e., models) about the workload: either a *highly optimistic* one where inputs are always stochastic (well-behaved), and we are only worried about the common case; or, a *highly pessimistic* one, where we are only concerned with pathologies and worst-case inputs.

Therein lies the problem.

> **Incongruity**: We must commit to a design at *deployment time*, but workload characteristics can always change at *run time*. Consequently, when the workload inevitably deviates from our expectations (and whichever model we committed to), we are left with a suboptimal design, and not-so-great outcomes.

Looking back at Figure 3.1, the drawback of deploying *Design B* is clear: most workloads *are*, in fact, typically stochastic and well-behaved, and so the vast majority of times, optimizing for the worst-case leaves 90% of the throughput on the table. As network operators, we are thus incentivized to choose *Design A*, but this is accompanied by significant disparity between average- and worst-case performance (100 Gbps versus 1 Gbps), which, as we saw earlier, is the root cause of ACAs.

We argue the problem fundamentally lies not in the designs themselves, but rather in the false dichotomy created by our classical model of packet processing workloads. As we will see in §3.3, revisiting this design question through the lens of a **realistic** model (where the input is *usually* stochastic but *allowed* to contain some amount of adversarial traffic) allows us to devise a system that achieves *Design A*'s common-case performance while still providing worst-case resilience guarantees similar to *Design B*.

3.2 Background and Motivation

Algorithmic complexity attacks target a system's underlying algorithms and/or data-structures, using specially-crafted inputs to trigger the system's worst-case behavior [2, 22, 45, 136]. While the attacker's input patterns and the resulting behavior may vary from design to design, the ultimate goal of these attacks is the same: to overload the system with large amounts of wasteful work, inhibiting its ability to serve innocent user traffic. The key difference between an ACA and a traditional volumetric DoS attack is that in an ACA, the attacker can induce the system to perform a *large* amount of wasteful work by introducing a *small* input that costs little to produce. In a volumetric DoS attack, the attacker must craft a *large* amount of input to overload the system, which requires the investment of physical resources to produce this traffic. Colloquially, an ACA provides "more bang for one's buck".

Example: Consider the following, simplified example drawn from Pigasus [162]. Pigasus is a hybrid FPGA+CPU, 100Gbps IDS, and it implements partial TCP reassembly in order to detect attacks that span across multiple packets in a TCP bytestream. As shown in the Figure 3.2, Pigasus stores packets from out-of-order flows in a linked list. When a packet corresponding to an out-of-order flow arrives, the reassembly engine traverses its linked list to find the appropriate insertion location (using the packet sequence number), performs insertion, and, if possible, releases any in-order segments.

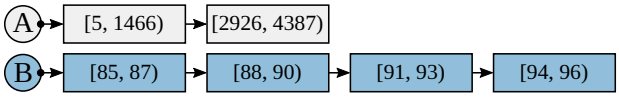


Figure 3.2: TCP reassembly implemented using a linked list [162]. Each node in the list represents a range of packet sequence numbers.

Let us assume for the sake of exposition that most connections look like flow A in Figure 3.2, with exactly two packets in the linked list and only one "gap" in the sequence number space. When a re-transmitted or re-ordered packet arrives to fill in a gap in the sequence number space (*e.g.*, a packet with sequence number 1466 in flow A), it takes one iteration of pointer chasing to reach the right index in the linked list. To mount an ACA, an attacker might transmit a sequence of packets leading to a scenario more like flow B: should a packet arrive with index 97, it would take four iterations of pointer chasing — or four times as many cycles as in the typical case — to find the appropriate insertion location.

We refer to the amount of work the system performs to process a packet as the packet's *job size*, with the average "innocent" packet's job size J_I and attack job sizes averaging J_A. Now let us assume the system is operating at capacity: there are some C packets per second arriving at the system, with an average job size of J_I per packet. If some of those packets are instead sized $J_A > J_I$, the system will be unable to keep up with the offered load and be forced to drop some packets. If an attacker injects one packet of sized $J_A = 10$, and all other packets are $J_I = 2$, then the system will be forced to drop 5 innocent packets in order to process the additional attack packet. In our simulations with Pigasus (§3.6.2), we found that in practice, an attacker could force Pigasus' reassembly engine to drop roughly 300 innocent bits for every bit of input attack traffic.

3.2.1 High Pervasiveness and Potency of ACAs

Unfortunately, the literature is full of examples of packet processing engines vulnerable to ACAs. For example, in 2020, researchers showed that they could slow the popular open-source software switch, Open vSwitch, to support just 1% of its typical throughput using a small 1 Mbps attack stream designed to exploit algorithmic complexity. The attack exploited a vulnerability in the Tuple-Space Search (TSS) [139] algorithm for packet classification known as "Tuple Space Explosion" (TSE) [46, 47].

In 2018, [143] identified a vulnerability in the Linux kernel's TCP reassembly logic. Although the Linux implementation uses a more sophisticated data-structure to manage out-of-order flows (Red-Black Trees), the bug allowed malicious peers to consume an excessive number of CPU cycles using specially-crafted inputs. The bug was addressed by a patch that streamlined processing enough to render the attack "not critical" [54]; while this may be sufficient for

the current line rate supported by kernel networking, the vulnerability will inevitably resurface alongside the next generation of line-rates.

An entire sub-literature of research [2, 22, 45, 136] describes attacks on deep-packet inspection (DPI) engines (*e.g.*, Pigasus [162], Snort [120], Suricata [49]) via Regular expression Denial of Service (ReDoS). A ReDoS attack crafts packets with payloads that are carefully designed to traverse multiple states in regular expression automata — the more states the packet triggers in the automata, the larger the J_A for that packet. Previous work has shown that an attacker responsible for only 10% of the traffic entering a regular expression engine can slow down legitimate traffic by up to 500% [2]. The literature is rife with other examples: ACAs that exploit decompression algorithms, sorting, hash tables, *etc.* [70, 88, 111, 112].

3.2.2 Existing Defenses are Insufficient

Resource isolation is insufficient to prevent ACAs in a networked setting. Many systems aim to shield users from the actions of other (potentially malicious) users by allocating each one a fixed slice of the shared resource (i.e., *resource isolation*). Unfortunately, the networking equivalent of resource isolation — *Fair Queueing* (FQ) [51] — can be easily circumvented, making it incapable of providing performance isolation. FQ schedules packets for processing in such a way as to divide service time equally between classes of traffic — service time might be divided evenly by network connection, by class of traffic (*e.g.*, HTTP vs VOIP traffic), or by sender. At first glance, it might appear that this would prevent an attacker from consuming more than their "fair share" of service time. Unfortunately, on the Internet, attackers have numerous ways to easily *spoof* the source IP address of their traffic — leading to the appearance that the attack traffic originates from *multiple* users.

Existing, algorithms-specific solutions lead to undesirable tradeoffs. The *de facto* mitigation technique for ACAs in network subsystems is to shrink the gap between J_I, the innocent job size, and J_A, the worst-case attack job size. However, this leads to undesirable trade-offs between common-case usability and ACA resilience.

Returning to the flow reassembly case, one might enforce that no linked list ever extends further than a chain of four packets, and if additional out-of-order packets arrive, the flow is simply reset. This approach mitigates the ACA: where a malicious packet might have led to the loss of n innocent packets in the base design, we can bound J_A to bring it closer to J_I and limit the malicious packet to only cause a loss of $m < n$ packets. Unfortunately, imposing a maximum length on the reassembler limits usability in the common case: we reduce J_A, but we also limit the NF's ability to handle innocent highly out-of-order flows, *even in scenarios where the system has excess capacity and can feasibly service them.* Thus, the system designer is left with two equally unappealing alternatives. They can either set a higher limit on J_A, allowing the system to service a wider range of flows but leaving it more vulnerable to ACAs, or they can set a lower limit on J_A, thereby sacrificing the ability to serve certain innocent flows for the sake of higher ACA resilience.

Network subsystems today come with a variety of such patches in an effort to restrict J_A, and sacrifice some property or the other (*e.g.*, common-case performance, or memory efficiency)

in exchange for ACA resilience. Additionally, the application-specific nature of these patches means that there is no *general* solution for mitigating ACAs — every patch must be constructed from scratch for each new ACA. This motivates our search for an attack mitigation strategy that is both general, and obviates the need to make undesirable tradeoffs in order to achieve resiliency against ACAs.

3.3 Incongruity-Aware Workload Model

In order to facilitate a first-principles analysis of algorithmic complexity attacks, we start by formulating a theoretical model to capture workload dynamics in §3.3.1. Next, we characterize the adversary's capabilities and our threat model in §3.3.2. In §3.3.3, we formally define the *Displacement Factor* (DF), a novel metric to quantify the harm induced by ACAs.

3.3.1 System Model

Packets and jobs: At the heart of our model is a packet processor that serves packets appearing on an ingress link of capacity R Gbps. Each packet requires a certain amount of time to be processed (*e.g.*, due to computation, I/O, memory lookups, *etc.*), and thus can be characterized by two independent variables: a packet size (in bits) and a job size (in seconds). For convenience, we also tag each packet with a class: class I packets correspond to innocent traffic and class A packets correspond to adversarial traffic; however, note that this tag is only relevant for the purpose of our analysis, and is not visible to the underlying system.

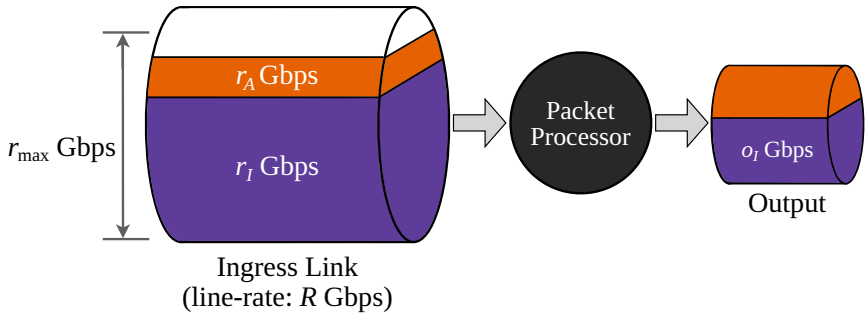


Figure 3.3: Refined model treating the workload as a superposition of *purely stochastic* innocent traffic (purple) and a sliver of *adversarial* traffic (orange).

We assume that packets belonging to innocent traffic follow certain packet and job size distributions, with P and J denoting continuous random variables sampled from these distributions, respectively. Let $f_P(p)$ and $f_J(j)$ denote their probability density functions (pdf),[1] and $\mathbb{E}[P]$ and $\mathbb{E}[J]$ denote the corresponding expectations. Table 3.1 contains a summary of the notations used in the model.

Goodput: Let r_I denote the input traffic rate (in Gbps) for class I traffic on the ingress link. For simplicity, we assume that packet arrivals have a constant inter-arrival time; i.e., the inter-

[1]In general, the packet size and job size may be correlated, and we use $f_{P,J}(p, j)$ to denote the joint pdf.

Notation	Description
R	Link capacity (in Gbps)
P	Packet size of class I traffic (random variable)
J	Job size of class I traffic (random variable)
$f_P(p)$	Probability density function of packet size P
$f_J(j)$	Probability density function of job size J
$P_{\min}$, $P_{\max}$	Minimum, maximum packet sizes
$J_{\max}$	Maximum job size
r_I	Input traffic rate (in Gbps) for class I traffic
$r_{\max}$	Maximum serviceable traffic rate
o_I	Output traffic rate (in Gbps) for class I traffic
$\alpha(r_I)$	Displacement Factor (DF)

Table 3.1: Summary of notations used in the model.

arrival time is $\frac{\mathbb{E}[P]}{r_I}$ seconds for innocent traffic. We define the system *goodput*, denoted as o_I, as the output traffic rate corresponding to class I traffic; i.e., the *useful* throughput that the system can sustain. Note that the system is designed to serve innocent traffic, and the maximum serviceable traffic rate without dropping packets is given by $r_{max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$ (in Gbps). Thus, in the absence of any adversarial traffic, the goodput $o_I = r_I$ when $r_I \leq r_{\max}$. The system model is depicted in Figure 3.3.

3.3.2 Threat Model

In order to model algorithmic complexity attacks, we allow a rate-limited adversary to inject a stream of *adversarial* (class A) traffic into the ingress link. Let r_A denote the input traffic rate for class A traffic. To enforce line-rate semantics, we impose the constraint $r_I + r_A \leq R$. Our threat model assumes an attacker that is overpowered relative to what we believe a practical attacker is capable of. In particular, we assume that the adversary is aware of all aspects of the underlying system ("transparent" model), as well as the innocent packet and job size distributions, and always uses the *optimal attack strategy*. In particular, the adversary crafts packets with the best choice of packet size and job size to maximize the harm to the system, where the harm is measured by reduction in goodput as defined in §3.3.3. The adversary is *not* capable of: (a) inspecting individual packets as they appear on the ingress link, (b) affecting the job sizes of class I packets (*e.g.*, by tainting shared state), or (c) amplifying their attack bandwidth using other means (*e.g.*, reflection-based amplification).

3.3.3 Quantifying Vulnerability

We measure the harm induced by the adversary using the volume of innocent traffic "displaced" under a given attack traffic input rate r_A. Specifically, we write the goodput o_I as $o_I(r_I, r_A)$ here to explicitly express its dependence on r_I and r_A. Then the volume of innocent traffic displaced is $o_I(r_I, 0) - o_I(r_I, r_A)$, i.e., how far the goodput deviates from the goodput in the absence of an adversary ($r_A = 0$). We then quantify the vulnerability of the system using

the *Displacement Factor* (DF), α, defined as the adversary's payoff relative to the amount of resources they invest:

$$DF = \frac{\text{Innocent traffic displaced (Gbps)}}{\text{Attack bandwidth used (Gbps)}}$$

A DF of 5 means an attacker can force the packet processor to drop 5 bps of innocent traffic for every 1 bps of attack traffic injected into the system. More formally, we can write the DF as follows:

$$\alpha(r_I) = \sup_{r_A} \frac{o_I(r_I, 0) - o_I(r_I, r_A)}{r_A}. \tag{3.1}$$

We take the supremum over the attack traffic rate r_A to capture the adversary's most efficient attack.

3.4 Mitigating ACAs using Scheduling

In this section, we demonstrate how scheduling can be used to effectively mitigate ACAs in a networked setting. As a starting point, we first consider two commonly-used scheduling policies, First-Come First-Served (FCFS) and Fair Queueing (FQ). We show in §3.4.1 and §3.4.2 below that under both FCFS and FQ, the DFs become unbounded in some regimes of system parameters. Consequently, systems that use FCFS or FQ scheduling *cannot* rely on the scheduler to protect against ACAs.

To build intuition as to how a job-size based scheduling policy can limit the harm induced by the adversary, we then present a scheduling policy called Shortest Job First (SJF). We show that SJF has a DF upper bounded by a constant that is independent of $J_{\max}$, improving upon both FCFS and FQ; however, this constant grows as the average packet size for innocent traffic, $\mathbb{E}[P]$, increases. We then present the policy underlying SurgeProtector: Packet-Size Weighted Shortest Job First (WSJF). We show that WSJF further removes the dependence on $\mathbb{E}[P]$, achieving a maximum DF of 1.

3.4.1 First-Come First-Serve (FCFS)

As the name suggests, *First-Come First-Serve* (FCFS) serves jobs in the order that they appear on the ingress link. Under FCFS, in order to maximize harm, the adversary crafts packets with the smallest possible packet size, $P_{\min}$, and the largest possible job size, $J_{\max}$.

As depicted in Figure 3.4, using small-sized packets encoding large jobs enables an attacker to consume a significant fraction of CPU (i.e., service time) despite using only a small amount of NIC time (i.e., attack bandwidth), throttling goodput. Intuitively, this happens because FCFS serves jobs in the order of arrival regardless of their sizes. Therefore, if an adversary can craft packets with arbitrarily large job sizes, they can also reduce the traffic rate for innocent packets to an arbitrarily large degree. We show in Claim 3 below that the adversary can achieve unbounded DF under FCFS as $\frac{J_{\max}}{P_{\min}}$ becomes large.

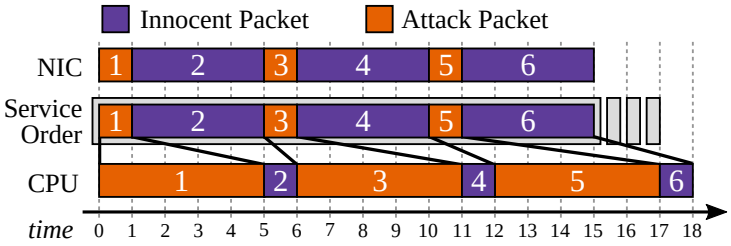


Figure 3.4: FCFS fails to protect against ACAs. Using minimum-sized packets encoding maximum-sized jobs allows the attacker to induce significant amounts of work in the system.

Claim 3 (DF of FCFS). *Under FCFS, for any innocent input traffic rate r_I and any packet size and job size distributions, the Displacement Factor $\alpha_{FCFS}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$.*

Proof. Consider any innocent input traffic rate r_I and any packet size and job size distributions with expectations $\mathbb{E}[P]$ and $\mathbb{E}[J]$. Observe that, over any time period of length T seconds, the number of *class I* packets appearing on the ingress link is $N_I = \frac{r_I T}{\mathbb{E}[P]}$. Similarly, the number of *class A* packets appearing on the link over the same period is $N_A = \frac{r_A T}{P_{\min}}$. FCFS guarantees that these $(N_I + N_A)$ jobs will be scheduled before any jobs that arrive afterwards. Also, the total time required to serve these jobs is $(N_I \cdot \mathbb{E}[J] + N_A \cdot J_{\max})$ seconds, yielding in expectation $N_I \cdot \mathbb{E}[P]$ bits worth of innocent traffic on the egress link. Thus, in the long-run, the goodput o_I can be upper-bounded as follows:

$$o_I(r_I, r_A) \leq \lim_{T\to\infty} \frac{N_I \cdot \mathbb{E}[P]}{N_I \cdot \mathbb{E}[J] + N_A \cdot J_{\max}} = \frac{r_I}{r_I \cdot \frac{\mathbb{E}[J]}{\mathbb{E}[P]} + r_A \cdot \frac{J_{\max}}{P_{\min}}}.$$

We can then lower-bound the DF $\alpha_{\text{FCFS}}(r_I)$ as follows:

$$\alpha_{\text{FCFS}}(r_I) = \sup_{r_A} \frac{\min\{r_I, r_{\max}\} - o_I(r_I, r_A)}{r_A} \tag{3.2}$$

$$\geq \sup_{r_A} \frac{1}{r_A} \left(\min\{r_I, r_{\max}\} - \frac{r_I}{r_I \cdot \frac{\mathbb{E}[J]}{\mathbb{E}[P]} + r_A \cdot \frac{J_{\max}}{P_{\min}}} \right) \tag{3.3}$$

$$\geq \frac{J_{\max}}{P_{\min}} \frac{\min\{r_I, r_{\max}\}}{2r_I \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)}, \tag{3.4}$$

where recall that $r_{\max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$, (3.2) is true since the goodput is $\min\{r_I, r_{\max}\}$ under FCFS in the absence of adversarial traffic, (3.3) applies the upper bound on $o_I(r_I, r_A)$, and (3.4) is obtained by setting r_A as follows: $r_A = \frac{r_I P_{\min}}{J_{\max}} \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)$. Therefore, $\alpha_{\text{FCFS}}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$. □

3.4.2 Fair Queueing (FQ)

Fair Queueing is a scheduling algorithm that is widely employed in switches and network processors. FQ and its variants (*e.g.*, WFQ, DRFQ [65]) ensure that one or more shared resources (*e.g.*, network throughput, processor time, *etc.*) are evenly partitioned among a number of competing flows. While this scheme performs well when these flows are operated by good faith

users seeking fair arbitration over a shared, limited resource, it does not translate well to the adversarial setting. The fundamental problem is that FQ only guarantees equitability across *flows*, thereby allowing a malicious user to occupy a disproportionately high fraction of the shared resource(s) by spawning more flows. Further, using FQ at source IP granularity is also insufficient because of the possibility of source address spoofing.

As depicted in Figure 3.5, using small-sized packets across a large number of flows enables an attacker to consume a significant fraction of service time using only a small amount of attack bandwidth. As we show in the proof for Claim 4, the DF under FQ ultimately scales with $\frac{J_{\max}}{P_{\min}}$, and, as in the case of FCFS, can become unbounded.

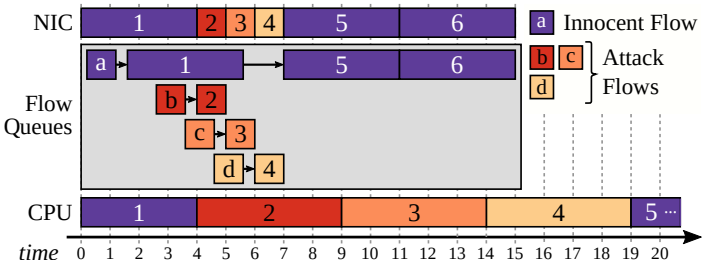


Figure 3.5: FQ fails to protect against ACAs. Since FQ allocates service time *per flow*, the attacker can trick the FQ scheduler into giving them a large fraction of service time using multiple flows. In steady-state, the attacker receives 75% of the total service time despite using a small attack bandwidth.

Claim 4 (DF of FQ). *Under FQ, for any innocent input traffic rate r_I and any packet size and job size distributions, the Displacement Factor $\alpha_{FQ}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$.*

Proof. Assume that the input traffic rate for innocent traffic, r_I, is split equally among k innocent flows, while each packet of adversarial traffic corresponds to a distinct attack flow. As in FCFS, the adversary maximizes the harm to the system by crafting packets with the smallest possible packet size $P_{\min}$ and the largest possible job size $J_{\max}$.

Consider the state of the system at time $T > J_{\max}$. Observe that, in expectation, the *maximum number* of innocent jobs in each of the k flow queues with virtual clock $\leq T$ is $N_I = \frac{T}{\mathbb{E}[J]}$. Conversely, the number of adversarial jobs with virtual clock $\leq T$ is given by $N_A = \frac{(T-J_{\max})r_A}{P_{\min}}$. FQ ensures that all $(N_A + k \cdot N_I)$ jobs will be scheduled before any jobs that arrive afterwards. Also, the total time required to serve these jobs is given by the expression: $\frac{(T-J_{\max})r_A}{P_{\min}} \cdot J_{\max} + k \cdot T$. Then, the goodput o_I can be upper-bounded as follows:

$$o_I(r_I, r_A) \leq \lim_{T\to\infty} \frac{k \cdot \frac{T}{\mathbb{E}[J]} \cdot \mathbb{E}[P]}{\frac{(T-J_{\max})r_A}{P_{\min}} \cdot J_{\max} + k \cdot T} = \frac{k \cdot r_{\max}}{r_A \cdot \frac{J_{\max}}{P_{\min}} + k},$$

where recall that $r_{\max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$. We can then lower-bound the DF $\alpha_{\text{FQ}}(r_I)$ as follows:

$$\alpha_{\text{FQ}}(r_I) = \sup_{r_A} \frac{\min\{r_I, r_{\max}\} - o_I(r_I, r_A)}{r_A} \tag{3.5}$$

$$\geq \sup_{r_A} \left(\min\{r_I, r_{\max}\} - \frac{k \cdot r_{\max}}{r_A \cdot \frac{J_{\max}}{P_{\min}} + k} \right) \tag{3.6}$$

$$\geq \frac{J_{\max}}{P_{\min}} \frac{\min\{r_I, r_{\max}\}}{2kr_{\max}\left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}}\right)}, \tag{3.7}$$

where (3.5) is true since the goodput is $\min\{r_I, r_{\max}\}$ under FQ in the absence of adversarial traffic, (3.6) applies the upper bound on $o_I(r_I, r_A)$, and (3.7) is obtained by setting $r_A = \frac{kr_{\max}P_{\min}}{J_{\max}}\left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}}\right)$. Therefore, $\alpha_{\text{FQ}}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$. □

3.4.3 Shortest Job First (SJF)

FCFS's obliviousness to job sizes and FQ's focus on per-flow fairness leaves them both susceptible to ACAs. In order to prevent ACAs, we need a scheduling policy that considers job sizes without being vulnerable to flow inflation. *Shortest Job First* (SJF) is a popular policy for scheduling jobs in a non-preemptive system. As the name suggests, at any instant, SJF prioritizes the queued job with the smallest (initial) job size.

We show in Theorem 2 below that the DF under SJF is upper bounded by a small constant independent of both $J_{\max}$ and $f_J(j)$. The intuition behind why SJF works well is simple: if the adversary produces packets whose jobs are *too expensive* to process, they will simply be de-prioritized and never end up being served. Instead, if the adversary produces packets whose jobs are *too cheap*, they will fail to push the system into overload. As depicted in Figure 3.6, the attacker's optimal strategy is to pick a job size corresponding to a "sweet spot" in the innocent job size distribution, and use minimum-sized packets to inflate their packet rate (and, consequently, the total work injected into the system). This allows them to displace *some* innocent traffic, achieving a worst-case constant DF. As we show in the proof for Theorem 2, the DF under SJF scales as the ratio between the average packet size for innocent traffic, $\mathbb{E}[P]$, and the minimum packet size, $P_{\min}$.

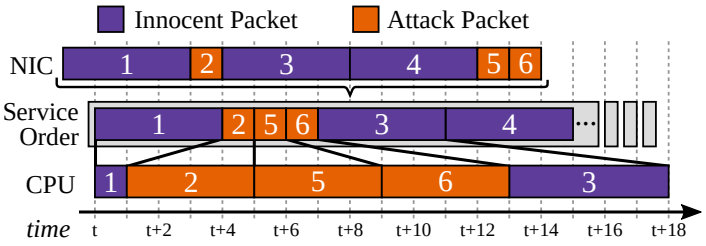


Figure 3.6: In order to exploit SJF, the attacker uses minimum-sized packets with a job size (i.e., CPU time) between that of packets 1 and 3. The attack packets (i.e., 2, 5, 6) are scheduled before more expensive ones (3, 4), pushing the system into overload and displacing packet 4.

