Rearchitecting the End Host Network for the Terabit Per Second Era

by

Annus Zulfiqar

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2026

Annus Zulfiqar

zulfiqaa@umich.edu

ORCID iD: 0000-0003-0612-4939

To my parents, Zarqa and Zulfiqar Ali,
and my brother, Danish.

ACKNOWLEDGEMENTS

A PhD is five years on paper. In lived experience, it is something harder to name—a long education in persistence, in coping with uncertainty, in learning to keep going when the path disappears underfoot. And yet, it is only when sitting down to write the dissertation that one finally makes sense of the journey in retrospect. While this thesis presents a cohesive narrative, truth is that published research is highly processed—much of the uncertainty, false starts, (many) moments of wanting to quit, and invisible labor disappears from the final account. This section is a small attempt to acknowledge all those people behind this polished narrative. To everyone who supported me along the way: I am deeply grateful, and to those that I forget to mention, I apologize in advance but your contributions I sincerely acknowledge nonetheless.

First and foremost, I am grateful to my advisor, *Muhammad Shahbaz*. Your mentorship fundamentally shaped how I think about research and instilled in me the most essential meta-skills for a researcher—clarity of thinking, work ethic, and persistence. Central to this clarity is the art of asking powerful questions—ones that generate new knowledge. Deceptively simple questions like "why does the Megaflow cache still miss?," when asked relentlessly, reveal inherent, often unspoken assumptions embedded in how a system is designed and understood. Questioning novelty with "why didn't anybody else do it if it was this simple?" to understand the constraints that once made an idea impractical, to think historically, and recognizing when new opportunities emerge. And ambitious questions like "what can we do to make this work?" that drive the search for the principles, abstractions, and architectures needed to realize an idea—and to commit the time and effort to bring it to life. For teaching me to ask powerful questions, think clearly, pursue precision meticulously, communicate effectively, and chase ambitious ideas with intellectual rigor, I am deeply grateful.

Equally formative, in their own ways, were *Gianni Antichi* and *Ben Pfaff* who have been collaborators on every chapter of this dissertation and needless to suggest, this thesis would not exist without them. Gianni has an uncanny ability to step into the skeptical reviewer's shoes precisely when it matters most—his sharp, honest feedback has been instrumental in shaping the narrative of nearly every paper I have published. And Ben, despite being an industry giant in every sense of the word, has probably donated his time worth hundreds of thousands of dollars to me at this point, being actively involved in every project we've completed over the past five years. His guidance has been a quiet constant throughout this entire journey. Thank you Gianni and Ben!

I am grateful to my committee members—*Mosharaf Chowdhury*, *Ang Chen*, and *Jiasi Chen*—whose feedback sharpened both the technical contributions and the broader narrative of this dissertation. Thank you for your time and service on my committee.

I also want to thank my mentors on projects beyond this thesis—*Kunle Olukotun*, *Arpit Gupta*, *Walter Willinger*, *Shir Landau Feibish*, *Ori Rottenstreich*, *Josep Torrellas*, *William Tu*, *Luigi Nardi*, and *Nadeen Gebara*—for their insights, perspective, and encouragement at pivotal moments. Their influence quietly broadened the intellectual scope of my work throughout this PhD journey.

I am fortunate to have shared this journey with some incredible lab mates who I also collaborated with: *Marilyn Rego*, *Ali Imran*, *Advay Singh*, *Ertza Warraich*, *Bilal Saleem*, *Sripath Mishra*, *Murayyiam Parvez*, *Jingqi Huang*, *Chunao Liu*, *Enkeleda Bardhi*, and *Dominic Damoah*. It was your company, and of course, the group lunches, (plenty of) multi-nighter deadlines, whiteboard sessions, and everyday camaraderie that made the NextGArch lab a place of both rigor and belonging. Thank you everyone.

To my collaborators across Purdue, Stanford, and UIUC who were in the trenches alongside me—*Tushar Swamy*, *Alex Rucker*, *Venkat Kunaparaju*, *(Makis) Gerasimos Gerogiannis*, *Charles Block*, *Sylee Beltiukov*, *Dimitrios Merkouriadis*, *Filippos Tofalos*—thank you for your dedication, creativity, and countless hours spent building and evaluating complex systems. Research is demanding and working alongside you made it deeply rewarding.

I am equally grateful to friends across other labs both at UMich and back at Purdue—*Usman Jaffri*, *Chandan Bothra*, *Muhammad Ibrahim*, *Noman Abbasi*, *Muhammad Taha*, *Umakant Kulkarni*, *Zoha Khan*, *Gloria*, *Azam Ikram*, *Athena Terzoglu*, *Hasan Amin*, and *Arsalan Khan*—for the conversations, laughter, and perspective that kept me grounded.

My graduate program managers both at Purdue and later at UMich were extremely helpful in navigating the departmental logistics and formalities at every step. *Lacey Siefers* and *Jasmin Stubblefield*, I am thankful to both of you.

I would also like to acknowledge ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and the National Science Foundation awards CAREER-2338034 and CNS-2211381 for funding the work presented in this dissertation. Support also came in part from VMware by Broadcom, a Google Research Scholar Award, and research gifts from Intel and Google.

The spark in my intellectual curiosity was lit long before graduate school. I must thank my undergraduate professors, *Dr. Faisal Shafait* and *Dr. Muhammad Shahzad*, for inspiring me to pursue academic research and graduate school. I am grateful to my seniors and mentors from my undergrad days at NUST—*Shahrukh Qasim*, *Khurram Javed*, *Tooba Imtiaz*, *Talha Paracha*, and *Ferjad Naeem*—who guided me early on as a fledgling researcher. I am also thankful to close friends—*Hamza Mahmood*, *Ali Zeb*, *Muhammad Umar*, and *Ussama Zahid*—for their encouragement and belief.

Thanks to my former colleagues, *Asad Tariq* and *Hamza Dogar*, for their friendship and support. I am grateful to my managers and mentors, *Dr. Mansoor Shaukat* and *Dr. Shoab*, for shaping my early professional growth. Beyond the work itself, *Ashish Kashinath* and *Muhammad Haris Mughees* made my time at VMware in California something I look back on with genuine fondness—their friendship broadened my perspective and strengthened my confidence in bridging research and practice.

To my roommates—*Athul*, *Kalyan*, and *Cynthia*—thank you for making home a place of warmth and humor amid deadlines. To my dear and closest friend, *Rai Muhammad Danish*, thank you for unwavering friendship across time and distance.

A special thank you to *Sunny Sun*. Your support and companionship along with our shared fascination with penguins (*Soni*!), flying squirrels, raccoons, and cats gave me some of my most cherished memories and brought balance and warmth to my time at UMichigan.

Finally, and most importantly, I thank my family. My mother, *Zarqa Zulfiqar*, and my father, *Zulfiqar Ali*, have been my foundation—their belief in me unwavering, their love unconditional, and their support without limit. My brother, *Danish Zulfiqar*, has been a constant source of strength and pride. And finally, my cousin *Bilal Ahmed Bhatti*, who was the first in our family to pursue PhD in engineering, showed me that this path was possible before I dared to imagine walking it myself.

This dissertation stands on the shoulders of all of you. Thank you!

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

LIST OF APPENDICES

BIBLIOGRAPHIC NOTES

The material presented in Chapter 2 is a joint work with Ben Pfaff, Gianni Antichi, William Tu, and Muhammad Shahbaz, and has been published at ACM SIGCOMM CCR [184].

The material presented in Chapter 3 is a joint work with Venkat Kunaparaju, Ali Imran, Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. It has been published as a full paper at ACM ASPLOS 2025 [182]. Earlier versions appeared as posters at Hot Chips 2024 [180] and NSDI 2025 [181]. The work was also selected as a Google Summer of Code (GSoC) 2025 project under the P4 Language Consortium organization [45].

The material presented in Chapter 4 is a joint work with Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. A preliminary version was presented as a poster at ACM SIGCOMM 2025 [183].

ABSTRACT

Modern cloud data centers operate at massive scale, interconnecting millions of servers to support diverse, multi-tenant workloads. These environments require sophisticated, stateful network policies—such as isolation, load balancing, and access control—that must evolve rapidly while sustaining line-rate performance. Rather than embedding this complexity in the physical fabric, cloud providers implement it in software at the end hosts using virtual switches (vSwitches). By distributing policy enforcement across servers, vSwitches allow network functionality to scale independently of the underlying hardware. Collectively, these distributed vSwitches form the logical data center network, called the *end host network*.

Because every packet entering or leaving this network traverses a vSwitch, its efficiency directly determines the performance and scalability of the entire system. However, since their inception, modern vSwitches have been fundamentally CPU-centric systems, reflecting architectural assumptions from an earlier era. They were built around general-purpose CPUs, relying on aggressive caching to avoid expensive multi-stage packet processing, and assuming that continued improvements in CPU performance would accommodate rising link speeds and frequent policy updates.

However, with the slowdown of Moore's law and growth of link rates to 400 G per link—pushing per-host aggregate traffic volume toward 1 Tbps—these assumptions no longer hold. As a result, the cost of handling cache misses and cache evictions under policy updates have become the dominant performance bottlenecks, revealing fundamental scalability limits in the prevailing CPU-oriented vSwitch architecture.

Moreover, while SmartNICs are now pervasive in data centers, current deployments largely treat them as accelerators for the existing vSwitch architecture rather than as the primary execution target. Consequently, the fundamental organization of the vSwitch has remained unchanged, even though the hardware landscape has shifted.

In this dissertation, we revisit the virtual switch (vSwitch) from first principles and present GIGAFLOW and KAIRO, two systems that together form a clean-slate vSwitch architecture for the emerging terabit-per-second era in a SmartNIC-native environment.

GIGAFLOW captures a new form of locality—*pipeline-aware locality*—in programmable vSwitch pipelines, inherent in the structure induced by ordered policy stages (e.g., L2, L3, ACL). Rather than caching entire flows, Gigaflow constructs cache entries over shared pipeline segments (sub-traversals) reused across many flows. By aligning caching with the structure of vSwitch packet pipelines, it increases effective rule space coverage by 450× within limited SmartNIC memory, reducing cache misses by up to 90%, while sustaining line-rate performance.

Kairo then rethinks cache eviction as an instance of the incremental view maintenance (IVM) problem. It represents policy tables as incremental dataflow *circuits* and propagates rule updates to identify and evict only affected cache entries. As a result, update cost scales with the size of the change rather than the number of cache entries, delivering 18× faster cache evictions on average and up to 130× lower stale cache processing.

Together, Gigaflow and Kairo redesign both the fast path and update path of the modern vSwitch as a SmartNIC-native architecture, eliminating reliance on faster CPUs to rearchitect the end host network for the terabit-per-second era.

CHAPTER 1

Introduction

“

The fast path/slow path split is often an alluring mirage. It’s easy enough to design a system with a well defined fast path and slow path. However, time and time again we see the fast path/slow path split fails to deliver results in real deployment.

Tom Herbert, XDPnet (Medium [49])

Modern public clouds comprise millions of servers supporting multi-tenant workloads such as serverless computing, real-time analytics, video streaming, and large-scale distributed ML. These workloads require complex, stateful network policies—including private virtual networks, L4 load balancing, custom address spaces, ACLs and security groups, and QoS enforcement—that traditional hardware cannot support at scale. These policies enforce isolation, steer traffic across dynamic service chains, maintain per-tenant address spaces, and react to failures or scaling events in real time, all while sustaining line-rate packet processing.

To provide this flexibility without complicating the physical network fabric, cloud providers move network functionality to the end hosts in the form of virtual switches (vSwitches) [121, 73, 36, 158]. vSwitches at the hosts allow the underlying fabric to remain simple and throughput-optimized, while network functionality scales independently with compute. Collectively, these distributed vSwitches form the logical data center network, which we refer to as the *end host network*, as shown in Figure 1.1.

Over the past decade, vSwitches have evolved into programmable, multi-table packet-pipeline processors capable of expressing intricate control flows across multiple policy stages [121, 149, 126, 105]. A packet entering the vSwitch may traverse a sequence of logical tables implementing L2 switching, L3 forwarding, ACL evaluation, encapsulation, metering, and other stateful transformations [121, 89, 148, 11, 13]. This flexibility is essential for modern virtualization platforms [73, 153], yet it introduces significant computational overhead. Evaluating the multi-table pipeline on every packet is prohibitively expensive in software, par-

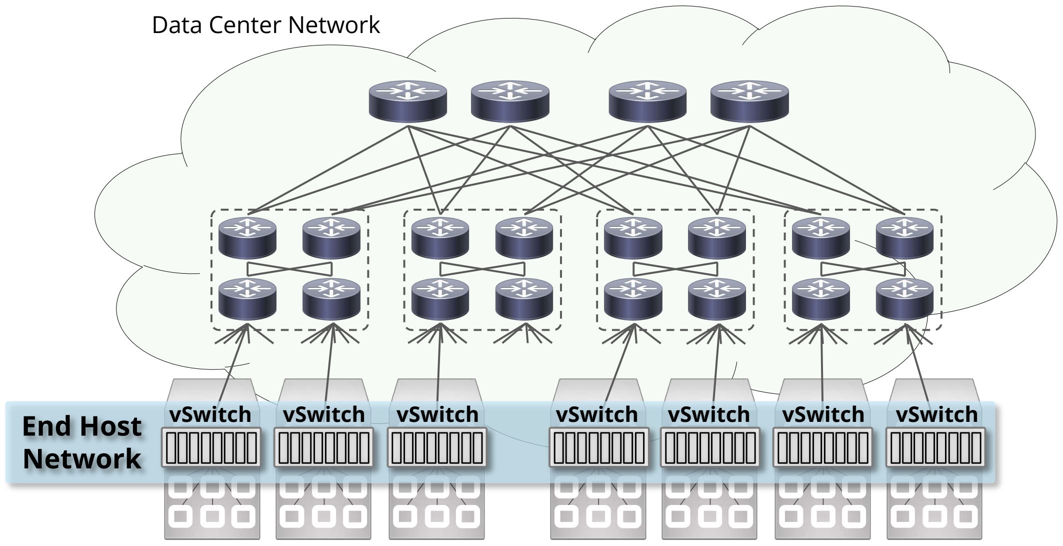


Figure 1.1: The end host network encompasses virtual switches (vSwitches) running at the end hosts throughout the data center network.

ticularly at high link rates [121, 184, 126]. As a result, vSwitch throughput is fundamentally bounded by CPU processing capacity and memory subsystem behavior [129, 48].

To reconcile flexibility with performance, vSwitch architectures adopted a split design consisting of a slow path and a fast path [121, 149, 184]. The slow path evaluates the full multi-table pipeline and installs derived cache entries in the fast path. The fast path, implemented as a single-lookup cache, allows subsequent packets matching a cached rule to bypass full pipeline execution [121, 36, 158]. This design was initially realized through exact-match Microflow caching, and later generalized to wildcard Megaflow caching to improve hit rates by aggregating flows that share header prefixes [121, 149, 126]. Subsequent work further optimized lookup performance within the fast path by replacing Tuple Space Search (TSS) classifiers with learned index models such as RQ-RMI [125, 126]. Together, these optimizations established a cache-centric vSwitch architecture in which overall performance depends critically on fast-path hit rate and handling cache misses as rare exceptions.

However, this traditional vSwitch architecture rested on two foundational assumptions:

> **Assumption #1:** *Multi-table packet processing on general-purpose CPUs is prohibitively expensive to perform at line rate, necessitating a single-table fast-path cache that collapses multi-table traversals into one lookup [182, 126, 184].*
> **Assumption #2:** *CPU performance would continue to scale with Moore's law, enabling a single core to sustain increasing link rates, cache misses, and cache eviction overhead [3, 48].*

Unfortunately, neither assumption holds in modern cloud environments anymore.

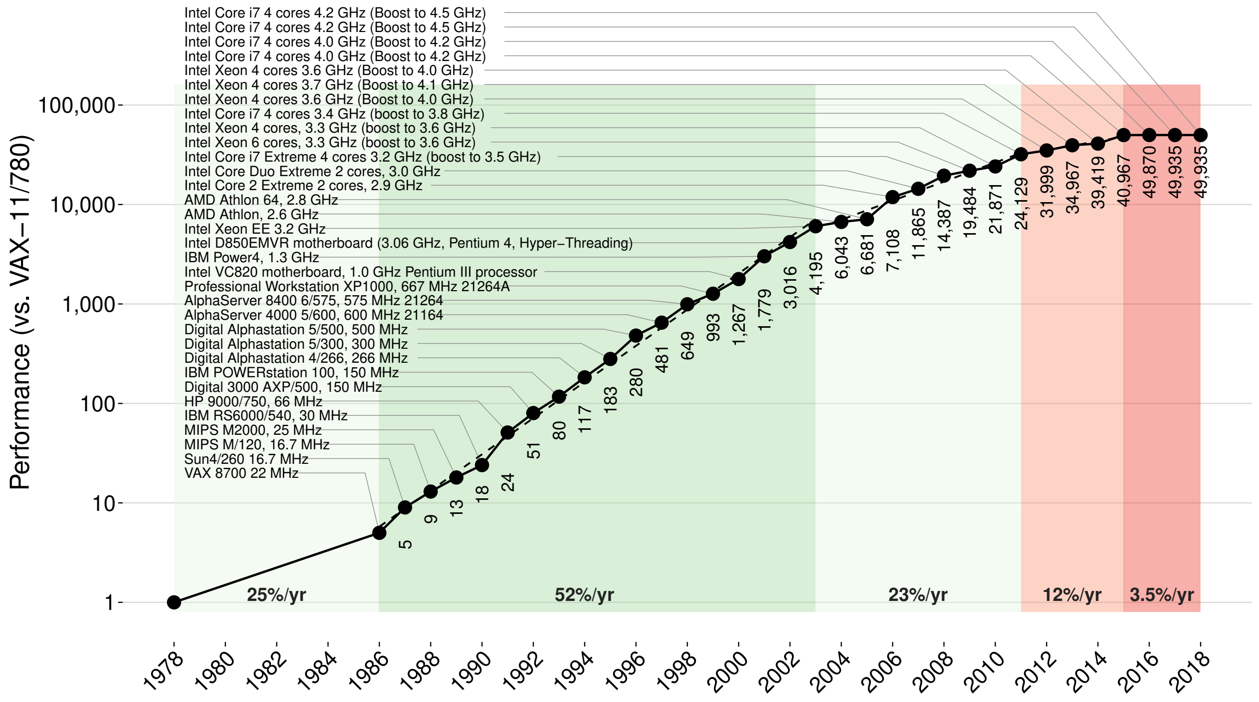


Figure 1.2: Moore's law has slowed down significantly since 2010s [48, 3, 161, 167], leading to stagnated CPU performance gains over the last 15 years.

While single-thread CPU performance improvements have slowed substantially [3] (Figure 1.2), network link rates have grown drastically from 10 Gbps to 400 Gbps per interface (Figure 1.3), pushing per-host aggregate throughput toward 1 Tbps. A single cache miss that costs tens of microseconds in software now represents a two-orders-of-magnitude penalty compared to hardware forwarding latencies of hundreds of nanoseconds. As a result, even modest miss rates translate directly into throughput collapse at high link speeds. In practice, pure software vSwitch datapaths remain constrained to roughly 10 Gbps per core under realistic workloads [121, 149, 126]. The cost of handling cache misses has therefore become a first-order scalability bottleneck.

To mitigate CPU bottlenecks, industry has shifted toward SmartNIC and DPU-based acceleration, as evident through recent products from major players, including Amazon Nitro [4], Nvidia Bluefield [99, 100], AMD Pensando DPU [7, 5], and Intel IPU [57]. These devices integrate programmable packet-processing pipelines capable of executing match-action logic at line rate. In principle, SmartNICs can offload the fast path entirely, eliminating CPU involvement for cache hits. In practice, however, hardware wildcard caches are severely capacity constrained, typically supporting only tens of thousands of entries due to power and silicon area limitations [85, 157]. Consequently, only a subset of traffic can be offloaded, and the majority of flows still incur expensive CPU processing on cache misses [157, 46, 133, 33]. Thus, overall end host network performance remains tightly coupled to cache effectiveness.

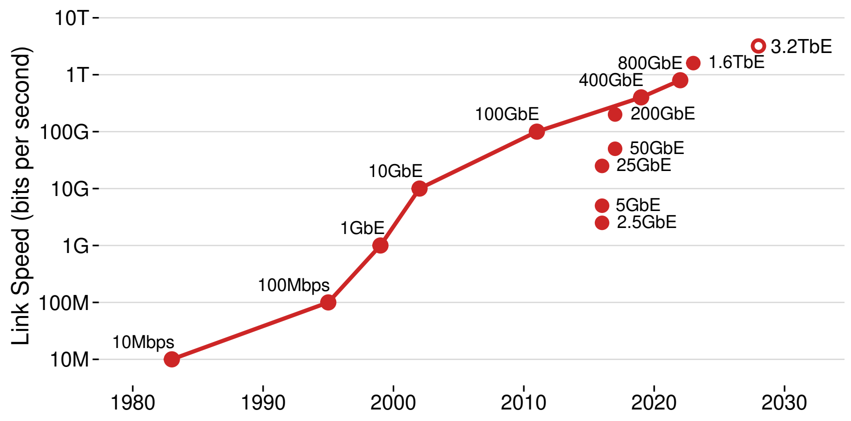


Figure 1.3: Ethernet link speeds since 1980s: 400 Gbps interfaces are now available on dual-port SmartNICs and multiple such NICs are installed on each end host.

At the same time, a second scalability bottleneck has emerged in the update path. vSwitch pipelines store wildcarded rules with priorities, so even small policy updates can invalidate arbitrary subsets of cached entries [66, 121]. Such updates arise frequently in practice due to traffic engineering, monitoring, load balancing, and security events [15, 150, 80, 91, 21].

Modern vSwitches address invalidation by re-evaluating cached entries from scratch through the slow path pipeline—a mechanism called cache *revalidating*—to detect and evict stale entries in $O(E)$ time, where E is the number of cache entries [121, 184, 66]. During this revalidation phase, incoming traffic may continue to match invalidated cache entries, leading to temporary policy inconsistencies and stale-cache processing. As link speeds rise and cache sizes grow, the cost of revalidation increases proportionally: a one-second eviction lag that was negligible at 10 Gbps results in 40× more stale-cache processing at 400 Gbps. To bound this overhead, production systems cap cache sizes and dynamically shrink them when revalidation exceeds latency thresholds [66]. This trade-off sacrifices hit rate and throughput in order to preserve processing correctness.

These trends reveal a fundamental tension in contemporary vSwitch design. End-to-end performance hinges on high cache hit rates and large effective cache capacity while processing correctness under frequent rule updates requires fast, scalable cache eviction. Yet prevailing architectures couple datapath performance and update cost to cache size and CPU scalability. In the emerging terabit-per-second era, this coupling has become untenable.

The path forward requires rejecting the premise that cache performance (e.g., hit rate) and cache eviction cost are immutable trade-offs to be managed rather than problems to be solved. What is needed is an architecture that treats SmartNIC hardware capabilities not merely as an offload for a CPU-centric vSwitch design, but as the primary forwarding substrate—one whose caching philosophy and update semantics must be rethought from the ground up based on the realities of modern cloud networks.

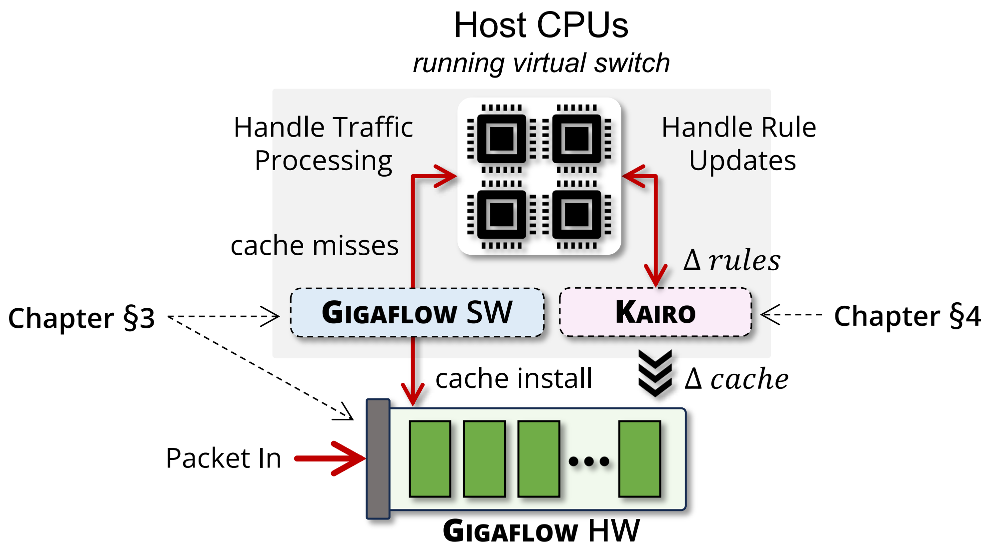


Figure 1.4: Contributions of this dissertation: Gigaflow (§3) introduces a high hit rate SmartNIC-native cache architecture and Kairo (§4) proposes a fast cache eviction scheme based on incremental view maintenance.

Escaping this tension requires solving both problems at their root. We aim to rearchitect the vSwitch fast path and update path for a SmartNIC-native, terabit-per-second era. We distill our argument into the following thesis:

> ❏ **Thesis Statement**
> *The performance ceiling of modern vSwitches is not a fundamental limit but an artifact of two outdated assumptions: that hardware caches must collapse multi-table pipelines into a single lookup, and that cache correctness requires full revalidation on every rule update. Discarding both assumptions—by exploiting pipeline-aware locality to build a multi-table SmartNIC cache, and by using incremental computation to make eviction fast and precise—breaks this ceiling to unlock a new tier of vSwitch performance.*

We realize this vision through two complementary systems, Gigaflow and Kairo (Figure 1.4), each excising one flawed assumption at its root: the former exploits vSwitch pipeline-aware locality to build a multi-table SmartNIC cache that captures bigger rule spaces within limited TCAM memory, and the latter replaces cache revalidation with incremental view maintenance to make cache eviction fast and precise.

1.1 Summary of Contributions

As shown in Figure 1.4, Gigaflow exploits pipeline-aware locality to build a multi-table hardware cache on SmartNICs, while Kairo replaces brute-force cache revalidation with incremental view maintenance (IVM) to make eviction fast, correct, and scalable. Together, they form a cohesive redesign of the end host network's core forwarding substrate for the terabit-per-second era.