Theorem 2 (DF of SJF). *Under SJF, for any innocent input traffic rate r_I and any packet size and job size distribution, the Displacement Factor is upper bounded as:*

$$\alpha_{SJF}(r_I) \leq \frac{\mathbb{E}[P]}{P_{\min}} \cdot \rho,$$

where $\rho = \min\left(\frac{r_I}{r_{\max}}, 1\right) \in [0, 1]$ is the load on the system due to innocent traffic.

Optimal attack strategy. We first characterize the optimal attack strategy of the adversary under SJF for a given innocent input traffic rate r_I and a given adversarial input traffic rate r_A. It is easy to see that the adversary should craft packets with the smallest possible packet size $P_{\min}$ since the job scheduling under SJF does not depend on packet sizes.

To reason about the optimal choice of adversarial job sizes, we first consider the case where the adversary picks certain job size J_A. Then innocent jobs with size $\leq J_A$ and adversarial jobs have priority over innocent jobs with size $> J_A$. Therefore, innocent jobs with size $> J_A$ will be 'displaced', i.e., never get served, if r_A is large enough to overload the processor with innocent jobs with size $\leq J_A$ and adversarial jobs. Consequently, the goodput consists of innocent packets whose job sizes are no larger than J_A.

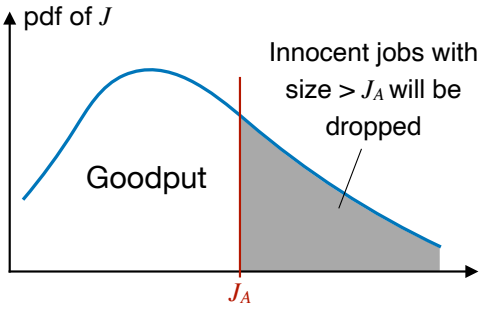


Figure 3.7: Optimal choice of adversarial job size J_A.

We now argue that the adversary only needs to pick one deterministic job size without loss of optimality. To see this, suppose the adversary crafts packets whose job sizes are either J_A or J'_A, where $J_A < J'_A$. But if the adversary swaps the packets with job size J'_A for packets with job size J_A, they can only displace more or equal innocent traffic. Therefore, we can restrict our attention to attack strategies with one deterministic job size.

We characterize the adversary's optimal choice for the job size in Lemma 2 below. Here for simplicity, we assume that the innocent packet size P and job size J are independent. We will remove this assumption when we present WSJF next. Recall that the pdf of the innocent job size J is denoted by $f_J(\cdot)$.

Lemma 2 (Optimal Attack Strategy for SJF). *Consider the SJF policy for job scheduling and any innocent input traffic rate r_I and any adversarial input traffic rate r_A. Then the adversary can minimize the goodput by choosing the job size, J_A, to be the solution of the following equation if the solution satisfies $J_A \leq J_{\max}$:*

$$\frac{r_I}{\mathbb{E}[P]} \int_0^{J_A} j \cdot f_J(j)\, dj \; + \frac{r_A}{P_{\min}} \cdot J_A = 1, \tag{3.8}$$

and $J_A = J_{\max}$ otherwise.

Proof. We have argued that the adversary only needs to pick one deterministic job size. It remains to show that the r_A given in the lemma minimizes the goodput. In (3.8), if the solution satisfies $J_A \leq J_{\max}$, the term $\frac{r_I}{\mathbb{E}[P]} \int_0^{J_A} j \cdot f_J(j) dj$ is the workload for the processor contributed

by innocent packets with job size $\leq J_A$. Since these packets get served by the processor, they constitute the goodput. We consider the following two cases: (i) *The adversary picks a job size larger than* J_A. In this case, more innocent jobs will get served since smaller jobs are prioritized, resulting in a larger goodput. (ii) *The adversary picks a job size* $J'_A < J_A$. In this case, the total workload from innocent jobs with size $\leq J'_A$ and adversarial jobs is given by

$$\frac{r_I}{\mathbb{E}[P]} \int_0^{J'_A} j \cdot f_J(j) dj + \frac{r_A}{P_{\min}} \cdot J'_A < 1.$$

So some innocent jobs with size $> J'_A$ will also get served. More precisely, the processor has more capacity left for innocent jobs when the adversarial job size is J'_A compared to when the adversarial job size is J_A. Thus the goodput is higher under J'_A. Combining the two cases, it follows that the solution J_A to (3.8) is the optimal choice for the adversary.

When the solution to (3.8) satisfies $J_A > J_{\max}$, the adversary cannot displace any innocent traffic no matter what the job size is. So simply setting $J_A = J_{\max}$ is an optimal choice. □

The remainder of the proof for Theorem 2 is very similar to that for Theorem 3 in the next section, so we elide this part of the proof for the sake of brevity. Unlike FCFS and FQ, SJF *does* impose an upper bound on the DF, limiting the extent that an attacker can cause harm to the system. SJF has an upper bound that depends on $\frac{\mathbb{E}[P]}{P_{\min}}$, which is approximately a factor of 8 given typical innocent packet size distributions. We show that we can further improve this bound using *Weighted* SJF.

3.4.4 Weighted Shortest Job First (WSJF)

A fundamental limitation of the policies described so far is that they altogether ignore the *packet size* information encoded in an incoming packet. This enables an adversary to greatly inflate their job arrival rate using minimum-sized packets, leading to either an unbounded DF (in the case of FCFS and FQ), or one that scales inversely with $P_{\min}$ (in the case of SJF). Then, a natural question is: *can we do better by leveraging this readily-available information?*

Here, we propose to use *Packet-Size Weighted Shortest Job First* (WSJF), a variant of SJF that prioritizes the packet with the smallest *job-to-packet-size ratio*. We show in Theorem 3 below that the DF under WSJF is *at most 1*, which implies that the attacker must invest at least as much attack bandwidth into the attack as the innocent bandwidth they wish to displace. The intuition is that *WSJF minimizes the system's work-per-bit, thereby preventing an adversary from consuming a high fraction of processing cycles unless they invest a proportionally high bandwidth into the attack*. To concretize this notion, consider the scenario depicted in Figure 3.8. For the sake of simplicity, let us assume that all innocent packets have a job size of 1 unit time and that the system is operating at capacity. Also assume that in steady-state, the WSJF queue contains 3 packets with packet sizes $P_1 = 5$, $P_2 = 3$, and $P_3 = 1$. Observe that WSJF would serve these packets in decreasing order of packet size (i.e., P_1 before P_2, and P_2 before P_3), corresponding to scenario S_0.

Consider an attacker that seeks to displace a single innocent packet (i.e., with packet size P_3) from this queue with one of their own. In order to do this, the attacker *must* inject an attack packet of size $P_4 \geq P_3$ with a job size of 1 (scenario S_1); a smaller job size would introduce slack in the system load (allowing it to periodically serve innocent packets of size P_3 as well), while a smaller packet size would result in the attack packet never being served. Thus, the attacker is forced to inject as many bits as they wish to displace.

Suppose, instead, that the attacker wishes to displace *two* innocent packets (with sizes P_2 and P_3). The attacker now has two options: they can either inject two packets with sizes $P_5 \geq P_2$ and $P_4 \geq P_2$ and unit job size each (scenario S_2), or a single packet of size $P_6 \geq 2P_2$ and a job size of 2. Once again, the attacker's bandwidth investment matches or exceeds the displaced goodput. As we demonstrate in the proof for Theorem 3, this result generalizes to any load, as well as any job and packet size distributions of innocent traffic.

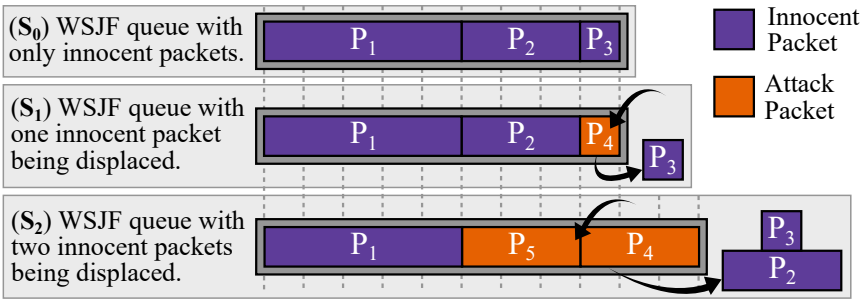


Figure 3.8: WSJF service order in steady state.

Theorem 3 (DF of WSJF). *Under WSJF, for any innocent input traffic rate r_I and any packet size and job size distribution, the Displacement Factor is upper bounded as:*

$$\alpha_{WSJF}(r_I) \leq \rho \leq 1,$$

where $\rho = \min\left(\frac{r_I}{r_{\max}},\ 1\right) \in [0,\ 1]$ is the load on the system due to innocent traffic.

Optimal attack strategy. We again first characterize the optimal attack strategy of the adversary under WSJF for a given innocent input traffic rate r_I and a given adversarial input traffic rate r_A. Under WSJF, the harm that an adversary can induce is fully determined by the job-to-packet-size ratio of the adversarial traffic, denoted as Z_A, as opposed to the individual values of job size and packet size. To see this, note that WSJF schedules jobs solely based on their job-to-packet-size ratios, and that the rate at which the adversary generates work for the processor is $r_A \cdot Z_A$, which also depends on the job size and packet size only through their ratio. Therefore, we assume that the adversary uses packet size $P_{\min}$ without loss of optimality, and picks a job-to-packet-size ratio Z_A that results in job size $Z_A \cdot P_{\min}$.

The reasoning for the optimal choice of Z_A is similar to that for the optimal choice of J_A under SJF. The only difference is that under WSJF, whether an innocent packet gets displaced or not is determined by its job-to-packet-size ratio rather than its job size. Following similar arguments, we establish Lemma 3 below, whose proof is omitted for the sake of brevity. Here

we use $f_{P,J}(p, j)$ to denote the joint pdf of the innocent packet size and job size. Note that we do not make independence assumptions between them.

Lemma 3 (Optimal Attack Strategy for WSJF). *Consider the WSJF policy for job scheduling and any innocent input traffic rate r_I and any adversarial input traffic rate r_A. Then the adversary can minimize the goodput by choosing the job-to-packet-size ratio, Z_A, to be the solution of the following equation if the solution satisfies $Z_A \cdot P_{\min} \leq J_{\max}$:*

$$\frac{r_I}{\mathbb{E}[P]} \int_{P_{\min}}^{P_{\max}} \int_{0}^{p \cdot Z_A} j \cdot f_{P,J}(p, j)\, dj \;\; dp \; + r_A \cdot Z_A = 1, \tag{3.9}$$

and $Z_A = \frac{J_{\max}}{P_{\min}}$ otherwise.

Displacement Factor of WSJF

Proof. We divide the discussion into two cases: $r_I < r_{\max}$ (underloaded by innocent traffic) and $r_I \geq r_{\max}$ (overloaded by innocent traffic).

Case 1 ($r_I < r_{\max}$): Consider a period of T seconds, with a total of N innocent packets arriving during this period. Let $S = \{(p_1, j_1), (p_2, j_2), \ldots, (p_N, j_N)\}$ denote this set of arrivals, where $p_i \in [P_{\min},\ P_{\max}]$ and $j_i \in [0,\ J_{\max}]$ denote the packet size and job size corresponding to the i'th packet, respectively. Without loss of generality, we choose the index of each packet, i, such that $\frac{j_i}{p_i} \leq \frac{j_{i+1}}{p_{i+1}}\ \forall i$.

We now turn to the service order of these N innocent packets under WSJF. In particular, note that since WSJF serves packets in *increasing order of their job-size-to-packet-size-ratio*, packet *1* is served before packet *2*, packet *2* before packet *3*, and so on. Further, since we assumed that $r_I < r_{\max}$, it follows that in steady state (i.e., for sufficiently large T), all N jobs will be served. Now, consider an adversary who wishes to displace $k \in \{1, \ldots, N\}$ innocent packets. In order to do this, they must inject some $x \geq 0$ attack packets with packet size p_A and job size j_A. Note that the attacker's input traffic rate can be written as: $r_A = \lim_{T \to \infty} \frac{x \cdot p_A}{T}$. Now, in order to both be served *and* displace k innocent packets, x, p_A, and j_A must satisfy the following constraints with probability 1:

$$\frac{j_A}{p_A} \leq \frac{j_{N-k+1}}{p_{N-k+1}}, \tag{3.10}$$

$$\sum_{i=1}^{N-k} j_i + x \cdot j_A \geq T - o(T), \tag{3.11}$$

where (3.11) further implies that

$$x \geq \frac{1}{j_A} \left(T - o(T) - \sum_{i=1}^{N-k} j_i \right). \tag{3.12}$$

In particular, (3.12) ensures that the adversarial workload pushes the system to capacity (otherwise, this slack would be applied towards serving additional traffic, implying that the adversary would fail to displace k innocent packets). Similarly, (3.10) ensures that all x adversarial

packets are served *before* packets $\{N-k+1,\ \ldots,\ N\}$ (otherwise, some of the last k innocent packets would be prioritized over the adversary's traffic).

Let g denote the *number of innocent bits displaced* by the adversary using $x \cdot p_A$ bits of their own traffic. We have: $g(k) = \sum_{i=N-k+1}^{N} p_i$. Now, in steady state, the displacement factor under WSJF can be expressed as follows with probability 1:

$$\begin{aligned}
\alpha_{\text{WSJF}}(r_I,\ r_A) &= \frac{r_I - o_I(r_I, r_A)}{r_A} \\
&= \lim_{T\to\infty} \frac{g(k)}{x \cdot p_A} \\
&= \lim_{T\to\infty} \frac{\sum_{i=N-k+1}^{N} p_i}{x \cdot p_A} && (3.13) \\
&\leq \lim_{T\to\infty} \frac{\sum_{i=N-k+1}^{N} p_i}{T - \sum_{i=1}^{N-k} j_i} \cdot \frac{j_A}{p_A} && (3.14) \\
&\leq \lim_{T\to\infty} \frac{\overbrace{\sum_{i=N-k+1}^{N} p_i \cdot \frac{j_{N-k+1}}{p_{N-k+1}}}^{\text{Term (R1)}}}{T - \sum_{i=1}^{N-k} j_i}, && (3.15)
\end{aligned}$$

where (3.14) is obtained by substituting the expression for x we derived in (3.12) into (3.13), and (3.15) is obtained by substituting the expression for $\frac{j_A}{p_A}$ we derived in (3.10) into (3.14). Now, since $\frac{j_i}{p_i} \leq \frac{j_{i+1}}{p_{i+1}}$ implying that $p_{i+1} \cdot \frac{j_i}{p_i} \leq j_{i+1}\ \forall i$, we can upper-bound Term (R1) as follows:

$$\begin{aligned}
\sum_{i=N-k+1}^{N} p_i \cdot \frac{j_{N-k+1}}{p_{N-k+1}} &\leq \sum_{i=N-k+1}^{N} j_i \\
&\leq \sum_{i=1}^{N} j_i \ - \sum_{i=1}^{N-k} j_i.
\end{aligned}$$

Substituting this expression back into (3.15), we have:

$$\alpha_{\text{WSJF}}(r_I,\ r_A) \leq \lim_{T\to\infty} \frac{\sum_{i=1}^{N} j_i - \sum_{i=1}^{N-k} j_i}{T - \sum_{i=1}^{N-k} j_i}.$$

Observe that the RHS is of the form $h(x) = \frac{t-x}{T-x}$, where $t = \sum_{i=1}^{N} j_i \leq T$ (i.e., the cumulative service time for innocent packets in the absence of adversarial traffic, which is constant for a given r_I), and $x = \sum_{i=1}^{N-k} j_i \in [0,\ t]$. Since h is a decreasing function of x on its domain, it follows that $\alpha_{\text{WSJF}}(r_I,\ r_A)$ achieves its maximum value when $x = 0$. Therefore, we can write:

$$\alpha_{\text{WSJF}}(r_I) \ \leq \ \lim_{T\to\infty} h(0) = \lim_{T\to\infty} \frac{t}{T} \leq \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} j_i. \qquad (3.16)$$

Now, for a given distribution of innocent packets and job sizes, the input rate and maximum serviceable rate for innocent traffic (r_I and $r_{\max}$, respectively), can be expressed as follows:

$$r_I = \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} p_i, \text{ w.p.1}, \tag{3.17}$$

$$r_{max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]} = \lim_{T\to\infty} \frac{\sum_{i=1}^{N} p_i}{N} \frac{1}{\frac{\sum_{i=1}^{N} j_i}{N}} = \lim_{T\to\infty} \frac{\sum_i p_i}{\sum_i j_i}, \text{ w.p.1}. \tag{3.18}$$

Then, we can define the *load* on the system due to innocent traffic, ρ, as follows:

$$\rho(r_I) = \frac{r_I}{r_{\max}} = \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} j_i \leq 1, \text{ w.p.1}. \tag{3.19}$$

Observe that (3.19) is identical to the RHS of (3.16). Thus, we can rewrite the maximum DF under WSJF: $\alpha_{\text{WSJF}}(r_I) \leq \rho$, as required.

Case 2 ($r_I \geq r_{\max}$): In this case, one can verify that there exists $K < N$ such that the system is underloaded with respect to packets with a job-size-to-packet-size-ratio of $\frac{j_K}{p_K}$ (i.e., the distribution of innocent traffic served is effectively truncated at this point). Following the same arguments as of those for Case 1 (the worst case being $r_I = r_{\max}$, corresponding to $\rho = 1$), we have:

$$\alpha_{\text{WSJF}}(r_I) \leq \rho.$$

Combining Case 1 and Case 2 completes the proof of Theorem 3.

□

To summarize, WSJF guarantees that for every 1 bps of innocent traffic that the attacker wishes to displace, they must invest *at least* 1 bps of their own bandwidth into the attack, significantly mitigating the potency of ACAs.

3.5 SurgeProtector Implementation

SurgeProtector interposes a WSJF scheduler in front of variable-time subsystems (*e.g.*, the reassembler discussed in §3.2). WSJF meets both our initial goals of *generality* and *not limiting the innocent traffic that can be served*. First and foremost, it provides a provable upper bound on the DF that is *independent of the underlying algorithms*. WSJF is a drop-in solution that can be applied to any algorithm, and hence, it is general.[2] Second, where many ACA solutions, *e.g.*, drop packets from flows that are determined to be too expensive to process, WSJF guarantees that all connections will be served so long as there is system capacity to do so (i.e., it is starvation-free when the system is at or below capacity). Hence, WSJF does not place any

[2]This assumes, for the moment, *a priori* knowledge of the packet processing time, which must be calculated based on the underlying algorithm. We address this point in more detail in §3.5.2.

limitations on innocent traffic under normal operation. In overload, the most computationally expensive packets *are* dropped,[3] but overall this *minimizes the rate of innocent traffic that is denied service.*

In order to validate our theoretical findings in the context of a real system, we incorporate SurgeProtector into the open-source Pigasus IDS [162]. A simplified block diagram of Pigasus is depicted in Figure 3.9. In §3.2, we briefly introduced the linked-list based design of Pigasus's FPGA-based TCP Reassembly engine (labelled ①), and demonstrated how it can be exploited by an adversary. It turns out that a second component of the IDS — the CPU-side Full Matcher (labelled ②) — is also vulnerable to a different type of complexity attack.

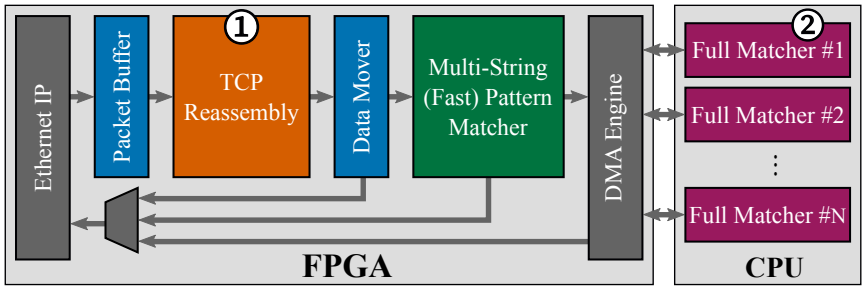


Figure 3.9: The simplified Pigasus IDS pipeline.

We begin in §3.5.1 with a brief overview of the two vulnerable components in Pigasus that we sought to protect. Over the course of implementing SurgeProtector, we encountered the following three important practical challenges. First, how do we predict job sizes? Second, since flow reordering is often undesirable, how do we guarantee in-order delivery for packets of the same flow? Finally, how do we ensure that the scheduler itself does not present a target for ACAs? We frame the implementation details of the SurgeProtector scheduler in the context of these three questions (§3.5.2–§3.5.4).

3.5.1 Overview of Vulnerable Components

FPGA-based TCP Reassembly: Recall that the goal of TCP reassembly is to reconstruct an in-order TCP bytestream from a sequence of out-of-order packets. The Pigasus reassembler, which is FPGA-based, prioritizes memory efficiency, and employs a linked list-based design to manage out-of-order flow state. While this achieves excellent memory utilization, the worst-case linear complexity of linked-list operations makes it susceptible to ACAs.

An example of this is depicted in Figure 3.10. When a new packet arrives (with PSN range [35, 50) in the example below), the reassembler linearly scans the list and inserts the node

[3] At this point, one might wonder: if WSJF drops the most expensive jobs in overload, why doesn't the adversary simply use less expensive jobs, thereby cajoling the scheduler into exclusively serving their traffic (*e.g.*, if innocent packets have a job size of 10 units, the adversary uses packets with a job size of 1 unit)? From an adversarial perspective, this turns out be an inefficient strategy; the adversary must now send 10× the number of packets to displace innocent traffic, corresponding to 10× as much attack bandwidth. In particular, this devolves the DoS attack into a *volumetric* one, which defeats the purpose of using an ACA in the first place.

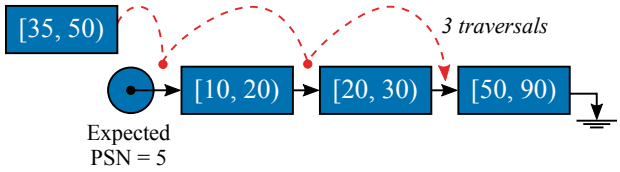


Figure 3.10: Linked-list state for an out-of-order flow.

at the appropriate position. In order to exploit this, an attacker crafts *highly out-of-order flows*, linearly increasing the number of traversals required for each subsequent attack packet. Finally, they use *minimum-sized packets* (with a 1-byte TCP payload) to inflate their packet arrival rate, maximizing the work injected into the system.

CPU-based Full Matching: As a signature-based IDS, Pigasus identifies malicious flows by comparing packet payloads against a database of known attack signatures ('rules'). To achieve high performance, it does so in two stages: it first uses a number of fast filters in hardware (i.e., the Multi-String Pattern Matcher) to quickly filter out innocent traffic; it then relays the remaining (small) fraction of possibly-malicious traffic to the CPU to perform more expensive regex analysis ('Full Matching' [162]). While Pigasus' first stage operates in constant time (and hence is not vulnerable to ACAs), the CPU-side Full Matching stage involves variable-time computation that is also input-dependent, making it vulnerable to ACAs.

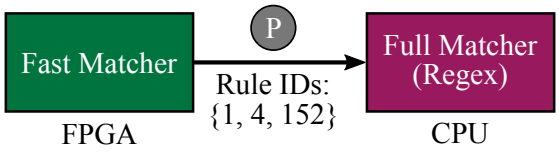


Figure 3.11: Pigasus Full Matching pipeline.

Pigasus's Full Matching pipeline is depicted in Figure 3.11. During the first stage, in addition to filtering out innocent packets, Pigasus also generates a list of candidate rules that the packet may ultimately match on. It then sends this list, along with the packet payload, to the CPU for processing. The CPU sequentially processes each rule in the list, stopping at the first rule that results in a match. The processing result (i.e., indicating whether to drop or forward the packet) is subsequently relayed back to the FPGA.

An attacker can exploit this by crafting attack packets that either: (a) result in a *large number of matches in the Fast Matching stage* (requiring the Full Matcher to evaluate many rules), (b) *trigger a regex search with super-linear runtime* in the Full Matcher (i.e., ReDoS-style attacks [36, 49, 156]), or both.

3.5.2 Predicting Job Sizes

SurgeProtector schedules packets based on job sizes, but, in practice, the time required to process a packet is not known *a priori*. A common approach to solve this problem — and one we employ in this work — is to use *heuristics* for job size estimation [93, 96]. In particular, we use the following heuristics to estimate job sizes for our target applications:

TCP Reassembly: a packet's job size is estimated as the *length of the out-of-order linked-list for the corresponding flow*. Despite its simplicity, this heuristic has two salient properties: first, since the number of traversals can never exceed the length of the linked-list, the estimate always *upper-bounds* a given packet's true job size; second, since the heuristic is computed on a per-flow basis, the adversary cannot affect the quality of estimates for innocent flows.

Full Matching: if K denotes the list of candidate rules identified by the fast matching stage, then the job size is estimated as $J = \Sigma_{k \in K}(z_k \cdot p)$, where z_k denotes the *maximum job-size-to-packet-size ratio observed for rule k thus far*, and p denotes the packet payload size. By using historical run-time data as feedback, the heuristic function 'learns' which rules are prone to complexity attacks and selectively deprioritizes them.

We implement and evaluate SurgeProtector using both these heuristics in §3.6.1. It is worthwhile to note that neither of these heuristics is 'ideal' in a theoretical sense. For example, in the case of TCP Reassembly, there *may* exist innocent TCP flows on the Internet for which the heuristic consistently overestimates job sizes by a significant margin, allowing the attacker to unfairly displace them. Similarly, in the case of Full Matching, an attacker *may* be able to manipulate the outcome of the heuristic for every rule, potentially causing large prediction errors for subsequent innocent packets.

In practice, this does not appear to be the case. For instance, in the case of TCP Reassembly, the heuristic yields accurate job size estimates for the vast majority of TCP flows, limiting the additional harm that an adversary can induce. Similarly, in the case of Full Matching, most rules don't have large variance in their job-size-to-packet-size ratios. We explore this further in §3.6.2, where we empirically evaluate the effect of using heuristics on SurgeProtector's DF upper-bound. *Empirically, we find that for both applications, the adversary's DF increases by no more than 5% of the upper-bound even when the adversary has perfect knowledge of the actual and heuristic-estimated job size distributions.* We leave the exploration of adversary-proof job size heuristics for arbitrary packet processing engines to future work (§3.7).