1.1.1 A Cache Architecture for the Terabit Per Second Era

Limitations of Prior Work. Traditional vSwitch fast paths collapse the entire multi-table pipeline into a single wildcard lookup using Megaflow caching [121, 149]. While effective in software, this design maps poorly onto SmartNIC hardware, where TCAM capacity is severely constrained—typically supporting only tens of thousands of wildcard entries due to power and silicon area limitations [85, 157]. As a result, hardware wildcard caches suffer from high miss rates, forcing the majority of traffic back to the CPU for processing [182, 157]. Furthermore, existing caching schemes exploit only temporal and spatial locality derived from traffic patterns, leaving the rich structural information encoded in the vSwitch pipeline entirely unexploited [182, 126].

Key Insights. GIGAFLOW rethinks fast-path caching in modern vSwitches for programmable SmartNICs as the execution target. Our key insight is that vSwitch pipelines exhibit *pipeline-aware locality* beyond traditional temporal and spatial locality [182]. Each packet follows a unique sequence of table lookups, or *traversal*, and different traversals often share common subsequences across policy stages [182, 149]. Rather than caching entire traversals as monolithic wildcard entries, GIGAFLOW decomposes them into reusable sub-traversals and distributes these segments across multiple hardware cache tables. By exploiting disjoint header matching and composing sub-traversals through Longest Traversal Matching (LTM), GIGAFLOW expands effective rule-space coverage through cross-product combinations within the SmartNIC pipeline.

Results. GIGAFLOW's design increases cache hit rate by up to 51% (25% on average), reduces misses by up to 90%, and captures up to 450× more rule space within the same hardware memory budget, all while operating at line rate [182, 42].

Broader Impact. GIGAFLOW has been published at ASPLOS 2025 [182] and has received traction in both open-source and industry settings. Our reference implementation for Open vSwitch (OVS) is publicly available [115, 42], lowering the barrier for adoption by researchers and practitioners working on future SmartNIC cache designs. GIGAFLOW has been presented to networking research teams at Intel and IBM, showcased at Hot Chips '24 [180] and the Google Networking Summit '25, and demoed at the P4 Workshop '25 [113, 114]. A SmartNIC offload implementation targeting the AMD Xilinx Alveo U250 datacenter accelerator [172] was also completed under Google Summer of Code '25 [45], in collaboration with the P4 Language Consortium. Ongoing discussions with a major chip manufacturer are exploring the integration of Gigaflow cache into upcoming SmartNIC products, reflecting its potential to influence next-generation hardware design.

1.1.2 A Cache Eviction Scheme for the Terabit Per Second Era

Limitations of Prior Work. Modern vSwitches rely on iterative revalidation to evict stale cache entries following rule updates [121, 66]. This approach re-executes each cached entry's traversal through the slow-path pipeline to detect staleness, incurring $O(E)$ cost proportional to the number of cache entries E [121, 66]. As link rates grow and cache sizes scale, the eviction window widens proportionally: a revalidation lag that was negligible at 10 Gbps results in orders of magnitude more stale-cache processing at 400 Gbps and beyond [182, 183]. To bound this overhead, production systems cap cache sizes and dynamically shrink them when revalidation exceeds latency thresholds [66]—a trade-off that sacrifices hit rate and throughput to preserve correctness.

Key Insights. We formulate vSwitch cache eviction as an instance of Incremental View Maintenance (IVM), a technique widely used in database systems to maintain materialized views under streaming updates [86, 2, 52, 54, 53, 72]. We model vSwitch rule tables as a database and the fast-path cache as a materialized view of the forwarding pipeline. Instead of iterating over all E cache entries upon each rule update, Kairo computes only the incremental effect ΔC induced by a rule change ΔR, where typically $\Delta R \ll E$. To realize this abstraction, Kairo expresses policy tables as incremental dataflow circuits using DBSP [20, 34, 41]. Rule updates and slow-path packets are treated as streams that propagate through these circuits, identifying and evicting only the affected cache entries. We further introduce early-exit gates and propagation timers to bound time-to-eviction (TTE) while ensuring eventual consistency.

Results. When integrated into Open vSwitch (OVS) [121, 43], Kairo delivers 18× faster updates on average, reduces stale-cache processing by 35.5× (up to 130×), and scales to millions of entries without degrading end-to-end performance.

Broader Impact. When we shared our results with researchers who have worked extensively on both virtual switching and incremental view maintenance, they were surprised to see that vSwitch rule tables and fast-path caches are amenable to an IVM framing, and that this connection yields such significant reductions in eviction cost and stale-cache processing. We hope this work opens a new line of thinking at the intersection of database systems and network data planes: the machinery developed for maintaining materialized views under streaming updates maps naturally onto the problem of cache correctness in programmable packet pipelines. As vSwitch deployments scale toward terabit-per-second link rates and cache sizes grow to millions of entries, we believe incremental computation will become an essential primitive for sustaining correctness without sacrificing performance.

The remainder of this dissertation is organized as follows. We provide a background on the evolution of vSwitch architecture in §2, covering the slow path and fast path design, traditional caching schemes Microflow and Megaflow, and the shift toward SmartNIC-based offloads. Then, we present GIGAFLOW in §3 in depth, including its design, the pipeline-aware locality insight, the Longest Traversal Matching mechanism, and a thorough evaluation across real-world vSwitch pipelines. Finally, we present KAIRO in §4, detailing its formulation of cache eviction as incremental view maintenance, the DBSP-based circuit design, early-exit gates, and its end-to-end performance results when integrated into Open vSwitch (OVS).

CHAPTER 2

Understanding the End Host Network

“

Since datapath flow table is full, lots of packets . . . keep going to (the slow path) . . . handlers are even losing packets since they can not keep up (with cache misses) . . . (consequently) OVS may not send LACP packets in time . . . (or) OVS itself might shut one of the interfaces down . . . The main problem is the number of datapath flows that slows down the whole system and makes OVS to delay or drop LACP packets.

Ilya Maximets, Red Hat (OVS Mailing List [87])

Cloud data centers enforce network policy directly at the servers rather than in the physical fabric using virtual switches (vSwitch) that processes all traffic to and from its hosted workloads, implementing isolation, load balancing, and access control in software. The collection of vSwitches deployed across a data center's servers forms the *end host network*. This chapter examines the internal architecture of the vSwitch, their slow path/fast path bifurcation and their roles in handling traffic that demands stateful, rule-based processing.

2.1 Introduction

When we think of Software-Defined Networking (SDN), we typically think of a data plane managed by a logically centralized control plane. This separation is pervasive throughout our modern computing landscape, in areas ranging from data centers [154, 10, 106, 107, 146], to wide-area [65], to 4G/5G mobile core [117], to service meshes [31], and more. The control plane performs computationally taxing decision-making on a per-flow basis (using a general-purpose CPU), and the data plane caches the outcome of the decision as flow rules and applies it to every packet (using a dedicated ASIC), Figure 2.1(a).

This separation of the control plane and data plane is how SDN works in theory. However, in almost all real settings, a third component links the control plane and the data plane, an

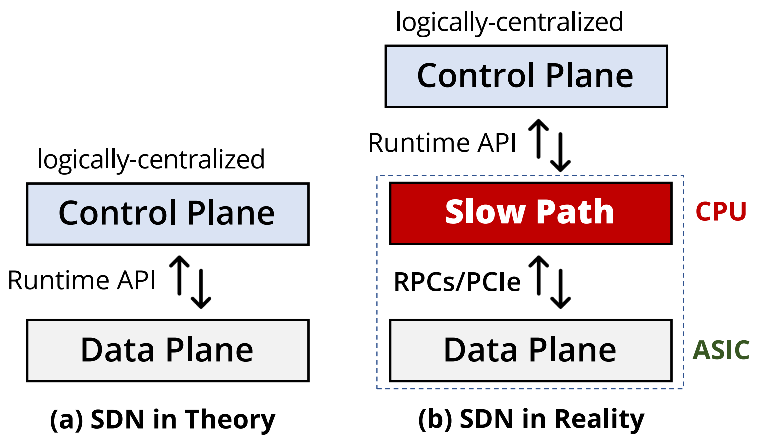


Figure 2.1: Slow path is the backbone of SDN systems, including vSwitches and the end host network.

entity known as the *slow path*[1], as shown in Figure 2.1(b). For example, the slow path is the switch OS [38] in hardware switches (e.g., Intel Tofino [59, 60]), the userspace logic in virtual switches (e.g., OVS [121, 108]), the PGW/SMF in 4G/5G networks [117], and an infrastructure layer in service meshes (e.g., Istio [61]).

These slow paths are not just responsible for populating data-plane caches (e.g., flow tables) with updates from the control plane, but also perform myriads of other critical tasks for the correct and timely operation of the environment they are running in [132, 36, 121]. For example, without the OS, a hardware switch would fail to update its tables quickly in response to changing network conditions (e.g., link failures and microbursts) [176, 50]. Similarly, the absence of the userspace logic in virtual switches [121] would lead to excessive load on the control plane, which would have to handle flow setup for each new flow. The same holds true for 4G/5G networks and service meshes.

So far, as a networking community, our focus has been on accelerating the data plane using dedicated silicon chips (e.g., Tofino for hardware switches [59, 60] or FPGA-based SmartNICs for virtual switches [100, 99, 58, 169]). Similar efforts are underway in accelerating the user plane of the 4G/5G mobile core [116, 81] and the data plane (i.e., sidecar proxy [128]) of service meshes using programmable ASICs [63]. This trend of accelerating data planes using ASICs is likely to continue as link rates rise beyond 200 Gbps, both at end-hosts and in the core of the network. Increasing link rates are not just straining the data plane but also the slow path. Likewise, the evolving compute and application landscapes (i.e., mega-scale, multi-tenant clouds and highly disaggregated micro-services) are further stressing the slow path when scaling to an ever-increasing number of tenants and services [138].

We argue and show that state-of-the-art CPU-based platforms alone, whether on the end

[1]A slow path is operating at a much faster timescale than a control plane—the "slow" in slow path is in comparison to the data plane, similar to what slow start is in TCP [118].

host, SmartNIC, or switch, are a poor fit for the slow path. In this chapter, we make the case for accelerating the slow path in the end-host network using Open vSwitch (OVS) as an exemplar (§2.2).

In OVS, the slow path serves three purposes: (1) to abstract away the resource limitations of the data-plane ASIC, providing the control plane with an abstraction of any-size flow table while installing only the active subset in the data plane as cache entries, (2) to update these caches with evictions when policy rules are updated in the vSwitch, and (3) to handle infrastructure protocols (such as BFD, LACP, IGMP, and OpenFlow) that a data-plane ASIC (e.g., a switch or NIC) otherwise cannot handle, as a kind of exception handler.

Ideally, these operations should all run in the data plane; however, today's data-plane accelerators are optimized for forwarding user traffic only, using match-action table (MAT) pipelines [17, 93]. They treat these flow handling and cache maintenance operations as exceptions, which are handled by a (CPU-based) slow path [93]. Hence, the slow path handles a *subset* of the data-plane traffic and operations that require *exceptional* processing. These *exceptions* include the sub-line-rate processing that requires complex control-flow, compute, or memory resources, which the data plane is not designed to handle. These slow-path processing requirements are difficult (or impossible) to express using MATs, so the slow path today is implemented on a CPU [184, 121].

At first glance, this division of labor may seem like the right approach, that, clearly, high-volume user traffic should be handled in the data-plane ASIC, and slow-path traffic and operations be processed on a CPU. However, as we show in §2.2, slow-path traffic has grown commensurately with user traffic over the years. Allocating more CPU resources directly reflects as higher operational expenses (OPEX), a cause of primary concern for the network operators running today's mega-scale cloud data centers because of per-core pricing [155]. That's why operators of these data centers aim at provisioning a single CPU core for their virtual switches [149], making the rest available for revenue-generating tenant workloads.

Offloading all slow path operations to recent ARM-based SmartNICs at the end-hosts is also not feasible: (1) they would not scale to ever-increasing link rates (exceeding 200 Gbps) [69], and (2) their maximum power usage is orders of magnitude higher than an equivalent ASIC-based NIC [135, 136, 5, 170], further raising OPEX. In short, for slow paths to scale in efficiency and performance in the next-generation networks, we must move away from purely CPU-based platforms to accelerated ones, similar to what we have been doing for the data plane (i.e., with switching ASICs) [17].

To make our case, we study Open vSwitch (§2.2), a production-quality virtual switch, and thoroughly evaluate its slow path (§2.3).

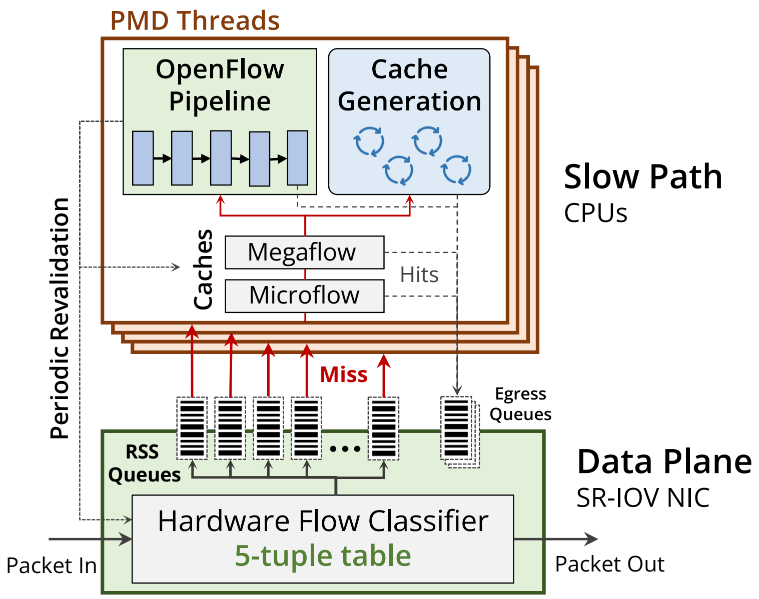


Figure 2.2: The design of Open vSwitch (OVS) [121, 149]: data plane offloaded to an SR-IOV NIC, and slow path running on server CPUs.

2.2 Understanding the vSwitch Architecture: OVS

Popular and widely used virtual switches, including OVS [121, 149], Microsoft VFP [36], and Google Snap [83], use a slow path to separate (and couple) the control and data plane. Over time, the per-packet (data-plane) processing functions have been offloaded to dedicated hardware ASICs (e.g., OVS offload [101, 171, 75, 46], Accelnet for VFP [37], 1RMA for Snap [139]). The slow path, on the other hand, has remained on general-purpose CPUs, where it is responsible for the creation and deletion of flows, maintaining large flow caches to abstract away resource limits of data-plane ASICs, performing cache evictions when rules are updated, executing control protocols (e.g., for link failure detection, multicast management, and link bonding and rebalancing, described in more detail in §2.3), and operations like packet sampling.

To study the challenges associated with running a slow path on CPUs, we picked Open vSwitch (OVS) [121]—a widely used open-source virtual switch inside cloud data centers—as our case study. OVS consists of a slow path and a fast path. Initially, the fast path used to run as software in the Linux kernel [120] or userspace [109, 149]. However, today it is offloaded to the NIC's data plane, as a flow-table classifier, to handle the rising link rates (200 Gbps) entering the datacenter end-hosts [131]. Popular NIC vendors now support OVS fast-path offload on their data-plane ASICs (e.g., Nvidia ConnectX-6 Dx [101], Intel IPU SmartNIC [75], and Xilinx Alveo SN1000 SmartNIC [171]). Yet, the slow path still runs on CPUs, either host CPUs or SmartNIC's onboard CPUs [46].

2.2.1 The vSwitch Fast Path

Hardware Flow Table. When a packet belonging to a new flow arrives at the NIC for the first time, it misses in the hardware flow table. The NIC enqueues the packet in one of the RSS queues, picked using the hash of the packet's 5-tuple. Each of these queues is connected to a "poll-mode driver" (PMD) thread, over PCIe. To sustain high throughput, the OVS slow path runs multiple PMD threads, each with its own set of caches, control processes, and OpenFlow pipeline. Upon entering the slow path, the packet again misses in two layers of flow caches: Microflow and Megaflow.

Software Cache Hierarchy. In OVS, a hierarchy of these caches is utilized to handle incoming packets [121]. The packet is first looked up in the Microflow cache, an exact-match cache designed to capture the temporal locality of traffic (i.e., packets from a specific flow arriving frequently), making it particularly effective for high-bandwidth "elephant" flows. On a miss, it is looked up in the Megaflow cache, which holds wildcard rules that leverage spatial locality (i.e., packets from flows with matching header prefixes), allowing a single entry to match an entire class of flows. The key insight is that subsequent packets with matching header fields (exact or wildcard) can be processed directly from the cache, requiring only a single lookup. Finally, on a miss in both caches, the packet traverses the full OpenFlow pipeline, after which the resulting traversals are processed and installed into both caches.

2.2.2 The vSwitch Slow Path

The OVS slow path performs four main tasks, shown in Figure 2.2: executing OpenFlow pipelines on new incoming flows; creating and maintaining flow caches, including the hardware flow tables; revalidating the cache when rules are updated to evict stale cache; and processing control packets pertaining to the infrastructure.

The OpenFlow Pipeline. The pipeline implements a sequence of OpenFlow-compliant flow tables [148] that network operators can configure from the centralized control plane. At runtime, the control plane specifies the flow rules (i.e., the match and action instructions) and other properties of the pipeline using the OpenFlow and OVSDB APIs [148, 119, 156]. OVS passes each missed packet through the OpenFlow pipeline table by table, composing each table's actions into a final set of actions, plus a set of caching instructions. The slow path executes these instructions on these packets and, if the instructions permit, installs a rule in the micro-/mega-flow cache of its PMD thread and the hardware flow table in the NIC. Later packets for the same flow match these rules. The slow path also periodically revalidates the caches and the hardware flow table, to ensure correctness against up-to-date rules in the OpenFlow pipeline.

Megaflow Cache Construction. OVS maintains a flow data structure (representing input packet fields) and a wildcard data structure, which is *inline* updated with each table lookup to represent all fields matched on during the slow-path traversal. Actions from successful lookups are applied immediately, and next tables to lookup are determined based on `resubmit` actions. After the slow-path lookup terminates, the initial and final (modified) flow are compared (`commit` operation) to determine `setfield` actions for the cache entry. The match predicate is constructed using a bitwise `AND` operation between the input flow and the final wildcard[2].

Cache Revalidation. OVS relies on an *iterative revalidation* scheme to evict stale cache entries. This approach scans cached entries one by one, checking each against the updated rule set [121, 66]. Because revalidation traverses the entire cache, its cost grows with cache size and is highly sensitive to update patterns. Moreover, it must be throttled to avoid interfering with fast-path packet processing. As a result, iterative revalidation frequently becomes a scalability bottleneck under dynamic workloads, making low time-to-eviction (TTE) difficult to achieve in practice (more details in §4).

Control Protocols. The OVS slow path also needs to process various infrastructure (control) protocols, such as those for link-failure detection (CFM [56] and BFD [70]), link bonding and rebalancing (LACP [55]), and multicast group management (IGMP) [24]. It also runs protocols for packet sampling (e.g., sFlow and IPFIX) for sending packets to the remote collector or the control plane.

2.3 The vSwitch Bottlenecks

In this section, we give an overview of bottlenecks pertaining to traffic trends and cache revalidation under rule updates. More evaluate these bottlenecks in more detail in §3 and §4 when we introduce GIGAFLOW cache and KAIRO cache eviction, respectively.

2.3.1 Traffic Patterns and Emerging Trends

Bottlenecks arise when slow-path traffic, such as cache misses and control packets, arrives faster than it can be processed. Cache misses in the data-plane ASIC are a common source of bottlenecks that has received attention both in industry [121] and research [125, 126]. Most often, these misses are because of the NIC's data plane's limited memory and feature support (i.e., it can only do 5-tuple matches), which causes flows matching on other fields to be processed by the slow path. The amount of incoming traffic a slow path can handle is also

[2]Note that the wildcard is only used for creating Megaflow cache; it is not needed for the OpenFlow lookup itself.

proportional to the number of PMD threads required (1 thread per core); as the traffic rate increases, the load on these threads also increases. With sufficient load, congestion in these threads causes HoL blocking and queuing delays, which in turn increases flow setup time. We show in §2.4 that increasing the number of PMD threads does not help.

Changing traffic patterns can also cause high miss rates. For example, some traffic patterns (such as microflows with varying source and destination addresses or packets for P2P rendezvous applications) can cause excessive misses [121]. Similarly, configuring rules to construct a complicated OpenFlow pipeline in the slow path, without following OVS best practices, can render the megaflow cache ineffective, so that each entry essentially ends up doing an exact match (leading to more misses). Bad actors can also exploit the flow tables remotely, by taking advantage of OVS classifier weaknesses to send pathological traffic that causes cache explosion [29].

2.3.2 Periodic Cache Revalidation

The slow path periodically revalidates the data plane cache, usually every second, to ensure correctness against up-to-date OpenFlow rules in the slow path. OVS revalidates in dedicated slow-path threads, which visit all the data-plane cache entries. For each entry, they simulate a traversal of the OpenFlow pipeline and compare the output to what is currently installed in the cache, and update the entry if necessary. The slow-path revalidation threads, therefore, must be able to verify the whole cache in under one second. And, if the cache entries exceed a prescribed limit, then further cache-flow entries cannot be installed—resulting in all packets of new arriving flows consistently entering the slow-path (as demonstrated in §2.4), which hampers performance due to an overloaded slow path.

2.3.3 Infrastructure Packet-processing Overhead

The packet-processing load of control protocols can also produce bottlenecks when these packets are hogged behind normal traffic. The worst case happens for the BFD and CFM protocols when monitoring the liveness of peers. When slow-path processing falls behind, e.g., due to HoL blocking under excessive load (e.g., cache misses), it can prevent BFD and CFM packets from being processed and replied to in a timely fashion. This can cause a host's peers to conclude that the link is down and that they should choose an alternate host, which increases the traffic to that host and potentially causes further disruption. Using longer protocol timeouts eases the issue, but it can potentially slow down the detection of genuine network or host failures.

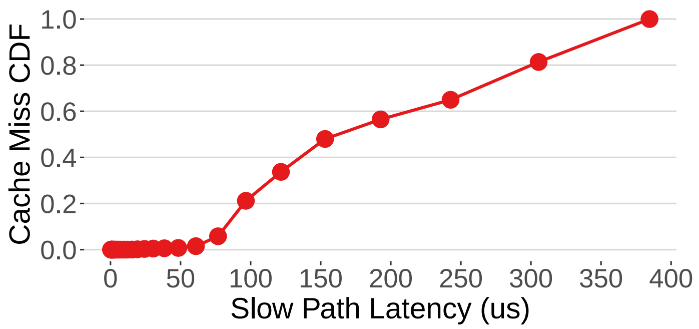


Figure 2.3: Empirical distribution of cache miss processing times. We only show a quarter of the total number of cache misses that are processed in under 400μs. The rest take more than 400μs (not shown here).

2.4 Implications of vSwitch Bottlenecks

To understand the efficiency and performance implications of the slow path, we set up a testbed with two machines equipped with dual-port ConnectX-6 DX SmartNICs [100] connected back-to-back. Both machines have two Intel Xeon Platinum 8358P processors with 64 cores, running at 2.60 GHz with 512 GB of memory. They both run Ubuntu 22.04 with kernel version 5.15.0-33-generic. We also set up OVS 2.17.90 with DPDK 22.03.0-rc1 and Mellanox OFED 5.6-1.0.3 drivers (`mlx5_core`), and configured SR-IOV on the NIC with OVS offload. One machine ran the TRex traffic generator and a custom DPDK script to generate a ClassBench-rules-compatible CAIDA [23] traffic profile, on-the fly [143, 126], to the other machine, which is the device under test (DUT).

• **Cache misses cause highly skewed slow-path processing times with long tails.** We installed 200K rules in OVS, and tested its cycles per cache-miss performance against real-world CAIDA [23] traffic profile taken from the Equinix datacenter in January 2019. A custom DPDK script [126] took CAIDA's traffic profile (e.g., flow locality, inter-arrival times) and generated compatible packets on-the-fly by replacing the 5-tuples from the traffic with 5-tuples from ClassBench rules to maintain the properties of a realistic traffic trace. We configured OVS to use 4 RXQs for the physical interface with 4 PMD threads, and generated the traffic at 10 Gbps to send a total of 790 million packets. With these rules, we observed a total of 893K cache misses in OVS, 1.5% of which were dropped at all four PMD threads. Figure 2.3 shows a cumulative distribution of the CPU cycles consumed by the cache misses in the OVS slow path. Worth noting is the two orders of magnitude variation (between median and tail) in CPU processing times for these cache misses[3] (all matching on 5 tuples). A cache miss can take anywhere from 30K to millions of clock cycles in slow-path processing. These

[3]Figure 2.3 shows only 25% of the actual upcalls. The OVS performance counters aggregate all upcalls consuming over 1M clock cycles. These upcalls end up on the right side of this CDF (not shown).

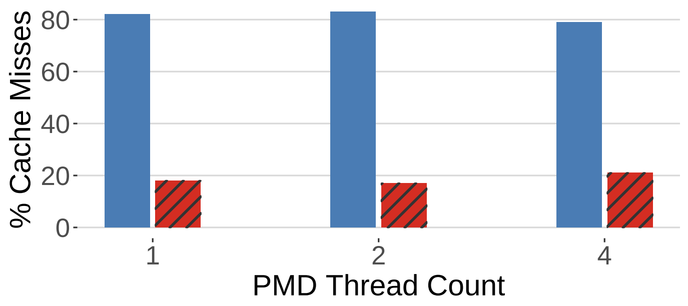


Figure 2.4: Number of served (blue) and dropped (red) cache misses vs. number of PMD threads. The number of PMD and revalidator threads are kept equal.

processing times will be even higher in a realistic environment, with OpenFlow rules matching other header fields (31 in a representative NSX deployment [109]), not just a 5-tuple. These rules can further burden the slow-path, leading to even higher tail latencies. The impact on applications' performance can be significant: Memcached suffers by over 50% performance hit in the presence of 50 μs added delay [122].

• **Increasing flow rules saturate caches and increase the revalidation cost.** We installed 450K ClassBench rules (utilizing ACL seed-5 profile) and used OVS internal counters to measure the number of cache misses when OVS processes CAIDA traffic (with rule-compatible headers) at 25 Mpps. We sent traffic for 12 seconds and repeated it with different PMD thread counts (while having equal or more RXQs configured on the ports). The number of revalidator threads are also kept equal to the number of PMD threads. The number of successfully processed cache misses (that resulted in a cached entry) versus the dropped (did not result in a cached entry) are shown in Figure 2.4. Cache misses are dropped due to the cost of revalidation (§2.3), which increases with larger cache sizes. To keep the cost within limits, additional cache misses are dropped, which causes all later packets of those flows to enter the slow path. The consequence of this bottleneck is wasted CPU cycles in processing all the subsequent *non-cachable* upcalls, which could have been avoided if the revalidation cost were affordable.

• **Larger caches improve vSwitch performance but make revalidation prohibitively expensive.** To understand the cost of cache revalidation as cache sizes grow, we set up the Cord OFDPA pipeline (OFD, 10 tables) [103] (Table 3.1 describes vSwitch pipelines in more detail) in OVS and generate a workload of 1M unique flows. We then apply rule updates at runtime and measure the time-to-eviction (TTE) of the cache. As shown in Figure 2.5, larger caches significantly improve hit rates and reduce cache misses. However, TTE increases linearly with cache size, leading to longer periods of stale cache usage (i.e., outdated entries used after rule changes, leading to incorrect forwarding). Moreover, deeper pipelines exacerbate the cost since each cache entry triggers a complete pipeline traversal

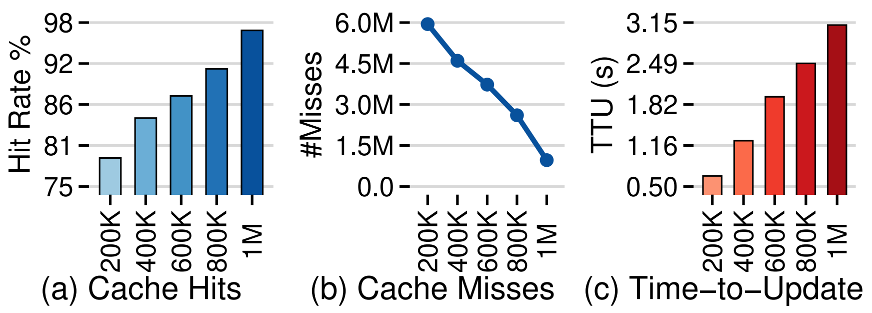


Figure 2.5: Cache hits (%), cache misses, and time-to-update vSwitch (TTU) with increasing Megaflow cache size using the Cord OFDPA pipeline (OFD) [103].

during revalidation. For instance, revalidating 500K entries takes 1.62 s in Cord OFDPA (OFD, 10 tables) [103], compared to 3.60 s in Antrea OVS (ANT, 22 tables) [10, 11, 12].

• **CPU-based slow paths struggle to keep up with increasing traffic volume.** These are two further consequences of the results shown in Figures 2.3 and 2.4. First, the volume of slow-path traffic (cached or dropped) on a single PMD core is about 8K requests per second when running at a link rate of 25 Mpps. With link rates projected to exceed 200 Gbps (288 Mpps for minimum-sized packets), the slow-path traffic would rise to roughly 92K requests per second, which on a state-of-the-art server CPU, running at 2.6 GHz translates to 28K clock cycles (10.8 μs) per request. Figure 2.3 indicates that, even with simple rule matching, there is an order of magnitude of variation in completion times for different slow-path requests, making a single PMD core dedicated to handle slow-path traffic grossly insufficient, hence calling for CPU scaling just to handle the slow-path traffic.

Second, scaling the number of cores only partially addresses the problem. As shown in Figure 2.4, as we increase the PMD thread count, the slow-path traffic volume does not reduce since cache misses occur when new flows arrive regardless of having more cache space (§2.3). The revalidation cost of flow caches demands limited cache sizes and increasing the revalidation threads doesn't benefit. Moreover, the dropped traffic volume also increases and the number of clock cycles left for slow-path processing per request per CPU at higher rates does not benefit much.

• **Uncertainties in control-packet processing yields devastating datacenter-wide consequences.** OVS runs the BFD protocol for link failure and fault detection. Typical production environments configure pairwise BFD among virtual tunnel endpoints. If a BFD packet is delayed while waiting behind long slow-path queues (Figure 2.3), OVS might falsely label a link down and, as a consequence, stop forwarding traffic to a node. Similarly, OVS handles IGMP snooping in the slow path to detect multicast routers. Delayed processing of IGMP packets would cause multicast packets to be incorrectly delivered to obsolete ports, increasing traffic on those links and exacerbating slow-path traffic volume. Creating dedicated queues for processing such packets is not feasible since these packets are tunneled in real environments (i.e., encapsulated in a VxLAN/Geneve tunnel header), and most hardware NICs cannot classify and prioritize using inner protocol headers [57, 99, 169, 7].

2.5 Discussion

The bottlenecks described in this chapter share a common root: today's Megaflow cache is too small to accommodate realistic flow volumes in emerging workloads and applications, and making it larger only exacerbates the cost of cache eviction in face of diverse update patterns. A high hit-rate cache that can accommodate the full diversity of production rule sets—without driving up miss rates or saturating PMD threads—is therefore the first-order need. Such a cache must exploit new, previously unexplored forms of localities (e.g., from the structure of vSwitch OpenFlow pipelines) to maximize rule space coverage in the cache and reduce cache misses.

At the same time, revalidation as a cache eviction mechanism cannot scale with cache size. Re-evaluating the entire cache on every rule update, regardless of which cached entries are actually invalidated, is fundamentally wasteful. What is needed is an eviction mechanism that treats cache correctness as an incremental problem: when a rule changes, only the entries whose forwarding behavior is affected should be evicted, leaving the rest untouched. This is precisely the insight behind incremental view maintenance, where updates propagate only the delta of a change through the system rather than recomputing from scratch.

We address the first problem in §3 and present Gigaflow, a pipeline-aware cache that improves hit rates by exploiting locality in vSwitch pipeline traversals. Next, we address the second problem in §4 and present Kairo, which replaces iterative revalidation with incremental view maintenance using DBSP [20] that bounds eviction cost to the scope of a rule change, independent of cache size.

CHAPTER 3

Gigaflow: Pipeline-Aware Sub-Traversal Caching for Modern vSwitches

“
The core idea in this paper is very interesting, making a strong argument for the multi-cache design for matching in SmartNICs. This idea could have immediate impact on real cloud workloads as the flow table size and performance is a key factor in cloud application performance.

Anonymous Reviewer #2, ASPLOS 2025

3.1 Introduction

Since the late 90s, the end-host networking stack in datacenter networks has transformed into a switching substrate built around virtual switches (vSwitches) [121, 73]. These vSwitches act as the last-hop layer in the modern distributed computing landscape—routing traffic to and from virtual machines (VMs) and containers, connecting them to the outside world [73, 121, 36, 158]. Early incarnations of these vSwitches primarily resulted in mimicking the functionalities of fixed-function hardware switches as hardcoded software switches (e.g., Linux Bridge and iptables) [39, 160]. We have come a long way since then (a) through a series of software optimizations aimed at maximizing CPU performance [121, 125, 126] to (b) leveraging hardware offloads using modern SmartNICs [100, 99, 58, 169, 46]. Nevertheless, the challenge persists: *these vSwitches struggle to scale effectively with emerging workloads and increasing link rates* [184, 126].

In software, applying the entire multi-table pipeline—comprising a sequence of transformations based on network policies (e.g., L2, L3, or ACL)—within a vSwitch for each incoming

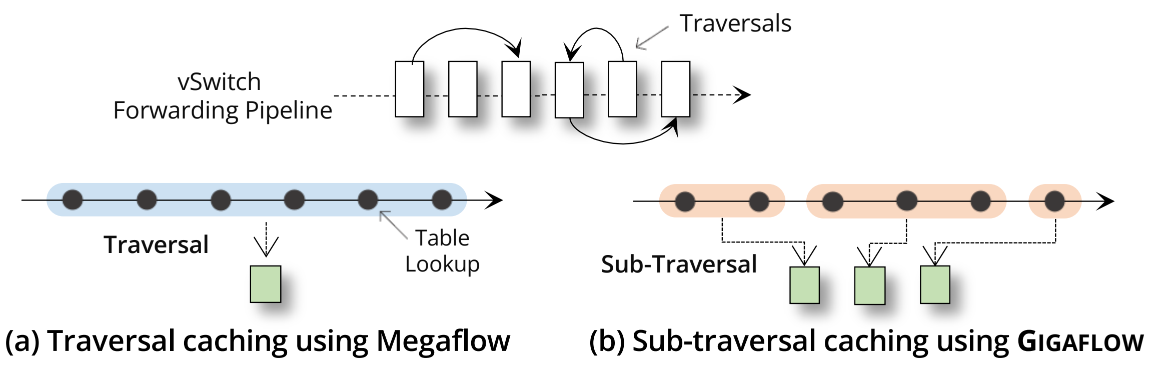

Traversals
vSwitch Forwarding Pipeline
Traversal
Table Lookup
Sub-Traversal
(a) Traversal caching using Megaflow
(b) Sub-traversal caching using GIGAFLOW

Figure 3.1: (a) A *traversal* is a complete sequence of table lookups through the vSwitch pipeline that generates a Megaflow rule. (b) A *sub-traversal* is a subset of these lookups within a traversal, capturing smaller, reusable segments shared across multiple flows.

packet is prohibitively expensive. CPUs struggled with handling multiple lookups, resulting in significant performance degradation with each additional lookup [121]. Early optimizations addressed this by introducing single-lookup caches, namely Microflow and Megaflow [121]. In the Microflow cache, the first packet in a flow is processed by the multi-table pipeline to generate a single exact-match rule, which is then cached, allowing subsequent packets to bypass the full pipeline. Later, Microflow caches evolved into wildcard-based Megaflow caches, which improved aggregated vSwitch throughput by allowing a broader range of traffic patterns to be handled within the cache.

More recent work has focused on enhancing lookup speeds within Megaflow caches by replacing the existing Tuple Space Search (TSS) classifier [121, 140] with compact machine learning models, such as the Range Query-Recursive Model Index (RQ-RMI) [125, 126]. Despite these improvements, CPU limitations—particularly the stagnating performance growth with the slowdown of Moore's Law [48, 3, 161, 144, 167]—continue to restrict the overall throughput of vSwitches to less than 10 Gbps per core [121, 149, 126].

There is an urgent push within the networking industry to shift from CPU-based virtual switching to SmartNICs, as evident through recent products from major players, including Amazon Nitro [4], Nvidia Bluefield [99, 100], AMD Pensando DPU [7, 5], and Intel IPU [57]. Equipped with a hardware (HW) cache, these NICs can process and route traffic directly to and from the virtual endpoints (e.g., using SR-IOV [102]), effectively bypassing software-based processing. These NICs can reach link speeds of 400 Gbps and higher [99] when the matching rule is present in the HW cache.[1]

Yet, the main challenge with these NICs is the limited size of their HW caches, typically holding between 10–50K wildcard rules—far fewer than the software-based caches [85, 157, 121].

[1]Note, the on-NIC ARM cores are typically less powerful and result in even poorer performance compared to the server cores [26] (see §3.6).

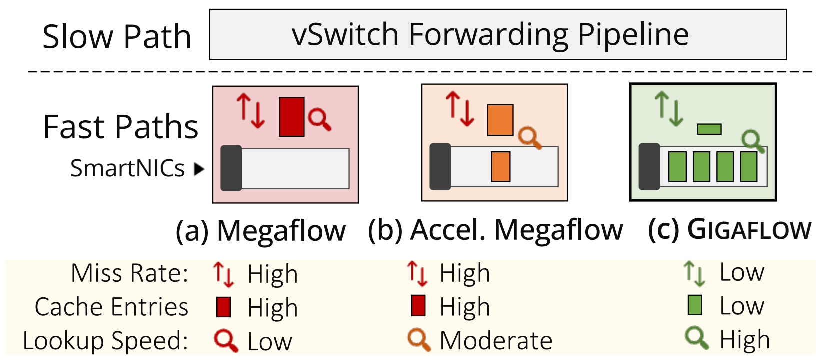


Figure 3.2: Comparing Megaflow cache, hardware-accelerated Megaflow, and GIGAFLOW, in terms of cache miss rates, cache entries, and average lookup speed.

This limitation is attributed to the restricted power budget of SmartNICs, typically around 75 W [170], and the complexity of integrating large TCAMs on-chip [151, 142, 127, 79, 96]. As a result, despite their performance advantages, high miss rates within these caches result in significantly lower overall aggregate throughput. Thus, most implementations today use HW caches to handle only a small subset of traffic—primarily long and bursty flows—while directing the bulk of other traffic to software for processing [157, 46, 133, 33].

In this chapter, we offer a fresh perspective on caching rules within SmartNICs. Traditionally, two assumptions have guided the design of caching in vSwitches:

> **Assumption #1:** *Multi-table lookups are prohibitively expensive, necessitating a single-lookup cache (e.g., Megaflow) [121, 126].*
> **Assumption #2:** *Cache-rule generation relies solely on the locality derived from traffic patterns (i.e., temporal and spatial) [121, 126, 149, 36].*

We argue that it is time to revisit these assumptions. First, unlike CPUs, modern SmartNICs can perform multi-table lookups directly in hardware at line speeds, similar to high-performance network switches (e.g., RMT [17, 59], dRMT [27], and Trident5 [19, 18]), allowing for cross-product rule combinations that greatly expand rule-space coverage beyond physical table limits. Second, the vSwitch pipelines are programmable (e.g., using P4 [16] and OpenFlow [89]), letting operators configure the types and precise ordering of policies to apply (e.g., L2, L3, or ACL). This flexibility introduces a new, powerful form of locality—*pipeline-aware locality*—that extends beyond the traditional temporal and spatial localities.

In conventional caching, Microflow rules capture temporal locality by caching frequent packets from the same exact flow, while Megaflow rules use wildcards to capture spatial locality, grouping flows with overlapping headers. Pipeline-aware locality, however, leverages the structure of the vSwitch pipeline itself, enabling cache rules based on shared sequences and operations within the pipeline.

To exploit this pipeline-aware locality, we extend the concept of a *traversal* [121, 36, 73, 62, 149, 184] in vSwitches—a unique sequence of table lookups through the vSwitch pipeline that generates a Megaflow rule [121, 126, 62], reflecting both the order of table lookups and the rules matched within those tables (Figure 3.1a). We decompose each traversal into smaller, reusable segments, known as *sub-traversals* (Figure 3.1b), which represent common lookup patterns shared across flows, capturing frequently repeated sequences within the pipeline. By caching these sub-traversals and strategically installing them across SmartNIC tables, we allow multiple flows to reuse shared segments of the pipeline, creating a more efficient and scalable caching strategy.

Building on these insights—(a) line-rate multi-table lookups in SmartNICs and (b) Megaflow rules constructed from overlapping sub-traversals—we design Gigaflow (Figure 3.2), a caching subsystem that maximizes rule-space coverage and cache hit rate while maintaining line-rate performance:

• To ensure that the sub-traversal rules in the SmartNIC tables preserve the correct sequence of the original vSwitch pipeline for a given flow, Gigaflow introduces Longest Traversal Matching (LTM) with table tags (§3.4.2). By selecting the longest matching sub-traversal in each table, LTM ensures the correctness of the lookup operations, with table tags maintaining the proper sequence order. This allows Gigaflow to support partial matches across tables without violating pipeline semantics.

• To identify the most common sub-traversals for maximum rule-space coverage in the SmartNIC, Gigaflow leverages the disjointedness property of vSwitch pipelines (§3.4.3), i.e., separating sub-traversals at boundaries with disjoint headers (e.g., across L2 and L3 headers). This stateless approach effectively captures pipeline-aware locality while keeping the processing cost constant, irrespective of the number of cached rules. Crucially, this partitioning requires no changes to the vSwitch pipeline itself, making Gigaflow deployable as a drop-in caching layer.

We implement Gigaflow as a new caching subsystem inside Open vSwitch (OVS) [121] and deploy it on P4-programmable FPGA-based SmartNICs using Xilinx's P4SDNet [9] with the OpenNIC codebase [8, 147] and Alveo U250 FPGAs [172, 169]. Our experiments use five real-world vSwitch pipelines with diverse traversal patterns (Table 3.1), along with Classbench [143] rulesets, and CAIDA [23] traffic traces. The results demonstrate up to 51% improvement in hit rate (25% on average), up to 90% fewer cache misses, and up to 450× greater rule-space coverage with 18% fewer cache entries, all while maintaining line-rate performance. Our Gigaflow prototype is available as open source[2].

[2]Codebase: https://github.com/gigaflow-vswitch

3.2 Background

Traditionally (Figure 3.3a), a NIC would forward all incoming packets up the host's hypervisor stack to a virtual switch (vSwitch), which applies steering policies (as flow rules) to route the packets to the destined virtual machines (VMs) or containers [121, 36]. Thus, being on the critical path demanded that the vSwitch be able to swiftly process packets with minimal overhead—as time spent processing packets here would directly impact the performance of the VMs, hosting tenant's workloads [179, 77, 94, 159, 44, 173].

3.2.1 Programmable vSwitch Pipelines

Early vSwitches, such as Linux Bridge [39] and iptables [160], proved inadequate in supporting advanced use cases (e.g., network virtualization [73]) crucial for modern cloud environments [121]. Hosting multiple tenants' workloads and VMs demanded sophisticated flow policies for filtering, isolating, and steering traffic across the shared infrastructure [121, 73]. For instance, data centers required a new protocol (VXLAN [162]) to partition the network into finer overlays than VLAN [163] or other traditional protocols could achieve. To express these policies, vSwitches, like Open vSwitch (OVS) [121], VMware's VDS [152], and Microsoft's Virtual Filtering Platform (VFP) [36], began exposing a programmable pipeline of match-action tables (MATs) through a standardized interface (e.g., OpenFlow [89]). Instead of relying on fixed pipelines, operators could now program these switches and specify, as a sequence of flow rules, what policies to execute and in what order.

3.2.1.1 vSwitch Caches: Microflow & Megaflow

CPUs, however, struggled to execute these programmable multi-table pipelines efficiently, leading to considerable performance degradation compared to earlier counterparts [121, 184, 126]. This led to a series of innovations in the development of fast single-lookup flow caches (such as Microflow/Megaflow in OVS [121], and Unified Flow Table (UFT) in VFP [36]), which store the results of packets processed through the multi-table pipeline (e.g., *traversal*) as cache entries. The main insight was that subsequent packets with matching header fields (exact or wildcard) could be processed directly from the cache, requiring only a single lookup.

For example, in OVS, a hierarchy of these caches is utilized to handle incoming packets [121]. The packet is initially checked in the exact-match Microflow cache, designed to capture the temporal locality of traffic (i.e., packets from a specific flow arriving frequently). Next, it is looked up in a wildcard-match Megaflow cache, which leverages spatial locality (i.e., packets from flows with matching prefixes arriving closer in time). Finally, it progresses through the vSwitch multi-table pipeline, and if the instructions permit, corresponding rules are installed

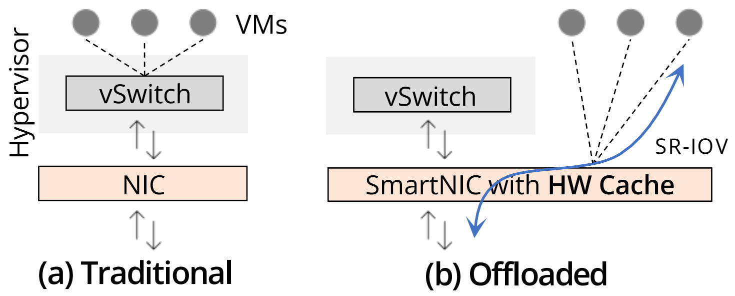


Figure 3.3: The modern end-host virtual switches offload traffic steering to and from VMs to SmartNICs, with hardware (HW) cache, for high-speed networking.

in the two caches. These caches function effectively when they have large sizes (i.e., to capture large flow-rule space) but quickly lose their advantage when the size is limited (see §3.6).

3.2.2 Accelerating vSwitches via SmartNICs

Achieving high speeds with CPUs is a tall order, especially when compounded by the need to utilize minimal CPUs, typically just one, for virtual switching in public clouds [57, 100, 99, 169, 136]. Regardless, several seminal works came forth including kernel-bypass techniques (like DPDK [32] and Netmap [145]), caching schemes (e.g., exact/wildcard matching) [121, 149], and more recent ML-based packet classifications [125, 126] to reduce the processing cycles spent by vSwitches on packet handling. Yet, despite these innovations, the deceleration of Moore's Law [3] and the intrinsic constraints of CPUs (i.e., the von Neumann bottleneck [164]), restricts the maximum achievable performance to less than 10 Gbps per CPU [121, 149, 126].

To overcome these limitations, the networking community is now turning towards SmartNICs in an effort to offload packet steering from vSwitches—taking them off the critical path [4, 100, 99, 7, 5, 57, 84, 92]. A hardware (HW) cache inside SmartNICs (Figure 3.3b) stores and applies steering policies to incoming packets and routes them directly to their designated endpoints (e.g., VMs and containers) using single root input/output virtualization (SR-IOV) support [102]. Unlike CPUs, these caches utilize a match-action table (MAT) architecture based on TCAMs [57, 99], capable of processing packets at link speeds exceeding 400 Gbps. This new configuration resembles the operation of modern switches, featuring an ASIC dedicated to line-rate packet processing, complemented by an on-switch CPU responsible for programming entries within the ASIC [99, 57, 170]. In our case, it is the vSwitch, running on the CPU (host or ARM SoC), that programs the SmartNIC HW cache.

The pressing need to scale end-host networking and to break past the restrictions of CPUs has, thus, resulted in the introduction of a variety of such SmartNIC products by various NIC vendors and cloud providers. These include Amazon Nitro [4], Nvidia Bluefield [99, 100], AMD Pensando DPU [7, 5], Intel IPU [57], Marvell LiquidIO [84], and Microsoft Fungible DPU [92].

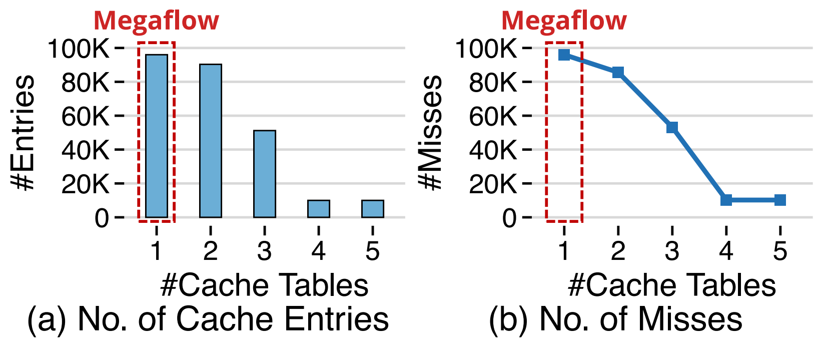


Figure 3.4: Increasing the cache tables (K) reduces cache entries (a) and misses (b), cutting vSwitch CPU usage by an average of 3× while covering 335× more rule space, compared to Megaflow; tested against 100,000 flows (see §3.6).

However, despite their raw computing power, these SmartNICs are now constrained by the size and utilization (i.e., hit rate) of their HW caches [85, 84, 17, 27, 99, 136].

3.3 Motivation & Key Insights

To propel end-host networking into the era of 400 Gbps+ packet processing, it is crucial that the majority of traffic be handled by the HW caches in the SmartNICs. Doing so, however, requires a radical rethinking of how we develop and offload vSwitch caches today. Simply increasing the size of these HW caches would not suffice, as evident by most recent SmartNIC products (typically, holding 10–50K rules) [85, 84]. The limited PCIe power budget of 75 W [169, 170, 135] and the operational complexity and cost of integrating TCAMs on-chip [127], restrict the number of rule entries these HW caches can accommodate. This begs the question, *is there a way to scale the rule efficiency and performance of these SmartNICs?*

We believe there is. Until now, all our efforts have been driven by the caching assumptions highlighted earlier in §3.1 (i.e., single-lookup flow caches are essential, as multi-table pipelines result in higher classification load and to improve hit rate, the only available locality for cache-rule generation is derived exclusively from the traffic). However, two recent trends invalidate these assumptions today.

3.3.1 Multi-Table Lookups & SmartNIC Offloads

Similar to data center switching ASICs (e.g., RMT [17, 59]), SmartNIC switching ASICs (e.g., on the Nvidia Bluefield [99]) can execute multi-table lookups at line rate with sub-μs latencies [169, 170].

At first glance, this capability might suggest that we could simply offload the entire vSwitch pipeline onto the SmartNIC. However, several limitations make this approach infeasible: (1) the vSwitch multi-table pipelines are intricate, offering an abstraction of any-size flow tables

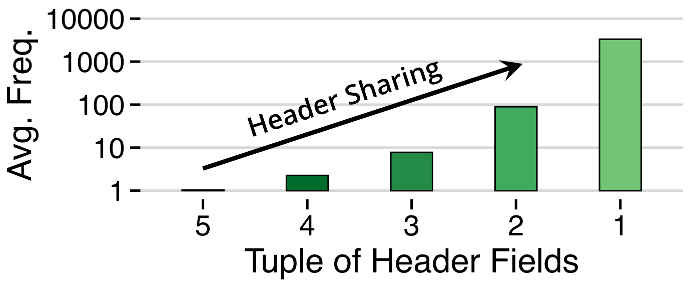


Figure 3.5: The average frequency of a tuple of header fields reoccurring in the Classbench ruleset (which includes 200,000 rules) increases as the number of the matching header fields decreases ($5 \rightarrow 1$), thereby increasing the potential for header sharing.

with deep pipelines (e.g., 256 tables [121]) and complex control flow structures (e.g., loops) to support operator policies [121, 73], and (2) the low power rating and restricted chip area of SmartNICs limit how many tables they can support (typically 4–8) [169, 57, 75]. Additionally, recirculations within SmartNIC ASICs directly affect throughput and latency, resulting in sub-line-rate processing [59, 60].

> ❏ **Key Insight: I1**
>
> *Rather than directly offloading the vSwitch pipeline as-is, we can trace the control flow of a packet through the pipeline, yielding a linear sequence of lookups (i.e., a traversal). The goal is then to map this traversal (involving N tables) onto the SmartNIC pipeline with K cache tables, where $K \ll N$.*

A Megaflow cache corresponds to a mapping with K=1 (Figure 3.1a), which simplifies the mapping problem significantly [121]. In a Megaflow cache, each table's actions are traced and composed into a final set of actions, along with a matching wildcard, and cached as a single rule. We generalize this concept to K>1 by breaking a traversal into K sub-traversals (Figure 3.1b), which are then mapped across the SmartNIC's multi-table pipeline. The difficulty, however, lies in identifying optimal breakpoints for sub-traversals to maximize sharing among incoming flows. For example, using a real-world vSwitch pipeline (OLS; Table 3.1), Classbench ruleset [143], and CAIDA traffic traces [23], setting K=4 reduces cache misses by up to 90% (Figure 3.4a) and increases rule-space coverage by 335×, all while requiring only 10,000 entries (Figure 3.4b).