3.5.3 Keeping (TCP) Flows In-Order

Keeping packets within the same TCP flow in order is necessary to avoid degrading application performance [31, 52, 89, 134, 162]. While FCFS and FQ (along with its variants) guarantee that same-flow packets are served in-order, SJF and WSJF do not. In this section, we explore how to augment SurgeProtector to provide in-order service.

As a natural starting point, consider the following extension to WSJF, which we will refer to as *WSJF Head-of-Queue* (WSJF-HoQ). This policy maintains independent queues for each flow, with incoming packets being appended to the *end* of the corresponding flow queue. At any moment, the policy prioritizes the flow whose *leading* (Head-of-Queue) packet has the smallest job-to-packet size ratio; clearly, this maintains the desired in-order property. Then, we can ask: is this WSJF/FCFS hybrid a good policy?

Unfortunately, WSJF-HoQ turns out to be a poor strategy in the adversarial setting. The problem is as follows: while an innocent flow's packets may *typically* have a small job-to-packet

size ratio (making this flow a good candidate for service), eventually, a HoQ packet with a large job-to-packet size ratio will stifle the likelihood of the *entire* flow ever being served. Here, the adversary's optimal strategy is simply to send small packets encoding large jobs and wait for this situation to arise.

The fundamental problem with WSJF-HoQ is that *it evaluates entire flows on the basis of one packet, which may not be a good estimator of a flow's candidacy for service.* Based on this observation, we develop another variant of WSJF (hereafter referred to as *WSJF-Inorder*), which predicates its scheduling decision on all queued packets in the flow queue. As before, the policy maintains independent queues for each flow, with incoming packets appended to the tail of the corresponding queue. In scheduling, the policy computes a *rank* for each flow, f, and prioritizes the flow with the *lowest* rank:

$$Rank(f) = \frac{\Sigma_i J_i(f)}{\Sigma_i P_i(f)},$$

where $J_i(f)$ and $P_i(f)$ denote the job size and packet size of the i'th packet currently in f's flow queue, respectively. Thus, a flow's rank represents its *outstanding work per bit.* In the limit, this converges to $\frac{\mathbf{E}[J(f)]}{\mathbf{E}[P(f)]}$, the long-running average of the flow's inverse-throughput; by minimizing this quantity, WSJF-Inorder maximizes the overall throughput. Consequently, if an adversary wants the policy to consistently schedule (their) large jobs, they must offset the resulting work with a proportionally large number of packet bits, effectively reducing the DF they can achieve. *We use WSJF-Inorder to protect the TCP Reassembly component in Pigasus.*

3.5.4 Designing Adversary-Proof Schedulers

The final practical issue that we need to address is how to make sure that the scheduler itself will not expose a novel attack surface. While simple policies like FCFS can be implemented with minimal overhead, in order to implement WSJF we must be able to determine which packet has the minimum job-to-packet size ratio on a packet-by-packet basis. If this is done inefficiently, the scheduler itself may become a bottleneck. Another potential problem is that we can only hold a finite number of outstanding packets at any given time. Once the packet buffer becomes full, the system must drop packets in a way that cannot be exploited by an attacker.

There is extensive literature on designing efficient priority queues for packet scheduling in both hardware and software [6, 124, 127, 135, 148]. However, these schedulers typically handle buffer space exhaustion by simply dropping any incoming packet when the buffer is full [135]. While this approach simplifies their design — since they only need to support either Extract-Min or Extract-Max operations, and not both — it does not work well in the adversarial setting. For instance, suppose that we use PIFO [135] to implement WSJF and drop all incoming packets once we run out of buffer space. In this scheme, an attacker can quickly fill up the queue (with minimally-sized packets encoding maximally-sized jobs), eventually leaving the scheduler with no alternative but to pick the attacker's packets. *To avoid this issue, the scheduler must use Extract-Min to decide which packet to process next, and Extract-Max to decide which packet to drop once it runs out of buffer space.*

We augment the highly-efficient Hierarchical FFS (Find First Set) Queue [124, 148] to provide both EXTRACT-MIN and EXTRACT-MAX functionality by using a BSF (Bit Scan Forward) instruction to find the minimum element in each bitmap, and a BSR (Bit Scan Reverse) instruction[4] to find the maximum element. Figure 3.12 depicts the data-structure. An HFFS queue using 32-bit bitmaps and a height of h can represent 32^h unique priorities, and guarantees a worst-case run-time of $O(h)$ (i.e., constant) for all queue operations (INSERT, EXTRACT-MIN, and EXTRACT-MAX).

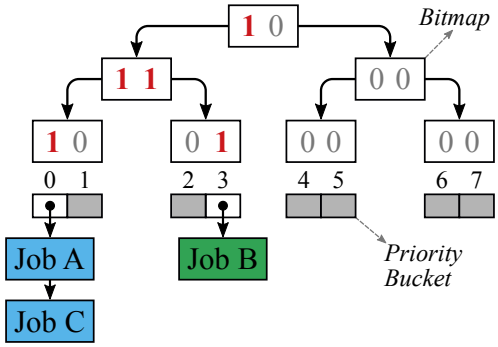


Figure 3.12: A Hierarchical FFS Queue implemented using 2-bit bitmaps and height of $h = 3$. A '1' in any bitmap indicates a non-empty priority bucket in the subtree rooted at that node. In order to find the *min* (or *max*) priority bucket, we recursively follow the leftmost (or rightmost) set bit.

In order to enable SURGEPROTECTOR to work in a general context, we implement the Hierarchical FFS Queue in both software and hardware. In Pigasus, the software and hardware implementations are used to realize WSJF queueing for Full Matching and TCP Reassembly, respectively. The software version is implemented in C++, and is further evaluated in §3.6.3. The hardware version of the HFFS priority queue is another key contribution of this thesis, and we describe it in detail in Chapter 4.

3.6 Evaluation

In this section, we evaluate the effectiveness of using SURGEPROTECTOR to defend against ACAs on the TCP Reassembler and Full Matching stage of the Pigasus IDS. We also evaluate the robustness of the Hierarchical FFS Queue (used to implement WSJF) against attacks targeting the scheduler itself.

3.6.1 SURGEPROTECTOR + Pigasus

How effective is SURGEPROTECTOR at mitigating ACAs on the TCP Reassembler? To answer this question, we emulate an adversary targeting Pigasus' TCP Reassembler using highly out-of-order attack flows, and measure the achieved performance in two modes of operation: using Pigasus' default scheduling policy (FCFS), and using SURGEPROTECTOR. For the purpose of this experiment, we use a synthetic trace containing innocent flows sampled from the 2014 CAIDA San Jose dataset [147], and 50 artificially-crafted attack flows.

[4]On modern CPUs, both BSR/BSF translate to single μops with a fixed latency of 3–5 cycles [68].

The attack flows are crafted as follows: we send 1B TCP packets with *alternating sequence numbers* starting with the ISN (i.e., ISN, ISN+2, ISN+4, and so on). With a sequence of N such packets, we can emulate an average adversarial job size corresponding to $\frac{1}{N}\sum_{i=0}^{N-1} i = \frac{(N-1)}{2}$ traversals. We use the optimal adversarial strategy for each mode of operation. In particular, for FCFS, we let N grow to Pigasus' maximum TCP window size of 16KB (by design, Pigasus will drop the flow at this point), then start over. For WSJF, we solve Equation 3.9 to determine the optimal adversarial job size, then choose N so as to achieve, on average, the corresponding number of traversals.

Empirically, we find that the maximum serviceable traffic rate of the system (i.e., $r_{\max}$) is 12 Gbps, and we fix the input rate for innocent traffic to 10 Gbps (corresponding to ~83% load). Figure 3.13 depicts the steady-state goodput in each mode of operation as we sweep the adversary's attack rate.

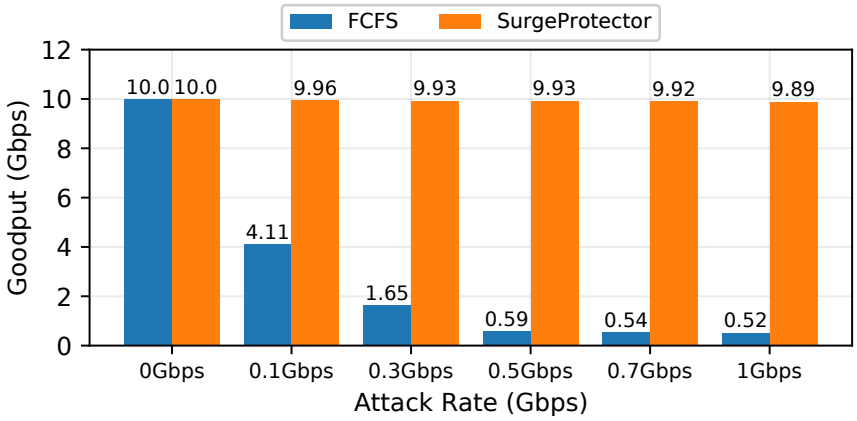


Figure 3.13: Goodput of Pigasus' TCP Reassembler under FCFS and SurgeProtector.

We observe that the goodput under FCFS drops significantly as the attack rate increases (*e.g.*, with an attack rate of 0.1 Gbps, the adversary is able to displace ~5.9 Gbps of innocent traffic). Conversely, with SurgeProtector, the goodput remains steady despite the increasing attack rate; in the worst case, at most 0.11 Gbps of innocent traffic is displaced.

In lieu of precise knowledge about the system design or the innocent traffic distribution, a practical adversary may also choose to 'probe' the space of attack parameters to determine the most effective adversarial strategy. In order to evaluate performance in this scenario, we emulate an adversary who incrementally changes the degree of out-of-orderness of attack flows while keeping the attack rate fixed at 0.3 Gbps. Figure 3.14 depicts the steady-state TCP Reassembly goodput with FCFS and SurgeProtector as we sweep the out-of-orderness of attack flows (measured in terms of the maximum number of concurrent out-of-order packets within each attack flow).

As expected, the goodput under FCFS gradually decreases as the attack flows become increasingly out-of-order (corresponding to larger job sizes per packet), while the goodput under SurgeProtector remains relatively unchanged.

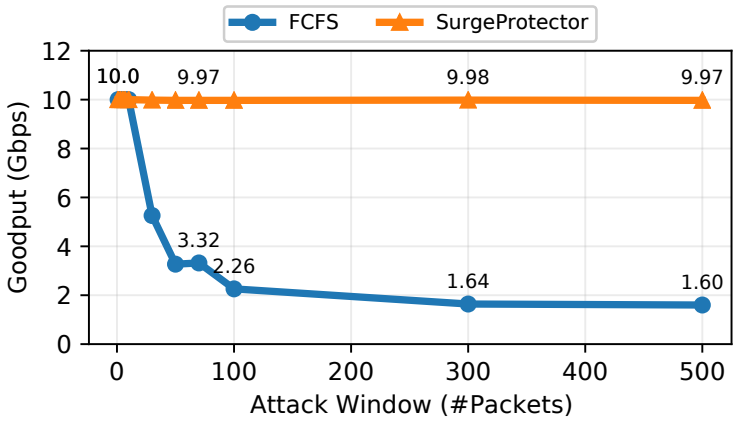


Figure 3.14: Goodput of Pigasus' TCP Reassembler for different degrees of out-of-orderness of attack flows.

How effective is SurgeProtector at mitigating ACAs on the Full Matching stage? As before, we answer this question by emulating an adversary targeting Pigasus' Full Matching stage, and measure the goodput under FCFS and SurgeProtector. We use a synthetic trace containing innocent flows sampled from all the traces used in [162]. In order to generate attack traffic, we pick the packet payload with the largest job size among all packets in the dataset, and craft an attack flow using this payload for every packet. Figure 3.15 depicts the steady-state goodput in each mode of operation as we sweep the adversary's attack rate.

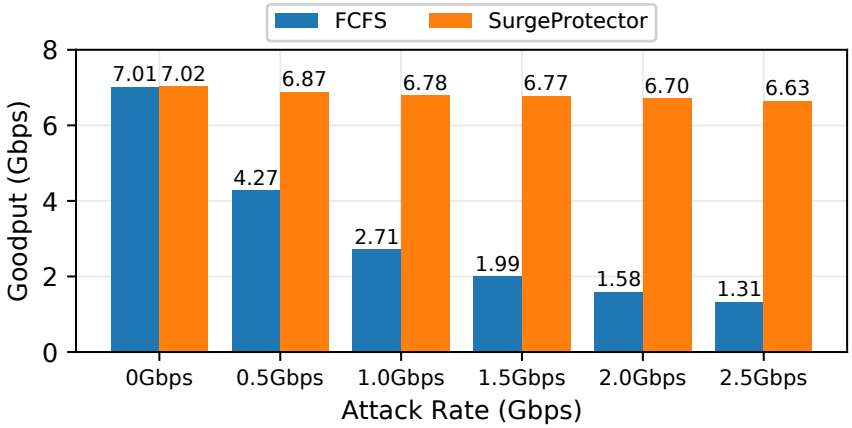


Figure 3.15: Goodput of Pigasus' Full Matcher under FCFS and SurgeProtector.

Once again, we observe that SurgeProtector significantly reduces the impact of the attack on innocent traffic compared to FCFS. In particular, we observe a maximum reduction in goodput of 0.4 Gbps for SurgeProtector (compared to 5.7 Gbps for FCFS).

3.6.2 SurgeProtector in Simulation

While the empirical evaluation in §3.6.1 demonstrates the efficacy of SurgeProtector in the context of a real system, it focuses a small number of attack input rates with just two scheduling

policies. In order to analyze a wider range of scheduling policies, applications, and a truly optimal adversary (i.e., one who is not constrained by the space of "practical" attack strategies[5]), we turn to an adversarial scheduling simulator that we developed in-house. The event-driven simulator, implemented in C++, is capable of modeling G/G/1/k queueing systems, supports both trace-driven and synthetic workloads, and exposes a convenient interface for plugging in a wide range of simulated application backends. An overview of the simulator pipeline is depicted in Figure 3.16.

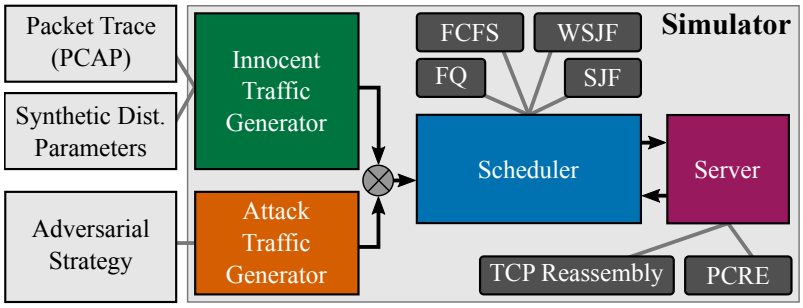


Figure 3.16: Simulator pipeline.

In order to quickly explore the space of different policies and heuristics for a variety of NFs, the simulator framework allows users to develop and 'run' their own simulated applications on the Server. It also provides traffic-generation modules for innocent and attack traffic, and includes tools for computing the optimal adversarial strategy under SJF and WSJF given innocent job and packet size distributions. We use now use the simulator to address several research questions of interest.

What is the worst-case DF an optimal adversary can achieve assuming the true job size is known *a priori*? Unlike the empirical setting, the simulator allows us to determine the *true* job size ahead of time. In the following simulated experiments, we use this information for the purpose of scheduling. In the context of TCP Reassembly, Figure 3.17 depicts the goodput and Displacement Factor achieved by different scheduling policies for various combinations of the input rate (r_I) and attack rate (r_A).[6] Each column corresponds to a certain, fixed r_I (going from 1 Gbps on the left, to 5 Gbps, and 10 Gbps). On the X-axis, we sweep the input attack rate from 10 Mbps to 10 Gbps. The bottom row depicts the steady-state goodput (in Gbps) as a function of the attack rate, while the top row depicts the corresponding DF.

[5]In particular, a practical adversary may not be powerful enough to craft packets with a *specific* job size. For instance, in the case of TCP reassembly, an adversary cannot, in practice, force K linked-list traversals on *every* attack packet; instead, they must settle for a uniform distribution over $\{0, \ldots, 2K+1\}$ (see §3.6.1), resulting in an *average* job size corresponding to K traversals. Simulation allows us to model a more powerful adversary who can precisely control their packets' job sizes.

[6]Note that, for each configuration, we use the attack parameters (P_A and J_A) corresponding to the optimal adversarial strategy. As we saw in §3.4.1 and §3.4.2, for both FCFS and FQ this corresponds to using minimally-sized packets encoding maximally-sized jobs. For SJF and WSJF, we (numerically) solve Equation 3.8 and Equation 3.9 to determine these quantities.

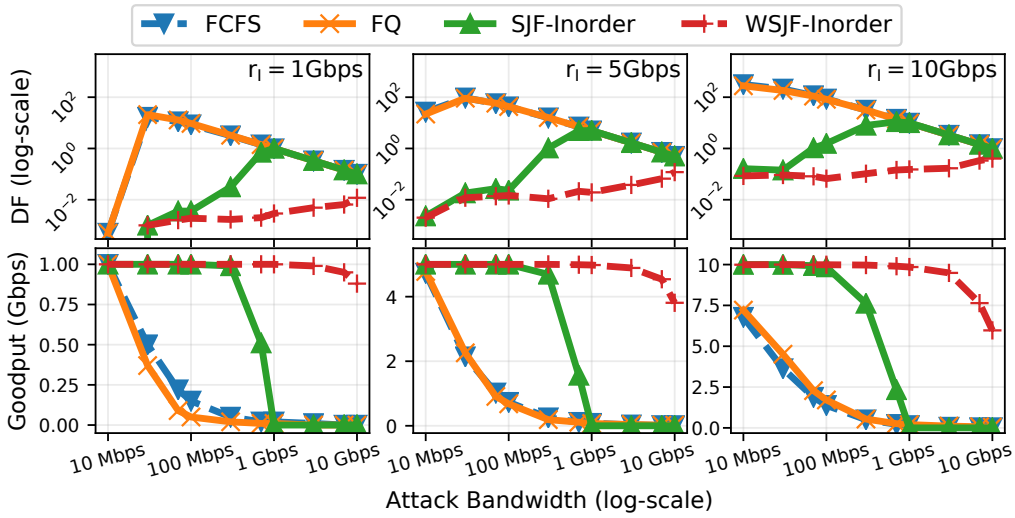


Figure 3.17: Goodput and Displacement Factor (DF) for TCP Reassembly. Left to right: Increasing innocent input rate, r_I, from 1 Gbps to 10 Gbps.

Looking at the bottom row, we observe a sharp drop-off in goodput for both FCFS and FQ even with a small attack bandwidth. For instance, with just 30 Mbps of attack traffic, an adversary is able to displace roughly half the system goodput, regardless of the innocent input rate. Correspondingly, we see a maximum displacement factor of 313 and 278 for FCFS and FQ, respectively. SJF is initially competitive, but we observe a performance cliff when the attack rate is sufficiently large; with 0.7 Gbps of attack traffic, an adversary is able to consistently displace over 50% of the goodput, corresponding to a maximum displacement factor of 11 (recall that the theoretical bound is $\alpha_{SJF} \leq \frac{\mathbf{E}[P]}{P_{\min}} \cdot \rho \approx 16$). Finally, we see that WSJF consistently outperforms the other policies, yielding a low degradation in goodput even with a high fraction of attack traffic. We observe a worst-case displacement factor of 0.4 for this application, implying that the *adversary must use over 2.5 bps of their own bandwidth in order to displace 1 bps of innocent traffic*, a considerable improvement over FCFS and FQ.

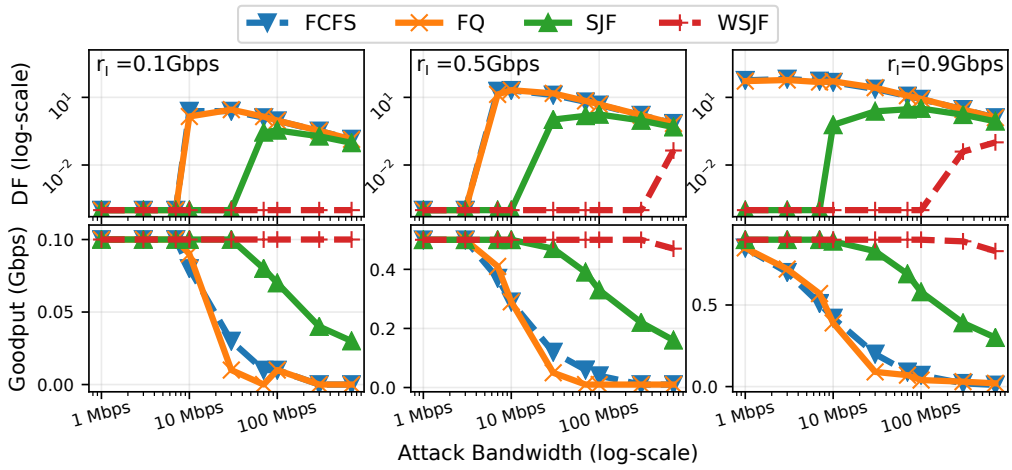


Figure 3.18: Goodput and Displacement Factor (DF) for Pigasus Full Matching. Left to right: Increasing innocent input rate (r_I) from 0.1 Gbps to 0.9 Gbps.

Similarly, Figure 3.18 depicts the goodput and DF achieved by the different scheduling policies in the context of Pigasus' Full Matching stage. The format of the figure is identical to that of Figure 3.17. Looking at the bottom row, we observe a gradual decrease in goodput for FCFS and FQ as the input attack rate increases from 1 Mbps to 1 Gbps. Overall, we observe a maximum displacement factor of 82 and 75 for FCFS and FQ, respectively. While we don't observe a goodput 'cliff' that we saw for SJF earlier, the adversary is consistently able to displace roughly 50% of the system goodput using an attack bandwidth of 100 Mbps, with a maximum observed displacement factor of 3. Finally, WSJF consistently outperforms the other policies, achieving a maximum DF of 0.1.

How does using a heuristic affect the DF achieved by WSJF? In the above simulated experiments, we assumed *a priori* knowledge of a packet's true job size at the time of scheduling. However, given that this information is rarely (if ever) available ahead of time in real systems, we would like to know the impact of using a *heuristic* on the achievable DF. While deriving an analytical answer to this question is beyond the scope of this work, we address it empirically here. For both Pigasus components (TCP Reassembly and Full Matching), we evaluate the difference in DFs achieved under WSJF *with* and *without* their respective heuristics. We assume that the adversary has knowledge of both the actual and estimated job size distributions, and uses job sizes which displace the maximum innocent traffic under the heuristic.[7] While we note that an attacker with such detailed knowledge of the system state likely does not exist, we find that our heuristics perform well even in the face of such an overpowered attacker.

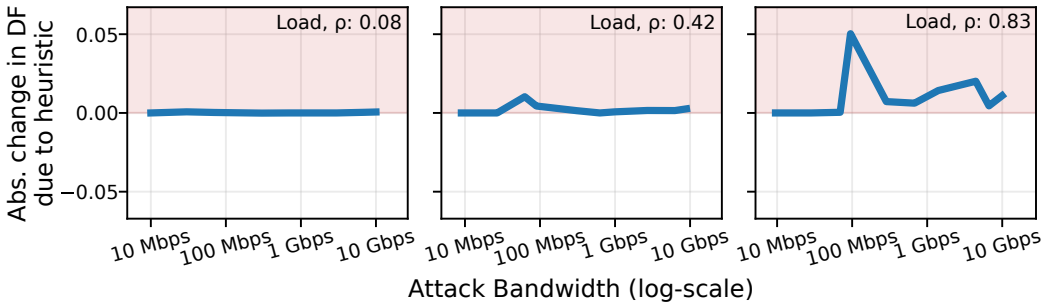


Figure 3.19: Absolute change in DF achieved by WSJF-Inorder due to the heuristic. Portions highlighted in red indicate regions where the heuristic does worse than the baseline.

In the context of TCP Reassembly, Figure 3.19 depicts the effect of using the heuristic (described in §3.5.2) on the achieved DF under WSJF-Inorder. On the x-axis, we sweep the adversary's attack bandwidth (r_A), and on the y-axis we plot the change in DF when using the heuristic (compared to using the true job size, computed offline). We see that using the heuristic increases the DF by at most 0.05 compared to the ideal case. Empirically, we find that this simple heuristic is both an excellent estimator of job size for innocent traffic and largely robust to any subversion attempts by the adversary. We observe similar results (<5% change) for the Full Matching stage.

[7] In practice, this involves a brute-force search over the joint distribution of estimated and actual job sizes for innocent traffic.

3.6.3 SurgeProtector Scheduler

A key component of the SurgeProtector scheduler is the Hierarchical FFS Queue (§3.5.4) used to implement WSJF. As described earlier, a poorly-designed heap may itself expose a novel attack surface for an adversary to exploit. In this section, we evaluate SurgeProtector against attacks targeting the software heap implementation.

There are two attack vectors we must consider. First, the adversary may flood the fixed-size queue with large attack jobs, causing innocent jobs arriving later to be dropped. Second, the adversary may attempt to inflate their packet arrival rate (using minimally-sized attack packets) beyond what the queue can sustain. Combining these ideas, the adversarial strategy is clear: use minimum-sized packets encoding large jobs.

As basis for this discussion, we consider three WSJF queue designs: a standard, bounded Fibonacci heap that supports Extract-Min operations (but no Extract-Max); a double-ended priority queue [125] (DEPQ, implemented using a pair of Fibonacci heaps) that supports both Extract-Min and Extract-Max operations in worst-case logarithmic time; and finally, the Hierarchical FFS Queue. For the purpose of evaluation, the packet size and job size for innocent traffic are sampled *i.i.d.* from Gaussian distributions (with an average packet size, $\mathbb{E}[P]$, of 1250 bytes, and an average job size, $\mathbb{E}[J]$, of 1μs). We set the maximum job size, J_{max}, to 10μs.

Finally, the experiment setup is as follows. For each of the three heap designs, we pin a process running a software implementation of the heap to a single core on an Intel Xeon E5-2620 CPU operating at 2.1 GHz, where it consumes packets from a 100G Ethernet link via DPDK. The packets (encoding the job size in μs) are dispatched to a different core, which emulates "running" the job by sleeping for a period corresponding to the job size. A third core is responsible for profiling the application goodput. Figure 3.20 depicts how the goodput varies with the input attack rate for the three heap implementations.

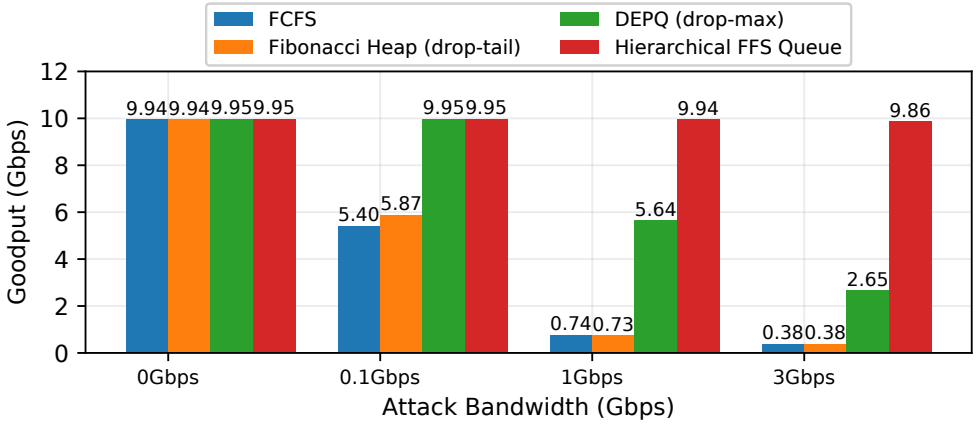


Figure 3.20: Goodput for different heap implementations.

First, in the case of the standard Fibonacci heap, we observe a large performance cliff when the attack rate reaches a certain threshold. The reason is that, once the queue becomes full, dropping at the tail causes a significant fraction of subsequent innocent arrivals to be dropped. Conversely, while the DEPQ is capable of selectively dropping large jobs, the worst-case log-

arithmic cost of EXTRACT-* operations imposes a significant performance penalty, resulting in a gradual degradation in goodput. Finally, we observe that the Hierarchical FFS Queue's goodput remains largely unchanged regardless of the input attack rate. Overall, we find that the HFFQ's EXTRACT-MAX functionality, in conjunction with the worst-case constant complexity of all operations, makes the Hierarchical FFS Queue robust to these kinds of attacks.

3.7 Limitations and Open Questions

This work opens up a broad range of theoretical and practical questions, and we are only able to answer some of them.

Optimality and Multi-Server Settings: An important theoretical question relates to the existence of an *optimal* adversarial scheduling policy. In this work, we have shown that WSJF, the policy underpinning SURGEPROTECTOR, achieves a DF that is always upper-bounded by 1. However, devising a policy that is always optimal (i.e., one which minimizes the DF for any load and choice of traffic parameters) — or proving its existence thereof — remains an open problem. Additionally, we have only considered a queueing system with *one* server; we do not currently know how the ACA mitigation problem scales with more than one server.

Heuristics: As described in §3.5.2, any practical implementation of SURGEPROTECTOR must rely on application-specific heuristic functions for estimating job sizes. Our experience implementing SURGEPROTECTOR in the context of TCP reassembly and IDS/IPS Full Matching suggests that even simple, easy-to-compute heuristics can be powerful job sizes estimators. However, the design of heuristics for a broader range of packet processing engines remains an open problem. In particular, there are two questions of interest. First, is there some fundamental property of network subsystems that makes job size estimation feasible? Second, for subsystems in which job size estimation *is* feasible, how do we reason about the efficacy of different heuristic functions? Parallel work in our group [40] has formalized sufficient criteria for an 'ideal' heuristic, and has shown that non-ideal heuristics can still provide an upper-bound on the DF achievable under WSJF.

Preemption: In this work, we have only explored the space of *non-preemptive* scheduling (i.e., a job, once started, must run to completion). However, given recent advances in the design and implementation of lightweight preemption handlers [34], it is reasonable to ask: can we do even better with preemptive scheduling policies? This is particularly relevant for subsystems where developing accurate heuristics is challenging. In this case, preemption may help tolerate some error in job size estimates by allowing the scheduler an additional degree of freedom (*e.g.*, by preempting jobs that far exceed their job size estimates).

Fairness: As we have seen in §3.4.2, fair queueing is fundamentally vulnerable to ACAs because of the adversary's ability to spawn many flows. However, fairness is an important consideration for many systems. While WSJF alone does not provide any fairness guarantees, we conjecture that an augmentation of this policy (*e.g.*, using FQ as a second-stage queueing discipline, or switching between the two based on some goodput watermark) may be able to provide both ACA resilience and flow-level fairness.

Memory Complexity Attacks: Finally, we have not considered the impact of ACAs on *memory*. In many systems, memory is just as precious (and exhaustible) a resource as processing cycles, and may be an important consideration in the design and analysis of adversarial scheduling policies for packet processors.

3.8 Related Work

ACAs and mitigation: Crosby et al. were the first to characterize ACAs as DoS vectors in [45], and empirically evaluated their impact in the context of an IDS. Others have since explored ACAs on several applications, including hash tables [16, 22], automata-based string pattern-matching [130], regular expression matching [49, 129, 156], PDF decompression [70], and TCP reassembly [52, 143]. [112] provides both an excellent survey of prior work and a novel approach for automatically crafting ACAs in a domain-independent manner (using fuzzing).

Many works have proposed *application-specific* mitigation strategies. For example, [133] implements TCP reassembly by maintaining statically-allocated, fixed-sized buffers for each flow; this renders the design impervious to ACAs at the cost of significantly higher memory overhead (every flow is allocated 64KB of memory regardless of its peak usage). Similarly, many regular expression engines restrict the number of states a single packet may invoke to avoid ReDoS attacks [137] (limiting the length of regular expressions in the common case). Other systems place a cap on the number of cycles spent decompressing a file or webpage for deep packet inspection [67] (limiting the size of files or web pages that can be served). Still other systems rely on universal hashing to prevent attacks on hash tables [45] (imposing computational and memory overheads). In a slightly different direction, [2] leverages a multi-core architecture to mitigate ACAs on DPI engines.

Scheduling: Scheduling and queueing theory has garnered significant research attention in recent years. While the vast majority of queueing literature focuses on optimizing various response time metrics in stochastic settings, some recent works in OS and packet scheduling are notable due to the focus on fairness and performance isolation. In particular, Fair Queueing (FQ) [51] aims to equitably partition the available link bandwidth between multiple contending flows. Dominant Resource Fair Queueing (DRFQ) [65] generalizes this idea to multiple resources [64], and [151] provides a low-overhead approximation to DRFQ. However, as described in §3.4.2, FQ and its variants are ineffective in the adversarial setting [157].

Recent works have also explored the use of queueing theory to analyze *volumetric* DoS attacks (*e.g.*, SYN-floods). [152] proposes a two-dimensional embedded Markov chain to model DoS attacks, and derives various performance metrics (*e.g.*, connection loss probability) by analyzing its stationary distribution. Along these lines, [33] evaluates how dynamic TCP timeouts can be used as a mitigation strategy against SYN-floods. [118] proposes a composite model to jointly analyze memory and bandwidth resource exhaustion during an attack. More recently, [61] derived the feasibility criteria for a successful volume-based DDoS attack on a multi-hop network following the Join-the-Shortest-Queue (JSQ) policy. We reiterate that the distinguishing factor here is the *type* of DoS attack considered in this work: complexity-based instead of volumetric.

Finally, we are aware of two works that consider the ACA mitigation problem from a queueing theoretic perspective, and are therefore most closely related to this work. First, [84] models DoS attacks using an M/M/1/k queueing model with the goal of detecting both flood- and complexity-based attacks. However, they only perform analysis for FCFS, and they only consider exponentially-distributed service times (which may not be an accurate assumption in the adversarial setting). Second, [22] analyzes the impact of using two different hashing schemes on the efficacy of ACAs on hash tables. They also develop a metric called the 'Vulnerability Factor' to quantify the impact of ACAs. However, they limit their analysis to FCFS. Moreover, since their analysis is based on a job's *average waiting time*, they are fundamentally constrained to scenarios where the system is *not* overloaded.

To the best of our knowledge, this is the first work to analyze scheduling policies beyond the simple FCFS and to propose a policy-based mitigation strategy for ACAs.

3.9 Conclusion

Network services on the Internet are prone to algorithmic complexity attacks (ACAs), a potent class of Denial-of-Service (DoS) attacks. In this chapter, we described how these attacks are, in fact, an artifact of *workload incongruity* in our classical model of packet processing. Refining this model (treating the input as a superposition of stochastic innocent traffic and rate-limited adversarial traffic) allows us to systematically reason about the attacker's actions and their collateral on innocent users. Using this abstraction, we develop SurgeProtector, a framework to mitigate temporal ACAs on network subsystems using novel insights from adversarial scheduling theory. WSJF, the scheduling algorithm underpinning SurgeProtector, provides *provable* upper bounds on the maximum "harm per unit effort" an adversary can induce, regardless of the underlying algorithm, the load on the system, and the parameters of the innocent traffic distribution. Our proofs and evaluation show that SurgeProtector, provides resilience to ACAs without limiting the system in ways that existing *ad hoc* mitigation strategies do.

"People don't want to buy a quarter-inch drill. They want a quarter-inch hole!"

THEODORE LEVITT

Chapter 4

Deployable Hardware Packet Scheduling

Packet scheduling, the problem of deciding what *order* and/or *time* network packets ought to be served or transmitted, has long occupied a prominent position in networking literature. Prior work in this space has resulted in a rich repertoire of packet scheduling algorithms with a variety of different properties: fairness and starvation avoidance [24], attack resilience [13], burst reduction [116], optimal flow completion time [8], *etc.* At the heart of these algorithms is a **priority queue** data-structure, which allows the scheduler to map packets' relative order (or scheduling time) to unique priorities, sort them, and subsequently extract the highest-priority item (i.e., the next packet to schedule) from the queue.

Despite these strong theoretical foundations, network switches and NICs have historically failed to provide anything beyond a small suite of simple scheduling algorithms. The key problem is the *complexity associated with implementing a fast, scalable, and generic priority queue in hardware*. PIFO [135], published in 2016, was a foundational paper in this regard: it showed that packet priorities can almost always be determined at enqueue time (reducing functional complexity), and that queue operations can be parallelized by leveraging the spatial nature of switch or NIC hardware (i.e., by unrolling the operations in *space* instead of *time*). Since then, several prior works have improved upon PIFO's design using more sophisticated parallelization techniques [23, 131, 158]. Nonetheless, despite having been a "solved" problem for almost a decade, none of these designs were ever *deployed* in practice. This made us wonder: *why?*

In this chapter, we argue that the reason is **design incongruity**: a *fundamental* mismatch between the operational constraints imposed on the scheduler by modern switches and SmartNICs, and the objectives that hardware designers today are (implicitly) optimizing for. We present a quantitative analysis of the *minimum* performance, scalability, and functionality requirements for deployment today (§4.1), and show that there is a *significant* gap between what existing designs *provide* and what the surrounding dataplane *expects*. Based on our analysis, we find that the shortest path to hardware deployment is an unconventional priority queue design (previously used in software) that altogether eschews comparison-based sorting.

The remainder of this chapter is organized as follows. In §4.2, we present an overview of the Bitmapped Bucket Queue (BBQ), a hardware priority queue design that leverages *integer priority queueing* to circumvent the performance-scalability barrier imposed by comparison-based sorting. We then provide a more detailed description of the architecture in §4.3, focusing on the challenges due to parallelism and hazards. In §4.5, we describe extensions to the BBQ design that (a) counteract the latency artifacts introduced by pipelining, and (b) allow it to operate over dynamic priority ranges. In §4.6 we evaluate BBQ, showing that it enables, for the first time, line-rate packet scheduling for 100K+ flows on a commodity FPGA-based SmartNIC, and a state-of-the-art 32×400 Gbps switch [108]. In §4.7, we discuss how the design can be incorporated into modern networks. We then discuss limitations and future work in §4.8, describe related work in §4.9, and conclude in §4.10.

4.1 Background and Motivation

More than ever, there is a need for programmable packet scheduling in hardware. Switches have long offered a limited set of packet scheduling algorithms and NICs are increasingly taking over dataplane tasks traditionally performed in software [121], including end-host packet scheduling [95, 116, 140]. In the case of switches, having a programmable packet scheduler could not only vastly expand the catalog of scheduling algorithms available to network operators, but also pave the way for faster innovation and customization [32, 135]. With NICs, system administrators already *expect* to be able to customize the packet scheduler that runs on the end host [121].

PIFO [135] is a seminal work in this regard. It makes the observation that, when considering a single node [101], all scheduling algorithms can be expressed in how they make two decisions: *which* packet to schedule next and *when* to schedule it. The authors observe that for many scheduling algorithms this behavior can be captured simply with a hardware priority queue. This priority queue can be used in one of two ways: it can implement work conserving algorithms by sorting flows by rank, or it can implement non-working conserving algorithms by sorting flows by scheduling time. While PIFO was initially conceived to run on switches, it has also been shown to be a useful primitive to schedule packets on NICs [95, 140].

4.1.1 Design Incongruity

Unfortunately, existing solutions for programmable packet scheduling in hardware fail to meet the requirements for both state-of-the-art NICs and switches. In what follows, we elaborate on these two use cases, highlighting how their stringent performance requirements are at odds with existing proposals for programmable hardware packet schedulers.

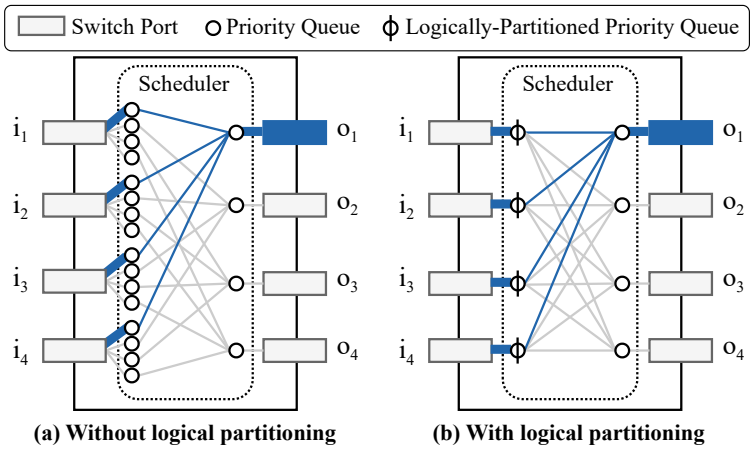
Line-Rate Switches

In order to support packet scheduling without impeding the rest of the switch's dataplane functionality, any realistic proposal for a scheduler ought to satisfy two key requirements:

(A1) Match the switch's aggregated packet rate: For scheduling in output-queued switches, the worst-case throughput demand corresponds to the scenario where ingress traffic from all

ports is incident on a single egress port (i.e., incast behavior). In order to avoid backpressuring the switch fabric, the packet scheduler must be able to process packets at the same rate as the switch backplane (i.e., its aggregated packet rate) at *any* given port. For instance, in NVIDIA's Spectrum SN4700 (a state-of-the-art 400 GbE switch with 32 ports), the packet scheduler for a single port must be able to handle an aggregated packet rate of 8.4 *billion* pps [108].

(A2) Allow every port to address all buffered packets: In order to efficiently utilize on-chip memory, switches use a shared packet buffer that is dynamically partitioned between its output ports [135]. In the worst case, every packet in the shared buffer might be destined for the same output port, and each packet might map to a different flow to be scheduled. If the packet scheduler at a given port could only address a subset of buffered elements, it would impose additional constraints on the switch's ability to handle such bursts. Thus, a second key requirement for schedulers is the ability to allow any output port to address *every* buffered packet. In modern switches, shared packet buffers are provisioned for *hundreds of thousands* of packets [108].



(a) Without logical partitioning (b) With logical partitioning

Figure 4.1: Scheduler architecture using priority queues without (left), and with (right) support for logical partitioning. In a k-port switch using priority queues without logical partitioning, we need k^2+k priority queues in order to both implement output queueing *and* absorb incast traffic from all the ports. Support for logical partitioning reduces the number to $2k$.

In 2016, a single physical PIFO instance could handle the full aggregated packet rate for a 64-port 10 GbE switch (1 Bpps) [135]. Today, state-of-the-art switches, *e.g.*, the SN4700, offer 400 GbE line rates with 32 ports (20× higher aggregated throughput). Given the large (and ever-widening) gap between network and processing speeds, a single priority queue instance can no longer sustain aggregated packet rates. Consequently, to satisfy (A1), packet schedulers for modern switches must "scale out" using a mesh of priority queues.

Unfortunately, *priority queue designs that do not support logical partitioning, i.e., the ability to multiplex several independent queues atop a single physical queue, require prohibitively large meshes.* To the best of our knowledge, PIFO and PIEO are the only existing design that support logical partitioning, while more recent proposals (*e.g.*, BMW-Tree [158]) do not. Generously assuming that a single priority queue instance could handle 400 Gbps line-rate input in a k-port switch, these designs would require, at minimum, a $(k^2 + k)$ mesh of instances to realize per-output-port scheduling while supporting the full aggregated throughput due to incasts; an example mesh for $k = 4$ is depicted in Figure 4.1 (a). Further, *every instance would have to be provisioned with hundreds of thousands of queue elements* to satisfy (A2). Overall, a 32-port switch operating at 400 Gbps line-rate would require at least $32^2 + 32$ BMW-RPU instances,

each provisioned with 100K+ entries, to implement priority queueing alone. Based on synthesis numbers reported by the authors and considering that a switch chip area ranges from 200 mm^2 to 800 mm^2 [135] this corresponds to $1.5 - 6\times$ the total area.

The ability to logically partition a physical priority queue enables a significantly simpler mesh architecture. Assuming, again, that each priority queue instance can sustain 400 Gbps input, we would only need $2k$ instances, as depicted in Figure 4.1 (b). Each physical instance in the first layer ingests packets from a single ingress port, but enqueues into one of k *logically independent queues* corresponding to the k possible destination ports. The second layer then periodically schedules packets among their k inputs, feeding traffic to their respective egress ports. *Despite the fact that PIFO benefits from this architecture, its inability to scale beyond 2,048 elements means that it does not satisfy (A2). Conversely, while PIEO scales marginally better, it does not provide the requisite performance, violating (A1).*

SmartNICs in the Public Cloud

Another key driver for programmable packet schedulers in hardware are SmartNICs in the public cloud, either ASIC [95, 116, 121, 140] or FPGA-based [11, 58, 60, 73, 99, 121, 122]. Packet schedulers for such NICs ought to satisfy three requirements:

(B1) Scale to tens of thousands of flows: The packet scheduler on the SmartNIC may need to implement scheduling policies for a large number of active (concurrent) flows. This might seem surprising because NIC packet buffers are typically orders of magnitude smaller than those used in switches; however, unlike switches, modern SmartNICs may be required to make scheduling decisions for flows not just in their local TX or RX queues, but also those residing in *host memory*. This is a popular theme in the cloud setting where, in order to save valuable CPU cycles, the hypervisor dataplane (including scheduling functionality) is offloaded to the NIC [58]. The packet scheduler aboard the NIC is then responsible for deciding which backlogged flow queues in host memory to serve at any point, with the number of scheduling candidates scaling as (tenants $\times$ flows per tenant). For instance, across 1M+ VMs in Azure, VFP [57] reports 4.8K active connections *per VM* at the 99th percentile, and as high as 12K at P99.9. We expect the scalability problem to become all the more apparent given trends of increasing core counts [9] (and therefore potential for multi-tenancy), and as more services that traditionally used multiple physical NIC queues (*e.g.*, RDMA) become amenable to virtualization [71].

(B2) Sustain 100GbE+ line-rates: While state-of-the-art NICs have lower throughput requirements compared to switches, they still need to support line rates of 100 Gbps and beyond [107]. This is particularly relevant in the context of public clouds because network bandwidth is a commoditized resource and an important driver for many high-performance cloud-based applications [122].

(B3) Implement scheduling both across and within tenants: Finally, in the context of multi-tenant clouds, the NIC scheduler should be able to provide, at minimum, the ability to schedule traffic *across* tenants (to enforce cloud providers' policy requirements, *e.g.*, fairness or bandwidth quotas) and *within* tenants (to provide application-level priority queueing), without imposing significant resource overhead.

Recent priority queue designs (*e.g.*, PIEO [131], BMW-Tree [158]) symbolize a concerted effort towards addressing the *scalability* requirement outlined in (B1). For instance, we can synthesize a PIEO instance with up to 64K entries, and a BMW-RPU instance with 350K entries on a state-of-the-art FPGA (§4.6.2). Unfortunately, *these designs do not meet both the performance (B2) and provider policy (B3) requirements.*

In the context of (B2), the problem with existing designs is that performance degrades rapidly as the number of elements increases, a fundamental tradeoff associated with comparison based sorting. Moreover, scaling out using a mesh is not feasible because NICs are considerably smaller and more resource-poor than switches; as such, single-instance performance is the key factor in determining feasibility. For example, [158] reports that a single BMW-RPU instance can sustain 200 Mpps with 85K elements using a 28nm ASIC process; this is sufficient to drive line rate at 100 GbE (148.8 Mpps), but not at 200 GbE. The picture is even more dire for packet scheduling on FPGA-based SmartNICs[58, 73]. For instance, as we will show in §4.6.2, BMW-Tree achieves a packet rate of 55 Mpps for 85K elements on a state-of-the-art, Intel Stratix 10 MX FPGA, 37% of line rate even at 100 Gbps.

A second, more fundamental problem with these designs is that it is impossible to disentangle their *function* (implementing priority queueing) from their *form* (a fixed tree [29, 78, 158] of queue elements). As a result, implementing n distinct priority queues (*e.g.*, for n tenants) requires duplicating the underlying data structure, imposing significant resource overhead, fragmentation of queue memory, or both. As before, the key enabler for (B3) is *logical partitioning*.

Incongruity: In the context of modern deployments (both switches and SmartNICs), existing hardware priority queue designs are not viable alternatives for packet scheduling because they treat one or more operational requirements as *objectives that can be traded-off* rather than *constraints that must be met* — whether in terms of scaling (*e.g.*, PIFO), performance (*e.g.*, PIEO), or their ability to provide logical partitioning (*e.g.*, BMW-Tree).

4.1.2 Exploring a Different Tradeoff

In order to circumvent the performance-scalability barrier, in this work we explore a different tradeoff: sacrificing a small amount of *precision* to achieve the best of both worlds. In this regard, we are motivated by prior work's observation that a large fraction of networked applications *do not* require particularly high precision [6, 128]. For instance, both VLAN and DSCP support up to 8 traffic classes (3-bit priority tags), priority-qdisc (*tc-prio*) in the Linux kernel provides at most 16 priority bands, and state-of-the-art commercial switches support up to 32 strict-priority queues [6]. These, in turn, provide sufficiently fine-grained priority queueing to support a variety of higher-level abstractions: transport protocols and frameworks (*e.g.*, Homa [102], PASE [104], PIAS [15]), congestion and interference controllers (RC3 [100], QJump [69]), and high-performance overlay networks (*e.g.*, GRIN [3], SLIM [163]). In what follows, we describe the design of a highly scalable and performant priority queue exploiting this observation.

4.2 BBQ Overview

BBQ is a new hardware-based priority queue architecture for packet scheduling that is designed with three goals in mind: (1) *scalability*, the maximum number of concurrent flows that the queue can support, (2) *performance*, the maximum steady-state packet rate that the queue can sustain, and (3) *logical partitioning*, the ability to multiplex several logical queues atop a single physical queue. In this section, we give an overview of BBQ's design before diving into the architectural details in §4.3. We start with a brief introduction to the data structure underlying BBQ in §4.2.1, followed by a high-level description of the BBQ primitive in §4.2.2. Finally, in §4.2.3, we describe the challenges the hardware architecture must address in order to meet our system goals.

4.2.1 Data Structure

Integer Priority Queueing: BBQ leverages an *Integer Priority Queue* (IPQ) scheme to alleviate the tension between scalability and performance (§4.1). Unlike traditional priority queues where elements can have arbitrary priorities, an IPQ requires elements to map to a *finite set of integer priorities*, called its *priority span*; for an IPQ that supports P integer priorities, the span is represented by the set $\{0, \ldots, P-1\}$. Quantizing the priority range allows IPQs to implement a simple counting sort-like algorithm: the IPQ maintains an array of P *priority buckets*, each representing a unique priority in its span; when a new element is enqueued, it is inserted into the bucket indexed by the element's priority.

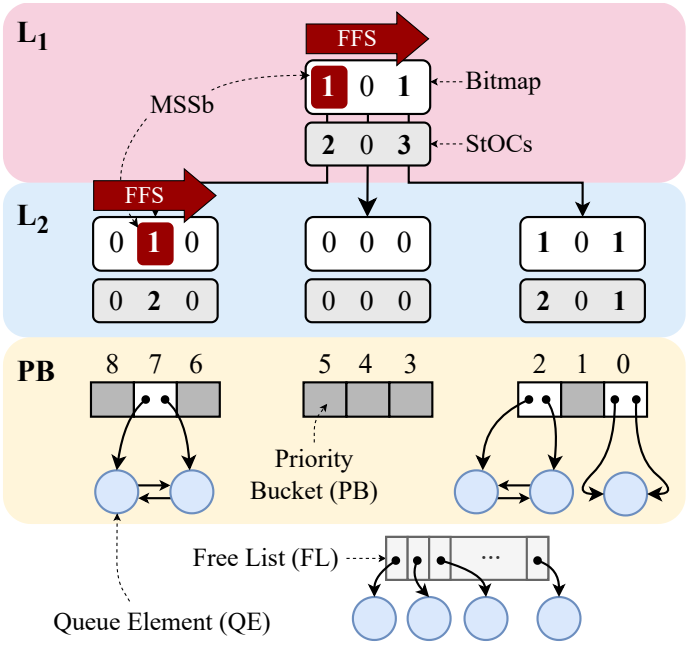
The only remaining challenge is to find the right priority bucket to dequeue from. Since only a subset of buckets may be occupied at any time, dequeueing entails finding the *highest-priority bucket containing at least one element*. A naive approach is to sequentially check buckets in decreasing order of priority, stopping at the first non-empty bucket; however, this may turn out to be expensive, necessitating $O(P)$ sub-operations in the worst case.