3.3.2 Pipeline-Aware Locality & Programmable vSwitches

The vSwitch pipelines are programmable (e.g., via P4 [16] and OpenFlow [89]); network operators can specify which policies to apply and in what order—forming a sequence of table lookups through the vSwitch pipeline, with each table serving a distinct function (e.g., L2, L3, or ACL). Ideally, the most frequently and commonly used sub-sequences of these table lookups should be selected as candidate sub-traversals for installation on the SmartNIC's cache tables. However, identifying these optimal sub-traversals requires real-time awareness of traffic patterns and the cached sub-traversals; a process that becomes progressively more costly as the cache size increases.

> ❑ **Key Insight: I2**
>
> *Another approach to identifying frequently taken sub-traversals within the vSwitch pipeline is to examine the fields each table matches on. While it may seem counter-intuitive initially, breaking these sub-traversals across tables—matching on disjoint header fields—could result in sub-traversals that are frequently traversed (or shared) by incoming traffic flows.*

For instance, as shown in Figure 3.5, a sub-traversal matching between 1-4 headers can be shared by 856 flows (on average) in the Classbench [143] ruleset, which depicts a datacenter environment. However, the average sharing drops to about 1.03 when matching on the entire 5-tuple with a Megaflow cache. This indicates that breaking sub-traversals into segments of smaller, disjoint header fields would yield fewer misses and cache entries (Figure 3.4).

3.4 Design of Gigaflow

Building on these insights—caching sub-traversals from the vSwitch pipeline in the SmartNIC (**I1**) and leveraging disjointedness for pipeline-aware locality (**I2**)—we introduce Gigaflow, a novel system featuring a multi-table caching scheme in the SmartNIC, called Longest Traversal Match (LTM), §3.4.2. Gigaflow populates this cache by partitioning traversals into disjoint sub-traversals (with no overlapping fields), ensuring maximum rule-space coverage, §3.4.3.

3.4.1 Workflow

As illustrated in Figure 3.6a, Gigaflow's LTM cache in the SmartNIC consists of K feed-forward lookup tables based on the P4-programmable RMT architecture ($\mathrm{GF}_1, \mathrm{GF}_2, \ldots, \mathrm{GF}_K$), which is widely supported by commercial SmartNICs today [57, 99, 136, 169, 172]. When an incoming packet misses the LTM cache, it is sent to the vSwitch, where its flow signature, F (a set of header fields), is extracted and processed through the userspace forwarding pipeline (§3.4.3.1). This process generates a *traversal*, which includes the accessed vSwitch pipeline tables ($T_1, T_2, \ldots, T_N$), the modified flow after each table (F^i), and a wildcard indicating the matched header bits (W_i), as shown in Figure 3.6b.

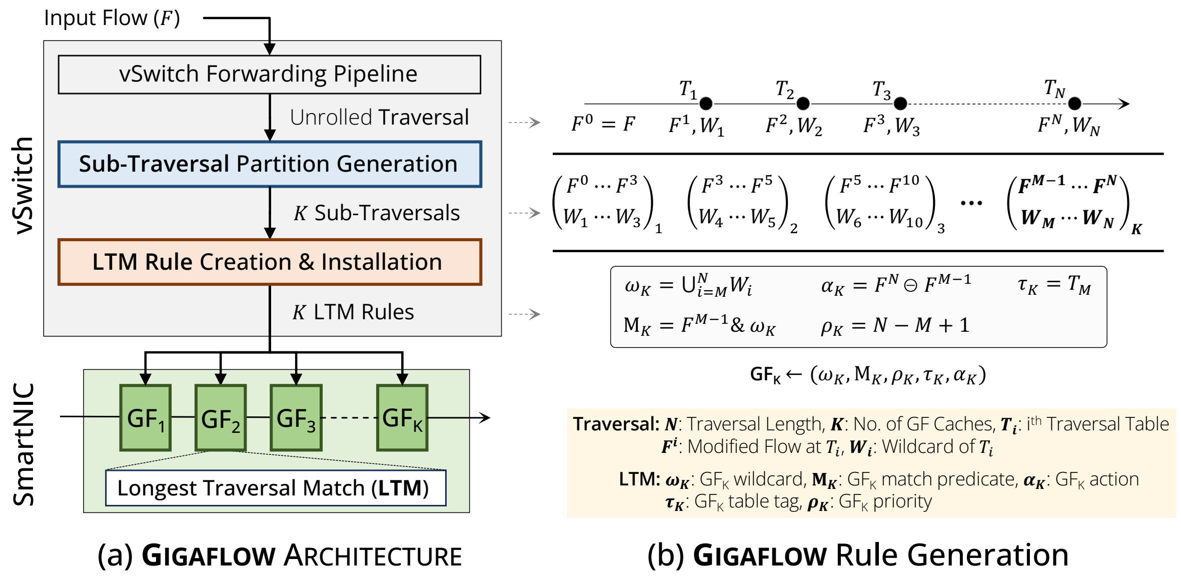

Input Flow (F)
vSwitch
vSwitch Forwarding Pipeline
Unrolled Traversal
Sub-Traversal Partition Generation
K Sub-Traversals
LTM Rule Creation & Installation
K LTM Rules
SmartNIC
GF1 GF2 GF3 GFK
Longest Traversal Match (LTM)
$F^0 = F$, F^1, W_1, F^2, W_2, F^3, W_3, F^N, W_N
$\omega_K = \cup_{i=M}^{N} W_i$ $\alpha_K = F^N \ominus F^{M-1}$ $\tau_K = T_M$
$M_K = F^{M-1} \& \omega_K$ $\rho_K = N - M + 1$
$GF_K \leftarrow (\omega_K, M_K, \rho_K, \tau_K, \alpha_K)$
Traversal: N: Traversal Length, K: No. of GF Caches, T_i: ith Traversal Table
F^i: Modified Flow at T_i, W_i: Wildcard of T_i
LTM: ω_K: GFK wildcard, M_K: GFK match predicate, α_K: GFK action
τ_K: GFK table tag, ρ_K: GFK priority
(a) GIGAFLOW ARCHITECTURE
(b) GIGAFLOW Rule Generation

Figure 3.6: High-level GIGAFLOW architecture: the input flow (F) traverses the vSwitch userspace pipeline to produce an unrolled traversal, which is then partitioned into sub-traversals and stored as LTM rules in GIGAFLOW caches.

Next, GIGAFLOW divides the traversal into up to K disjoint sub-traversals, selecting a partitioning scheme that maximizes disjointness between adjacent sub-traversals (§3.4.3.2). These sub-traversals are converted into LTM rules, each containing a match predicate (M_k), a wildcard (ω_k), a match priority (ρ_k), a table tag (τ_k), and an action (α_k), Figure 3.6b. The rules are subsequently installed into the LTM cache, allowing incoming packets to leverage shared segments of the pipeline for efficient processing (§3.4.3.3).

For instance, Figure 3.7 depicts two traversals generated from different flow signatures. Both initially follow the same sequence of vSwitch tables, $T_1 = 1 \rightarrow T_2 = 3 \rightarrow T_3 = 2$, matching on common Ethernet headers (ETH) and an IP/24 prefix. They diverge at the fourth table ($T_4 = 6$), where one matches on the TCP destination port and the other on a more specific IP/32 address. By partitioning these traversals into sub-traversals, GIGAFLOW installs the resulting rules into the LTM cache, capturing both the original flows and potential new flows that combine segments of the original traversals (indicated by the purple paths in Figure 3.7)—allowing direct handling in the SmartNIC without traversing the entire vSwitch pipeline. We will use this as a running example throughout the remainder of the chapter.

3.4.2 GIGAFLOW SmartNIC Offload

Sub-traversal caching imposes additional challenges for ensuring accurate and consistent lookups within SmartNICs, especially in RMT-based architectures [17, 27, 59, 60]. Unlike traditional traversal caching (Figure 3.1a), which consolidates the entire vSwitch processing

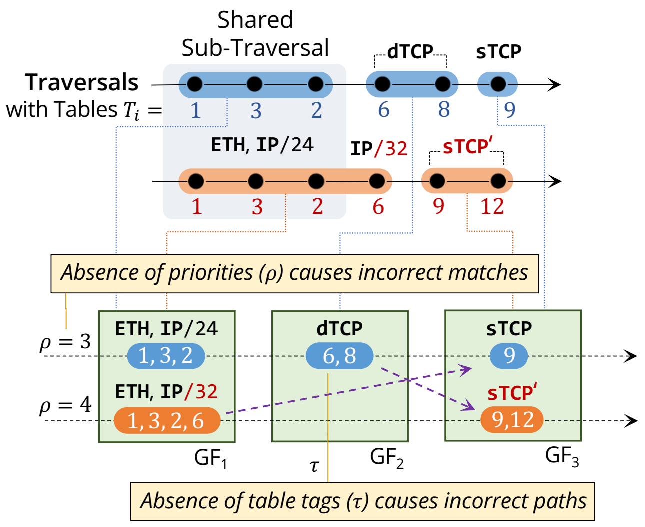


Figure 3.7: Example GIGAFLOW processing with two traversals partitioned across three cache tables, capturing four flows.

into a single, non-overlapping Megaflow rule [121, 126, 108], sub-traversal caching (Figure 3.1b) distributes rules across multiple cache tables. This distribution can lead to overlapping sub-traversals from different flows, resulting in multiple matches within the same table and making it harder to accurately reflect the original vSwitch traversal sequence. While Megaflow caching maintains a clear one-to-one mapping between the rule and its traversal, sub-traversal caching must handle these complexities to ensure accurate and consistent matching.

3.4.2.1 Longest Traversal Matching (LTM)

To tackle these challenges, we introduce Longest Traversal Matching (LTM), a multi-table caching scheme for SmartNICs in GIGAFLOW. LTM performs two key tasks: (1) selecting the correct matching rule within a cache table, and (2) ensuring that the chosen sub-traversals maintain the correct sequence of the original vSwitch traversal as the packet progresses through cache tables.

In Figure 3.7, a packet from the second traversal matches both rules in GF_1 since they share the same ETH headers and match both IP/24 and IP/32 prefixes. The challenge is determining which rule to select. Similarly, in GF_2, the packet could match the TCP destination port (dTCP), but this rule does not correspond to the original vSwitch traversal—thus leading to incorrect and inconsistent behavior.

LTM resolves these conflicts by selecting the rule associated with the sub-traversal that spans the most tables in the original vSwitch pipeline; hence, the name “Longest Traversal Matching.” This ensures that the packet matches the most specific rule from the original traversal. In our earlier example, the packet would match the second rule (ETH and IP/32) in GF_1 as it spans more tables.

```
table LTM_Table {
    key = {
        meta.table_tag : exact;     // Table Tag (τ)
        meta.src_port  : ternary;   // Ingress Port
        hdr.eth.smac   : ternary;   // ETH Src
        hdr.eth.dmac   : ternary;   // ETH Dst
        hdr.eth.type   : ternary;   // ETH Type
        hdr.ipv4.src   : ternary;   // IP Src
        hdr.ipv4.dst   : ternary;   // IP Dst
        hdr.ipv4.proto : ternary;   // IP Protocol
        meta.tp_src    : ternary;   // TCP Src
        meta.tp_dst    : ternary;   // TCP Dst
    }
    actions = {
        set_ethernet;               // Modify ETH Fields
        set_ip;                     // Modify IP Fields
        set_transport;              // Modify TCP Fields
        update_table_tag;           // Next Sub-Traversal
        forward;                    // Forward Pkt to Egress
        drop;                       // Drop Pkt
        NoAction;
    }
    size           = NUM_ENTRIES;
    default_action = NoAction;
}
```

Figure 3.8: An LTM table, as defined in P4 [16], performs an exact matches on tag τ while allowing wildcard matches on other header fields. Actions modify header fields, update the tag, and either forward or drop the packet. When a miss occurs, GIGAFLOW sends the packet to the userspace vSwitch pipeline.

To maintain the correct traversal sequence across cache tables, LTM appends a "table ID" to the match predicate. The table ID represents the expected next table in the original vSwitch traversal. At each cache stage, the packet must match both the rule and the expected table ID, filtering out sub-traversals that are not part of the original sequence. This approach ensures consistency—causing a miss in GF_2 but correctly selecting the second rule in GF_3 for its matching TCP source port.

LTM operates similarly to Longest Prefix Matching (LPM) [67] in packet routing: while LPM selects the longest prefix for precise routing, LTM prioritizes sub-traversals that span the most tables in the vSwitch pipeline. By focusing on longer matches, LTM ensures accurate alignment with the original traversal, enabling consistent and efficient packet processing within SmartNICs.

3.4.2.2 LTM on P4-programmable RMT Architecture

GIGAFLOW utilizes the P4-based RMT architecture [17] to implement Longest Traversal Matching (LTM) in SmartNICs, using RMT's rule priority and metadata features for efficient processing. LTM assigns priorities (ρ) based on sub-traversal length, with longer sub-traversals receiving higher priority. For example, in Figure 3.7, the first rule in GF_1 has a priority of

$\rho = 3$, while the second has $\rho = 4$. LTM also uses P4 metadata to manage the "table ID" (i.e., tag τ), which indicates the next expected table. When a packet belonging to the second traversal exits GF_1, τ updates to 9, directing it to skip GF_2 and proceed to GF_3. This design ensures accurate traversal sequences while operating at line rate in SmartNICs.

Figure 3.8 illustrates an LTM table definition in P4 [16]. All LTM tables share a homogeneous structure, performing an exact match on the tag τ (8-bit metadata) and ternary matches on additional fields, such as the ingress port and all five-tuple headers. LTM rules use these wildcarded fields and match them based on the LTM priority ρ. This uniform design enables any table in the SmartNIC to support flexible sub-traversal matching without being restricted to specific vSwitch pipeline stages or networking layers, preserving the programmability of the vSwitch pipeline.

3.4.3 Gigaflow vSwitch Processing

When a packet misses the Gigaflow cache in the SmartNIC, it is forwarded to the vSwitch for processing. The vSwitch then executes three main steps: (1) constructing a traversal based on the vSwitch pipeline and packet's flow signature (§3.4.3.1), (2) partitioning the traversal into sub-traversals (§3.4.3.2), and (3) generating the final LTM rules for caching in the SmartNIC (§3.4.3.3).

3.4.3.1 Building vSwitch Pipeline Traversal

To construct the traversal, the vSwitch gathers information as the packet moves through its pipeline (Figure 3.6b). It collects the incoming and modified outgoing flows (F^i), the sequence of table IDs (T_i) processed, and the matching wildcards (W_i).

The traversal is composed of three vectors: $\mathbf{T}$ for table IDs, $\mathbf{F}$ for modified flows after each lookup, and $\mathbf{W}$ for matched fields. Additionally, the vSwitch identifies available Gigaflow caches for mapping, represented as $\text{GF} = \{\text{GF}_k \mid \text{GF}_k \textsf{ is not full}, k \in [1, K]\}$. Thus, the complete traversal vector is defined as $< \mathbf{T}, \mathbf{F}, \mathbf{W}, \mathbf{GF} >$.

3.4.3.2 Generating Sub-Traversal Partitions

This stage takes the traversal vector $< \mathbf{T}, \mathbf{F}, \mathbf{W}, \mathbf{GF} >$ and partitions it into sub-traversals, selecting the configuration with the highest pair-wise disjointedness between adjacent sub-traversals. The key criterion driving this partitioning is the *disjointedness* between sub-traversals. Intuitively, placing sub-traversals that match on different header fields into separate cache tables maximizes the number of independent rule combinations, broadening coverage of the flow space. We define this property formally below.

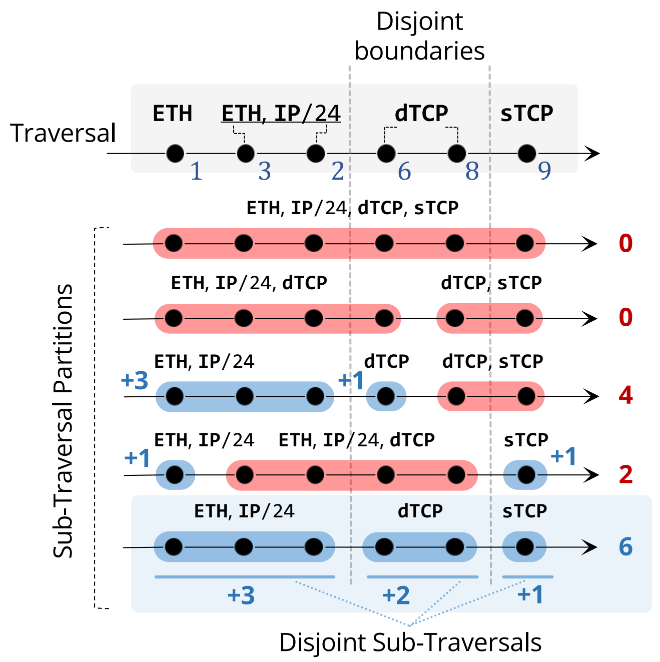


Figure 3.9: An example traversal showing disjoint field boundaries and various sub-traversal partitions. Partitions crossing the boundary (in red) receive a score of 0, while those within the boundary are scored by their length. The partition with the highest total score represents the most disjoint configuration.

> **❏ Disjointedness Property**
>
> *Two sub-traversals are considered disjoint if they have no matching fields in common. For example, if one sub-traversal matches on Ethernet headers while another matches on TCP ports, they are disjoint, as in Figure 3.7. When creating cache entries, it is best to place disjoint sub-traversals in separate tables and merge those that share matching fields (Figure 3.5). This approach maximizes the number of independent rule combinations, allowing broader coverage of flow space and improving cache efficiency, as we evaluate in §3.6.*

To illustrate sub-traversal partition generation and selection, Figure 3.9 shows potential partitions for the first traversal from Figure 3.7 that GIGAFLOW explores to find the optimal partition. The disjoint field boundaries in the figure separate ETH, IP/24, and TCP source/destination ports.

A disjoint set of sub-traversals can be seen as grouping vSwitch pipeline table rules with overlapping matching fields. In the optimal partition (the last one in Figure 3.9), the first sub-traversal matches Ethernet/IP headers, the second targets TCP destination ports, and the third handles TCP source ports. In contrast, sub-optimal partitions cross the disjoint field boundaries (highlighted in red), combining disjoint fields within the same sub-traversal.

This observation leads to an effective method for identifying optimal sub-traversals:

> ❏ **Partitioning Algorithm**
>
> *Explore all possible partitions and evaluate each sub-traversal for tables with overlapping fields. If fields overlap, assign a score equal to the sub-traversal's length (i.e., the number of tables it spans); if fields are disjoint, assign a score of 0. The total score for a partition is the sum of its sub-traversal scores, with the partition achieving the highest score selected as optimal. A dynamic program can do this exploration in $O(N \times K)$ time, N being the traversal length and K being the number of* Gigaflow *tables.*

The optimal partition has two key characteristics: (1) it maximizes disjointedness by separating distinct field sets and (2) it prioritizes longer sub-traversals, reducing the number of Gigaflow cache entries needed per traversal. In Figure 3.9, the optimal partition achieves a score of 6, compared to other alternatives. We further show in §3.6 that many real-world vSwitch pipelines (Table 3.1) offer similar opportunities for creating disjoint sub-traversals.

3.4.3.3 Creating and Installing LTM Rules

Gigaflow's partition generator produces one partition with up to K sub-traversals, each converted into an LTM cache entry. To create Gigaflow entries, the matching wildcard (ω_k) is generated by performing a bitwise union (OR) of wildcards (W_i) from all tables in the sub-traversal. The match predicate (M_k) is constructed through a bitwise intersection (AND) of this wildcard with the initial flow at the start of the sub-traversal. The actions (α_k) are defined by calculating the `commit`, which represents the differences between the initial flow (F^i) and the final flow after the sub-traversal, recorded as setfield actions in the LTM cache.

LTM sets the priority (ρ_k) of each sub-traversal rule based on its length (§3.4.2). The match predicate is updated to include an exact match on the table tag (τ_k), which corresponds to its starting vSwitch pipeline table ID (T_i). The actions also update the table tag to the ID of the next expected table, ensuring that packets follow the correct traversal sequence.

Managing vSwitch Pipeline Rule Dependencies. In Gigaflow, a cache hit must reflect the highest-priority traversal path in the vSwitch pipeline. To ensure this, Gigaflow combines LTM priorities (ρ) with additional field bits added to the wildcard of each cache entry, similar to Megaflow [121], preventing matches with higher-priority rules.

For instance, consider a packet destined for 192.168.21.27 with vSwitch rules prioritized by IP prefixes: (400: 192.168.14.15/32), (300: 192.168.14.0/24), (200: 192.168.0.0/16), and (100: 192.0.0.0/8). As the packet misses the first two rules but matches the third, Gigaflow adjusts the wildcard to account for the highest-priority rule that could still match, resulting in a wildcard of 255.255.240.0. The final cache entry becomes 192.168.21.27 AND 255.255.240.0

$\rightarrow$ 192.168.16.0/20. By adding these bits to the cache entry's wildcard, Gigaflow ensures that packets match only the target vSwitch rule for this flow, preventing any simultaneous matches with higher-priority rules and effectively blocking lower-priority hits.

3.4.4 Gigaflow Revalidation and Updates

Gigaflow handles cache revalidation through two mechanisms: (1) evicting entries when vSwitch pipeline rules are updated and (2) removing expired entries when incoming traffic no longer utilizes them.

3.4.4.1 Eviction Due to vSwitch Pipeline Rule Updates

When vSwitch pipeline rules change, Gigaflow revalidation identifies inconsistent entries without immediate rule pushes to the cache. Instead, it uses revalidation cycles, where Gigaflow picks the table tag (τ) of an entry, sends its parent flow to the corresponding vSwitch pipeline table, and validates it up to the length of its sub-traversal. If the actions and match predicate of the revalidated entry differ from the stored Gigaflow entry, it is marked for eviction.

3.4.4.2 Eviction Due to Timeouts

Gigaflow uses a *max-idle* parameter, similar to OVS [121], to remove stale entries. However, instead of evicting entire parent traversals, Gigaflow selectively evicts only stale sub-traversals. Caching sub-traversals enables faster revalidation, as sub-traversals are shorter than complete traversals, reducing revalidation overhead. This makes Gigaflow more efficient than traditional traversal caches like Megaflow in OVS (§3.6.3).

3.5 Implementation

We implement Gigaflow as a caching subsystem within the widely-used Open vSwitch (OVS) [121], adding support for both a software backend and SmartNIC offload. Using OVS [43] (`commit:4f933301`), we modified approximately 10,000 lines of code (LoCs) across 45 files, written in C/C++. About 30% of these changes are dedicated to Gigaflow's sub-traversal partitioning and rule-creation engine. Moreover, our software backend utilizes the OVS DPDK framework (`v21.11.0`) [40], with an additional 10% LoCs.

For the SmartNIC offload, we used production-grade Xilinx OpenNIC [8] implementation with NetFPGA-PLUS [147] (`commit:f03b61c1`) with the P4-programmable FPGA-based SmartNIC—Alveo U250—as the hardware backend. The Xilinx Alveo series are FPGA-powered SmartNICs optimized for datacenter acceleration [172, 169]. We wrote 350 lines of P4 [16] code, compiling it to Verilog using the P4SDNet toolchain (`v2020.1`) [9] and Vivado

SDK (`v2020.2`) [6], provided by Xilinx, to create GIGAFLOW's caching pipeline with four match-action tables (MATs). Each table performs ternary matches on ten standard network header fields (e.g., Ethernet, IP, TCP/UDP) and includes an exact-match field for the table tag (τ), along with priorities (ρ) and actions (e.g., to modify the tag).

Overall, our SmartNIC implementation utilizes 47% of the FPGA's lookup tables (LUTs), 33% of flip-flops (FFs), and 49% of on-chip memory (BRAM/URAM). The setup consumes 38 W on-chip power and is synthesized for 100 G line-rate performance. These numbers are reported from the Xilinx Vivado SDK (v2020.2) post-implementation thermal and power report (junction temperature set at 95 °C, typical for SmartNICs [111, 134]). The deviation from tapeout is within ±25% [168]. Finally, to update SmartNIC tables at runtime, we incorporated PCIe read/write routines—generated by P4SDNet for our P4 program—into the OVS offload APIs (i.e., `queue_netdev_flow_put` and `rte_flow`) [43, 40], taking around 700 LoCs.

3.6 Evaluation

We evaluate GIGAFLOW's end-to-end cache efficiency (e.g., forwarding latency, hit rate, cache misses) and performance (§3.6.2) and conduct microbenchmarks (§3.6.3) to ascertain the impact of its various design choices (e.g., number of tables, partitioning algorithm) on its performance under various scenarios.

3.6.1 Experiment Setup

3.6.1.1 Testbed Environment

Our testbed consists of two servers connected back-to-back, serving as a device under test (DUT) and a traffic generator, respectively. These servers are equipped with a 64-core Intel Xeon Platinum 8358P CPU @ 2.60 GHz with 512 GB RAM, running Proxmox VE (`v8.0.4`) [123]. Each of these hosts a dual-port Intel XL710 10/40G NIC as well as Nvidia's Connectx-6 100G NIC and a Bluefield-2 DPU (with ARM SoC). The DUT server also comes equipped with a Xilinx Alveo U250 FPGA for testing our GIGAFLOW prototype. All these NICs and FPGAs are connected using a P4 Tofino-1 switch [59] to connect them depending upon the configured baseline implementations.

3.6.1.2 Baseline Configurations

Our baselines include various configurations of Open vSwitch (OVS) drawn from the standard industry deployments [110, 98, 97]: OVS/Kernel, OVS/DPDK, and OVS/Megaflow-Offload; including our GIGAFLOW implementation, OVS/GIGAFLOW-Offload. We evaluate the OVS

Pipeline	Description	Tables	Traversals
OFD	OpenFlow Data Plane Abstraction (OFDPA) [103] provides integration of HW/SW switches in CORD.	10	5
PSC	A combined L2/L3 forwarding and ACL enforcment OVS pipeline as used for benchmarking in Pisces [137].	7	2
OLS	OVN logical switch [112] manages virtual network topologies with logical segments using OVS.	30	23
ANT	Antrea [10, 11, 12] implements networking and security policies using OVS for a Kubernetes cluster.	22	20
OTL	OpenFlow Table Type Patterns (TTP) [105] for configuring L2L3-ACL policies in OVS.	8	11

Table 3.1: Real-world Open vSwitch (OVS) pipelines, depicting vSwitch tables and unique traversals.