Building efficient IPQs using Find-First Set: One approach to improve the run-time efficiency of dequeue operations is to encode the occupancy of the IPQ's priority buckets as a P-bit wide bitmap, with '0's representing empty buckets, and '1's representing buckets containing at least one element. Then, the *most-significant set bit* (MSSb) in the bitmap yields the required bucket to dequeue from. Finding the MSSb, an operation known as *Find First Set* (FFS), can be performed with $\Theta(\log P)$ simple bit operations (bit-shifts and additions) using a binary search algorithm. Unfortunately, scaling to large values of P (*e.g.*, 32K) using FFS Queues quickly becomes impractical due to constraints on the maximum word size that the hardware can efficiently operate upon. For example, general-purpose processors provide FFS intrinsics for at most 64-bit words; similarly, implementing FFS on bitmaps larger than 64 bits would result in low operating frequency even in more specialized circuits (*e.g.*, ASICs or FPGAs).

Scaling to larger priority spans using Hierarchical FFS: [148] novelly observed a recursive structure to the problem: given an array of *bitmaps* ordered by priority, the *highest-priority non-zero bitmap* can also be identified via a single FFS operation using a schema similar to the one described above. This observation naturally leads to a *Hierarchical* FFS (HFFS) Queue.

The idea is to construct a tree of bitmaps, with leaf nodes representing regular FFS Queues. The bitmaps are encoded such that, at any level in the tree, *a '1' in any bit position indicates a non-empty priority bucket in the subtree rooted at that node.* Now, to dequeue the highest-priority element, we recursively perform FFS at each level of the tree starting with the root, following the MSSb until we arrive at the required priority bucket. BBQ uses a variant of the HFFS Queue as its underlying data-structure.

4.2.2 BBQ Primitive



At the heart of BBQ is a priority index structure inspired by HFFS Queues: a perfect w-ary tree of w-bit bitmaps representing the queue occupancy. The bitmap tree in a BBQ can be composed of an arbitrary number of levels, which in turn dictates the queue's priority span. In general, for a BBQ with w-bit bitmaps and $D \geq 1$ levels, the number of supported priorities is $P = w^D$. Figure 4.2 depicts a BBQ with $w = 3$ and $D = 2$, representing $P = 3^2 = 9$ unique priorities.

Figure 4.2: 2-level BBQ with $w = 3$ bit bitmaps. To dequeue the highest-priority element, we recursively perform FFS at each level starting with the root, following the most-significant set bit (MSSb) until we arrive at the required priority bucket.

The bitmap tree has a recursive structure: at the leaf level of the tree (*e.g.*, the L_2 bitmaps in Figure 4.2), each bit maps to a unique *priority bucket*; at non-leaf levels (*e.g.*, L_1), each bit maps to a unique subtree of bitmaps. In either case, we maintain the invariant that a bit in any bitmap is 1 *if and only if* there is a priority bucket containing at least one element in the corresponding subtree. In BBQ, we additionally associate with every bit a *subtree occupancy counter* (StOC) that indicates the total number of elements in that subtree; a StOC is non-zero if the corresponding bit is 1, and vice versa. As we will see in §4.3, the design choice of storing additional counters enables us to achieve stall-free execution of the BBQ pipeline, yielding high performance (i.e., full pipelining) with a relatively small memory overhead.

IPQs group queue elements (QEs) with identical priority in the same priority bucket (PB). In BBQ, PBs are implemented as *doubly-linked lists* of QEs. In particular, each PB stores a pair of pointers: HEAD and TAIL, pointing to the first and last QE in the linked list, respectively. QEs themselves are composed of two attributes: (1) a pair of pointers (PREV and NEXT) to other QEs, allowing them to interface with the PBs' doubly-linked lists, and (2) a DATA field to store

arbitrary, user-supplied identifiers.[1]

The final component of the BBQ is the *Free List* (FL): a FIFO queue containing pointers to QEs that are currently unallocated (i.e., *not* enqueued in the BBQ). Table 4.1 depicts the operations supported by BBQ.

Operation	Description
ENQUEUE(X, p)	Inserts the given data, X, into the BBQ with priority *p*.
DEQUEUE(t) → (Y, p) or ∅ t ∈ {MAX, MIN}	Extracts the data, Y, corresponding to either the *highest-priority* QE (when t is MAX) or the *lowest-priority* QE (when t is MIN) currently enqueued in the BBQ. Returns ∅ if empty.

Table 4.1: Priority Queue operations supported by BBQ.

4.2.3 Goals and Challenges

Recall that we sought out to build BBQ with three key goals in mind: *scalability*, *performance*, and *logical partitioning*. In this section, we describe how BBQ meets these goals, and the challenges the underlying hardware architecture must address to achieve them.

Scalability. Using an IPQ-based design breaks the dependence between run-time complexity of operations and queue size, allowing BBQ to support a large number of QEs without necessitating a fundamental performance trade-off. In many ways, scalability "falls out" of this high-level design choice, allowing us to explicitly optimize for the other goals.

Performance. While the data structure underlying BBQ is conceptually simple, realizing it in hardware in a manner that achieves high performance remains a challenging proposition.

The overall performance of the hardware queue (measured in packets per second) is the product of two independent metrics: (C_1) $f_{\max}$ or the maximum frequency that the queue operates at, which is dictated by its *critical path* (i.e., the worst-case combinational delay in the hardware circuit), and (C_2) the number of operations that can be issued every cycle.[2] Given the logical complexity of the queue operations (i.e., ENQUEUE or DEQUEUE), it is impractical to perform them in a single hardware clock cycle because it would significantly throttle $f_{\max}$, hurting (C_1). Instead, operations must be divided into a sequence of n *stages*, where each stage involves a smaller quantum of work. While this improves $f_{\max}$ of the resulting circuit by reducing combinational delay, doing so naively (*e.g.*, trying to avoid concurrency problems by issuing one operation every n cycles) would slash (C_2) by a factor of n, once again degrading performance.

[1] BBQ, like most priority queues, is agnostic of the data contained in the QEs. The DATA attribute has a configurable bit-width and could be used to store a pointer to a packet, flow ID, or even a reference to another BBQ. Unless specified otherwise, we will assume that QEs represent *flows* [135].

[2] In architectural terms, this is the queue's IPC (instructions-per-cycle).

The key to high performance — and simultaneously the most challenging aspect of BBQ's architectural design — is *pipelined parallelism*. In this context, pipelining refers to designing the hardware architecture such that *multiple stages* may simultaneously be active at the same time, thereby allowing operations to be issued fewer than n cycles apart. Unfortunately, there are several sources of complexity that make it non-trivial to achieve a high degree of pipelined parallelism: practical limitations on the number of R/W ports on physical memory blocks, data hazards (i.e., ephemeral memory dependencies between active pipeline stages), and control hazards (i.e., logical and algorithmic dependencies between stages).

Our key finding in this context is that by carefully architecting BBQ's hardware pipeline, we can, in fact, achieve fully-pipelined execution (i.e., guaranteed throughput of 1 operation per cycle) *without* compromising on the maximum clock frequency. This is realized by: (a) employing deep pipelining to preserve $f_{\max}$, and (b) using a variety of architectural techniques (speculation, write-forwarding, and instruction coloring) to handle pipeline hazards without stalling or discarding operations. We describe the BBQ pipeline in detail in §4.3.

Logical Partitioning. Full decoupling between BBQ's queue memory (i.e., its QEs) and its priority index structure (i.e., the bitmap tree) gives BBQ a unique opportunity to provide logical partitioning with no resource overhead. The idea is to *treat the bitmap tree as a collection of n disjoint subtrees, each of which maps to an independent BBQ*. This effectively partitions the original BBQ's priority span into n disjoint regions; then, in order to DEQUEUE an element from a *logical* BBQ, we simply "mask" the appropriate bits in the bitmap tree while performing FFS such that we only traverse down the corresponding subtree.

This technique allows us to fully reuse *all* of the physical BBQ's resources without any performance degradation or fragmentation of queue memory. There is, however, a cost in terms of precision, because each logical BBQ can only address $\frac{1}{n}$'th of the priority span of the underlying instance. For use-cases where the number of logical partitions is not too large (*e.g.*, 32–64 port switches [108], or cloud servers hosting 16–128 tenants), this is simply a matter of appropriately provisioning the priority span of the underlying BBQ.

Since logical partitioning is, (a) a key enabler for building efficient priority queue meshes for line-rate switches and realizing hierarchical scheduling in multi-tenant cloud NICs (§4.7), and (b) adds negligible overhead in the BBQ datapath, we natively support this feature in the BBQ primitive. We describe logical partitioning (both for prior work, as well as BBQ) in more detail in §4.4. Logical partitioning also enables us to extend the BBQ primitive to operate over dynamic priority ranges with zero overhead, an idea we describe in §4.5.2.

4.3 BBQ Architecture

In this section, we describe the architectural details that enable us to map BBQ's design to hardware in a manner that achieves our performance goals. We begin by describing the hardware pipeline (§4.3.1), followed by a description of the hurdles we encountered while trying to fully pipeline the design in (§4.3.2), and finally some key implementation details that reduce memory pressure and further improve $f_{\max}$ (§4.3.3).

Cycle	Description	PHR
0	Register inputs If ENQUEUE: $F \leftarrow$ FreeList.POP ( ) // Pop free list	
1	Compute L_1 bitmap index ↳ Read the corresponding L_1 StOC	L_1
2	Compute, Write: L_1 StOC $\rightsquigarrow$ L_1 bitmap Read L_2 bitmap	L_2
3	// Read delay for L_2 bitmap	
4	Compute L_2 bitmap index ↳ Read the corresponding L_2 StOC	
5	// Read delay for L_2 StOC	
6	Compute, Write: L_2 StOC $\rightsquigarrow$ L_2 bitmap Read the corresponding PB	PB
7	// Read delay for PB	
8	If DEQUEUE: (a) Read $X \leftarrow \text{QE}_{\text{DATA}}[\text{PB . TAIL}^{\text{new}}]$ (b) Read $Y \leftarrow \text{QE}_{\text{PREV}}[\text{PB . TAIL}^{\text{new}}]$	
9	// Read delay for QE_{DATA} and QE_{PREV}	
10	If ENQUEUE: // Enqueue at the HEAD (a) $\text{QE}_{\text{DATA}}[F] \leftarrow$ Data to enqueue (b) Write $\text{QE}_{\text{NEXT}}[F] \leftarrow$ PB . HEAD (c) Write $\text{QE}_{\text{PREV}}[\text{PB . HEAD}] \leftarrow F$ (d) Write PB . HEAD $^{\text{new}} \leftarrow F$ If DEQUEUE: // Dequeue from TAIL (a) FreeList.PUSH(PB . TAIL) (b) Write PB . $\text{TAIL}^{\text{new}} \leftarrow Y$ (c) Output X	

Table 4.2: 11-stage hardware pipeline for a 2-level BBQ (without operation coloring) highlighting independent pipeline hazard regions (PHRs). ↳ and $\rightsquigarrow$ indicate same-stage dependencies, which result in more complex combinational logic. In general, a BBQ with $D > 1$ levels entails a pipeline depth of $p = 7 + 4 \times (D - 1)$ stages.

4.3.1 Hardware Pipeline

In principle, enqueueing and dequeueing follow a similar blueprint, yielding an intuitive algorithm for mapping them to a unified datapath: for each level of the tree starting with the root (i.e., L_1), compute the bitmap index (for DEQUEUE(t), the index is computed by performing FFS on the bitmap, while for ENQUEUE(x, p), it is computed using simple bit manipulations on p), increment/decrement the corresponding StOC, then update the bitmap; finally, enqueue into or dequeue from the doubly-linked list corresponding to the target priority bucket. To maximize performance, we take a careful two-pronged approach.

(1) Maximizing $f_{\max}$: Our first objective is to maximize $f_{\max}$, or the maximum clock frequency at which the BBQ circuit can operate. To do this, we use a *deep pipeline* where individual stages are designed to do both *little* and *roughly equal* amounts of work. Table 4.2 depicts the events that occur at cycle-level in a 11-stage pipeline for a 2-level BBQ.[3] By load balancing expensive operations across stages, we minimize the number and severity of same-stage dependencies (depicted by $\hookrightarrow$ and $\rightsquigarrow$). For instance, chaining FFS and StOC updates (multi-bit addition or subtraction) would result in a large combinational delay, so we split this work across stages (*e.g.*, cycles 1 and 2).

(2) Maximizing operations per cycle: As described in §4.2.3, high $f_{\max}$ is only useful if we are not rate-limited by the pipeline latency or even a portion of it. Our second objective is to *fully pipeline* the BBQ design so it can concurrently process as many operations as there are pipeline stages, thereby achieving its maximum rate of 1 op/cycle. There are several challenges we encounter in this process, which we address in detail next.

4.3.2 A Fully-Pipelined Architecture

Pipeline Hazard Regions: The first key enabler for BBQ's stall-free architecture is the design choice of associating every bit in the bitmap tree with a *subtree occupancy counter*. Recall that for a given bit, the associated StOC indicates the *total number of elements contained in the corresponding subtree*.

To understand how this enables pipelining, consider a strawman design with n pipeline stages where we *only* store the bitmaps for each level of the HFFS tree, but not the associated StOCs. Here, the earliest time we know whether a DEQUEUE operation causes a priority bucket to become empty is when the operation is committed (i.e., the final pipeline stage). Now, consider what happens if this causes a bit in any bitmap along the path to that priority bucket to flip (i.e., become '0'). Any subsequent DEQUEUE operations in the pipeline may have been routed along the tree based on stale state, creating an incorrigible *control hazard*. As a result, we would have to either: (a) discard and re-issue the subsequent DEQUEUE operation(s), hurting worst-case performance, or (b) only issue DEQUEUEs every n cycles, defeating the purpose of pipelining altogether.

[3] Since the L_1 level has a small memory footprint, we choose to store the associated metadata (bitmaps and StOCs) in registers with single-cycle access latency. The larger arrays (*e.g.*, L_2 bitmaps and StOCs, priority buckets, and queue elements) use substantially more memory and involve more complex address decoding logic; as such, we store these in SRAM with a 2-cycle access latency in order to optimize $f_{\max}$.

Instead, StOCs allow us to divide the BBQ pipeline into independent *pipeline hazard regions* (PHRs) mapping to different levels of the bitmap tree, as shown in Table 4.2. When exiting a PHR, the outcome of every operation (either an ENQUEUE or a DEQUEUE) is committed to the StOC. This has two implications: (1) two operations can only be conflicted (i.e., have data or control dependencies between them) if they are in a PHR at the same time, and (2) conflicts are limited to intra-PHR state (*e.g.*, the bitmap or StOC data at that tree level). As a result, we only have to address intra-PHR hazards (i.e., dependencies between active operations in the same PHR), which are far more localized — and therefore more tractable — than hazards spanning the entire pipeline. We characterize our implementation of StOCs in detail in §4.3.3.

While StOCs *enable* us to achieve stall-free operation, they are not sufficient to guarantee a fully-pipelined design on their own. In what follows, we describe three types of intra-PHR hazards we encountered in our endeavor to fully-pipeline BBQ, and the architectural techniques used to address them.

(H_1) Data Hazards: The simplest form of hazards we encounter are *data* hazards, where one stage of the pipeline either: (a) issues a memory read, or (b) waits on completion of a memory read at an address that is concurrently updated by a different pipeline stage.

For instance, consider the BBQ pipeline depicted in Table 4.2. If Stage 2 issues a read for an L_2 bitmap that is simultaneously being modified by Stage 6, it will receive either a stale or invalid value 2 cycles later.[4] Similarly, if Stage 3 has a read in progress for the same memory address, it will receive a stale value on the next cycle. This is a common problem in processor design, where the standard solution — and the one we use here — is to perform ***write forwarding*** from a later pipeline stage to its predecessors when a read-after-write conflict occurs.

In order to resolve data hazards, we need to first compute *whether* and *which pairs of* operations in a PHR access the same state (*e.g.*, bitmaps, StOCs, PBs). BBQ exploits the hierarchical nature of the queue to make this computation efficient: since bitmap and StOC addresses also have a hierarchical structure to them (*e.g.*, the address of an L_3 bitmap is generated by splicing together the address of its L_2 parent and its own index in the parent bitmap), we can both reduce address comparator logic and improve $f_{\max}$ by memoizing address conflicts at higher levels and propagating them down the pipeline; then, when we need to compute address conflicts for lower levels, we reuse the memoized results, necessitating comparison of only the lower address bits.

(H_2) Non-atomic bitmap accesses: A second type of intra-PHR hazard arises due to the non-atomic nature of bitmap accesses, causing back-to-back DEQUEUE operations to be routed down incorrect paths in the bitmap tree.

To illustrate this problem, consider the example depicted in Figure 4.3. Initially, at ①, both the MSb and LSb of an L_2 bitmap are set, and the corresponding StOC values are 1 and 2, respectively. Now, consider two DEQUEUE–MAX operations, OP_A and OP_B, issued one cycle apart. Since the highest-priority (left-hand) subtree has just one element and becomes empty

[4]Certain hardware platforms may guarantee a consistent memory view, but this is not true in general (*e.g.*, FPGA SRAM).

after the first DEQUEUE operation (i.e., OP_A), we would *expect* to see OP_B to be routed down the right-hand subtree (corresponding to the LSb). Instead, we find that *both* operations are incorrectly routed down the left-hand subtree. The problem arises at ③, the moment OP_B and OP_A reach Cycles 4 and 5 of the pipeline, respectively. At this point, OP_A is waiting on the read for the MSb's StOC (issued one cycle earlier, at ②) to complete, while OP_B computes the same MSSb as OP_A and issues a read for the same StOC. It is only on the following cycle — at ④, when OP_A decrements the MSb's StOC down to 0 — that we discover that the left-hand subtree becomes empty, and that OP_B *should* have been steered to the right-hand subtree instead; unfortunately, it is far too late by this point. Note that this is not simply a rare performance issue that can be addressed by, *e.g.*, discarding trailing operations in case of conflicts; rather, it is a *correctness* bug. In this case, since OP_B may have already been "committed" to StOCs earlier in the pipeline, the operation cannot simply be discarded. Once again, we would have to either: (a) stall the pipeline, hurting worst-case performance, or (b) enforce that DEQUEUE operations are issued at least 2 cycles apart, throttling the queue's DEQUEUE throughput.

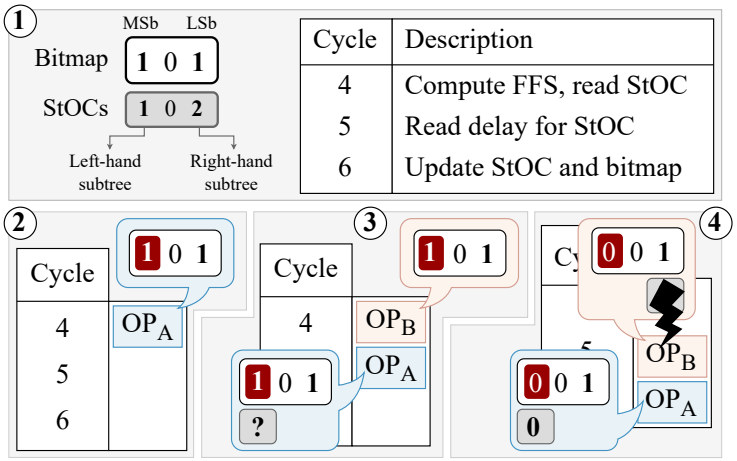


Figure 4.3: Non-atomic read-modify-write accesses to bitmaps cause OP_B (the second of two consecutive DEQUEUE–MAX operations) to be incorrectly routed to the left-hand subtree.

To address this problem, we adopt another technique from the architecture literature: ***speculation***. The idea is as follows: within L_i PHRs, if we have two back-to-back DEQUEUE–MAX operations (say, OP_A and OP_B, issued in that order, respectively) that access the same bitmap, *we compute the bitmap index for OP_B speculating that OP_A causes the MSSb to become '0'*. Consequently, OP_B will issue a read for the next-MSSb. On the following cycle, if we find that OP_A indeed caused the MSSb to flip (i.e., the corresponding StOC becomes 0), our speculation was correct, and OP_B proceeds as usual. Otherwise, OP_B simply discards its own state, and adopts both the MSSb index and StOC values computed by OP_A for the remainder of the pipeline. The speculation logic is illustrated in Figure 4.4 (updated steps ③ and ④ are depicted in Figure 4.4.)

The key observation here is that instead of squandering OP_B's one available read on data

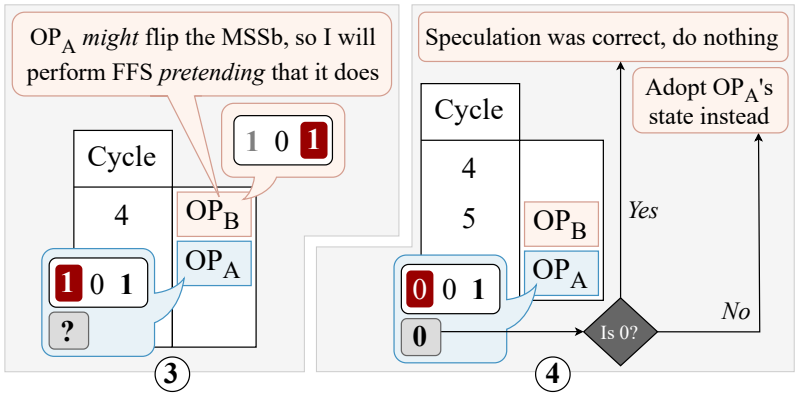


Figure 4.4: If there are conflicting operations in the L_i PHRs, operations issued later compute bit indices *speculating* that earlier operations will change the bitmap.

that are going to be available anyway (via inter-stage forwarding), we can "hedge" our bet on multiple bits simultaneously, ensuring that at least one of them yields the desired outcome. We also note that, while the description presented here involves only DEQUEUE-MAX operations for the sake of simplicity, the same technique generalizes to any combination and order of operations (i.e., ENQUEUE, DEQUEUE-MIN, and DEQUEUE-MAX).

(H_3) Non-atomic PB accesses: Conceptually similar to (H_2), the third and final type of hazard arises due to the non-atomic nature of priority bucket accesses, causing back-to-back DEQUEUE operations to potentially corrupt state within the *PB* PHR. To see why, consider stages $6-10$ of the BBQ pipeline. In Stage 10, DEQUEUE operations cause the PB's TAIL pointer to be updated (now denoted by PB.TAIL$^{\text{new}}$). In stage 8, DEQUEUE operations perform a read that is *supposed to be addressed* by the most up-to-date TAIL pointer for the corresponding PB. However, consider what happens when two back-to-back DEQUEUE operations (say, OP_A and OP_B, issued in that order, respectively) land at the same priority bucket. At Stage 9, OP_B is waiting on completion of the read addressed by PB.TAIL available on the previous cycle. However, OP_A, now at Stage 10, modifies the TAIL pointer, causing the read issued by OP_B (for QE_{DATA} and QE_{PREV}) to become stale. Unfortunately, write-forwarding does not help here because the stale variable (PB.TAIL) is being used to address other state, creating a control hazard.

To address this problem, we introduce the notion of ***operation coloring*** (inspired by *graph coloring*, problems where vertices in a graph must be assigned colors such that no two adjacent vertices have the same color), which works as follows. First, we tag each operation with a *Color* attribute, which assumes one of two values: *Purple* (■) or *Orange* (●). An operation's color determines which end of the PB's doubly-ended linked-list it interacts with: operations colored purple operate on the HEAD of the PB, while those colored orange operate on the TAIL. All ENQUEUE operations are always colored purple, while DEQUEUE operations are, by default, colored orange. Finally, we add a single constraint on color: *a DEQUEUE operation must not*

have the same color as a conflicting operation issued immediately before it (i.e., one cycle earlier). In the first cycle of the PB PHR (Cycle 7), if the active operation is a DEQUEUE that conflicts with another operation in the subsequent pipeline stage, it is recolored. Table 4.3 depicts the relevant portion of the BBQ pipeline post operation coloring (extraneous details are omitted for the sake of brevity).

C	Description	
7	// Color operation	
8	If DEQUEUE ■: Rd QE_{DATA}[PB . HEAD^{new}] Rd QE_{NEXT}[PB . HEAD^{new}]	If DEQUEUE ●: Rd QE_{DATA}[PB . TAIL^{new}] Rd QE_{PREV}[PB . TAIL^{new}]
9	// Read delay	
10	If ENQUEUE ■: Update PB . HEAD ^{new} If DEQUEUE ■: Update PB . HEAD ^{new}	If DEQUEUE ●: Update PB . TAIL ^{new}

Table 4.3: Updated stages 7 – 10 showing operation coloring.

Observe that, so long as the DEQUEUE operations are colored differently from any operation *immediately preceding them in the pipeline*, they will not incur stale reads. The key idea here is that each operation (whether an ENQUEUE or a DEQUEUE) only affects one pointer in the PB's (HEAD , TAIL) pair.[5] As a result, picking a mutually exclusive color also guarantees exclusivity on the data structure itself. Operations spaced more than one cycle apart can be safely handled via write forwarding. An artifact of this design choice is that back-to-back DEQUEUEs landing at the same priority bucket will not dequeue elements in FIFO order; however, since DEQUEUEs are bound to be interleaved with ENQUEUEs during typical operation, we do not expect this case to arise often.

Summary. Together, these optimizations realize a fully-pipelined priority queue architecture with both high $f_{\max}$, and a guaranteed throughput of 1 op/cycle independent of workload.

4.3.3 Implementing Subtree Occupancy Counters (StOC)

In §4.3.2, we described how every bit in BBQ's bitmap tree is associated with a StOC, which represents the *total number of elements contained in the corresponding subtree.* StOCs are an important component in BBQ because they are a key enabler for its fully-pipelined architecture. In this section, we characterize two practical details regarding our implementation of StOCs: how they are sized, and a general optimization technique that improves their performance.