Kernel and DPDK baselines on both the host CPU and the ARM SoC on the Nvidia DPU, while the Megaflow and GIGAFLOW offloads are tested using the Alveo U250 FPGA. Additionally, we examine Megaflow and GIGAFLOW performance using two different search algorithms: Tuple Space Search (TSS) and Neuvomatch (NM) [125, 126]. We provision a single CPU core for vSwitch software processing for all configurations, but the trend remains consistent with more CPU cores (Section 3.6).

3.6.1.3 Real-World vSwitch Pipelines

Table 3.1 lists the real-world pipelines used in our evaluations, each consisting of a few to dozens of tables and varying numbers of unique traversals. Using these pipelines, we analyze GIGAFLOW's ability to capture pipeline-aware locality and the potential for sub-traversal sharing in real-world scenarios.

3.6.1.4 Multi-Table Rulesets and Traffic Generation

We develop a tool, Pipebench, which uses real-world pipelines (Table 3.1) to generate multi-table rulesets and traffic traces, featuring both high and low locality traffic (e.g., with varying access patterns to shared sub-traversals). To accomplish this, we use Classbench [143], that provides a comprehensive list of rules with fully-populated wildcard header fields for Firewall, ACL, and IPSec policies in datacenter environments. To generate the multi-table ruleset, we first randomly select a traversal that includes tables and their matching fields for a specific pipeline. Next, we randomly choose a rule from Classbench and map its fields to each table in the traversal. This process is repeated to generate the complete ruleset.

Similarly, we sample rules from Classbench to generate two different traffic patterns using the CAIDA [23] traffic characteristics (i.e., flow sizes and inter-packet gaps) [126]. The first pattern is created by uniformly selecting rules and modifying the CAIDA packet

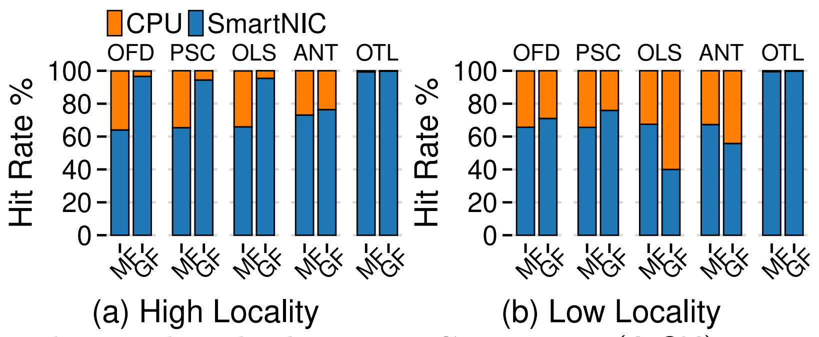


Figure 3.10: End-to-end cache hit rate: GIGAFLOW (4x8K) vs. Megaflow (32K) in high/low locality environments.

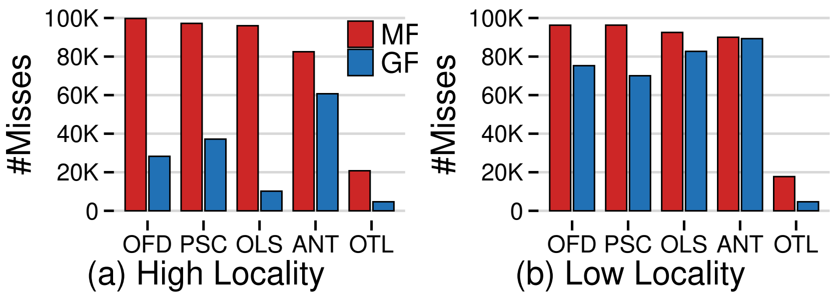


Figure 3.11: End-to-end cache misses: GIGAFLOW (4x8K) vs. Megaflow (32K) in high/low locality environments.

headers accordingly to reflect *low-locality traffic*. The second pattern selects rules based on the recurring frequency of a header tuple (Figure 3.5), producing *high-locality traffic* that increases the opportunity for sharing sub-traversals between flows.

3.6.2 End-to-End Analysis

3.6.2.1 Cache Behavior: Hit Rates, Misses, and Entries

In high-locality environments, GIGAFLOW consistently outperforms Megaflow across all five vSwitch pipelines (Table 3.1), achieving up to 51% higher hit rates (25% on average, Figure 3.10) and reducing cache misses by up to 90% (64% on average, Figure 3.11). These improvements stem from sub-traversal sharing, which leverages pipeline-aware locality. For instance, in the PSC pipeline, partitioning traversals by separating L2 (Ethernet) processing from ACL (IP/TCP) creates more sharing opportunities, resulting in higher hit rates. The extent of sub-traversal sharing depends on both the pipeline's design and the nature of incoming traffic (Figure 3.13). In low-locality traffic, with fewer sharing opportunities, hit rates decrease, and cache misses increase. However, GIGAFLOW still maintains performance comparable to Megaflow in pipelines, such as OFD, PSC, and OTL.

Sub-traversal sharing also affects cache utilization. In high-locality traffic, Megaflow uses

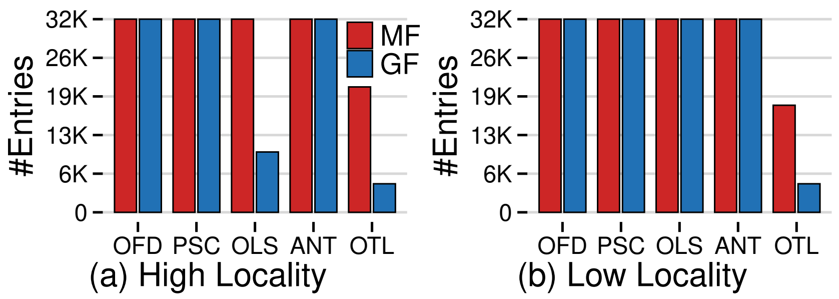


Figure 3.12: End-to-end cache entries: GIGAFLOW (4x8K) vs. Megaflow (32K) in high/low locality environments.

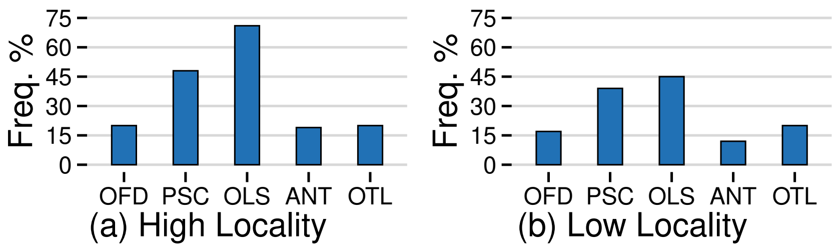


Figure 3.13: Frequency of sub-traversals reoccurring in GIGAFLOW (4x8K).

93% of available cache space, whereas GIGAFLOW uses only 76% on average (Figure 3.12). Even in low-locality environments—the worst-case scenario for sharing—GIGAFLOW still maintains comparable performance to Megaflow. Although sharing frequency decreases by an average of 25% (Figure 3.13), GIGAFLOW remains competitive. For OLS and ANT, the reduced sharing opportunities and higher cache usage slightly diminish GIGAFLOW's advantage over Megaflow in low-locality settings.

3.6.2.2 System Performance: Latency and CPU Usage

In high-locality environments, GIGAFLOW achieves significant reductions in average per-packet latency compared to Megaflow, with improvements of 29.14% for the OLS pipeline, 31% for OFD, and 27% for PSC (Figure 3.14). Although both GIGAFLOW and Megaflow offloads, using our FPGA-based SmartNIC, have the same hardware cache hit latency of about 9 μs, GIGAFLOW's higher cache hit rate minimizes the need to traverse the vSwitch multi-table pipeline, resulting in lower overall latency. In low-locality scenarios, where cache hits are less frequent, the latency improvements are more modest—6% for PSC, 1.67% for OFD, and 0.02% for OTL. Larger pipelines, like OLS and ANT, see higher latencies due to increased cache misses and the overhead associated with vSwitch processing, including pipeline lookups, sub-traversal partitioning, and LTM rule generation.

The CPU usage breakdown for vSwitch processing (Figure 3.15) reveals that larger pipelines, such as OLS and ANT, experience additional overhead when using GIGAFLOW; the

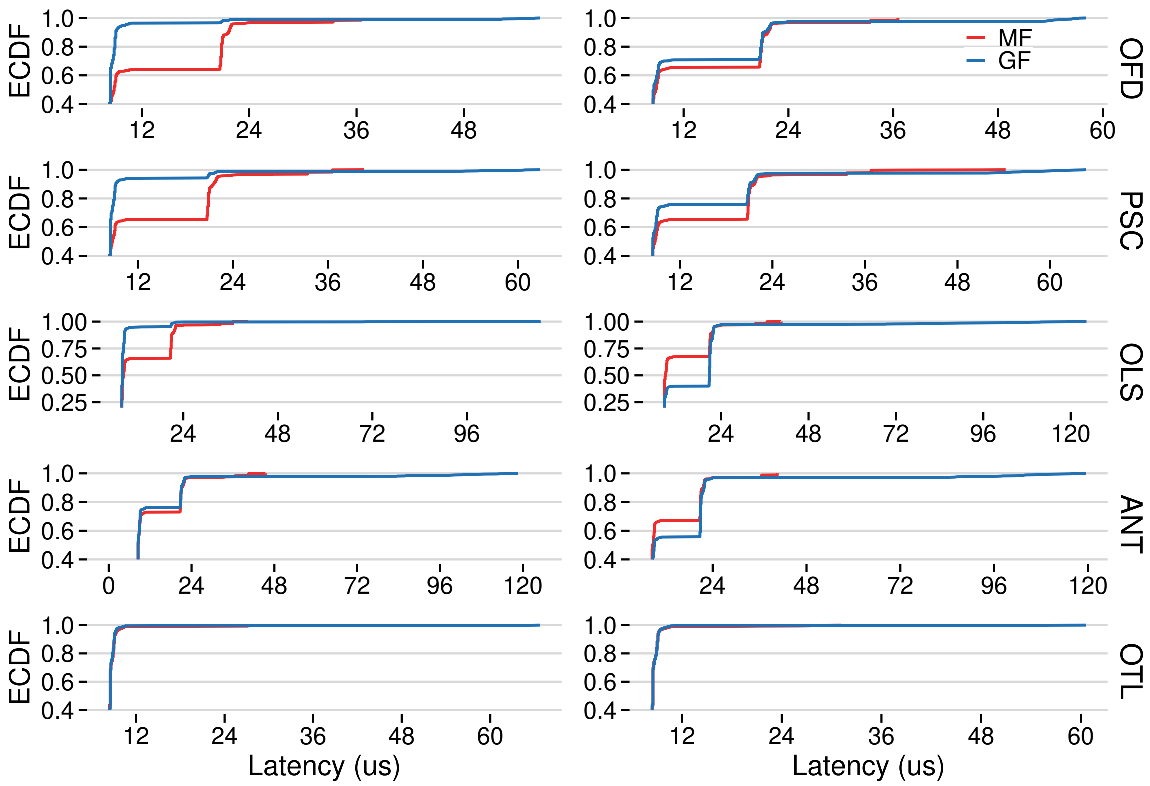


Figure 3.14: End-to-end latency: GIGAFLOW (4x8K) vs. Megaflow (32K) in high/low locality environments.

sub-traversal partitioning and LTM rule generation add 80% and 68% more processing time on top of the userspace forwarding pipeline, which Megaflow does not incur. In contrast, smaller pipelines like PSC, OTL, and OFD exhibit lower overheads of 28%, 23%, and 20%, respectively. Despite the added complexity, the substantial reduction in cache misses helps GIGAFLOW offset these overheads, particularly in high-locality settings (Figure 3.14).

3.6.3 Microbenchmarks

• **GIGAFLOW's performance improves with the addition of more SmartNIC tables, leading to fewer cache misses and reduced cache entries.** As shown in Figures 3.16 and 3.17, increasing the number of SmartNIC tables from 1 (Megaflow) to 5, each containing 100K entries, enhances GIGAFLOW's performance in both high- and low-locality environments. Different pipelines benefit from varying numbers of tables—OFD fully utilizes its disjointedness with just two tables, while PSC sees gains up to three tables. Larger pipelines, like OLS, continue to benefit up to four tables, where additional partitioning opportunities arise. However, adding more tables also increases vSwitch processing overhead, as the number of potential sub-traversal partitions to explore grows. Still, the overhead remains within 200 μs even for the larger pipeline, like OLS and ANT.

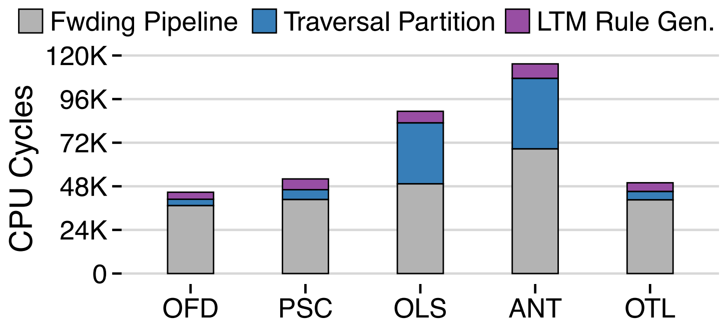


Figure 3.15: Average CPU cycle breakdown of vSwitch processing elements: vSwitch forwarding pipeline, sub-traversal partitioning, and LTM rule generation.

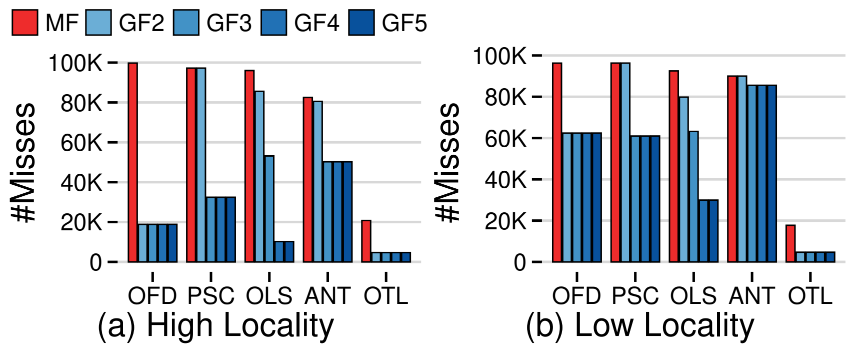


Figure 3.16: Cache misses with increasing number of GIGAFLOW tables (2–5): a 100K entry limit per table.

• **GIGAFLOW achieves orders of magnitude more rule space coverage than Megaflow with the same number of cache entries.** Using 4x8K tables, GIGAFLOW captures up to two orders of magnitude more rule space across all pipelines compared to Megaflow with 32K entries (same total cache size), Table 3.2. This is enabled by GIGAFLOW's shared sub-traversal caching, which allows cross-product rule combinations across multiple tables, vastly increasing coverage. For example, pipelines like OFD, OLS, and PSC see 459×, 337×, and 156× greater rule space coverage, respectively. Even pipelines with less partitioning potential, such as ANT and OTL, show improvements of 40× and 1.5×.

• **GIGAFLOW's disjoint partitioning (DP) achieves performance close to an ideal 1-1 mapping approach, but with 2.8× fewer cache entries** As shown in Figure 3.18, using the OLS pipeline with 100K unique flows and GIGAFLOW (4x8K), a random partitioning approach (RND) reduces cache misses by 11% compared to Megaflow, while consuming the entire cache. In contrast, disjoint partitioning (DP) reduces cache misses by 89% while using just 31% of the cache entries. We also compare against an ideal approach (1-1 mapping), where we assume each table in the traversal has a corresponding table in the SmartNIC. With this 1-1 mapping, cache misses are reduced by 94%, slightly better than disjoint partitioning, but consumes 2.8× more entries.

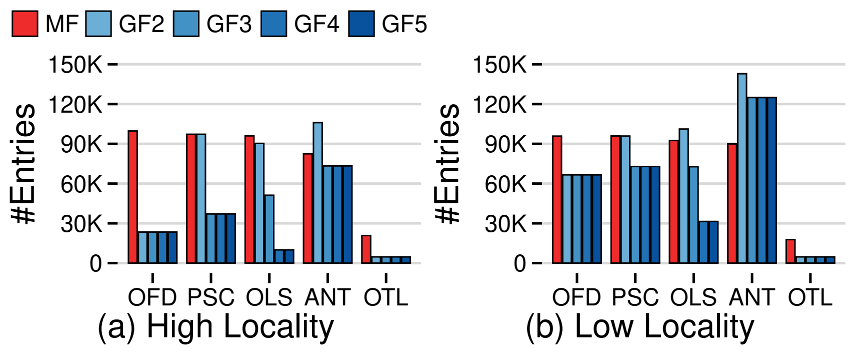


Figure 3.17: Cache entries with increasing number of GIGAFLOW tables (2–5): a 100K entry limit per table.

	OFD	PSC	OLS	ANT	OTL
Megaflow	32K	32K	32K	32K	32K
GIGAFLOW	14.7M	4.9M	10.8M	1.3M	48K

Table 3.2: GIGAFLOW (4x8K) vs. Megaflow (32K) maximum rule-space coverage with high-locality.

• **GIGAFLOW outperforms software cache optimizations through its extensive rule-space coverage in the SmartNIC.** We evaluate GIGAFLOW (4x8K) against Megaflow (32K) using two software cache search algorithms—Tuple Space Search (TSS) [121, 140] and Nuevomatch (NM) [125, 126]—on the PSC pipeline with a 100K unique flow traffic trace. As shown in Figure 3.19, in high-locality settings, Megaflow with NM reduces average latency from 13.4 μs to 12.5 μs (a 6.8% improvement over TSS). In contrast, GIGAFLOW with TSS achieves 9.8 μs latency—21.6% faster than Megaflow with NM. Adding NM to GIGAFLOW yields a slightly further reduction to 9.65 μs.

• **GIGAFLOW maintains a high hit rate with dynamically arriving workloads.** We evaluate GIGAFLOW (4x8K) against Megaflow (32K) on the PSC pipeline with dynamically arriving workloads; the vSwitch rules and the incoming traffic patterns (e.g., arrival times) remain the same, as new traffic flows arrive. We test using 100K unique flows divided across two workloads (50K each) in a high-locality environment. At time 5 min (Figure 3.20), we introduce the second workload, which causes Megaflow's hit rate to drop drastically from 84% to 61.18%, while GIGAFLOW sustains its performance at 93.26% due to significantly higher rule space coverage (Table 3.2). In a low-locality scenario, both caches perform similarly.

• **GIGAFLOW achieves the lowest latency and $2\times$ faster cache revalidation compared to OVS baselines.** GIGAFLOW offers the lowest cache hit latency among all OVS configurations, with both OVS/GIGAFLOW-Offload and OVS/Megaflow-Offload achieving 8.62 $\pm$ 0.4 μs. In contrast, OVS/DPDK on a host CPU has a higher latency of 12.61 $\pm$ 1.1 μs,

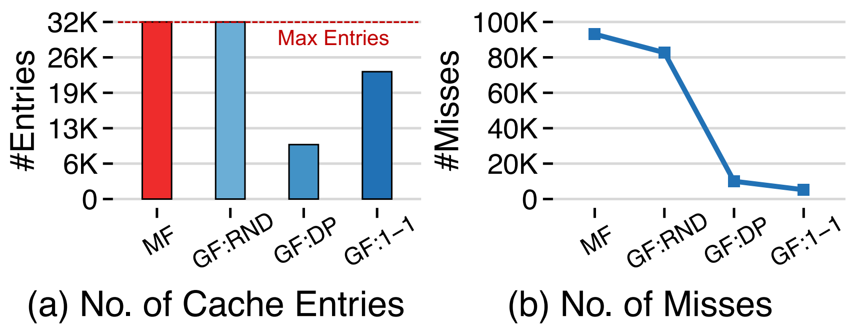


Figure 3.18: Comparing GIGAFLOW (4x8K) with partitioning schemes: Random (RND), Disjoint Paritioning (DP), and 1-1 mapping (1-1), using the OLS pipeline.

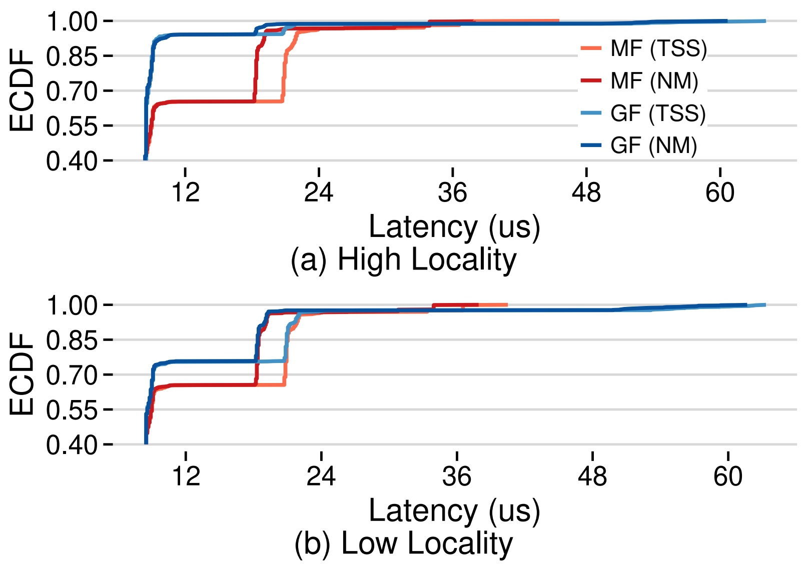


Figure 3.19: Comparison of Megaflow and GIGAFLOW with two search algorithms: original Tuple Space Search (TSS) [121, 140] and Nuevomatch (NM) [125, 126].

while the Bluefield-DPU's ARM cores perform even slower at 51.26 $\pm$ 9.7 μs. OVS/Kernel on both the host and Bluefield-DPU show much higher latencies, at 671.48 $\pm$ 13.4 μs and 3,606.37 $\pm$ 237.1 μs, respectively. Additionally, GIGAFLOW revalidates caches twice as fast as Megaflow. For instance, revalidating the Megaflow cache (32K entries) with the OLS pipeline takes 527 ms, while GIGAFLOW (4x8K) completes the task in 272 ms, about 2$\times$ faster.

• **GIGAFLOW cache performance scales proportionally with increasing CPU cores.** As data centers become more conservative in CPU allocation for infrastructure tasks like vSwitch processing—often restricting it to a single core—we explore how GIGAFLOW performs with multiple CPU cores. OVS allows per-CPU distribution of cache misses from the SmartNIC using RSS, balancing misses across cores. Figure 3.21 shows that, like Megaflow, GIGAFLOW reduces cache misses per core proportionally with increased cores. However, GIGAFLOW achieves this with a lower total CPU load across different pipelines.

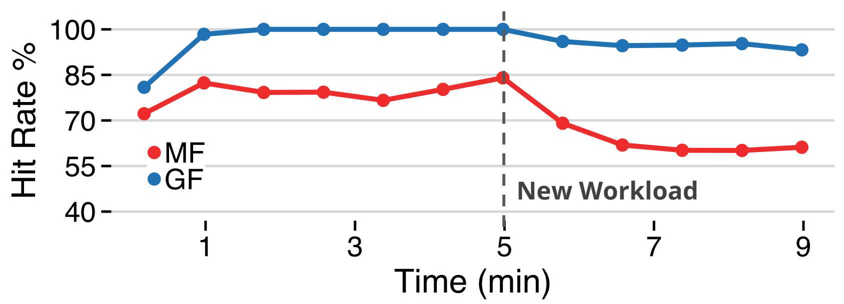


Figure 3.20: End-to-end cache hit rate with dynamic workloads: Gigaflow (4x8K) vs. Megaflow (32K) operating in a high-locality environment; a new workload with 50K unique flows is introduced at time 5 min.

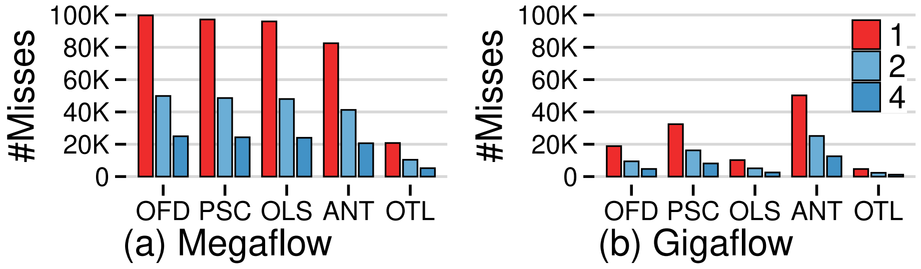


Figure 3.21: As the number of CPU cores for vSwitch processing increases, Gigaflow achieves performance gains similar to Megaflow.

3.7 Limitations & Future Work

Traffic-Profile-Guided Optimizations. In low-locality environments, Gigaflow may underperform compared to Megaflow since it depends solely on vSwitch pipelines to identify sub-traversal sharing opportunities. To address this, Gigaflow can use profile-guided optimization by periodically sampling packets from the SmartNIC to assess potential sub-traversal sharing in the traffic. If sharing opportunities are limited, Gigaflow could switch to using Megaflow entries in the cache, maintaining baseline performance. This adaptive fallback would ensure that Gigaflow never degrades below Megaflow performance regardless of the traffic environment.

Alternative Methods for Sub-Traversal Partitioning. Gigaflow employs a disjoint partitioning (DP) algorithm to divide traversals based on field-level separation. While our results show that DP incurs up to 2× the processing cost over traditional approaches, this overhead depends on both the number of Gigaflow tables and the traversal length. With approaches like Nuevomatch [125, 126] showing benefits of machine learning in network processing, we could incorporate it in finding more efficient Gigaflow mappings (e.g., by optimizing traversal partitioning based on traffic patterns). Such learned optimizers could also jointly consider traffic locality and pipeline structure to produce partitionings that better amortize the mapping cost at runtime.

3.8 Related Work

3.8.1 Software Flow Caching

Tuple Space Search (TSS) [140] is widely used for implementing lookups in vSwitches for both OpenFlow tables and caches [121], operating with a cost of $\mathsf{O(M)}$, where M represents the number of unique masks. SoftFlow [62] extended OVS by integrating middlebox functions, such as load balancing and stateful firewalls, allowing complex software actions and multi-stage processing via packet re-injection between the cache and vSwitch pipeline (i.e., slow path). Nuevomatch [125, 126] introduced RQ-RMI models [74], which use small perceptron trees to predict the lookup index with bounded error, lowering the lookup cost to $\mathsf{O(1)}$, but without affecting the cache miss volume. While these works focus on accelerating cache lookups, none address the structural inefficiency of how cache entries are generated from pipeline traversals. Instead, GIGAFLOW focuses on optimizing the cache misses.

3.8.2 Systems for Cache Management

OVS [121] prevents rule overlaps by adding extra bits to the Megaflow wildcard, ensuring unique match predicates for each traversal, allowing a single Megaflow cache entry per traversal. PipeCache [174] employs multi-staged hardware TCAMs, using simplified OpenFlow models to offload popular rules via a 1-to-1 mapping. Elixir [157] dynamically adjusts caching, adapting to predicted flow burstiness by keeping elephant flows in hardware and mice flows in software. However, these systems treat each cache entry independently and do not consider how traversals across pipeline stages share structural patterns. As a result, their per-entry coverage remains limited, leaving significant caching potential unexploited under diverse traffic conditions. These approaches manage *which* rules to cache but do not exploit the structure of pipeline traversals to increase per-entry coverage. In contrast, GIGAFLOW optimizes disjoint sub-traversal caching on SmartNICs, enhancing cache efficiency while sustaining line rate.

3.8.3 Cache Eviction Strategies

S3-FIFO [175] proposed a three-queue FIFO mechanism for cache eviction in software caches like Memcached, demonstrating better performance than LRU under high single-access loads. Inspired by S3-FIFO, SIEVE [177] developed a single-queue eviction strategy optimized for web cache workloads, further improving efficiency in managing software caches. Both approaches rely on access frequency and recency signals to guide eviction, making them well-suited to general-purpose caching scenarios. Yet neither accounts for the semantic relationships between cached entries, meaning that evicting one entry does not inform decisions about structurally related entries. While both schemes improve eviction decisions, they are agnostic to the structure of the cached rules and do not reduce the rate at which misses occur. In contrast, GIGAFLOW optimizes disjoint sub-traversal caching within SmartNICs, enhancing cache efficiency and supporting line-rate processing.

3.9 Conclusion

We presented GIGAFLOW, a multi-table sub-traversal cache architecture that leverages pipeline-aware locality to efficiently capture significantly larger rule spaces on SmartNICs. By leveraging disjointedness and optimizing cache storage, GIGAFLOW improves cache hit rate by up to 51% (average 25%) and reduces CPU-bound cache misses by up to 90% (average 64%) compared to Megaflow cache. GIGAFLOW delivers these improvements without compromising the line rate. We believe GIGAFLOW paves the way for more effective SmartNIC deployments, offering a scalable approach to manage and accelerate network processing in emerging cloud and AI data centers.

CHAPTER 4

KAIRO: Incremental Cache Eviction for Modern vSwitches

“

Effective incremental processing of a stream of updates, where the processing involves iterative computation and looping constructs is a tough problem. “For example, no previously published system can maintain in real time the strongly connected component structure in the graph induced by Twitter mentions.”

ADRIAN COLYER (THE MORNING PAPER [88, 90])

4.1 Introduction

To support emerging network virtualization use cases [73, 89, 148], vSwitches have evolved into multi-table packet pipeline processors [121, 149, 126], enabling intricate control flows for complex network policies [121, 182]. To balance this flexibility with high performance, vSwitches introduced the exact-match Microflow cache [121, 149, 126, 36, 37] as a fast-path layer, relegating full pipeline evaluation to a slow path for cache misses. The Megaflow cache [121, 149, 126] later reduced misses via wildcarding, and NuevomatchUP [126] further improved lookup speed by replacing Tuple Space Search (TSS) with ML-based learned index models [125, 126]. More recently, Gigaflow [182, 180] proposed a multi-table SmartNIC cache that exploits pipeline-aware locality to improve hit rates beyond Megaflow. Overall, this progression highlights a steady shift towards vSwitch architectures built around a highly optimized, cache-centric fast path, increasingly accelerated by specialized optimizations.

As caching becomes central to vSwitch performance, managing cache state—particularly the eviction of invalid entries—has emerged as a key challenge [121, 66] (Figure 4.1). Cache-related tasks, such as removing idle (but valid) entries to reclaim space and propagating fast-path statistics back to slow-path rules and counters, are latency-tolerant and can be

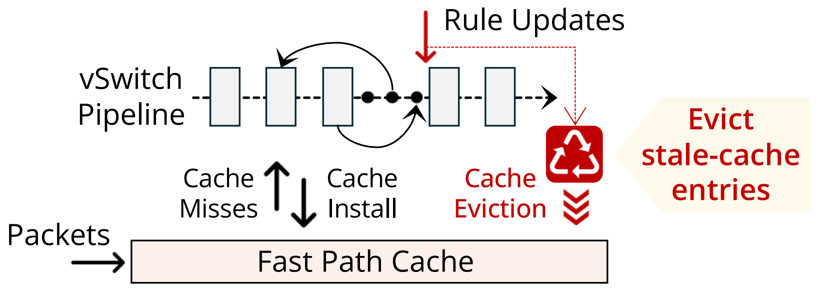


Figure 4.1: vSwitch cache eviction: rule updates invalidate cache entries which must be removed as quickly as possible to avoid stale cache processing.

performed periodically without affecting correctness. In contrast, evicting stale entries—those invalidated by recent rule updates—is critical for correctness and remains a major scalability bottleneck, especially at larger cache sizes, where such stale entries directly impact overall performance [182, 126, 184].

Evicting stale cache entries upon vSwitch rule updates is a fundamental and non-trivial challenge across all cache-based vSwitch architectures. These caches [121, 149] store *traversals* [182]—complete sequences of table lookups through the vSwitch pipeline—as individual cache entries. Because rules in the vSwitch pipeline are both wildcarded and prioritized, a single rule update in the slow path can invalidate multiple and arbitrary cache entries, making it difficult to precisely identify which entries are stale. In practice, vSwitch policy updates occur frequently enough (as shown in Table 4.3) that the eviction cost becomes critical—not just to maintain correctness, but also to sustain overall performance.

Naively re-evaluating the full flow space (Figure 4.2a, Bruteforce) is prohibitively expensive due to combinatorial growth, so instead, modern vSwitches limit the scope by re-evaluating only the cached flows [121]. This approach re-evaluates each cached entry and re-executes their traversal through the slow-path pipeline to detect and evict stale entries *iteratively* in $O(E)$ time [121, 184], where E is the number of installed cache entries (Figure 4.2b).

While iterative eviction eventually restores correctness, it introduces a variable window during which stale cache entries may still be present; this window grows with both the cache size and the depth of the vSwitch pipeline (i.e., traversal length). The impact of this delay grows proportionally at high link rates (e.g., 800 Gbps+) and under modern update patterns [182, 126] (see Table 4.3), where slow eviction can extend periods of inconsistency [121, 66], trigger incorrect routing decisions [141, 121], waste CPU cycles and bandwidth, and delay responses to network events [141, 176]. To mitigate these effects, many vSwitches cap the cache size (e.g., 200K entries [121, 43, 126, 182]) and dynamically shrink it at runtime if eviction takes longer than one second [66]—a trade-off that limits cache capacity to keep eviction overhead manageable, without compromising performance.

In this chapter, we introduce Kairo, which reimagines cache eviction as an instance of

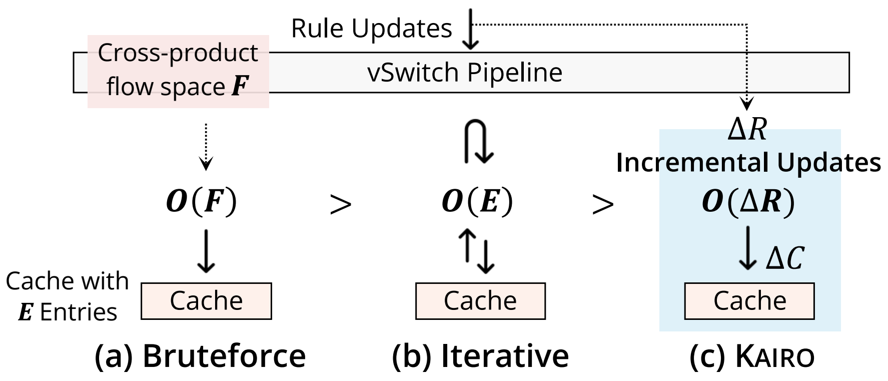


Figure 4.2: Cache eviction schemes: bruteforce recomputation grows with flow space F, iterative revalidation [121] with cache entries E, and KAIRO's incremental approach with update size ΔR, where $\Delta R \ll E \ll F$.

Incremental View Maintenance (IVM)—a technique widely used in database systems [86, 20, 41, 2, 52, 54, 53, 72]—by treating the vSwitch cache as a materialized view over the slow-path rule tables. Instead of iteratively reprocessing the entire cache on every rule update, KAIRO *incrementally* identifies just the cache entries affected by the update and evicts them (Figure 4.2c). This design decouples the eviction cost from cache size, enabling faster responses to the rule changes.

In database systems, a materialized view $V = Q(DB)$ represents the result of a query Q over a database DB. Instead of re-evaluating the entire query Q after every update to DB, Incremental View Maintenance (IVM) [86, 41] computes and applies only the change ΔV resulting from the update ΔDB.

Drawing on this model, we treat the vSwitch rule tables as the database and the cache as a materialized view of the forwarding pipeline. This framing is particularly effective for vSwitches, where rule updates typically modify only a small number of entries (tens to hundreds; see Table 4.3), while the cache may contain millions of flow entries [121, 182, 126], i.e., $\Delta R \ll E$. By targeting only these incremental changes, KAIRO performs cache eviction proportional to the number of updates, scaling far more efficiently than approaches that iterate over the entire cache.

To realize this, KAIRO builds on DBSP [20, 34, 41], a language and execution model for IVM with relational semantics. DBSP expresses queries as stateful dataflow circuits that continuously maintain results in response to streaming updates, making it well-suited for pipelines with frequent rule changes. It enables KAIRO to model rule tables as input streams and cache entries as output streams, forming a principled substrate for efficient cache eviction.

KAIRO implements a pipeline of incremental circuits that mirror the vSwitch rule tables. Rule insertions and deletions, as well as slow-path packets that lead to new cache entries, are treated as inputs to these circuits, while the resulting invalidated cache entries are emitted as output streams. When a rule update occurs, only the circuits for the affected tables are activated. Incremental lookups then determine the specific (`packet,matched_rule`) combinations affected by the update and evict the corresponding stale entries.

To minimize time-to-eviction (TTE) and reduce stale-cache processing, Kairo introduces novel early-exit *gates*: lightweight, stateful primitives that detect when an update can no longer affect downstream circuits (i.e., tables) and stop further computation. When a gate is closed, it holds its local state and delays propagation until a configurable timer expires, ensuring eventual convergence and consistency. This mechanism reduces immediate computation and postpones non-critical work, so that eviction cost scales with the actual impact of an update rather than with cache size or pipeline depth. Together, DBSP's incremental model and Kairo's gated, controlled execution enable efficient cache eviction—without re-evaluating the entire cache.

To realize incremental cache eviction, we make the following contributions:

- We formulate vSwitch cache eviction as an incremental view maintenance (IVM) problem and build Kairo, which shifts eviction cost from cache size to rule update size. Kairo models vSwitch table lookups as dataflow circuits using DBSP [20, 41], extending them to support incremental eviction via early-exit gates and propagation timers.

- We integrate Kairo into OVS [121, 43], demonstrating 18× faster updates on average, 35.5× reduction in stale-cache traffic (up to 130×), and support for millions of entries without degrading end-to-end performance.

- Kairo's codebase and documentation is available as open source (Appendix §B).

4.2 Background & Motivation

We provide a brief recap of vSwitch architecture evolution (see §2.2), followed by an overview of cache eviction in vSwitches and incremental view maintenance.

4.2.1 Evolution of vSwitch Architecture

Executing a full multi-table vSwitch pipeline in software is prohibitively expensive [121, 184, 149, 126, 182]. Each packet must undergo multiple dependent match-action lookups to enforce policies like L2 forwarding, L3 routing, and ACLs. This design places significant pressure on general-purpose CPUs [126, 182], which see sharp throughput degradation as pipeline depth increases [121].

To balance flexibility and performance, vSwitches adopted a bifurcated architecture: a slow path and a fast path [121, 108, 184, 36, 37, 149, 83]. The slow path processes packets through the full rule pipeline and installs matching rules as cache entries. The fast path then handles subsequent packets using a single lookup against this cache. This division amortizes policy evaluation across packets, marking a shift toward cache-centric vSwitch designs that prioritize throughput.

4.2.1.1 vSwitch Caches

Early fast-path designs relied on the exact-match Microflow cache [121, 149, 126], which exploits temporal locality by caching slow-path decisions on a per-flow basis. Subsequent packets in the same flow can bypass the full pipeline and hit the cache directly.

While Microflow caching is effective for stable flows, it performs poorly under diverse or bursty traffic patterns [121]. To address this, later designs introduced Megaflow caches [121, 149, 126], which generalize across flows by wildcarding irrelevant header fields—capturing spatial locality and boosting hit rates by aggregating similar flows.

This cache-centric philosophy appears across production-grade vSwitches. For instance, Microsoft VFP uses a Unified Flow Table (UFT) [36], while Gigaflow [182, 42] extends the concept to multi-table SmartNICs, exploiting pipeline-aware locality to improve both cache hit rates and throughput.

4.2.1.2 vSwitch Lookup Algorithms

On the slow path, packet classification within vSwitch tables is commonly performed using Tuple Space Search (TSS) [140, 121]. TSS organizes rules into tuple spaces—groups of rules that share the same wildcard pattern—and orders these spaces by the highest rule priority they contain.

Lookups proceed by scanning tuple spaces in descending priority order. If a match is found, it is tentatively selected, and the search continues only if remaining tuple spaces might contain higher-priority rules. If such a rule is found, it replaces the earlier match. The lookup terminates when no further tuple space can offer a better match.

TSS performs well when the number of wildcard patterns (i.e., tuple spaces) is small. However, as wildcard diversity increases, TSS becomes costly, prompting recent work to improve lookup performance using optimized or ML-based indexing. For example, Nuevo-matchUP [125, 126] uses a learned Range Query Recursive Model Index (RQ-RMI) to accelerate classification by learning how to predict rule matches more efficiently than traditional tuple-based search.

4.2.2 Cache Evictions in vSwitches

As caching has become central to vSwitch performance, managing the cache—especially evicting invalid entries—has emerged as a key challenge [121, 66] (Figure 4.1). Flow caches

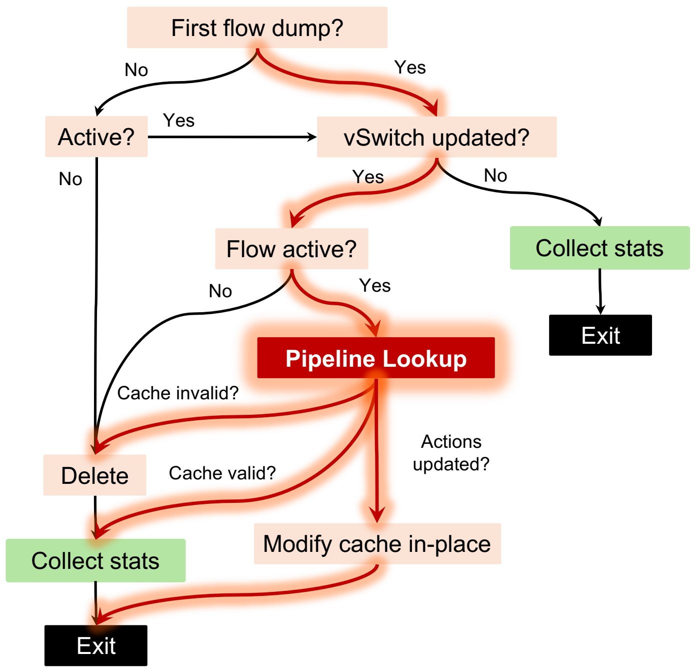


Figure 4.3: Iterative cache revalidation as implemented in the Open vSwitch (vSwitch) [184, 66]: vSwitch rule trigger the revalidation and cache entry is re-evaluated along the path highlighted with red arrows.

must be continuously updated to reflect changes in network policies, resource availability, and network topology (Table 4.3). When rules in the slow-path pipeline are modified, deleted, or reordered, some previously cached entries may no longer represent correct forwarding decisions. These outdated entries must be evicted to maintain correctness.

Not all cache maintenance operations are equally time-sensitive. For example, evicting idle (but still valid) entries to free space, or pushing fast-path statistics back to the slow path, are non-critical [121, 43, 66]. These tasks are performed periodically and do not affect packet forwarding correctness [121, 66]. In contrast, evicting stale cache entries—those invalidated by recent rule updates—is correctness-critical. Until these entries are removed, packets may be forwarded using outdated policies, causing misrouting, policy violations, or dropped traffic [183, 141, 176]. We define the latency between a rule update and the removal of all affected cache entries as the **time-to-eviction (TTE)**—a key but often overlooked performance metric in vSwitches.

High TTE leads directly to stale cache processing, where packets continue to match entries that should have been invalidated. Unlike cache misses, which simply trigger slow-path processing, stale cache hits actively result in incorrect behavior. Worse, they can persist for extended periods at scale. As caches grow and wildcard rules span broader sections of the flow space, the amount of stale traffic can become substantial—hurting both performance and correctness [182, 126, 184].

Today, vSwitches rely on an *iterative revalidation* scheme to evict stale cache entries (Figure 4.2b). This approach scans cached entries one by one, checking each against the updated rule set [121, 66], as shown in Figure 4.3. Because revalidation traverses the entire cache, its cost grows with cache size and is highly sensitive to update patterns. Moreover, it must be throttled to avoid interfering with fast-path packet processing. As a result, iterative revalidation frequently becomes a scalability bottleneck under dynamic workloads, making low TTE difficult to achieve in practice (§4.5).

These limitations underscore the need for *new cache eviction mechanisms* that can identify and remove stale entries both rapidly and selectively—without paying the high cost of re-evaluating the entire cache.

4.2.3 Incremental View Maintenance (IVM)

In database systems, a *materialized view* $V = Q(DB)$ stores the result of a query Q over a base database DB. When the DB changes by an update ΔDB, the view must be updated accordingly: $V' = V + \Delta V = Q(DB + \Delta DB)$. Recomputing the full query Q after each update is often prohibitively expensive, especially when DB is large and updates are frequent.

Incremental View Maintenance (IVM) avoids full recomputation by deriving an incremental query Q^{Δ} that computes only the change to the view. Given an update ΔDB, the incremental query produces a delta $\Delta V = Q^{\Delta}(\Delta DB)$, which is then applied to the existing view state [86, 20, 41]. A key idea in IVM is to retain auxiliary state—such as intermediate results, indexes, or partial aggregates—so that each update can be processed in time proportional to the update size, rather than the size of DB [20].

Modern IVM systems generalize this principle beyond classical select–project–join queries. They support richer relational semantics, streaming updates, and even iterative or recursive computations using dataflow-style execution models [20, 41, 34].

4.2.3.1 Towards Incremental Cache Eviction

This abstraction maps naturally to vSwitches. Here, the *database* corresponds to the collection of rule tables in the slow-path pipeline, while the *materialized view* is the flow cache that stores the computed outcomes of pipeline traversals for previously classified traffic.

A rule update Δr—such as an insertion, deletion, modification, or priority change—updates the rule tables and may invalidate cached forwarding decisions. Maintaining correctness, therefore, requires computing the corresponding cache evictions Δc, analogous to computing ΔV in IVM.

Crucially, Tuple Space Search (TSS) already lends itself to an incremental view-maintenance perspective. Within each vSwitch table, rules are partitioned into tuple spaces based on wildcard structure and stored in per-tuple dictionaries or hash tables, along with auxiliary metadata such as priority bounds that enable early termination during lookup [140, 121]. As a result, rule updates are localized to the affected tables and tuple spaces, rather than impacting the entire pipeline.

This aligns directly with the IVM goal of applying small deltas to the maintained state in response to updates, instead of rebuilding classifiers or revalidating the full cache. It provides a principled foundation for identifying and evicting the cache entries that a given update Δr can actually affect.

4.3 Design of Kairo

We now present the design of Kairo, its integration into production-grade vSwitch implementations (e.g., OVS [121]), and how it processes rule updates without disrupting the fast path. Next, we describe how Kairo uses DBSP [20, 41] to model general vSwitch pipeline lookups (e.g., OpenFlow [89, 148]), enabling incremental cache evictions. We introduce the design step by step: beginning with exact-match tuples (e.g., TSS) and extending them to support complete tables with wildcards and priorities, before scaling to full vSwitch pipelines with arbitrary control flows, as found in production systems today [121, 36, 126, 182].

4.3.1 Overview

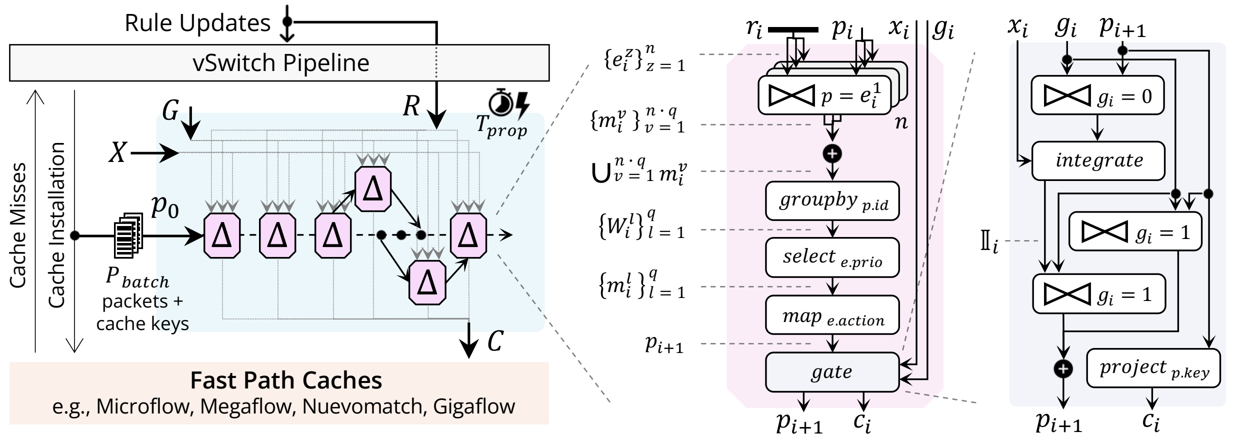
Figure 4.4a shows Kairo integrated into a modern vSwitch architecture as a dedicated cache-eviction thread that replaces existing logic for iteratively revalidating cache entries [121, 66]. The remainder of the cache-maintenance logic remains unchanged (§4.2) while Kairo replaces expensive revalidation to evict entries. When a rule update occurs, the vSwitch invokes Kairo, which briefly acquires a lock on the cache to evict affected entries off the critical path, without interrupting fast-path lookups.

Internally, Kairo implements a pipeline of incremental DBSP *circuits* (§4.3.2), as illustrated in Figure 4.4b. This pipeline mirrors the vSwitch table pipeline. It exposes a streaming interface to the vSwitch: rule updates for all N tables ($r_0, r_1, \ldots, r_{N-1}$, collectively forming a stream group R) and slow-path packets (i.e., cache misses) annotated with the hash keys of their resulting cache entries (p_0) are streamed into Kairo as inputs (Figure 4.4a). Each rule update is routed directly to the circuit for its corresponding table (e.g., r_i for table i), while the packet stream p_0 is fed into the first circuit, reflecting standard vSwitch processing order.

To control exactly which subset of circuits executes for a given update, Kairo equips each circuit with a *gate* (§4.3.5), Figure 4.4c, activated by gate-enable streams ($g_0, g_1, \ldots, g_{N-1}$, stream group G). During execution, each circuit incrementally computes cache evictions

Legend

R, C	Rule updates and cache evictions stream groups for all tables	$p_{\{id,key\}}$	Packet ID and cache key	$\{e_i^z\}_{z=1}^n$	Per-tuple rules ***e*** for table ***i***
G, X	Gate enable and state clear stream groups for all tables	$e_{\{prio,action\}}$	Rule's priority/action from tuple ***e***	$\bigcup_{v=1}^{n\cdot q} m_i^v$	Union of per-tuple matches ***m*** for up to ***q*** packet IDs from table ***i***
p_i, p_{i+1}	Packet streams for table ***i*** & ***i* + 1**	T_{prop}, P_{batch}	Propagation timer & pkt batch size	$\{W_i^l\}_{l=1}^q$	Match groups ***W*** for ***q*** pkt IDs
r_i, g_i, x_i, c_i	Rules, gate enable, state clear, & cache eviction streams for table ***i***	n	Number of TSS tuples in table ***i***	$\{m_i^l\}_{l=1}^q$	Highest priority matches by pkt ID
		q	Number of unique pkt IDs	$\mathbb{I}_i$	Accumulated pkt state in table ***i***



(a) vSwitch Integrated with Kairo
for top-down *incremental* cache evictions

(b) Kairo vSwitch Table **(c) Kairo Gate**
both represented as incremental circuits in DBSP

Figure 4.4: (a) Modern vSwitch architecture with Kairo: the vSwitch pipeline handles cache misses and streams *slow-path* packets (with cache keys p_0) and rule updates R to Kairo, along with control signals G and X, to incrementally generate cache evictions C. (b, c) Kairo's circuit representation of a vSwitch table and an early-exit gate in DBSP.

in response to rule updates and emits them via an eviction output stream c_i. The eviction streams from all circuits ($c_0, c_1, \ldots, c_{N-1}$, stream group C in Figure 4.4a) are consumed by the vSwitch to remove only the affected stale entries from the cache. Gates allow early exits by stopping packets from propagating to downstream circuits that would not contribute further evictions. This lets Kairo prioritize eviction work while deferring unnecessary computation. Any deferred packet state is kept locally and later released to ensure consistency across the pipeline, coordinated by a configurable propagation timer T_{prop}. Once convergence is complete, each circuit resets this state using clear streams ($x_0, x_1, \ldots, x_{N-1}$, stream group X) before handling the next set of rule updates.

We now introduce the incremental view maintenance (IVM) engine underlying Kairo: the DBSP language and its runtime [20].

4.3.2 Kairo's IVM Engine: DBSP

To support incremental cache eviction, Kairo builds on DBSP [20, 41], a language and execution model for incremental view maintenance based on stateful dataflow *circuits*. DBSP represents computations as composable incremental circuits that continuously maintain their

Primitive	API / Method	Semantic Operation
`equi-join`	`Stream::join_index`	Indexed equi-join
`addition`	`Stream::plus`	Stream union
`groupby`	`Stream::map_index`	Group elements by key
`select`	`Stream::topk_desc`	Select descending top-k
`map`	`Stream::map`	Element-wise projection
`filter`	`Stream::filter`	Selection by predicate
`integrate`	`Stream::integrate`	Prefix-sum accumulation
`circuit_init`	`circuit::RootCircuit`	Root circuit context
`stream_in`	`IndexedZSetHandle`	Indexed stream ingress
`stream_out`	`OutputHandle<OrdZSet>`	Stream egress

Table 4.1: DBSP primitives and operators used in KAIRO and their general dataflow semantics.

outputs in response to streams of updates, with state, feedback, and deltas explicitly encoded. This execution model efficiently supports complex relational operations under high update rates, closely mirroring the structure of vSwitch pipelines with cascaded tables and persistent intermediate state.