[5]The only situations in which *both* pointers are affected is when the priority bucket becomes empty (i.e., due to a DEQUEUE), or a priority bucket that was *previously* empty becomes non-empty (i.e., due to an ENQUEUE). Speculation precludes the first possibility (attempting to DEQUEUE an empty PB), so we are left with the second corner case, which we handle explicitly.

Sizing: In order to handle the worst case (i.e., all elements in the BBQ being contained in a single priority bucket), every StOC must be provisioned to represent the range $[0,\ N]$, where N is the number of supported queue elements. Conventionally, N is configured to be a power of two (i.e., 2^k) to avoid wasting resources such as pointer address bits. However, in the case of BBQ, naively provisioning the queue with $N = 2^k$ elements would entail $(k+1)$-bit StOCs, with the most-significant bit (MSb) only ever being used to encode the maximum occupancy of 2^k. Instead, in BBQ, we snap N to a value of the form $(2^k - 1)$, allowing us to use k-bit StOCs. Thus, carefully sizing the queue (and deliberately wasting one element's worth of address space) saves 1 bit per StOC, yielding a sizeable reduction in memory footprint.

Waterlevel Bit Optimization: Since StOC bit-widths scale with the queue size, performing arithmetic or logical operations on these counters can be expensive for large BBQ instances. As we will see, these operations sometimes need to be chained together with other combinational logic in a single pipeline stage, which in turn inflates the critical path and significantly degrades $f_{\max}$. Here, we describe a general optimization technique that alleviates counter-related performance bottlenecks, yielding up to 17% higher $f_{\max}$ for some BBQ configurations.

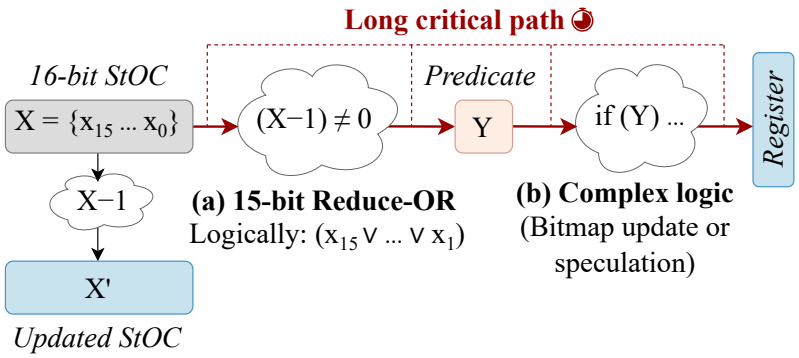


Figure 4.5: Dependency chain involving 16-bit counter logic for a DEQUEUE operation. The critical path comprises of (a) a 15-bit Reduce-OR (to determine whether the StOC becomes 0), chained with (b) more combinational logic (which uses (a) as a predicate).

To illustrate the problem, consider a DEQUEUE operation in stage 6 of the BBQ pipeline depicted in Table 4.2. This stage comprises of several sub-operations, two of which we will focus on here: (a) decrementing an L_2 StOC and checking if the resulting value becomes zero, and (b) updating the L_2 bitmap predicated on the result of (a). Note that (a) operates on a $\log_2(N+1)$-bit counter and (b) operates on a w-bit bitmap, and chaining these sub-operations together inevitably puts them on the critical path. The problem is further exacerbated if (b) entails more complex combinational logic (*e.g.*, resolving speculation outcomes, which, as shown in step ④ of Figure 4.4, follows the same blueprint). The critical path for this sequence of sub-operations is depicted in Figure 4.5.

To address this performance bottleneck, we augment every StOC in BBQ with an additional bit called the **waterlevel bit** (WLb). We maintain the invariant that the WLb is set to '1' if the current StOC value is greater than or equal to 2, otherwise it is set to '0'. The key idea is that *the WLb opportunistically memoizes the future result of (a)*,[6] allowing it to directly serve as the predicate for (b) instead of having to compute it from scratch (Figure 4.6). This avoids chaining the expensive Reduce-OR computation with other complex combinational logic, thereby shrinking the critical path. Moreover, the updated value of the WLb (corresponding to the current StOC value minus 2) can be efficiently computed using another 16-bit operation; however, since this is not chained with additional logic, it does not appear on the critical path.

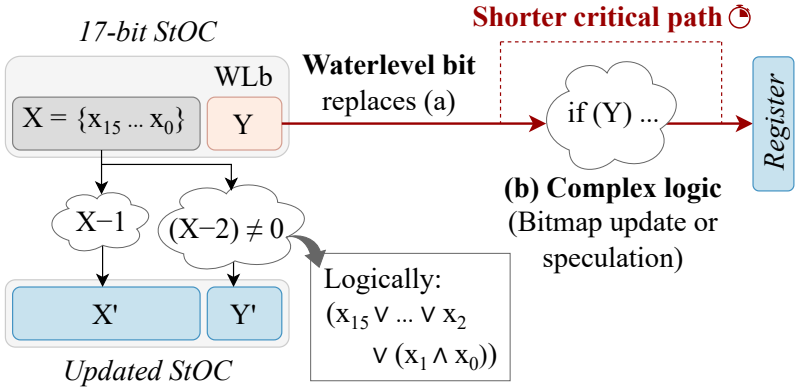


Figure 4.6: The waterlevel bit (WLb) replaces (a), improving $f_{\max}$ by removing the counter operations from the critical path.

In the context of BBQ, this optimization yields between $5-17\%$ higher $f_{\max}$ on the Stratix 10 MX FPGA for configurations with $(2^{17}-1)$ queue entries (i.e., 17-bit StOCs). This does come at a resource cost, since every StOC must now be one bit wider to accommodate the WLb (corresponding to approximately 4.75kB higher SRAM usage for a BBQ that supports 32K priorities with 8-bit bitmaps). However, we find that the resulting performance improvements justify this resource overhead, and we enable this optimization by default in the BBQ artifact.

Finally, we note that the underlying technique (i.e., memoizing useful counter arithmetic results in the counter structure itself) is a general one that may benefit *any* design which uses occupancy counters that might ultimately appear on the critical path (*e.g.*, BMW-Tree [158]).

4.4 Logical Partitioning in Practice

In §4.1.1, we motivated *logical partitioning* (i.e., the ability to multiplex several logical priority queues atop a single physical queue) as a key requirement for deployability in modern switches and SmartNICs. In this section, we first characterize the extent to which existing priority queue designs can realize logical partitioning (§4.4.1), followed by a detailed description of the mechanisms that enable BBQ to achieve this functionality (§4.4.2).

[6]Observe that, for positive values of X, the expression evaluated by (a), $(X-1) \neq 0$, is logically equivalent to $X \geq 2$, the value encoded in the WLb.

4.4.1 Existing Designs

The goal of logical partitioning is to allow a single, physical priority queue to emulate a collection of multiple, *logically-independent* priority queues. Simply realizing this abstraction is not particularly challenging, but doing so *efficiently* turns out to be a major impediment for most priority queue designs. We can evaluate efficiency along three axes: (1) *queue fragmentation*, or the worst-case fraction of queue elements (QEs) lost to external fragmentation when an instance is partitioned q ways; (2) *performance overhead*, or the throughput degradation resulting from logical partitioning; and, (3) *resource overhead*, or the resource cost (*e.g.*, logic, memory) required to support q logical partitions relative to an unpartitioned priority queue.

PIFO supports logical partitioning with zero queue fragmentation, a small performance overhead, and a resource overhead that scales linearly with queue size: Consider the example depicted in Figure 4.7, where a physical PIFO (provisioned with $N = 8$ QEs) is partitioned into $q = 2$ logical PIFOs. Per usual, PIFO maintains a sorted list of QEs ordered by priority [135]. To realize logical partitioning, PIFO annotates every QE with a *Logical PIFO ID* (in this case, q_0 or q_1) at enqueue time. Then, in order to dequeue from the i'th logical PIFO, it first "selects" the subset of elements with the corresponding ID (q_1 in our example), then performs priority decoding to extract the highest-priority element from that subset. Since every QE in the physical instance can always be independently addressed by every logical PIFO, queue memory is fully multiplexed, yielding zero fragmentation. Every element must be annotated with a $\log_2 q$ bit wide ID, resulting in a resource overhead that scales with queue size. Finally, selecting the appropriate subset of elements involves an extra comparator per element, incurring a small performance cost.

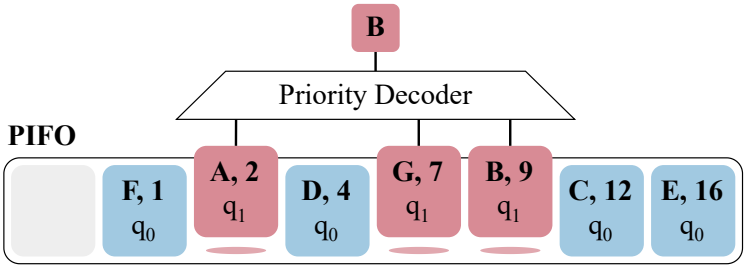


Figure 4.7: A single physical PIFO partitioned into 2 logical PIFOs: q_0 consisting of 4 elements, and q_1 consisting of 3 elements. Available queue memory is fully multiplexed among the logical PIFOs, resulting in zero fragmentation.

With modest changes, PIEO can support logical partitioning with zero queue fragmentation, and performance/resource overheads that scale with queue size: Architecturally, PIEO is organized as a matrix: an array of $2\sqrt{N}$ *sublists*, each consisting of $\sqrt{N}$ QEs sorted by priority. Besides standard priority queue operations, PIEO allows specifying *eligibility predicates*: a programmable function that "selects" a subset of elements to dequeue from. In principle, this is similar to the mechanism PIFO uses to implement logical partitioning (Figure 4.7), and an appropriate predicate function can be used in PIEO to the same effect. Unfortunately, the

vanilla PIEO design does not allow QEs belonging to different logical PIEOs to coexist in the same sublist. The reason is that, as a first step, PIEO must perform predicate filtering at sublist level, which only supports range-based queries (*e.g.*, $a \leq f \leq b$) but not set queries (*e.g.*, $f \in X$) that are required for logical partitioning. Consequently, it would incur external fragmentation at the granularity of sublists. However, we note that this is not a fundamental limitation, and with minor changes to the design (*e.g.*, by annotating every sublist with a q-bit bitmap representing the logical PIEO QEs contained therein), PIEO can, in theory, achieve logical partitioning with no external fragmentation while incurring a resource/performance cost similar to PIFO.

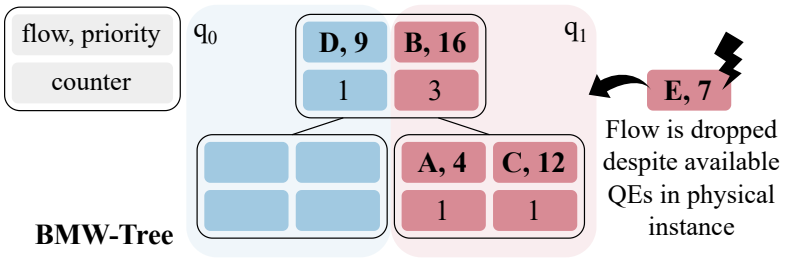


Figure 4.8: A 2-level 2-way BMW-Tree instance partitioned into 2 logical queues. Each logical queue must be mapped to a *physical* subtree, resulting in external fragmentation. Once a subtree becomes full, enqueues into the corresponding logical queue are impossible even if there are available QEs in other subtrees.

Other comparison-based priority queue designs cannot efficiently implement logical partitioning: PIFO and PIEO both satisfy a property that is key to achieving zero fragmentation in fixed-layout priority queues: maintaining a total ordering over elements at all times. *In their effort to leverage spatial and pipelined parallelism, other designs violate this property, inducing significant queue fragmentation.* We illustrate this point using the 2-way BMW-Tree depicted in Figure 4.8. To implement partitioning between $q = 2$ logical BMW-Trees, each logical queue must be statically mapped to a *physical* subtree as shown in the figure;[7] any other arrangement might result in inadvertently dequeueing from the wrong logical queue. The result is that, for a BMW-Tree instance of size N, each logical queue can address only $\frac{N}{q}$ QEs, incurring a fragmentation cost that scales with q. For instance, with $q = 8$ partitions, we would risk losing up to 87% of the queue memory to external fragmentation (and still not be able to support N flows). Instead, over-provisioning the physical instance to account for the worst case (all N elements being enqueued in a single logical queue, while the other 7 queues remain empty) would incur a 700% memory overhead. Other tree-like priority queue designs, such as pHeap [29] and Pipelined Heap [78], encounter precisely the same issue.

Summary. Overall, we find that besides the designs that maintain a total ordering over elements (PIFO and PIEO), *realizing logical partitioning in comparison-based designs incurs prohibitively high resource overhead, queue fragmentation cost, or both.*

[7]Heap property is intentionally violated at root level to enable partitioning.

4.4.2 Logical Partitioning in BBQ

Given a BBQ instance with w-bit bitmaps, we illustrate how to partition it into q logical BBQs by means of two exemplar configurations: one where $q \leq w$, and another where $q > w$.

(1) Fewer logical partitions than the bitmap width ($q \leq w$): Consider a 2-level BBQ with 4-bit bitmaps (i.e., $w = 4$, $D = 2$) that we would like to partition into $q = 2$ logical BBQs. The physical BBQ has a priority span of $P = 4^2 = 16$ priorities. As a first step, we partition this range equally between the two logical instances, allocating priorities $[0, 7]$ to queue q_0, and $[8, 15]$ to q_1. Observe that, in order to facilitate this split, the L_1 bitmap also needs to be partitioned as shown in Figure 4.9, with the lower two bits corresponding to q_0, and the upper two bits corresponding to q_1. Enqueueing an entry, X, into logical queue $i \in [0, 1]$ with relative priority $j \in [0, 7]$ is simple: first, we compute the *absolute* priority corresponding to the physical BBQ as $p = (i \times 8) + j$,[8] then perform ENQUEUE(X, p) as usual. In order to dequeue the highest (or lowest) priority entry from the i'th logical queue, we first mask the bits in the root bitmap *not* corresponding to q_i (*e.g.*, for $i = 1$, we would apply the mask 0xC), perform FFS on them, then proceed down the bitmap tree as in a typical DEQUEUE operation.

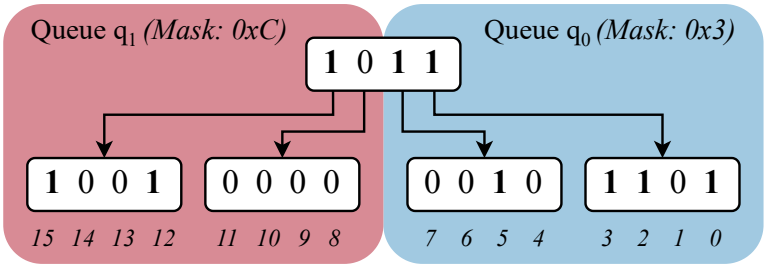


Figure 4.9: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 2 logical BBQs, q_0 and q_1. Each logical BBQ is allocated disjoint ranges of 8 priorities. To DEQUEUE from a logical BBQ, we first apply the corresponding *mask* to the L_1 bitmap before performing FFS on it.

(2) More logical partitions than the bitmap width ($q > w$): Consider again a ($w = 4$, $D = 2$) BBQ that we would now like to partition into $q = 8$ logical BBQs. Observe that each bit in the root bitmap now corresponds to *two* different logical BBQs, and therefore does not offer any discriminatory power.[9] Consequently, we eliminate this level of the bitmap tree; in its place, we insert a single pipeline stage that steers operations to their respective subtrees based on the logical queue index (*e.g.*, ENQUEUE and DEQUEUE operations on $i \in \{2, 3\}$ are steered to the second-from-right subtree). The updated bitmap tree structure is depicted in Figure 4.10. Each L_2 bitmap now maps to $q' = 2$ different logical BBQs, and we apply the idea described in (1) (since $q' \leq w$) to achieve this partitioning.

[8]Logically, this corresponds to simply concatenating together i and j.
[9]Since either of the logical BBQs contained therein may be empty, a '1' in any bit position of the L_1 bitmap provides no guarantee that a DEQUEUE operation on that subtree will succeed.

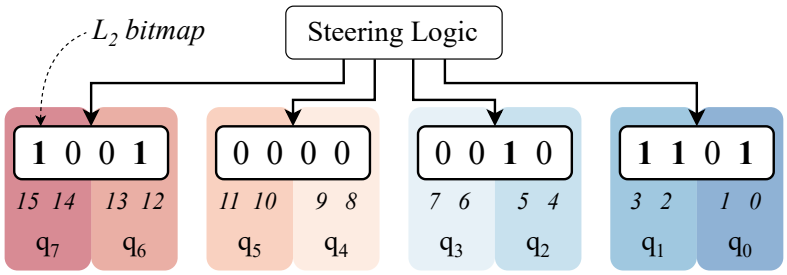


Figure 4.10: Bitmap tree for a 2-level BBQ with 4-bit bitmaps partitioned into 8 logical BBQs. The root (L_1) bitmap no longer adds any value, so we replace it with a *steering stage* that simply routes operations on logical BBQs to the corresponding subtree.

Thus, with nominal changes to the BBQ pipeline, we can support a broad range of LP configurations without any performance overhead. In contrast to PIFO and PIEO, the resource cost (corresponding to over-provisioning the priority range) scales with the *degree of logical partitioning* rather than queue size. Finally, full decoupling between its priority index structure (i.e., the bitmap tree) and QEs enables BBQ to achieve zero queue fragmentation. Overall, these techniques make it possible for BBQ to *efficiently* realize logical partitioning.

4.5 Extensions to the BBQ Primitive

In this section, we describe two useful extensions to the BBQ primitive that ameliorate some of the limitations of the original design.

4.5.1 BBQ$_\odot$: A Latency-Free BBQ

While deep pipelining is key to BBQ's high performance, the resulting *pipeline latency* (i.e., the number of clock cycles that elapse between when an operation is issued and when it completes) introduces a new source of error in the relative ordering of elements compared to an "ideal" priority queue.[10]

The problem manifests due to a confluence of two factors: (a) since it takes several cycles for an operation to traverse the pipeline, in order to use BBQ at full throughput (1 op/cycle), multiple operations need to be issued concurrently; and (b) for a pipeline of depth p, *the minimum delay* for a high-priority element to be dequeued, served, and re-enqueued into the queue is also p. *Consequently, any DEQUEUE operations that are issued in the p cycle interval that the highest-priority element is not present in the queue might ultimately dequeue lower-priority elements.* The aforementioned problem is inextricably tied to our decision of using a pipelined architecture, implying that the BBQ primitive alone cannot guarantee absolute accuracy at full throughput.

[10]We note, here, that there is a more obvious source of error in relative ordering that results from our decision to use an IPQ-based approach. Since IPQs only operate over fixed, integral priority ranges, *quantization* (i.e., "binning" elements into a smaller number of priority buckets than the original universe of priorities) might also cause the relative ordering to deviate from ideal. In keeping with convention, we call this loss in *precision* instead of *accuracy*.

However, we find that a simple augmentation to the BBQ primitive allows us to hide this latency and avoid the accuracy issues that come with it: use a tiny PIFO as a "cache" in front of the BBQ to hold the highest-priority elements. Whenever this tiny PIFO overflows, it "leaks" the lowest priority element to the BBQ. This PIFO only needs to be able to hold as many elements as the BBQ's pipeline depth (order of tens of elements). Because of its small size, this instance does not face the scalability limitations associated with the PIFO architecture and only adds a small footprint to the design.

The augmented design, BBQ$_{\odot}$, *provably guarantees zero loss in accuracy (i.e., any dequeued element is always the highest-priority one in the system at that time) while providing full throughput (1 op/cycle).* We can show this by proving the sufficient condition in Theorem 4: with a PIFO whose size exceeds the pipeline latency of the BBQ, the highest-priority element is always served from the PIFO, such that we never experience the accuracy or latency artifacts introduced by the accompanying BBQ. We start with a concrete example motivating the problem, then dive into BBQ$_{\odot}$'s design, followed by a proof of its correctness.

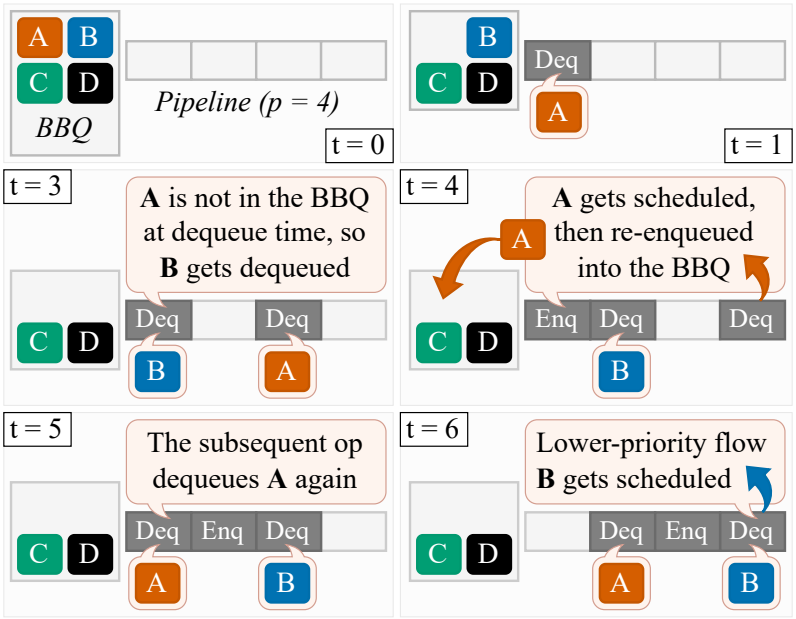


Figure 4.11: Issuing concurrent DEQUEUE requests, in combination with BBQ's pipeline latency, incorrectly causes a lower-priority flow, B, to be extracted (at $t = 3$) and scheduled (at $t = 6$).

Motivating Example: Consider the scenario depicted in Figure 4.11, where we use a BBQ instance with a pipeline latency of $p = 4$ cycles to implement strict priority scheduling at a bottleneck switch. There is a single high-priority flow, A, competing with 3 lower-priority flows (B, C, and D); if none of the flows are application-limited, we expect A to receive the full share of bandwidth, while the other flows should starve (i.e., never be served). Assume now that a single packet can be transmitted every other cycle (i.e., line rate corresponds to one packet every two cycles). Since it takes 4 cycles for the BBQ instance to complete a DEQUEUE request and yield the appropriate flow to schedule, we are faced with two alternatives for managing priority queue state.

First, when a flow is scheduled, we might re-enqueue the flow in the BBQ and immediately issue another DEQUEUE request, wait 4 cycles for the BBQ to respond with the next flow to schedule, and so on. This guarantees accuracy (i.e., the scheduled flow is always the highest-priority one), but implies that flows can only be scheduled every 4 cycles, wasting half the link bandwidth.[11]

The second option is to preemptively maintain *multiple* concurrent DEQUEUE requests in flight such that a flow is *always* available to be scheduled. In our example, this corresponds to issuing DEQUEUE operations 2 cycles apart such that a flow gets dequeued every other cycle (*e.g.*, at $t = 5$ in Figure 4.11). While this saturates the link bandwidth, it also implies that *not all active flows are enqueued in the BBQ at dequeue time.* For instance, at $t = 1$, the first issued DEQUEUE operation extracts flow A from the BBQ. Consequently, at $t = 3$, the next DEQUEUE operation results in the extraction of a lower-priority flow, B. Flow A eventually returns to the BBQ at $t = 4$, but it is far too late by this point: at $t = 6$, the second DEQUEUE operation completes, yielding B. In effect, this violates the strict priority scheduling requirement we sought to enforce.

BBQ$_{\odot}$ Design: BBQ$_{\odot}$ is composed of two components: a BBQ and a PIFO instance, which are connected as shown in Figure 4.12. To simplify the theoretical analysis of this system, we assume that both components run at the same clock frequency (say 250MHz), and the system clock, denoted by CLOCK$_{sys}$ runs at half the frequency. On each CLOCK$_{sys}$ cycle, we can insert one element into the BBQ$_{\odot}$, extract the highest-priority element, or both.

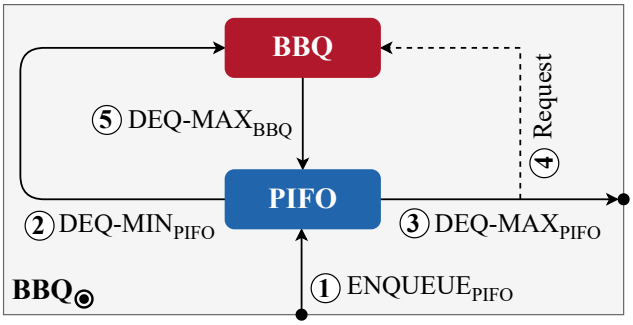


Figure 4.12: BBQ$_{\odot}$ design.

At a high level, our goal is to keep the PIFO full, only inserting into the BBQ when low-priority elements "spill over" from the PIFO. When a new element arrives at the system, it is first ENQUEUE'd into the PIFO ①; if this causes the PIFO to overflow its capacity (k), we perform a DEQUEUE-MIN$_{\texttt{PIFO}}$ operation ②, inserting the resulting element into the BBQ. Extracting the highest-priority element from the system involves two steps: (a) we perform a DEQUEUE-MAX$_{\texttt{PIFO}}$ operation to get the highest-priority element in the PIFO ③, which completes immediately, and (b) we issue a DEQUEUE-MAX$_{\texttt{BBQ}}$ request to fetch the highest-priority element from the BBQ ④, which takes p timesteps to complete (where p is the pipeline latency of the BBQ in units of CLOCK$_{sys}$ cycles). Finally, when a DEQUEUE-MAX$_{\texttt{BBQ}}$ operation

[11] Alternatively, we might schedule *bursts* of packets at a time so as to hide the pipeline latency, but this effectively imposes a leaky bucket atop the underlying scheduling policy, which may not always be desirable.

completes ⑤, we insert the resulting element into the PIFO; as before, if this causes the PIFO to overflow, we move the lowest-priority element in the PIFO to the BBQ. Observe that on every $\text{CLOCK}_{\text{sys}}$ cycle (corresponding to 2 clock cycles for each component), we issue at most 2 BBQ operations and 4 PIFO operations (2 ENQUEUEs and 2 DEQUEUEs), which matches their respective operation throughputs.

Proof of Zero Accuracy Loss: In this section, we prove a sufficient condition for $\text{BBQ}_{\odot}$ to guarantee zero accuracy loss when using an appropriately-sized PIFO. We start by proving a lower bound on PIFO occupancy when the associated BBQ is not empty, followed by an invariant regarding the subset of priorities contained in the PIFO at any time. In the remainder of the section, we use $\mathcal{P}(t)$ to denote the set of elements contained in the PIFO at time t, and $|\mathcal{P}(t)|$ to denote the cardinality of this set (and transitively the PIFO occupancy).

Lemma 4 (Lower-Bound on PIFO Occupancy). *In a $\text{BBQ}_{\odot}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, if $|\mathcal{P}(t)| < (k-p)$, the BBQ is empty at time t.*

Proof. The intuition behind the Lemma is that, so long as the BBQ is *not* empty, it will prevent the PIFO occupancy from dropping below a certain threshold (corresponding to $k-p$). We prove this claim via contradiction, showing that if $|\mathcal{P}(t)| < (k-p)$ starting with a full PIFO (a necessary condition for the BBQ to be non-empty), at least one $\text{DEQUEUE-MAX}_{\text{BBQ}}$ request resulted in $\varnothing$ after p timesteps, implying that the BBQ is empty at time t.

Let t_1 denote the *latest* time that the PIFO was full, and let t_2 denote the *earliest* time such that $|\mathcal{P}(t_2)| < (k-p)$. Note that no elements are inserted in the BBQ in the period $[t_1, t_2]$; otherwise $\exists\ t_1' > t_1$ where the PIFO is still full, implying that t_1 was not the latest time. Assume towards a contradiction that a total of n_1 $\text{DEQUEUE-MAX}_{\text{PIFO}}$ operations and $n_2 \geq 0$ $\text{ENQUEUE}_{\text{PIFO}}$ operations were performed in $[t_1, t_2]$, and n_3 $\text{DEQUEUE-MAX}_{\text{BBQ}}$ operations completed in the same period, all of which returned non-$\varnothing$ values.

$$n_1 - (n_2 + n_3) > p, \tag{4.1}$$

$$n_3 \geq \max(0, n_1 - p) \tag{4.2}$$

where (4.1) is true because the difference between the number of departures from the PIFO (n_1) and the number of arrivals to the PIFO ($n_2 + n_3$) in $[t_1, t_2]$ must correspond to an occupancy drop from k to $|\mathcal{P}(t_2)| < k-p$, i.e., exceeding p. (4.2) is true because a $\text{DEQUEUE-MAX}_{\text{BBQ}}$ request is issued for every $\text{DEQUEUE-MAX}_{\text{PIFO}}$ operation, and the maximum number of requests still outstanding is at most p. Substituting (4.1) into (4.2), we get: $n_3 \geq (n_1 - p) > n_2 + n_3$, a contradiction. □

Theorem 4 (Priority Set Invariant for $\text{BBQ}_{\odot}$). *In a $\text{BBQ}_{\odot}$ instance composed of a BBQ with pipeline latency p cycles and a PIFO of size $k > p$, the top $(k-p)$ highest-priority elements are always in the PIFO.*

Proof. Given Lemma 4, we only need to consider the scenario where $(k-p) \leq |\mathcal{P}(t)| \leq k$. Assume that the BBQ is not empty, otherwise all $|\mathcal{P}(t)| \geq (k-p)$ elements in the PIFO are

trivially the highest-priority ones. Now, assume towards a contradiction that only the $m < (k-p)$ highest-priority elements in the system are in the PIFO at a certain time t_2. It follows that the highest-priority element in the BBQ, x, has a higher priority than the remaining $|\mathcal{P}(t_2)| - m$ elements in the PIFO at t_2. Observe that x may only have been inserted in the BBQ if, at the time of insertion, t_1: (1) the PIFO was full, and (2) all k elements in the PIFO had higher priority than x.

Now, for x to be the $(m+1)$'th highest-priority element in the system at time t_2, we must have performed $n = (k-m) > (k-(k-p)) = p$ number of DEQUEUE-MAX$_{\text{PIFO}}$ operations since t_1, implying that n $(> p)$ DEQUEUE-MAX$_{\text{BBQ}}$ operations were issued in the interval $[t_1, t_2]$. Since at most one DEQUEUE-MAX$_{\text{BBQ}}$ operation can be issued every timestep and each such operation takes exactly p timesteps to complete, it follows that *at least one* DEQUEUE-MAX$_{\text{BBQ}}$ operation completed and returned x. Thus, x is in the PIFO at time t_2, a contradiction. □

4.5.2 Dynamic Priority Ranges

As described in §4.2.1, the standard BBQ primitive operates over a priority range that is both *finite* and *static*. While this is sufficient in some contexts (*e.g.*, strict priority scheduling), many policies implicitly assume an infinite priority set (*e.g.*, fair queueing). IPQs such as BBQ are fundamentally incapable of upholding this assumption (§4.8). Fortunately, prior work has shown that in most cases, a *dynamic* — albeit finite — priority range is sufficient to realize these policies [124, 128]. In this section, we describe how BBQ can be extended to provide the abstraction of a *rolling priority window*.

To handle dynamic priority ranges, we directly adapt Eiffel's [124] idea of using a *Circular Hierarchical FFS-based Queue* (cFFS). The idea is to have two independent HFFS queues, each with priority span P, working in tandem: a primary HFFS queue, q_0, that stores elements with priorities $[0, P)$, and a secondary HFFS queue, q_1, mapping to elements with priorities in $[P, 2P)$ (i.e., just outside q_1's range). Together, these queues represent a *logical* priority window of $[0, 2P)$. Once the primary queue becomes completely empty, the logical priority window advances by P, and the queue designations are swapped, with q_0 (now the secondary queue) buffering elements with priorities in $[2P, 3P)$, and so on and so forth.

We follow precisely the same blueprint for BBQ, with logical partitioning enabling us to multiplex both q_0 and q_1 atop a single physical BBQ instance with no resource overhead (or modification to the primitive, for that matter). The only additional component required is a simple controller to orchestrate the two logical queues. We have not implemented this feature as part of our research artifact yet, but we do not expect it to have much impact on BBQ's performance.

4.6 Evaluation

We now evaluate BBQ. Our main goal is to understand BBQ's *performance* and *viability* for both ASICs and FPGA designs. We also compare BBQ with PIFO [135], PIEO [131], and BMW-Tree [158]. PIFO is the state-of-the-art hardware priority queue in terms of *throughput* while BMW-Tree is the state of the art in terms of *scalability*. Throughout this evaluation, we show that BBQ can surpass PIFO's throughput while achieving similar scalability to BMW-Tree.

4.6.1 Setup and Methodology

We implement BBQ in SystemVerilog. Given the recurring and composable structure of the design, we implement a Python script to automatically generate different configurations of BBQs by stitching together modular blocks of handwritten SystemVerilog code. Users can specify the number of levels in the tree (D), the bitwidth of every node (W), and the maximum number of elements (N). We synthesize BBQ targeting both an FPGA (Intel Stratix 10 MX FPGA [77]) as well as an ASIC. The Stratix 10 MX contains 702,720 Adaptive Logic Modules (ALMs), 140 Mb of SRAM, and two 100 Gb Ethernet ports. For comparison, we also synthesize PIFO, PIEO, and BMW-Tree targeting the same board. For each design and configuration, we conduct a bisection search to find the maximum clock frequency achievable with 3 MHz precision, picking the best synthesis across 10 seeds. To synthesize the FPGA we use Intel Quartus [76]. To synthesize the ASIC, we use Synopsys Design Compiler [141] using a 7 nm Standard Cell Library [146] based on the ASAP7 PDK [44].

All the designs we evaluate have deterministic performance that is independent of the workload. As such, our analysis focuses on the packet rate that each design is able to sustain, as well as the cost [123] (in terms of die area and FPGA resources).

4.6.2 FPGA

As described in §4.1.1, FPGA-based NICs are increasingly used to achieve programmable offloads [58, 60, 75, 121, 122], and having a programmable packet scheduler on the NIC would allow administrators to change the packet scheduling algorithms at run time. In this section, we evaluate how BBQ and the baseline designs perform, both in terms of throughput and FPGA resources. We also explore how the different BBQ design parameters affect its performance when running on an FPGA.

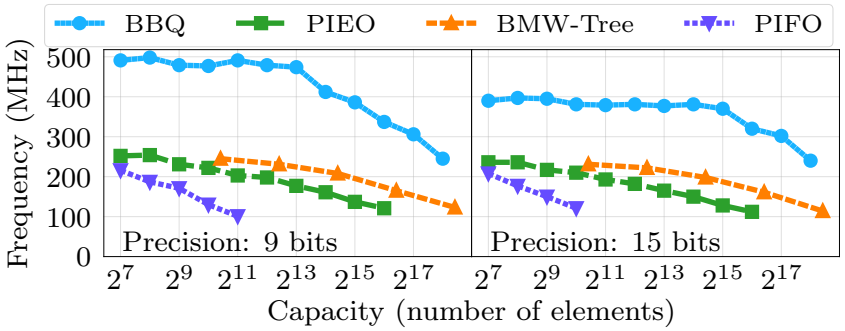


Figure 4.13: Clock frequency as we scale the queue capacity.

Throughput Scalability: Queue capacity can influence throughput by increasing the hardware critical path, which in turn reduces the maximum clock frequency that we can achieve with the design ($f_{\max}$). To understand this effect, we synthesize BBQ (with 8-bit bitmaps) and the baselines while changing both the queue capacity and the number of bits used to express the priorities (precision). We report both the clock frequency as well as the overall throughput achievable by each design.

Figure 4.13 shows the clock achievable by each design when we increase the queue capacity. BBQ achieves a clock as high as 500 MHz with 9-bit precision and 400 MHz with 15-bit precision, significantly higher than the baseline designs. Moreover, BBQ is able to scale to up to 2^{17} elements while still sustaining a 300 MHz clock. In comparison, PIFO can only scale to up to 2^{11} elements.

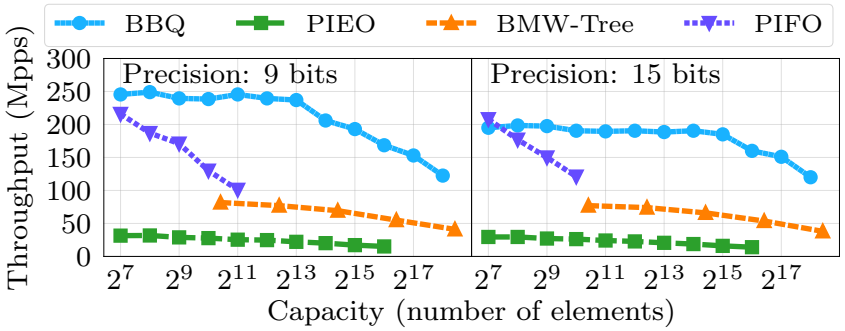


Figure 4.14: Throughput as we scale the queue capacity.

In addition to achieving a higher clock frequency, BBQ's fully pipelined design allows it to execute an operation (enqueue or dequeue) every clock cycle. As a result, the throughput difference is even higher compared to BMW-Tree (that consumes 1 cycle to enqueue and 2 cycle to dequeue) and PIEO (that consumes 4 cycles to enqueue and 4 cycles to dequeue). However, different from the other designs, PIFO is able to execute both an enqueue and a dequeue operation in the same cycle, allowing its packet rate to match its clock frequency. Figure 4.14 shows the throughput of the different designs for different queue capacities. BBQ is able to drive 100 Gbps line rate (148.8 Mpps) with as many as 2^{17} elements. Also note that PIFO is able to achieve similar throughput to BBQ, but only for small queues (128 elements or less).

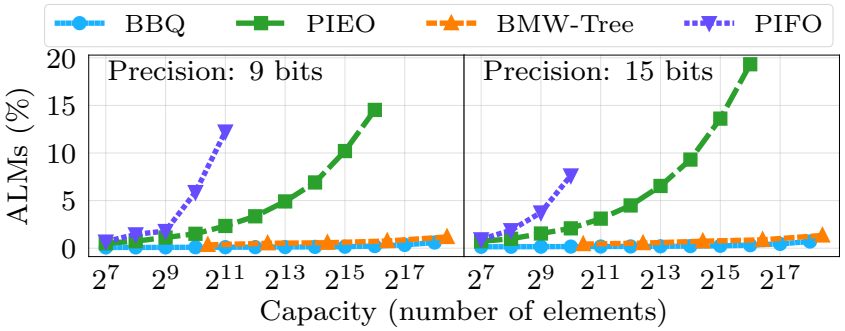


Figure 4.15: ALM utilization as we scale the queue capacity.

Resource Scalability: We also evaluate how the different designs scale in terms of FPGA resources. We report ALM utilization and SRAM blocks, both as a fraction of the overall number of resources available in the target FPGA. Figure 4.15 shows how the ALM utilization scales as we increase the number of elements that the queue can support. In BBQ, scaling the queue capacity has little effect on the logic utilization. This is a direct consequence of BBQ's use of an integer priority queue, which lets it avoid comparison-based sorting. BMW-Tree's hierarchical design also gives it much better scalability, using only slightly more ALM resources than BBQ. In contrast, both PIFO and PIEO use significantly more resources as we scale the capacity.

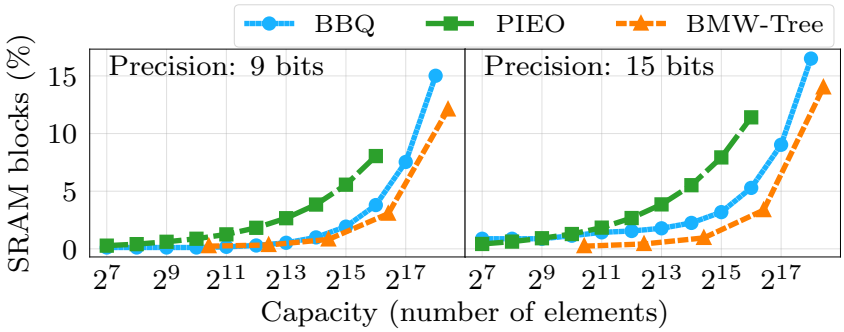


Figure 4.16: SRAM block utilization as we scale the queue capacity. PIFO is not included in the plot as it does not use SRAM.

While BBQ's ALM utilization remains low even for 131,072 elements (2^{17}), BBQ, like PIEO and BMW-Tree, relies on SRAM to store its elements. As a result, we expect SRAM utilization to increase with the queue capacity for all these designs. Figure 4.16 shows this effect. Note that BBQ's SRAM utilization is in between PIEO's and BMW-Tree's. However, PIEO and BMW-Tree require multiple copies in order to match BBQ throughput, which causes them to use vastly more SRAM if provisioned to meet the same performance target.

BBQ Sensitivity Analysis: To understand the impact of BBQ's configuration on performance, we perform a sensitivity analysis of the bitmap tree parameters: the number of levels (D), and the bitmap width (W) (recall that the number of priorities is computed as $P = W^D$), while keeping the number of elements fixed.

Figure 4.17 depicts how BBQ's throughput behaves as a function of the bitmap width. We sweep the number of levels in the bitmap tree from $D = 3$ to 15 (or until we reach 2^{15} priorities) for different bitmap widths. Then we plot the attained throughput for the corresponding priority count. We find that bitmap widths between 2 and 8 yield similar performance (with 4 being optimal), but this deteriorates as the bitmap width increases. In particular, starting with $W = 16$, *FFS computation* becomes the primary $f_{\max}$ bottleneck. We can similarly infer from the same graph that level count has little impact on performance; for instance, we observe that a $(D = 4, W = 2)$ BBQ achieves the same throughput as a $(D = 8, W = 2)$ BBQ despite the latter containing 16× as many priorities. Figure 4.18 shows the complementary view of the data, depicting change in throughput as a function of priorities for different numbers of tree levels.

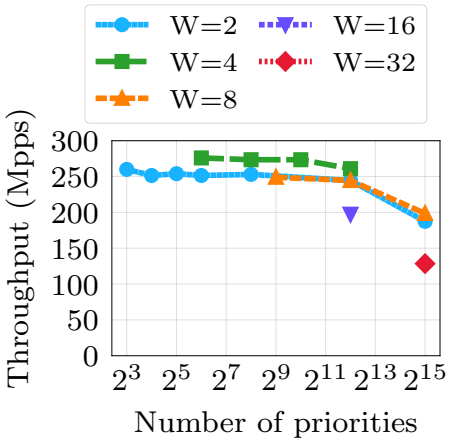


Figure 4.17: BBQ throughput as we increase the number of priorities when using different bitmap widths (W).

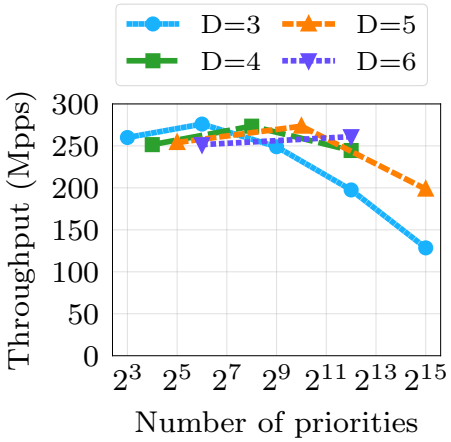


Figure 4.18: BBQ throughput as we increase the number of priorities when using different number of levels (D).

4.6.3 ASIC

We now evaluate BBQ in the context of designs targeting ASICs. Here we compare BBQ with PIFO for two reasons: (1) it is one of the two baseline designs that supports logical partitioning, which, as we discussed in §4.1.1, is essential to allow them to be efficiently incorporated into state-of-the-art switches, and (2) it offers the best throughput among all baseline designs.

We synthesize both BBQ and PIFO using a 7 nm process. PIFO only meets timing at 1 GHz with up to 2^{11} elements, which is consistent with [135]. BBQ meets timing at 3.1 GHz with 2^{17} elements, but we did not try scaling beyond this point. The difference in the clock frequency achieved by BBQ and PIFO means that BBQ is able to run at 55% higher throughput. To evaluate the cost of the design, we compare the chip area when synthesizing a single queue. We use the synthesis results to calculate the area used by the logic gates and estimate the SRAM area using the cost of 0.027mm^2/Mb reported by TSMC for their 7 nm process [38, 155].

				Area (mm^2)		
Design	Elements	Priorities	Clock	Logic	SRAM	Total
PIFO	2^{11}	2^9	1 GHz	0.043		0.043
	2^{11}	2^{15}	1 GHz	0.058		0.058
BBQ	2^{11}	2^9	3.1 GHz	0.00071	0.0029	0.0037
	2^{11}	2^{15}	3.1 GHz	0.00110	0.035	0.036
	2^{17}	2^9	3.1 GHz	0.00095	0.24	0.24
	2^{17}	2^{15}	3.1 GHz	0.00140	0.29	0.29

Table 4.4: Chip area for the different designs. BBQ uses little logic, causing its area to be primarily determined by SRAM.

Table 4.4 shows the chip area breakdown, divided in logic and SRAM for both BBQ and PIFO, using 9b and 15b priorities. BBQ uses very little area with logic; most of its area is taken up by SRAM. BBQ is not only able to scale to many more elements than PIFO, but also consumes less area when both are provisioned for the same capacity.

4.7 Applications

In §4.1, we motivated the need for a hardware priority queue capable of supporting packet scheduling for two rapidly emerging use-cases: terabit-scale switches, and NICs in multi-tenant cloud datacenters. Having evaluated its scalability, throughput, and resource usage, in this section we describe how BBQ can fill these application-level gaps.

4.7.1 Packet Scheduling on Switches

As described in §4.1.1, a key enabler for priority queue deployment in modern switches is the ability to realize *logical partitioning*, allowing the packet scheduler to leverage the simpler priority queue mesh architecture depicted in Figure 4.1(b). BBQ achieves this by treating disjoint subtrees in its priority index structure as independent queues (§4.4.2), enabling multiplexing of the underlying instance; as noted earlier, this comes at a cost in terms of precision (each logical queue gets $\frac{1}{k}$'th the original priority range), but with no resource overhead, performance penalty, or fragmentation of queue memory.

In terms of *performance* and *scalability*, we can synthesize BBQ instances with 100K+ entries and 32K priorities at 3.1GHz using a 7nm ASIC process; at two operations per packet, each BBQ instance can sustain a packet rate of 1.55Bpps. Our target switch (NVIDIA SN4700, 400GbE x32) supports an aggregated packet rate of 8.4Bpps, implying that a total of $\left\lceil \frac{8.4}{1.5} \right\rceil \times 2 = 12$ BBQs that are logically partitioned among the 32 output ports in a shared scheduler pipeline are sufficient to match the switch fabric's processing speed. Based on our ASIC synthesis results for a single BBQ (§4.6.3), we estimate that provisioning each of these 12 instances with 131K queue entries and 32K priorities (1K priorities per port) would require a total area of 3.48mm^2. Considering that a switch chip area ranges from 200mm^2 to 800mm^2 [135] this corresponds to 0.4–1.74% of the total chip area. Thus, BBQ's scalability, performance, and ability to be logically partitioned make it, for the first time, a *plausible* candidate to realize priority queueing in modern switches.

4.7.2 Packet Scheduling on Cloud SmartNICs

A key requirement for NIC-based packet schedulers in public clouds is the ability to independently perform scheduling both *across* tenants and *within* each tenant (§4.7.1). By allowing several logical BBQs (each representing one tenant) to share a single BBQ instance, we can efficiently use the available resources (*e.g.*, queue memory) without having to provision one priority queue per tenant. In order to enforce cross-tenant traffic policies, we instantiate another, smaller BBQ that stores references to the tenant BBQs. Conceptually, this corresponds to the two-level hierarchical scheduler depicted in Figure 4.19, with the lower and upper levels handling intra- and inter-tenant scheduling decisions, respectively.

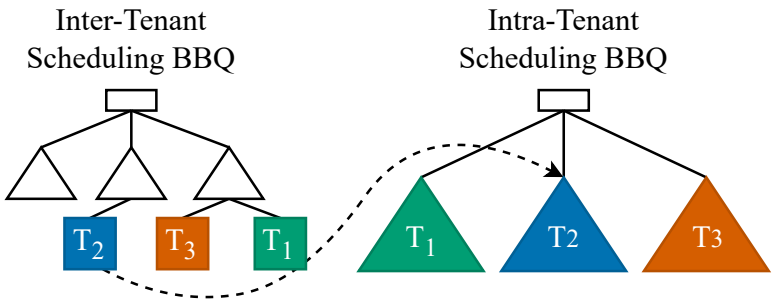


Figure 4.19: A two-level hierarchical NIC scheduler using BBQ. The first BBQ schedules traffic *across* tenants. The second BBQ (which is logically partitioned) is used to schedule traffic *within* each tenant.

Previously, we evaluated the feasibility of operating BBQ on an ASIC in the context of switches (§4.7.1). We now frame our discussion about scalability and performance for SmartNICs in the context of the more resource-constrained device family: FPGAs. On an Intel Stratix 10MX we can synthesize a BBQ instance with 100K+ entries and 32K priorities that meets timing at 302MHz, and uses 0.45% of the available ALMs and and 9% of the total FPGA SRAM, respectively. Consequently, a single instance can sustain 151Mpps, surpassing 100GbE line rate with minimum-sized packets (148.8 Mpps).

On FPGAs, scaling to higher line rates (*e.g.*, 400GbE [74]) is beyond the capability of any single priority queue instance, and would require augmenting the scheduler pipeline with multiple BBQs. However, given the resource cost of each instance relative to total FPGA resources, this is currently impractical; any priority queue design would have to significantly reduce its SRAM footprint (*e.g.*, offloading elements to DRAM) in order to make scaling out on FPGAs practical.

4.8 Limitations and Open Questions

Limitations: A key limitation of IPQ-based designs such as BBQ is that they operate over priority ranges that are both *finite*[12] and *static*. While we can, in fact, augment the vanilla BBQ primitive to support *dynamic* priority ranges (§4.5.2), boundedness of the priority span remains an immutable constraint. The fundamental reason is that we must map every priority in BBQ to a bucket in physical memory, so SRAM usage scales linearly with the priority span. Notably, this scheme becomes altogether impractical when the required precision grows beyond a certain threshold (*e.g.*, supporting 2^{32} priorities would require over 500MB of SRAM *just* to

[12]Technically, priority ranges are *always* finite (regardless of the underlying priority queue design) because they are ultimately upper-bounded by the maximum precision afforded by the priority tag (i.e., number of priority bits). However, the point here is that comparison-based priority queue designs (*e.g.*, PIFO) can, in principle, create the *illusion* of an infinite priority range using large priority tags; for instance, a time-based PIFO scheduler that uses nanosecond-granularity timestamps as priorities would require well over 500 years to exhaust a 64-bit priority range ($2^{64} = 1.8 \times 10^{19}$ priorities). Conversely, IPQs hit their priority scaling limits far earlier than any reasonable interpretation of infinity.

store bitmaps). In §4.6, we demonstrated the feasibility of synthesizing a BBQ instance with 15-bit priority tags (32K priorities), but we don't expect this number to scale much further. Thus, the ideal operating point for BBQ corresponds to a setting where we need to support a *large number of queue entries falling in a small (possibly dynamic) priority range*. Some priority queue architectures also enable richer abstractions (*e.g.*, PIEO's predicate-based filtering allows scheduling based on eligibility criteria such as virtual or wall-clock time [131]), which BBQ does not support.

Future Work: Today, we are at an inflection point with regard to the scalability of hardware priority queue designs. On the one hand, support for 100K+ queue entries is the culmination of a decade-long concerted effort towards jointly optimizing scalability and performance. On the other hand, this appears to be the end of the scalability roadmap: since every queue element must be stored *somewhere*, we are ultimately bottlenecked by available memory. For the sake of performance, today's designs exclusively use SRAM, which offers deterministic, single-cycle memory access. However, SRAM is a scarce resource, and even highly scalable designs such as BBQ and BMW-Tree [158] would require over 10% of the available FPGA SRAM (§4.6.2) to support 200K queue entries — a highly impractical proposition. However, given the trend of increasing multi-tenancy in datacenters, it is not far fetched to believe that schedulers will some day need priority queueing for 1M+ flows. A natural question then is: *how do we get there?* We believe the key to this lies in offloading queue entries to DRAM, which provides much slower (and non-deterministic) access latencies compared to SRAM, but is a far more abundant memory resource. The clean decoupling between BBQ's priority index structure and its queue memory (i.e., BBQ's ability to locate the highest-priority entry without needing access to the entry itself) makes it feasible to offload queue memory to DRAM, but there are several challenges that need to be addressed along the way. We leave it to future work to realize this lofty goal.