At a finer level of granularity, DBSP breaks down incremental computation into *operators* (Table 4.1), each responsible for maintaining its output as input deltas arrive. A canonical example is the incremental `equi-join` operator [20, 51], which joins two input streams a and b on a predicate (e.g., equality on a shared field), naturally modeling exact-match lookups in Tuple Space Search (TSS). DBSP incrementalizes this join by computing the change $\Delta(a \bowtie b)$ using the expression: $\Delta(a \bowtie b) = \Delta a \bowtie b + a \bowtie \Delta b + \Delta a \bowtie \Delta b$, which reduces work from $O(|a| \times |b|)$ to $O(\max(|\Delta a|, |\Delta b|))$.

Since these circuits are composed entirely of incremental operators, the circuits themselves inherit this incremental behavior. When applied to a circuit that mirrors the vSwitch table pipeline, KAIRO adopts this delta-driven execution model: rule updates trigger only localized computation on affected tables, allowing cache eviction to scale with update size rather than the size of the rule space or cache.

4.3.3 Incremental vSwitch Tables

We describe how vSwitch lookups based on Tuple Space Search (TSS) are mapped to incremental circuits.

4.3.3.1 Representing TSS Tuples (Exact-Match Tables)

As shown in Figure 4.4b, KAIRO uses DBSP's `equi-join` operator ($\bowtie$) to join packet and rule streams using a predicate based on the exact-match fields of rules that share the same wildcard pattern. For instance, a rule might match packets using a 24-bit IP prefix and a destination port. Each join produces a stream of matched pairs in the form (`packet,matched_rule`).

To enable efficient incremental execution, DBSP [20, 41] automatically builds and maintains indexes over input streams. These indexes allow `join` predicates to be evaluated incrementally as updates arrive, rather than recomputing matches from scratch.

4.3.3.2 Supporting Wildcards and Priorities

As in Tuple Space Search (TSS) [140, 121], KAIRO breaks down rule matching into a set of tuples using `equi-join` operators, where each operator corresponds to a group of rules that share the same wildcard pattern (i.e., a tuple). As shown in Figure 4.4b, the incoming rule stream r_i can contain rules from multiple tuples. To handle this, KAIRO partitions the stream into separate rule streams for each tuple: $\{e_i^z\}_{z=1}^n$, where n is the number of tuples in circuit table i.

Each `equi-join` operator then matches packets from the input packet stream p_i against its associated tuple rule stream e_i^z. This results in one match stream per tuple. If there are up to q distinct packets in p_i, the joins can produce as many as $n \cdot q$ total matches, represented as $\{m_i^v\}_{v=1}^{n \cdot q}$.

These n per-tuple match streams are merged using DBSP's `addition` operator $(+)$, creating a single stream M_i that contains all matches for all packets: $M_i = \bigcup_{v=1}^{n \cdot q} m_i^v$. KAIRO then uses a `groupby` operator to group these matches by packet identifier (e.g., 5-tuple), yielding $W_i = \texttt{groupby}_{p.id}(M_i) = \{W_i^l\}_{l=1}^q$. At this stage (as shown in Figure 4.4b), each group W_i^l contains all tuple matches for a single packet.

To enforce priority semantics, KAIRO applies a `select` operator to each group to retain only the match with the highest-priority rule: $\texttt{select}_{e.prio}(W_i) = \{m_i^l\}_{l=1}^q$. Internally, this operator sorts each group in descending priority order and selects the top match, ensuring that exactly one rule match is retained per packet—consistent with TSS behavior.

4.3.3.3 Applying Actions

Given the set of selected matches $\{m_i^l\}_{l=1}^q$, KAIRO applies the actions associated with each matched rule using DBSP's `map` operator (Table 4.1). For each match, `map` transforms the packet based on the action specified in the matched rule. This produces an output packet stream p_{i+1}, which is then passed to the next circuit in the pipeline, preserving the vSwitch semantics for table traversal in the slow path.

4.3.4 Incremental vSwitch Pipelines

Real-world vSwitch pipelines include control logic that directs packets through multiple tables, often with complex control flows (see Table 4.3). KAIRO extends incremental tables to support full pipelines by explicitly modeling packet resubmissions and tables' fan-in/fan-out using DBSP operators.

4.3.4.1 Handling Multi-Table Fan-Out

In vSwitch pipelines like OVS, the output packet stream p_{i+1} from table i may need to be forwarded to one or more downstream tables (i.e., fan-out), depending on the actions of individual rules. To support this, KAIRO uses a `filter` operator for each possible destination table. Each filter checks the resubmit action in the packet and selects only those packets directed to that specific table, forwarding them as the input stream for that table. This allows a single output stream to be dynamically split and routed to multiple tables, based on per-packet control decisions embedded in each rule's action.

4.3.4.2 Handling Multi-Table Fan-In

Conversely, a table may receive packets from multiple upstream tables. To support this fan-in behavior, KAIRO uses stream `addition` (+) to merge all incoming packet streams into a single input stream p_j for table j. This unified stream is then processed by table j using the same incremental steps: lookup, matching, and action execution, as described earlier (§4.3.3).

4.3.4.3 Supporting Resubmissions

vSwitch pipelines allow packets to be resubmitted to earlier tables, which introduces (bounded) control-flow cycles in the pipeline graph [121, 89, 148]. KAIRO handles these resubmissions by first applying a standard strongly connected component (SCC) condensation [166], followed by loop unrolling [165] to convert the cyclic pipeline into a time-expanded, acyclic graph.

Each SCC, representing a resubmission cycle, is locally unrolled into a fixed number of layers, where cyclic edges are replaced with forward edges between replicated layers. Edges entering the SCC connect to the first layer, while exit edges can originate from any layer—preserving the semantics that packets can leave the cycle after any resubmission.

This transformation results in a globally acyclic pipeline that preserves the original control flow and supports efficient incremental execution in DBSP, while incorporating multi-table fan-in and fan-out.

4.3.5 Early Exits with Gates

In practice, rule updates typically affect only a small portion of the vSwitch table pipeline—modifying specific policies in a few tables. However, even localized updates can cause packet propagation and trigger incremental execution across all downstream KAIRO circuits. This becomes particularly costly during large updates that invalidate many cache entries, as the resulting surge of packet updates can propagate through downstream circuits, incurring substantial computation that does not help with immediate cache eviction, Figure 4.4 (a,b).

To limit such unnecessary work and prioritize eviction, KAIRO introduces gates—*new* lightweight control operators embedded in each circuit that enable early pipeline exits while maintaining correctness.

Each circuit i includes a gate, as shown in Figure 4.4c, controlled by a gate-enable stream g_i (stream group G in Figure 4.4a). When the gate is open, $g_i = 1$, the output packet stream p_{i+1} proceeds to the next circuit. When the gate is closed, $g_i = 0$, KAIRO stops the packet stream from propagating further, since downstream tables will not trigger any additional cache evictions.

Importantly, the gate only affects downstream propagation—not local cache eviction. If there is a rule update for table i ($\Delta r_i > 0$), the circuit still emits any affected cache entries for eviction on its output stream c_i, regardless of the gate's state. This is done via the DBSP's `map` operator, which projects (extracts) the cache key from the packet and sends it back to the vSwitch for eviction.

When a gate is closed, packets that would normally move downstream are instead held locally in an accumulation buffer $\mathbb{I}_i$ (shown in Figure 4.4c), using DBSP's `integrate` operator. This deferred packet state is released when the gate reopens, ensuring that the pipeline eventually converges. To trigger this release, KAIRO uses a configurable propagation timer T_{prop} (Figure 4.4a), which opens all gates after a defined delay. Once convergence is complete, the circuit clears the accumulated packet state using a dedicated clear stream ($x_0, x_1, \ldots, x_{N-1}$, stream group X), activated briefly, $x_i = 1, \forall i \in \{0 \ldots N-1\}$, before the next rule updates.

4.3.6 Putting It All Together

Table 4.2 summarizes how KAIRO coordinates incremental execution at runtime across different update scenarios. It distinguishes between *rule updates*, which can invalidate cache entries and must be handled immediately, and *slow-path packet updates* (i.e., cache misses), which do not trigger eviction and only populate cache or update circuit state, so they can be safely deferred. Therefore, KAIRO batches slow-path packets using a configurable batch size P_{batch} (Figure 4.4a). These packets are injected into the pipeline only when the batch fills up or when the propagation timer T_{prop} expires (`S1`). In contrast, rule updates are processed immediately by streaming them into the appropriate circuits to trigger incremental cache evictions.

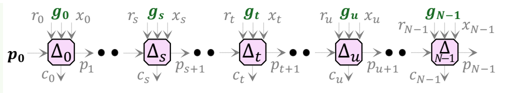
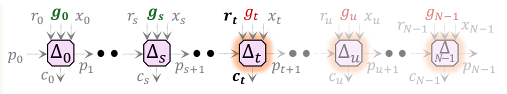
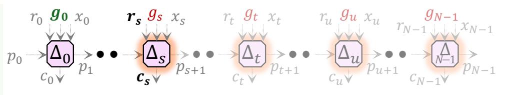
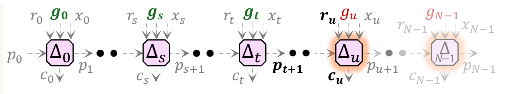
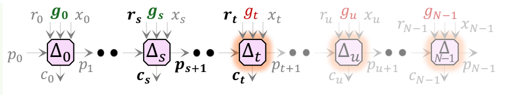
Update Scenario	Circuit Behavior	Kairo Streams & Signals	Kairo Circuit View
S1: T_{prop} expires or new P_{batch} packets arrive	All gates open; no cache evictions; state converges	$g_i = 1,\ \Delta r_i = 0,\ \Delta c_i = 0;$ $\Delta p_0 \geq 0;\ \Delta p_i \geq 0,$ $\forall i \in \{0 \dots N-1\}$	
S2: Back-to-back rule updates to table t	Gates $\geq t$ closed; t holds state & evicts cache	$g_i = 1\ \{i < t, 0 \text{ otherwise}\};$ $\Delta r_t > 0;\ \Delta c_t \geq 0;$ $\mathbb{I}_t > 0;\ \Delta p_{t+1} = 0$	
S3: Rule updates to table s, after updates to t	Gates $\geq s$ closed; s, t hold state & s evicts cache	$g_i = 1\ \{i < s\};\ \Delta r_s > 0;$ $\Delta c_s \geq 0; \mathbb{I}_s,\ \mathbb{I}_t > 0;$ $\Delta p_{s+1} = 0$	
S4: Rule updates to table u, after updates to t	Gates $\geq u$ closed; t passes state to u; u evicts cache	$g_i = 1\ \{i < u\};\ \Delta r_u > 0;$ $\Delta c_u \geq 0; \mathbb{I}_u > 0;$ $\Delta p_u = \mathbb{I}_t;\ \Delta p_{u+1} = 0$	
S5: Simultaneous rule updates to tables s and t	Gates $\geq t$ closed; t holds state; s & t evict cache	$g_i = 1\ \{i < t\};\ \Delta r_s,\ \Delta r_t > 0;$ $\Delta c_s,\ \Delta c_t \geq 0;\ \mathbb{I}_s = 0;\ \mathbb{I}_t > 0;$ $\Delta p_{s+1} \geq 0;\ \Delta p_{t+1} = 0$	

Table 4.2: Representative update scenarios in Kairo, illustrating how incremental circuits, streams, gates, and packet state retention interact. Additional scenarios can be expressed as combinations of these blocks.

To limit computation during rule updates, Kairo opens gates only up to the highest-indexed table that receives a rule update. As shown in Table 4.2, this ensures that only the relevant circuits—those contributing to cache evictions—are executed. Downstream circuits retain their packet state without running unnecessary computations. Once evictions are processed, Kairo reopens all gates—either after T_{prop} expires or when batched packets are released—allowing deferred packet state to flow through the pipeline and converge. This execution strategy enables Kairo to prioritize eviction work while ensuring eventual consistency of packet state across all pipeline stages.

Table 4.2 illustrates how Kairo behaves under common update scenarios. For instance, if there are back-to-back rule updates at a single table Δ_t (S2), Kairo closes the gates at and beyond table t, processes only up to circuit t to compute evictions ($\Delta c_t \geq 0$), and stores the current packet state locally as $\mathbb{I}_t$. Downstream packet propagation is halted ($\Delta p_{t+1} = 0$), avoiding unnecessary computation and allowing evictions to proceed quickly.

If a subsequent rule update then arrives at a downstream table Δ_u (S4), and T_{prop} has not expired yet, Kairo must now forward the retained packet state. It closes the gates at table u, releases the stored state from t to u (including any packet modifications), and has circuit u compute cache evictions based on its updated rules. As before, propagation stops beyond u, so only relevant circuits run. These scenarios in Table 4.2 show how Kairo uses gate control and packet state retention to limit execution to the actual scope of each rule update as it is applied, while still ensuring eventual consistency across the full pipeline.

4.3.7 Operational Concerns

We now highlight some practical considerations when using Kairo for vSwitch cache eviction.

4.3.7.1 Evictions Due to Idle Timeouts

Sometimes, cache entries are removed not because of rule updates, but because they have been idle for too long, based on vSwitch timeout policies (§4.2). To keep Kairo's internal state in sync, each such idle eviction is translated into a dummy packet deletion. This dummy packet is injected into Kairo's input packet stream p_0 (batched as part of P_{batch}). By processing these deletions, Kairo cleans up the corresponding state in its incremental circuits, ensuring that no stale packet state is retained in Kairo's circuits.

4.3.7.2 Handling Pipeline Updates

At runtime, vSwitch rule updates may introduce new tuples into existing tables or add entirely new tables to the pipeline. Kairo handles both cases as follows:

- *Handling new tuples.* A new tuple represents a wildcard group that previously did not exist or did not match any packets. Kairo initializes this new tuple with an empty match state and evaluates it against the current packet stream p_i of its associated table. This ensures the new tuple can immediately participate in incremental matching and cache eviction, without needing to replay or reprocess earlier rule state.

- *Handling new vSwitch tables.* Adding a new table involves inserting a new circuit into the processing pipeline. If a new table t is inserted between existing tables s and u, Kairo first breaks the existing stream from s to u by injecting packet deletions to discard any state derived from that stream. Then, it connects the output stream from s as the input to t. The output from t is forwarded to u as incremental updates, thereby integrating the new table into the pipeline. This preserves pipeline behavior while enabling the new table to start participating in incremental execution without replaying the full packet history.

4.3.7.3 Supporting Gigaflow's Multi-Table Cache Architecture

In Gigaflow's multi-table SmartNIC cache [182], a single packet may generate multiple, shared, Longest-Traversal Matching (LTM) cache entries (§3). Kairo supports such multi-table caches with a minimal extension to its execution model.

The key difference from the single-table case is that the input packet stream p_0 carries, in addition to the packet itself, a list of cache keys corresponding to all LTM entries produced by that packet along the pipeline. These keys propagate as normal through the incremental circuits together with the packet.

When a cache eviction is triggered in KAIRO, the first table whose circuit identifies the packet as stale evicts its corresponding cache entry. Because cache entries for downstream tables are causally dependent on upstream decisions, KAIRO evicts all LTM entries generated by that packet from the evicting table onward. This behavior matches GIGAFLOW's cache semantics while allowing KAIRO to reuse the same incremental pipeline and eviction logic without modification.

4.3.7.4 Security Concerns

A rapid stream of rule updates could potentially overwhelm KAIRO, resembling a denial-of-service (DoS) attack [129]. However, this vulnerability is not unique to KAIRO—it already exists in OVS, where frequent rule updates can saturate the revalidation thread. By speeding up rule update handling and stale-cache eviction, KAIRO actually reduces the time window in which such attacks could have an impact. As a result, it does not introduce new security risks beyond those already present in current vSwitch designs.

4.4 Implementation

We implement KAIRO in the widely adopted Open vSwitch (OVS) [121, 43] (`commit:4f933301`). To support KAIRO circuits, we build on Feldera's Rust-based implementation of DBSP [34, 41] (`commit:20d4ba57`). This avoids reimplementing DBSP in native C, much like how OVS itself leverages DPDK [109, 32] for its userspace fast path. We compile and run DBSP circuits using `cargo`, `rustc v1.83.0`, and `rustup v1.28.2`. Each vSwitch pipeline corresponds to approximately 3,200 lines of code (LoC): 77% implements the DBSP circuit logic, 12% handles ingress of packets and rules from OVS and egress of cache evictions back to OVS, and 11% maps OVS data types (e.g., rules, matches, actions) into Rust.

To interface with OVS, we compile KAIRO as a C-callable Rust shared library (`cdylib`) that exposes a minimal API via `extern "C"` functions. This library is linked directly into the OVS binary, allowing in-process invocation without the overhead of inter-process communication. We define a shared header that provides a C-compatible Application Binary Interface (ABI). It exposes a `KairoCtx` handle that encapsulates all circuit state, along with data structures for packets, rules, and cache keys using `(pointer,length)` batch semantics.

During OVS initialization, a long-lived `KairoCtx` instance is created and retained for the lifetime of the process. A dedicated cache-eviction thread synchronously invokes the

Rust entry points to: (1) stream batches of packets and rule updates into the circuit, and (2) trigger incremental evaluation of the DBSP pipeline. All input and output buffers are allocated and owned by OVS. Evicted cache keys are written by Rust into caller-provided buffers before returning control to OVS. This ensures clear memory ownership, predictable resource usage, and clean separation between circuit execution and the fast path.

4.5 Evaluation

We evaluate KAIRO along three key dimensions: (1) the volume of stale cache-processed traffic, (2) the speedup in time-to-eviction (TTE), and (3) the impact on fast-path forwarding performance. We compare KAIRO against multiple baselines in end-to-end scenarios (§4.5.2) and further conduct microbenchmarks to characterize its behavior under a range of workloads and system configurations (§4.5.3).

4.5.1 Experiment Setup

4.5.1.1 Testbed Environment

Our testbed consists of two servers connected back-to-back: one acting as the device under test (DUT) and the other as a traffic generator (TGEN). Both servers are equipped with a 64-core Intel Xeon Platinum 8358P CPU @ 2.60 GHz and 512 GB RAM, and run Proxmox VE (`v8.0.4`) [123]. Each machine hosts a dual-port Intel XL710 10/40 Gbps NIC and an Nvidia ConnectX-6 100 Gbps NIC. The DUT runs Open vSwitch (OVS) [121, 43] with our KAIRO prototype, while the TGEN runs a custom DPDK-based traffic generator provided with the open-source OVS pipelines used in prior work [182, 42].

4.5.1.2 Baseline Configurations

We evaluate KAIRO against three representative baselines. The first is *iterative* cache eviction, the de facto strategy used in OVS [121, 66, 43], which revalidates each cache entry individually in response to rule updates. The second is *bruteforce* batched cache recomputation, which recomputes the cache state by reevaluating the full flow space across the vSwitch pipeline (Figure 4.2). The third is a simplified variant of KAIRO that disables both early-exit gates and the state propagation timer, KAIRO (-Gate, -Prop).

We configure the packet batch size, P_{batch}, to 100 and the state propagation timer T_{prop} to 2s. This choice reflects the worst-case propagation time observed in our experiments: the largest OLS pipeline requires up to 1.9s to propagate state through the full circuit under P2 updates (see Table 4.3 and §4.5.3).

Pattern	Description	Reported Updates (# / %)	#Updates per Pipeline OFD	PSC	OLS	ANT
P1: Security	Firewall/ACL periodic rule changes [15, 150, 28], security responses (e.g., to DDoS)	1005 / **6.0%**	580	850	1740	850
P2: Pruning	Redundant/under-utilized ruleset cleanup [35] from enterprise firewalls	1668 / **10.0%**	970	1400	2900	1400
P3: Recovery	Traffic engineering [80, 64], fault recovery (link/node failures) to restore reachability	**300** / 2.1%	300	300	300	300
P4: Elasticity	Server load balancing [91], auto-scaling backends/pool membership changes	**110** / 0.8%	110	110	110	110
P5: Telemetry	Traffic monitoring by ISPs [21], temporary rules for debug/exceptions	**500** / 3.5%	500	500	500	500

Table 4.3: Rule update patterns with update size as number of rules or % of ruleset (reported numbers in bold) and their mapping to open-source vSwitch pipelines [182, 180]: Cord OFDPA (OFD) [103], Pisces (PSC) [137], OVN logical switch (OLS) [112], and Antrea OVS (ANT) [10, 11, 12].

For iterative revalidation, we directly measure the time required to revalidate the Megaflow cache in OVS [121, 126, 182] after rule updates. For bruteforce recomputation, we estimate time-to-eviction (TTE) as the product of the total flow space—i.e., the cross-product rule space induced by all pipeline traversals—and the time to revalidate a single cache entry. Unless otherwise noted, we provision one revalidation thread for OVS, while both Kairo and Kairo (-Gate, -Prop) run with a single thread; observed trends remain consistent with additional CPU threads.

4.5.1.3 Real-World vSwitch Pipelines, Rulesets, and Traffic

We evaluate using open-source vSwitch pipelines for OVS together with their corresponding rulesets and traffic traces, as used in prior work [182, 42]. These include Cord OFDPA (OFD, 10 tables) [103], Pisces (PSC, 7 tables) [137], OVN logical switch (OLS, 30 tables) [112], and Antrea OVS (ANT, 22 tables) [10, 11, 12], covering a range of real deployment scenarios. Traffic traces are derived from CAIDA [23] and consist of 100K flows per trace, with packet headers modified to match each pipeline's ruleset [126, 182].

4.5.1.4 Representative Rule Update Pattern Generation

Table 4.3 summarizes rule update patterns observed in production environments,[1] along with update sizes expressed either as a percentage of the overall ruleset or as an absolute number of rules (reported values shown in bold). These studies typically report long-term averages (e.g., monthly totals [15]) or qualitative observations (e.g., 10% of enterprise firewall rules are redundant [35]), rather than per-second update rates. Accordingly, we apply each update pattern as an atomic batch.

[1]While additional patterns may exist, we only cite sources that report concrete percentages or rule counts.

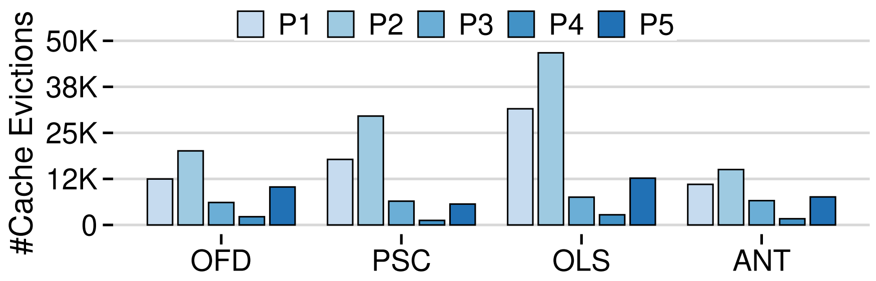


Figure 4.5: Cache evictions with 100K traffic flows across four pipelines and update patterns. All cache eviction schemes (e.g., iterative, KAIRO) remove the same cache entries.

For each pipeline, we realize an update pattern by mapping its update size to the corresponding ruleset, as shown in Table 4.3. We randomly select tables containing more rules than the target update size and sample a subset of rules equal to the update size to form the *update* ruleset; the remaining rules constitute the *base* ruleset. This produces a ruleset pair for each pattern and pipeline.

At runtime, we first install the base ruleset in OVS and replay the traffic trace until the cache is fully populated, cycling the trace for the duration of the experiment. We then add the update ruleset and measure time-to-eviction (TTE) and the number of cache entries evicted. Next, we remove the same rules and again measure TTE and evictions. We repeat this add–delete sequence and report TTE as the average over four eviction rounds, which evict identical cache entries, as shown in Figure 4.5.

4.5.2 End-to-End Performance

4.5.2.1 Stale Cache Traffic and Time-to-Eviction (TTE)

Figure 4.6a shows the number of stale cache-processed packets at 100 Gbps line rate for different cache eviction strategies.

Compared to iterative revalidation, the bruteforce recomputation significantly worsens stale cache processing—on average by 117.6× across all pipelines and update patterns. In contrast, KAIRO reduces stale traffic substantially: 28.4× average reduction for OFD, 32.1× for PSC, 30.1× for OLS, and 50.6× for ANT, relative to iterative revalidation. Across all patterns and pipelines, KAIRO delivers an average 35.5× reduction in stale cache traffic. The minimum observed improvement is 4.6× for OLS with P2 updates (reducing 3.37M stale packets to 0.74M), while the maximum is 129.6× for ANT with P4 updates (reducing 154.57K packets to 1.19K).

Stale cache processing is influenced by the line rate, packet size distribution, the ratio of cache evictions to total cache entries, and the time-to-eviction (TTE). Among these,

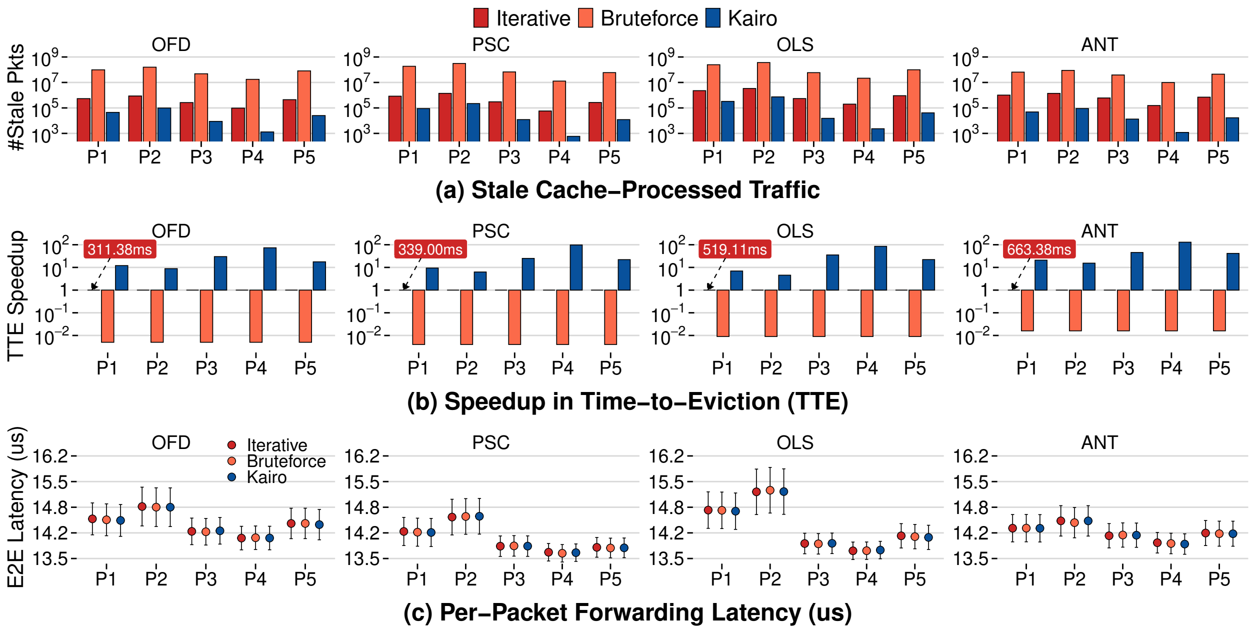


Figure 4.6: Stale cache-processed packets, time-to-eviction (TTE) speedup relative to iterative revalidation, and fast-path forwarding latency across pipelines and update patterns for various cache eviction algorithms.