4.9 Related Work

Counting priority index: The data structure used in BBQ is similar to the counting priority index (CPI) proposed by Wang and Lin [148]. They were the first to hypothesize that an integer priority queue could be used to speed up packet scheduling in both software and hardware. Unfortunately, CPI is not implementable in hardware in its original form as it fails to account for the many practical issues that arise when building a pipelined hardware design, e.g., memory access latency, hazards, and limited memory. As we discussed in §4.2.3 and §4.3, the challenging aspects of BBQ's design stem from these very issues. BBQ is also orthogonal to Eiffel [124], which deals with the practical issues of using a priority index to schedule packets in *software*.

Hardware priority queueing: PIFO [135] is the current state-of-the-art priority queue implementation in terms of throughput, and BMW-RPU [158] is the current state-of-the-art in terms of scalability. As confirmed in our evaluation, BBQ is able to match PIFO's throughput while scaling beyond BMW-RPU's maximum capacity. Another notable hardware priority queue design is pHeap [29], unfortunately pHeap can only process an operation every two clock cycles, which makes it unsuitable for line-rate switches. Further, besides PIFO and PIEO, none of the

designs that close the gap in terms of scalability (including BMW-Tree and pHeap) support logical partitioning efficiently, making them impractical for deployment in both switches and SmartNICs in the cloud setting. We characterize the efficiency that existing designs achieve in implementing logical partitioning in §4.4.1.

Approximate priority queueing: There is also a line of work that proposes approximating different scheduling algorithms to make them amenable to hardware implementation [6, 7, 62, 128, 161]. BBQ borrows from these works the observation that a small priority set (at the hardware level) is sufficient for most use cases. These works provide exciting theoretical insights and a path to implement some scheduling policies on existing programmable switches. In some ways, BBQ is complementary to some of these works; for instance, both SP-PIFO [6] and programmable calendar queueing (PCQ) [128] assume that the underlying switch hardware provides a certain number of strict-priority queues, and their accuracy (or in case of PCQ, precision) improves with the number of available queues. Today, switches provide 8–32 strict-priority queues [6]; however, as we have shown, BBQ's tree-based priority index structure enables scaling the priority count by 3 orders of magnitude. Thus, a switch that implements BBQ's priority index structure would benefit both SP-PIFO and PCQ. However, BBQ's design also shows that it is unnecessary to sacrifice accuracy in order to achieve scalability and speed.

4.10 Conclusion

PIFO's bold vision — a programmable packet scheduler that operates at line rate even on state-of-the-art network hardware — has been hampered by throughput, scalability, and functionality (or lack thereof) issues of existing priority queue designs. In this chapter, we used a quantitative analysis of modern switch and SmartNIC dataplanes to demonstrate that the lack of suitable priority queue candidates is, in fact, the result of *design incongruity*: a fundamental mismatch between prescribed hardware design objectives and operational requirements. Drawing inspiration from software-based priority queues with constant worst-case time-complexity, we designed and implemented BBQ, a hardware priority architecture, which, for the first time, makes it *feasible* to deploy programmable packet scheduling in modern networks.

Chapter 5

Conclusion

In this dissertation, we have argued that our contemporary models of systems are increasingly misaligned with operational realities, resulting in a problem we call **model incongruity**. Through detailed case studies of three foundational network subsystems — caches, packet processors, and schedulers — we showed that incongruities (a) subvert our ability to make intelligent design and deployment decisions, and (b) result in degraded performance, scalability bottlenecks, and security vulnerabilities. In each case, we demonstrated that modest refinements to existing models not only improve our ability to *understand* system behavior, but also provide actionable insights that ultimately enable us to *build* better systems.

In Chapter 2, we saw that the classical model of caching (which we have relied on since the 1960s) fails to account for *delayed hits*, impairing both *our* and our *algorithms'* ability to make the "right" caching decisions. Addressing this incongruity engendered two new caching algorithms: BELATEDLY, and MAD, offering significant latency improvements over the *status quo* with little-to-no overhead.

In Chapter 3, we turned to packet processing engines, which are commonly susceptible to algorithmic complexity attacks (ACAs). We showed that ACAs are fundamentally artifacts of *workload incongruity*, a transient mismatch between static designs (fixed at deployment time) and dynamic inputs (which can vary at run-time). In this context, we proposed an incongruity-aware workload model that apropos captures the reality of Internet traffic: a *superposition* of stochastic and adversarial traffic, instead of just one or the other. This formalization allowed us to develop SURGEPROTECTOR, a novel adversarial scheduling framework that *provably* defends network subsystems against ACAs.

Finally, in Chapter 4, we explored the limitations of programmable packet schedulers, using a quantitative analysis of modern network dataplanes to show that the glaring lack of deployment potential we have today stems from a deeper *design incongruity*: a fundamental disconnect between network requirements and hardware designers' objectives. We proposed BBQ, a novel priority queue based design which, for the first time, makes it *feasible* to deploy packet scheduling at line rate on modern switches and SmartNICs.

The models and systems presented in this thesis are not intended as final answers, but rather a small step toward bridging the ever-increasing gap between system complexity and model sophistication. As network line-rates and scalability demands continue to evolve at an inconceivable pace, it is only a matter of time before we must revisit these models once more; we hope that the lessons learned here will help guide similar endeavors in the future.

Looking forward, we believe there are three broad directions that are ripe for exploration.

- **A Library of System Models.** In this dissertation, we explored a small set of subsystems (network caches, packet processors, and hardware packet schedulers) that are ubiquitous in today's network deployments. Yet, these represent only a fraction of the components underpinning modern systems, which today comprise network interface cards, load-balancers, accelerators, storage devices, host and data-center fabrics, and so on. Indeed, the evidence presented in this thesis suggests that there are model incongruities lurking in each of these components, and systematically addressing them promises significant performance, scalability, and robustness improvements across the stack. Beyond *en masse* model refinement, our long-term vision is to develop and maintain a *library* of system models that designers can draw upon based on their deployment contexts. Such a resource would parallel the scientific mosaic [17] in the hard sciences (e.g., Physics), where theories are invoked selectively depending on the scale and precision required (Newton's Laws to reason about pendulums and push-carts, versus relativistic mechanics for satellite navigation).
- **Finding Incongruities Automatically.** In each of the three case studies presented in this dissertation, characterizing the underlying incongruity entailed substantial manual effort: reconstructing the best-known system model, identifying where and when it diverged from reality, and tracing back to the violated assumptions. A natural next step is to ask whether this process can be automated. Doing so will likely require a synthesis of formal methods, high-performance telemetry, and machine learning: formal tools to make model assumptions explicit, vast amounts of empirical measurements to test them against reality, and learning-based techniques to detect recurring mismatches that may point to structural flaws in the model. Such automation would not only accelerate the discovery of incongruities across a much broader range of systems (as a first step towards *fixing* them), but also democratize the methodology by making it more accessible to practitioners without deep modeling expertise.
- **Workload-Adaptive Dataplanes.** Today, the usual process of building systems entails committing upfront to a *single* design that is highly tailored to a *particular* objective (*e.g.*, memory efficiency, or worst-case performance) in the context of a *specific* workload. Unfortunately, this one-size-fits-all strategy leaves significant improvements on the table, whether in terms of performance or attack resilience (§3.1). The ability to quickly detect workload incongruity (as described above) also unlocks the possibility of quickly reacting to it: segueing between a *set of Pareto-optimal designs* in response to workload changes. In principle, this would allow system designers to optimize for multiple disjoint objectives in a workload-dependent manner (*e.g.*, "Optimize for objective X when the workload looks like Y"). There are a number of interesting research challenges that must be addressed to make this feasible, but solving them paves the way to building systems that truly work well in *any* context.

Bibliography

[1] K. Aasaraai and A. Moshovos. An efficient non-blocking data cache for soft processors. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 19–24, 2010.

[2] Yehuda Afek, Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. MCA2: Multi-Core Architecture for Mitigating Complexity Attacks. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, page 235–246, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316859. doi: 10.1145/2396556.2396603. URL https://doi.org/10.1145/2396556.2396603.

[3] Alexandru Agache, Razvan Deaconescu, and Costin Raiciu. Increasing datacenter network utilisation with GRIN. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 29–42, 2015.

[4] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[5] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *SODA*, pages 31–40, 1999.

[6] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 59–76, Santa Clara, CA, February 2020. USENIX Association. ISBN 9781939133137.

[7] Albert Gran Alcoz, Balázs Vass, Gábor Rétvári, and Laurent Vanbever. Everything matters in programmable packet scheduling. *arXiv preprint arXiv:2308.00797*, 2023.

[8] 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 SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2056-6.

[9] AMD. AMD EPYC 4th gen 9004 & 8004 series server processors – details, 2023. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html#specs.

[10] Wisnu Anggoro and John Torjo. *Boost. Asio C++ Network Programming*. Packt Publishing

Ltd, 2015.

[11] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 93–109, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.

[12] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '20, page 495–513, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405883. URL https://doi.org/10.1145/3387514.3405883.

[13] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. SurgeProtector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 723–738, New York, NY, USA, August 2022. Association for Computing Machinery.

[14] Nirav Atre, Hugo Sadok, and Justine Sherry. Bbq: A fast and scalable integer priority queue for hardware packet scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 455–475, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/atre.

[15] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, page 1–7, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332569. doi: 10.1145/2670518.2673871. URL https://doi.org/10.1145/2670518.2673871.

[16] Noa Bar-Yosef and Avishai Wool. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In Joaquim Filipe and Mohammad S. Obaidat, editors, *E-business and Telecommunications*, pages 162–174, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] Hakob Barseghyan. *The laws of scientific change*. Springer, 2015.

[18] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing hit density. In *USENIX NSDI.*, pages 1–14, 2018.

[19] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems*, pages 1–15. SIAM, 2020.

[20] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[21] Samson Belayneh and David R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, June 1996. ISSN 0163-5964. doi: 10.1145/381718.381727. URL https://doi.org/10.1145/381718.381727.

[22] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. *IEEE Transactions on Computers*, 62(5):1031–1043, 2013. doi: 10.1109/TC.2012.49.

[23] Imad Benacer, François-Raymond Boyer, and Yvon Savaria. A fast, single-instruction–multiple-data, scalable priority queue. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(10):1939–1952, 2018. doi: 10.1109/TVLSI.2018.2838044.

[24] J. C. R. Bennett and Hui Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*, volume 1 of *INFOCOM '96*, pages 120–128 vol.1, 1996.

[25] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *ACM HotNets*, pages 134–140, 2018.

[26] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[27] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *ACM POMACS*, 2(2):32, 2018.

[28] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*, pages 195–212, 2018.

[29] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000 Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2 of *INFOCOM 2000*, pages 538–547 vol.2, 2000.

[30] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[31] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, January 2002. ISSN 0146-4833. doi: 10.1145/510726.510728. URL https://doi.org/10.1145/510726.510728.

[32] 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 SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2056-6.

[33] Daniel Boteanu and José M. Fernandez. A Comprehensive Study of Queue Management as a DoS Counter-Measure. *Int. J. Inf. Secur.*, 12(5):347–382, October 2013. ISSN 1615-5262. doi: 10.1007/s10207-013-0197-6. URL https://doi.org/10.1007/s10207-013-0197-6.

[34] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477, 2020.

[35] Niv Buchbinder, Joseph Seffi Naor, et al. The design of competitive online algorithms via a primal–dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2–3):

93–263, 2009.

[36] Ben Caller. Regexploit: Dos-able regular expressions, Mar 2021. URL https://blog.doyensec.com/2021/03/11/regexploit.html.

[37] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[38] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, Hung-Jen Liao, Jhon-Jhy Liaw, Quincy Li, Chih-Yung Lin, Mu-Chi Chiang, and Shien-Yang Wu. A 7nm 256Mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-VMIN applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 206–207, 2017. doi: 10.1109/ISSCC.2017.7870333.

[39] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *USENIX ATC*, pages 379–392, 2016.

[40] Erica Chiang, Nirav Atre, and Hugo Sadok. Robust Heuristics: Attacks and Defenses for Job Size Estimation in WSJF Systems. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22 Demos and Posters)*. Association for Computing Machinery, August 2022. doi: 10.1145/3546037.3546062.

[41] Marek Chrobak, H Karloof, Tom Payne, and S Vishwnathan. New ressults on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[42] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[43] Olga Chuchuk. *Data access optimisation at CERN and in the Worldwide LHC Computing Grid (WLCG)*. Theses, Université Côte d'Azur, February 2024. URL https://theses.hal.science/tel-04620247.

[44] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016. ISSN 0026-2692. doi: https://doi.org/10.1016/j.mejo.2016.04.006. URL https://www.sciencedirect.com/science/article/pii/S002626921630026X.

[45] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association. URL https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks.

[46] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple Space Explosion: A Denial-of-Service Attack against a Software Packet Classifier. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 292–304, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369985. doi: 10.1145/3359989.3365431. URL https:

//doi.org/10.1145/3359989.3365431.

[47] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the Feasibility and Enhancement of the Tuple Space Explosion Attack against Open vSwitch, 2020.

[48] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (cxl) interconnect. *ACM Comput. Surv.*, 56(11), July 2024. ISSN 0360-0300. doi: 10.1145/3669900. URL https://doi.org/10.1145/3669900.

[49] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236027. URL https://doi.org/10.1145/3236024.3236027.

[50] Jeff Dean and R. Colin Scott. Numbers every programmer should know. https://people.eecs.berkeley.edu/ rcs/research/interactive_latency.html, 2024.

[51] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989. ISSN 0146-4833. doi: 10.1145/75247.75248. URL https://doi.org/10.1145/75247.75248.

[52] Sarang Dharmapurikar and Vern Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM '05, page 5, USA, 2005. USENIX Association.

[53] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[54] Eric Dumazet. Merge Branch 'tcp-robust-ooo'. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=1a4f14bab1868b443f0dd3c55b689a478f82e72e, 2018.

[55] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[56] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *IEEE INFOCOM*, volume 1, pages 107–116 vol.1, 1999.

[57] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 315–328, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9.

[58] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman

Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 51–66, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4.

[59] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *USENIX NSDI*, pages 51–66, 2018.

[60] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 38–46. IEEE, 2020. ISBN 978-1-72815-803-7.

[61] Xinzhe Fu and Eytan Modiano. Fundamental Limits of Volume-Based Network DoS Attacks. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), December 2019. doi: 10.1145/3366698. URL https://doi.org/10.1145/3366698.

[62] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 551–565, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4.

[63] Davy Genbrugge and Lieven Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Transactions on Computers*, 57(1):41–54, 2007.

[64] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, page 323–336, USA, 2011. USENIX Association.

[65] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 1–12, New York, NY, USA, 2012. ISBN 9781450314190. doi: 10.1145/2342356.2342358. URL https://doi.org/10.1145/2342356.2342358.

[66] K.-I. Goh and A.-L. Barabasi. Burstiness and memory in complex systems. *EPL (Europhysics Letters)*, 81(4):48002, jan 2008. doi: 10.1209/0295-5075/81/48002. URL https://doi.org/10.1209/0295-5075/81/48002.

[67] Google. Google Drive. https://support.google.com/a/answer/172541, 2021.

[68] Torbjörn Granlund. Instruction latencies and throughput for amd and intel x86 processors. *Technical report, KTH*, 2012.

[69] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*,

pages 1–14, 2015.

[70] Nathan Hauke and David Renardy. Denial of Service with a Fistful of Packets: Exploiting Algorithmic Complexity Vulnerabilities. https://www.blackhat.com/us-19/briefings/schedule/#denial-of-service-with-a-fistful-of-packets-exploiting-algorithmic-complexity-vulnerabilities-16445, 2019.

[71] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 1–14, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7.

[72] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 4 edition, 2011.

[73] Intel. Intel, Baidu drive intelligent infrastructure transformation, 2020. https://www.intel.com/content/www/us/en/newsroom/news/baidu-intelligent-infrastructure-transformation.html#gs.5vl4ru.

[74] Intel. Intel Agilex 7 FPGAs and SoCs product brief, 2023. https://www.intel.com/content/www/us/en/content-details/762901/intel-agilex-7-fpgas-and-socs-product-brief.html.

[75] Intel. Intel infrastructure processing unit (Intel IPU) platform (codename: Oak Springs Canyon), 2023. https://www.intel.com/content/www/us/en/products/platforms/details/oak-springs-canyon.html.

[76] Intel. FPGA Design Software – Intel Quartus Prime, 2023. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[77] Intel. Intel Stratix 10 MX 2100 FPGA, 2023. https://ark.intel.com/content/www/us/en/ark/products/210297/intel-stratix-10-mx-2100-fpga.html.

[78] Aggelos Ioannou and Manolis G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, April 2007. ISSN 1063-6692.

[79] SNIA IOTTA. Microsoft production server traces, 2011.

[80] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*, pages 78–89, 2016.

[81] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[82] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *IEEE ICDCS*, pages 254–261, 2000.

[83] Shudong Jin and Azer Bestavros. Greedydual⋆ web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24 (2):174–183, 2001.

[84] Suraiya Khan and Issa Traore. Queue-based Analysis of DoS Attacks. In *Proceedings from*

the Sixth Annual IEEE SMC Information Assurance Workshop, pages 266–273, 2005. doi: 10.1109/IAW.2005.1495962.

[85] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic External Memory for Switch Data Planes. In *ACM HotNets*, pages 1–7, 2018.

[86] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *ACM IMC*, pages 190–201, 2009.

[87] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ACM/IEEE ISCA*, page 81–87, Washington, DC, USA, 1981. IEEE Computer Society Press.

[88] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213874. URL https://doi.org/10.1145/3213846.3213874.

[89] Ka-Cheong Leung, Victor O. K. Li, and Daiqin Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):522–535, April 2007. ISSN 1045-9219. doi: 10.1109/TPDS.2007.1011. URL https://doi.org/10.1109/TPDS.2007.1011.

[90] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*, pages 1–15, 2015.

[91] Shang Li, Dhiraj Reddy, and Bruce Jacob. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 341–353, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364751. doi: 10.1145/3240302.3240315. URL https://doi.org/10.1145/3240302.3240315.

[92] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. Popularity-driven content caching. In *IEEE INFOCOM*, pages 1–9, 2016.

[93] Yin Li, Chuang Lin, Fengyuan Ren, and Yifeng Geng. H-PFSP: Efficient Hybrid Parallel PFSP Protected Scheduling for MapReduce System. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1099–1106, 2013. doi: 10.1109/TrustCom.2013.133.

[94] Shuang Liang, Ke Chen, Song Jiang, and Xiaodong Zhang. Cost-aware caching algorithms for distributed storage servers. In Andrzej Pelc, editor, *Distributed Computing*, pages 373–387, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-75142-7_29.

[95] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 243–259. USENIX Association, November 2020. ISBN 978-1-939133-19-9.

[96] Yang Liu, Yukun Zeng, and Xuefeng Piao. High-Responsive Scheduling with MapReduce

Performance Prediction on Hadoop YARN. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 238–247, 2016. doi: 10.1109/RTCSA.2016.51.

[97] Peter Manohar and Jalani Williams. Lower Bounds for Caching with Delayed Hits. arXiv:2006.00376 [cs.DS], 2020.

[98] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX FAST*, pages 115–130, 2003.

[99] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 753–766, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9420-8.

[100] Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Recursively cautious congestion control. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 373–385, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/mittal.

[101] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 501–521, 2016. ISBN 978-1-931971-29-4.

[102] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[103] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for CDNM-based live video delivery. In *ACM SIGCOMM*, pages 311–324, 2015.

[104] Ali Munir, Ghufran Baig, Syed Mohammad Irteza, Ihsan Ayyub Qazi, Alex X Liu, and Fahad Rafique Dogar. Pase: synthesizing existing transport strategies for near-optimal data center transport. *IEEE/ACM Transactions on Networking*, 25(1):320–334, 2016.

[105] A. Musa, Y. Sato, T. Soga, R. Egawa, H. Takizawa, K. Okabe, and H. Kobayashi. Effects of mshr and prefetch mechanisms on an on-chip cache of the vector architecture. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 335–342, 2008.

[106] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *USENIX NSDI*, March 2016.

[107] Nvidia. ConnectX-7 400G Adapters: Smart, accelerated networking for modern data center infrastructures, 2023. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[108] Nvidia. Nvidia spectrum sn4000 series switches, 2023. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/br-sn4000-series.pdf.

[109] Gurobi Optimization. Inc.,"gurobi optimizer reference manual," 2015, 2014.

[110] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[111] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 372–385, New York, NY, USA, 2018. ISBN 9781450355674. doi: 10.1145/3230543.3230573. URL https://doi.org/10.1145/3230543.3230573.

[112] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. ISBN 9781450349468. doi: 10.1145/3133956.3134073. URL https://doi.org/10.1145/3133956.3134073.

[113] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *USENIX NSDI*, pages 117–130, 2015.

[114] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[115] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ACM/IEEE ISCA*, pages 167–178, 2006.

[116] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 475–488, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6.

[117] Prabhakar Raghavan and Clark D Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[118] Simona Ramanauskaitė and Antanas Čenys. Composite DoS Attack Model/Jungtinis DoS Atakų Modelis. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, 4(1):20–26, 2012.

[119] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[120] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[121] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery. ISBN

978-1-4503-8438-4.

[122] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 1005–1025, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2.

[123] Hugo Sadok, Aurojit Panda, and Justine Sherry. Of apples and oranges: Fair comparisons in heterogenous systems evaluation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 1–8, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704154.

[124] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 17–32, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/saeed.

[125] Sartaj Sahni. Double-ended priority queues. In *Handbook of Data Structures and Applications*, 2004.

[126] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[127] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/sharma.

[128] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 685–699, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.

[129] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 225–235, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238159. URL https://doi.org/10.1145/3238147.3238159.

[130] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. Improving the Resilience of an IDS against Performance Throttling Attacks. In Angelos D. Keromytis and Roberto Di Pietro, editors, *Security and Privacy in Communication Networks*, pages 167–184, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36883-7.

[131] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 367–379, New York, NY, USA, 2019. Association for Com-

puting Machinery. ISBN 978-1-4503-5956-6.

[132] James Edward Sicolo. A multiported nonblocking cache for a superscalar uniprocessor. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[133] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[134] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCP's Burstiness with Flowlet Switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*. Citeseer, 2004.

[135] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. doi: 10.1145/2934872.2934899. URL https://doi.org/10.1145/2934872.2934899.

[136] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 89–98, 2006. doi: 10.1109/ACSAC.2006.17.

[137] Snort Project. SNORT Users Manual. https://www.snort.org/documents/snort-users-manual, 2020.

[138] Zhenyu Song, Daniel S. Berger, and Lloyd Wyatt LI, Kai. Learning relaxed belady for content distribution network caching. In *USENIX NSDI*, 2020.

[139] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 135–146, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131356. doi: 10.1145/316188.316216. URL https://doi.org/10.1145/316188.316216.

[140] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 33–46, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2.

[141] Synopsys. Design Compiler, 2023. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[142] Edward S Tam. *Improving cache performance via active management*. PhD thesis, University of Michigan, 1999.

[143] Juha-Matti Tilli. CVE-2018-5390: Linux Kernel TCP Reassembly Algorithm Lets Remote Users Consume Excessive CPU Resources on the Target System. https://ubuntu.com/security/cve-2018-5390, 2018.

[144] Eric Torng. A unified analysis of paging and caching. *Annual Symposium on Foundations*

of Computer Science - Proceedings, 08 1995.

[145] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *ACM/IEEE MICRO*, pages 409–422, 2006.

[146] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper. In *2017 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD, pages 992–998, 2017. doi: 10.1109/ICCAD.2017.8203889.

[147] Colby Walsworth, Emile Aben, K Claffy, and D Andersen. The caida anonymized 2019 internet traces, 2019.

[148] Hao Wang and Bill Lin. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, 2013.

[149] Jia Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–46, 1999.

[150] Justin Wang, Benjamin Berg, Daniel S Berger, and Siddhartha Sen. Maximizing page-level cache hit ratios in largeweb services. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):91–92, 2019.

[151] Wei Wang, Ben Liang, and Baochun Li. Low Complexity Multi-Resource Fair Queueing with Bounded Delay. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1914–1922, 2014. doi: 10.1109/INFOCOM.2014.6848131.

[152] Yang Wang, Chuang Lin, Quan-Lin Li, and Yuguang Fang. A Queueing Analysis for the Denial of Service (DoS) Attacks in Computer Networks. *Comput. Netw.*, 51(12):3564–3573, August 2007. ISSN 1389-1286. doi: 10.1016/j.comnet.2007.02.011. URL https://doi.org/10.1016/j.comnet.2007.02.011.

[153] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI*, pages 307–320, 2006.

[154] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions Electronic Computers*, 14(2):270–271, 1965.

[155] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm CMOS platform technology featuring 4th generation FinFET transistors with a 0.027um2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2.6.1–2.6.4, 2016. doi: 10.1109/IEDM.2016.7838333.

[156] Valentin Wüstholz, Oswaldo Olivo, Marijn J. Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis*

of Systems - Volume 10206, page 3–20, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662545799. doi: 10.1007/978-3-662-54580-5_1. URL https://doi.org/10.1007/978-3-662-54580-5_1.

[157] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-limiting network architecture. *ACM SIGCOMM Computer Communication Review*, 35(4):241–252, 2005.

[158] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. BMW tree: Large-scale, high-throughput and modular PIFO implementation using balanced multi-way sorting tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 208–219, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365.

[159] Neal E Young. On-line caching as cache size varies. In *ACM SODA*, 1991.

[160] Neal E Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[161] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 179–193, New York, NY, USA, August 2021. Association for Computing Machinery. ISBN 978-1-4503-8383-7.

[162] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[163] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.