TTE is the primary determinant. Under fixed experiment settings, all eviction schemes remove the same set of cache entries (Figure 4.5); as a result, TTE exhibits the same trend in Figure 4.6b as stale cache traffic. For example, bruteforce processing takes 56.8s for OFD, and iterative revalidation takes 311.4ms. In contrast, KAIRO completes eviction in just 18.66ms on average—a 16.7× improvement over the iterative scheme. Similar gains are observed for other pipelines: 13.8× for PSC, 11.1× for OLS, and 30.2× for ANT. KAIRO achieves these massive gains by incrementally computing updates in only the affected portions of the pipeline—leveraging gates and the propagation timer T_{prop} (§4.3).

4.5.2.2 Forwarding Latency and Throughput

As shown in Figure 4.6c, per-packet forwarding latency is unaffected by the choice of cache eviction scheme. This is because all eviction strategies operate off the critical path and evict the same set of entries. When a stale entry is evicted, it results in a cache miss, triggering slow-path installation for the affected flow. The eviction strategy only impacts the timing of this miss, not the forwarding behavior. The expected forwarding latency is given by: $L_{hit} + (1 - hit\_rate) \times miss\_penalty$ where $miss\_penalty = L_{miss} - L_{hit}$. (We ignore head-of-line blocking effects in cache miss processing.)

For any given update pattern, the hit rate is determined by the number of traffic flows and the volume of evictions/installations (Figure 4.5, §4.5.1). This causes variation in end-to-end

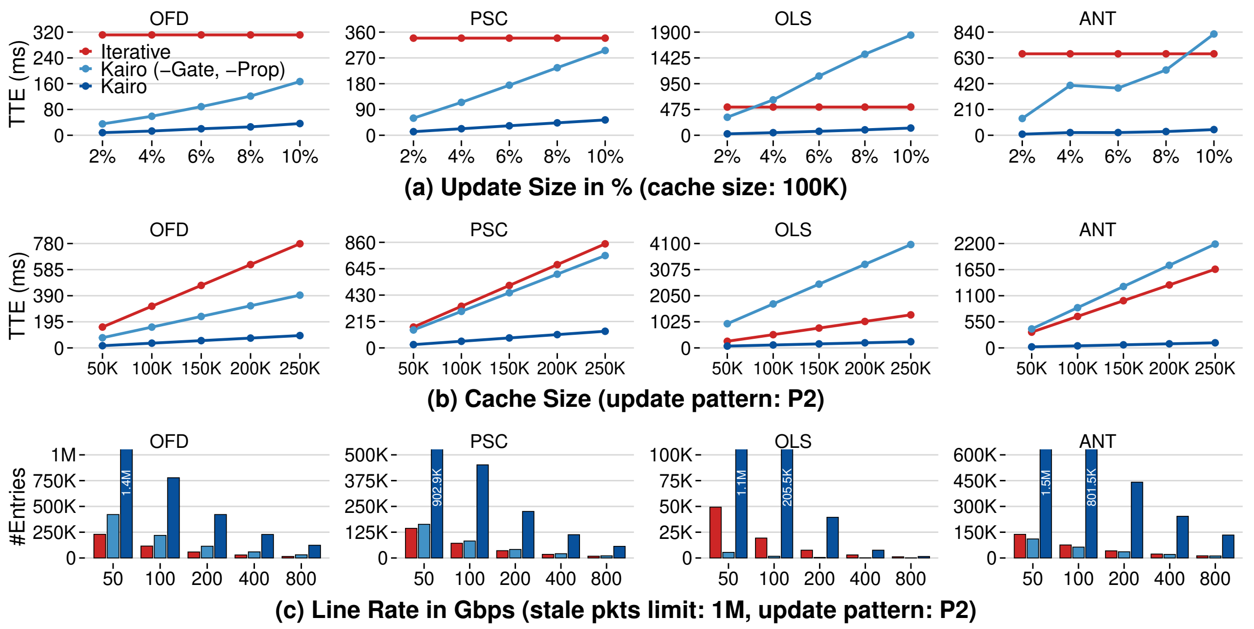


Figure 4.7: TTE against updates of increasing size as % of ruleset, TTE with increasing cache size with P2 updates, and maximum cache size vs. line rate with P2 updates and fixed 1M stale packets limit.

latency across patterns within a pipeline. However, for a given pattern, forwarding latency remains consistent regardless of the eviction scheme used. End-to-end throughput follows the same trend: all eviction schemes allow OVS to maintain line rate; the only difference lies in the volume of stale cache traffic processed per scheme.

4.5.3 Microbenchmarks

• Kairo's TTE scales with update size but remains much lower than iterative revalidation. Figure 4.7a shows how TTE changes with increasing update size across pipelines. Iterative revalidation remains insensitive to update size—it always scans the entire cache, yielding a constant TTE. Kairo (-Gate, -Prop), in contrast, scales with update size: it achieves lower TTE for small updates, but grows faster than the iterative scheme for complex pipelines like OLS and ANT. Kairo, by design, scales proportionally with update size and achieves the lowest TTE across all pipelines and update sizes. This behavior stems from how incremental view maintenance works: processing time scales with the size of the updates, both in terms of input (rule deltas) and output (affected cache entries) (Table 4.3, Figure 4.5).

At 2% updates, Kairo delivers TTE reductions of 38.6×, 27.1×, 20.3×, and 75.6× across OFD, PSC, OLS, and ANT, respectively. While at 10%, it is 8.6×, 6.3×, 3.9×, and 14.4× for the same pipelines. These benefits are due to Kairo's gated execution and localized

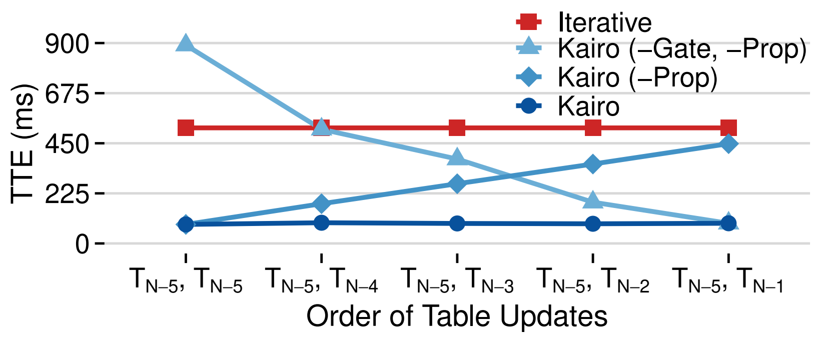


Figure 4.8: Comparing KAIRO optimizations: gates and T_{prop} timer with ordered P1 updates to OLS pipeline.

propagation, avoiding unnecessary computation in unaffected portions of the pipeline.

• **KAIRO exhibits weak TTE dependence on cache size relative to other cache eviction schemes.** In iterative revalidation, TTE grows linearly with cache size. As shown in Figure 4.7b for P2 updates, larger caches directly result in longer revalidation times and more stale traffic (Figure 4.6a). KAIRO, however, shows significantly slower TTE growth with increasing cache size. It consistently delivers an average 8.8× speedup over the iterative scheme across cache sizes ranging from 100K to 250K entries. At the low end, the improvement is 3.7× (for OLS at 50K entries), while at the high end, it reaches 15.4× (for ANT at 250K entries). This mild growth is expected: as traffic scales up, the number of evicted entries also increases, but due to KAIRO's incremental model, this growth is bounded and significantly less than that of baseline schemes (Figure 4.5).

• **For a fixed stale-cache-processing limit, KAIRO supports the largest caches of all baselines.** Given a fixed stale-cache-processing budget of 1M packets, Figure 4.7c shows the maximum cache size each scheme can support at various line rates. Cache size scales according to the following power-law expression: $cache\_size = (\frac{stale\_packets}{k \times line\_rate})^h$, where $cache\_size$ is in thousands, $line\_rate$ is in Gbps, and k, h are constants derived from Figure 4.7b. Owing to this massive reduction in stale-cache traffic (Figure 4.6a), KAIRO supports significantly larger Megaflow caches than the baseline, iterative scheme: 11.2× at 50 Gbps, 8.6× at 100 Gbps, 7.4× at 200 Gbps, 6.8× at 400 Gbps, and 6.7× at 800 Gbps, which is key to sustaining vSwitch performance at higher data rates [182, 126].

• **KAIRO's gates and state propagation timer play a crucial role in its performance gains.** We evaluate how KAIRO's optimizations affect TTE using a sequence of two P1 updates to the OLS pipeline, spaced 5s apart. Each run starts with a base ruleset. We then apply the first update to the fifth-to-last table in the pipeline, followed by a second update to successively later tables (e.g., fourth-to-last, third-to-last, and so on). Figure 4.8 shows the results: First, KAIRO (-Gate, -Prop) performs the worst when both updates hit

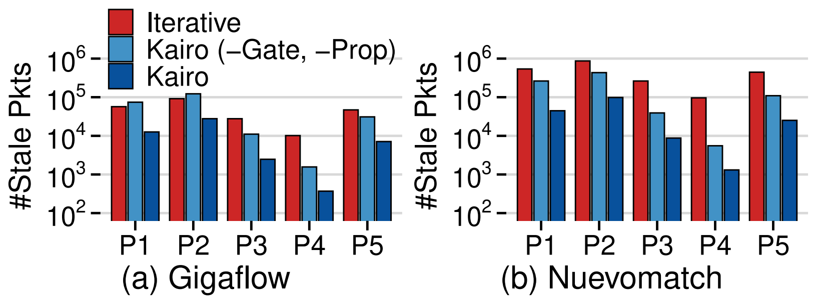


Figure 4.9: Stale cache–processed traffic in Gigaflow and Nuevomatch using KAIRO with OFD pipeline.

Overhead	OFD	PSC	OLS	ANT	Average
Setup Time	17.7%	18.1%	21.0%	20.9%	19.4%
CPU Utilization	-0.14%	-0.04%	1.10%	0.17%	0.27%
Memory Usage	17.7%	27.0%	71.3%	56.7%	43.2%

Table 4.4: KAIRO overheads: OVS-DPDK with 10 pinned CPU cores for 100 Gbps line rate, 200K traffic flows, and P2 updates at 10 s intervals.

the same table. It must propagate changes through the entire remaining pipeline, increasing TTE. Second, with gates enabled, TTE depends on the distance between the updated tables. Without the timer, TTE grows as this distance increases. Finally, enabling the propagation timer allows KAIRO to release accumulated state during idle periods. As a result, the second update has the same TTE as the first, regardless of distance.

• **KAIRO delivers comparable TTEs across pipelines of varying traversal lengths.** As shown in Figure 4.6b, iterative revalidation takes nearly twice as long to evict entries from ANT (22 tables) compared to OFD (10 tables), across all update patterns. This is expected: deeper traversals take longer to revalidate [182, 126, 121].

KAIRO, however, shows only modest variation in TTE. For P1, it takes 25.79ms for OFD versus 31.68ms for ANT, while for P3 it takes 10.38ms for OFD versus 14.55ms for ANT. This demonstrates that KAIRO's performance scales with update size rather than pipeline depth, allowing it to handle deeper pipelines efficiently.

• **KAIRO complements vSwitch architectures like Gigaflow and Nuevomatch.** We evaluate KAIRO using the OFD pipeline in combination with two recent cache architectures: (a) *Gigaflow* [182], with a high-locality SmartNIC cache (32K entries), and (b) *Nuevomatch* [126], with a Megaflow cache (100K entries).

Figure 4.9 shows that KAIRO reduces stale traffic on average by 28.4× for Nuevomatch and 10.6× for Gigaflow, with up to 73.4× and 27.4× improvement respectively under P4 updates. While Gigaflow's smaller, sub-traversal cache entries allow the baseline, iterative scheme to revalidate faster compared to Nuevomatch, KAIRO still achieves better TTE.

• **Kairo incurs modest resource overheads relative to traditional iterative revalidation.** We measure Kairo's resource overheads while running OVS-DPDK at 100 Gbps line rate with 10 pinned cores, 200K cached flows, and P2 updates every 10s. As shown in Table 4.4, Kairo circuits compile in about 10s, adding under 20% overhead relative to compiling OVS with DPDK, initializing the userspace bridge, and loading rules. At 200K flows—enough to saturate the default OVS cache [121, 43]—OVS with DPDK consumes about 3 GB of memory [129]. While Kairo increases memory usage, modern SmartNICs and end-hosts provide ample headroom [99, 7]. Lastly, Kairo's CPU impact is negligible across all pipelines.

4.6 Limitations & Future Work

4.6.1 Alternative Representations for Kairo Circuits

Kairo currently represents vSwitch pipelines using Tuple Space Search (TSS), which requires one `equi-join` operator per tuple. Since all tuples in a table receive the same input packet stream, this design inflates the amount of state maintained within Kairo's circuits and increases memory overhead as wildcard diversity grows. As future work, we plan to explore alternative circuit representations—by adopting different search algorithms or introducing new DBSP operators—that avoid this per-tuple overhead, reducing circuit state and Kairo's overall memory footprint.

4.6.2 Extending Kairo to support vSwitch lookups

Kairo maintains a full representation of the vSwitch pipeline state alongside OVS, which independently manages its own rule tables. While Kairo is designed exclusively for cache eviction, OVS also performs cache-miss processing using its table pipeline and generates non-overlapping wildcarded cache entries by removing overlaps from higher-priority rules during lookup [121, 126, 182]. These lookup-time operations are not currently modeled in Kairo. Extending Kairo to support them would further reduce redundant vSwitch state, resulting in a more efficient implementation.

4.6.3 Migrating Kairo from Rust to the OVS-Native Codebase

We implemented Kairo using Feldera's Rust-based DBSP implementation [41, 20], which introduces a non-trivial ABI boundary between Rust and the OVS C codebase. However, Kairo relies on only a small subset of DBSP operators. As future work, we can integrate limited DBSP support directly into OVS—similar to how OVS implements its internal data structures—within the scope of incremental cache eviction. This would simplify the implementation and eliminate the need for a cross-language interface.

4.7 Related Work

4.7.1 Cache Eviction Strategies

Prior work on cache eviction primarily improves replacement policies for software-managed caches under skewed access patterns [14, 25, 175, 78, 177, 178, 104]. S3-FIFO [175] showed that multi-queue FIFO outperforms LRU in single-access–heavy workloads, while SIEVE [177] distilled these benefits into a simpler single-queue design for web caches. These systems optimize eviction based on access locality. In contrast, KAIRO addresses a different dimension of the problem: efficiently evicting cache entries in response to rule updates, rather than access patterns.

4.7.2 Network Updates

Work on network updates focuses on safely and efficiently modifying forwarding behavior in SDN and data-center networks. Early systems prevent transient misconfigurations during rule changes [130, 47], while later work improves update efficiency via batching [30], bounded and selective updates [82], and scalable rule placement [68]. Others examine control-/data-plane limitations and automate rule distribution for monitoring and management [76, 1, 22]. Complementary work formalizes match-action pipelines to reason about rule-change effects [95]. These approaches target control-plane correctness and installation-time efficiency; by contrast, KAIRO addresses the data-plane cost of rule updates after installation, enabling efficient top-down eviction without full cache revalidation.

4.7.3 Incremental View Maintenance

IVM systems traditionally target relational queries and structured update models. Some precompute and propagate higher-order deltas for frequently updated views [2], while others develop dynamic algorithms with theoretical guarantees for maintaining conjunctive queries with inequalities [52, 54]. Follow-up work systematizes these approaches for dynamic datasets but remains centered on narrow relational classes [53], building on earlier algebraic foundations [72]. In contrast, DBSP [20, 41] offers a general, compositional model for incremental computation over stateful dataflow circuits, making it a natural fit for modeling vSwitch pipelines and enabling KAIRO to perform pipeline-aware cache evictions on rule updates.

4.8 Conclusion

We presented KAIRO, a cache-eviction mechanism for modern vSwitches that reframes rule-update–driven evictions as an instance of incremental view maintenance (IVM). By modeling vSwitch pipelines as incremental DBSP circuits, KAIRO enables precise, top-down evictions whose cost scales with the actual impact of a rule update, rather than with cache size or pipeline depth. Early-exit gates further limit immediate computation by deferring non-critical downstream work, while still ensuring eventual convergence of state. Together, these techniques decouple cache scalability from update intensity, making efficient eviction not only feasible but essential for sustaining correctness and performance in increasingly dynamic, 800 Gbps+ vSwitch deployments.

CHAPTER 5

Conclusion

"

Most creativity is a transition from one context into another where things are more surprising. There's an element of surprise, and especially in science, there is often laughter that goes along with the "Aha."

ALAN KAY (ACM QUEUE INTERVIEWS [124])

The thesis of this dissertation is simple to state and, we believe, now simple to defend. To return to the question that started this dissertation:

> ❏ **Thesis Statement**
> *The performance ceiling of modern vSwitches is not a fundamental limit but an artifact of two outdated assumptions: that hardware caches must collapse multi-table pipelines into a single lookup, and that cache correctness requires full revalidation on every rule update. Discarding both assumptions—by exploiting pipeline-aware locality to build a multi-table SmartNIC cache, and by using incremental computation to make eviction fast and precise—breaks this ceiling to unlock a new tier of vSwitch performance.*

We realized this thesis through two complementary systems, GIGAFLOW and KAIRO, each targeting one flawed assumption at its root. GIGAFLOW demonstrates that hardware cache hit rates need not be constrained by the single-table, traversal-cache design of Megaflow: by exploiting the structure of programmable vSwitch pipelines, it captures orders of magnitude more rule space within the limited SmartNIC TCAM budget. KAIRO demonstrates that cache eviction cost need not scale with the number of cached entries: by treating eviction as an instance of incremental view maintenance, it decouples time-to-eviction (TTE) from cache size and couples it instead to the size of the rule update that triggered the eviction. Together, they validate the thesis: the ceiling was not fundamental, and it has been broken.

5.1 Summary of Contributions

Gigaflow: Rethinking the Fast Path. The core insight behind Gigaflow is the recognition of a form of locality that existing vSwitch caching schemes had entirely overlooked. Temporal and spatial locality—the basis for Microflow and Megaflow caches—arise from traffic patterns. *Pipeline-aware locality* arises from the structure of the vSwitch pipeline itself: different flows tend to traverse overlapping sub-sequences of ordered policy stages (e.g., L2, L3, ACL), matching on distinct, often disjoint header fields at each stage. This structural regularity creates abundant sharing opportunities that a single-table Megaflow cache—which collapses the entire traversal into one monolithic wildcard—is incapable of exploiting.

Gigaflow decomposes each traversal into reusable *sub-traversals*, distributes them across a small number of SmartNIC cache tables, and composes them at lookup time through Longest Traversal Matching (LTM). A single set of cache entries can thereby match the cross-product of rule combinations across tables, dramatically expanding rule space coverage without increasing entry count. Modern programmable SmartNICs execute multi-table lookups at line rate using RMT-based architectures, rendering the assumption that such lookups are impractical a historical artifact. Evaluated against five real-world pipeline configurations, Gigaflow achieves up to 51% higher cache hit rates (25% on average), reduces cache misses by up to 90% (64% on average), and captures up to 450× more rule space than Megaflow within the same hardware memory budget, all while sustaining line-rate performance. Gigaflow has been published at ASPLOS 2025 [182], its artifact is publicly available [115, 42], and ongoing discussions with a major chip manufacturer are exploring integration of Gigaflow cache into future SmartNIC designs.

Kairo: Rethinking the Update Path. While Gigaflow addresses the fast path, Kairo addresses the cost of evicting stale cache entries in response to rule updates. The dominant approach, iterative revalidation, re-executes each cached entry's traversal through the slow path after every rule change—paying a cost $O(E)$ proportional to the number of cache entries, E, regardless of how many entries the update actually invalidates. As caches scale toward millions of entries at 800 Gbps, this becomes an insurmountable bottleneck.

Kairo reframes cache eviction as *Incremental View Maintenance* (IVM): vSwitch rule tables are the database, the fast-path cache is a materialized view, and a rule update ΔR induces a cache update ΔC. Kairo expresses each vSwitch table as an incremental dataflow circuit in DBSP; rule insertions and deletions are streamed in as deltas; only circuits for affected tables activate; and the circuits emit precisely the entries to evict. Eviction cost scales with ΔR, not E—and in practice, $\Delta R \ll E$. Early-exit gates further bound time-to-eviction by halting propagation when an update cannot affect downstream tables, deferring accumulated packet state to a propagation timer T_{prop} that ensures eventual convergence. Together, incremental circuits and gated execution allow Kairo to prioritize eviction work immediately while safely deferring non-critical downstream computation.

Evaluated across four real-world pipelines and five representative update patterns, Kairo delivers 18× faster eviction on average (up to 130×), reduces stale-cache traffic by 35.5× on average, and supports caches orders of magnitude larger than iterative revalidation permits at the same stale-traffic budget.

5.2 vSwitch Architecture for Terabit-Per-Second Era

It is worth being explicit about what Gigaflow and Kairo achieve together, because the significance of their combination is greater than the sum of the parts. Each chapter of this dissertation presents a system that solves one well-defined subproblem. But together, Gigaflow and Kairo constitute a clean-slate redesign of the two performance-critical paths through the modern vSwitch.

Gigaflow redesigns the *fast path*—the mechanism by which packets are matched against cached rules without traversing the full pipeline. It replaces the single-table, traversal-collapsing Megaflow cache with a multi-table, sub-traversal-composing architecture that treats the SmartNIC as the primary forwarding substrate rather than as a constrained accelerator for a CPU-centric design.

Kairo redesigns the *update path*—the mechanism by which stale cache entries are identified and evicted in response to rule changes. It replaces the scan-the-entire-cache revalidation loop with an incremental computation that propagates only the delta induced by each rule change through a circuit mirroring the pipeline structure.

The two systems are orthogonal in design and complementary in deployment: Gigaflow increases the hit rate and effective capacity of the cache, while Kairo makes it safe to operate with a large cache under frequent updates by ensuring that stale entries are evicted quickly and precisely. The result is an architecture in which the vSwitch can sustain high hit rates *and* maintain correctness under dynamic workloads—a combination that was structurally out of reach for the prevailing CPU-centric design. This dissertation thus proposes not two incremental improvements, but a coherent architectural rethinking of the end host network's core forwarding substrate.

5.3 Open Problems and Future Directions

Gigaflow and Kairo together open up a number of interesting directions that are worth pursuing. Each contribution surfaces new questions—about smarter caching strategies, more efficient circuit representations, and deeper integration with the existing vSwitch codebases—that we believe are as interesting as the problems solved here.

Traffic-Adaptive Caching in GIGAFLOW. GIGAFLOW's sub-traversal partitioning is driven entirely by the structure of the vSwitch pipeline and does not adapt to the traffic distribution at runtime. In low-locality environments, where sub-traversal sharing is infrequent, GIGAFLOW's advantage over Megaflow narrows. A natural extension is a traffic-profile-guided optimization that periodically samples traffic from the SmartNIC, estimates sub-traversal sharing frequency, and switches between GIGAFLOW and Megaflow caching strategies as locality conditions change. This would allow GIGAFLOW to degrade gracefully in adversarial traffic conditions rather than falling back to Megaflow by default.

Alternative Circuit Representations in KAIRO. KAIRO currently represents each vSwitch table as a circuit with a set of per-tuple `equi-join` operations, one per wildcard group, based on the Tuple Space Search (TSS) [140] representation. This design ties circuit complexity to wildcard diversity: as the number of distinct wildcard patterns in a table grows, so does the number of circuits and the memory they require. Future work should explore alternative circuit representations that avoid this per-tuple overhead, either by adopting different underlying search algorithms or by introducing new DBSP operators that support wildcard matching more directly.

Native Integration with the OVS Codebase. KAIRO is implemented as a Rust shared library linked into OVS via a C Application Binary Interface (ABI) boundary, building on Feldera's DBSP [20] implementation. While this approach enabled rapid prototyping and avoids reimplementing DBSP from scratch, it introduces a non-trivial foreign-function interface and cross-language memory management overhead. Since KAIRO relies on only a small subset of DBSP operators, a practical direction is to implement limited DBSP support natively within the OVS C codebase, analogous to how OVS implements its internal data structures, thereby eliminating the Rust dependency and simplifying deployment.

Extending the IVM Framing to vSwitch Lookups. KAIRO models only rule tables and cache entries for the purpose of eviction, but the OVS slow path maintains an independent representation of the same rule tables for cache-miss processing and wildcard generation [121, 184]. Extending KAIRO to also handle lookup-time operations—including the generation of non-overlapping wildcarded cache entries that accounts for higher-priority rule masking—would eliminate this redundancy and enable a more unified and efficient slow-path architecture.

Towards a General Slow-Path Accelerator. The slow-path bottlenecks exposed by GIGAFLOW and KAIRO are not unique to virtual switches. A slow path is the third component that links the control and data planes in almost all real SDN deployments [184]. For example, in hardware switches, the switch OS (e.g., Stratum [38], SONiC) handles flow installation, exception processing, and real-time reaction to network events such as link failures and microbursts [176, 50]. In 4G/5G mobile core, the session manager (SMF) [117] handles per-flow signaling, attach/detach events, and user-plane table updates as the network scales to trillions of connected devices. In service meshes, the control proxy (e.g., Istio [61]) manages timeouts, retries, circuit-breaking, service discovery, and adaptive load balancing—operations that already account for up to 44% of service-mesh CPU utilization [71]. In each of these domains, slow-path operations share a common character: they are stateful, compute-intensive, or too memory-hungry to express in a match-action pipeline, yet they must respond at much faster timescales than what a centralized control plane can provide [132, 36, 121]. As link rates push past 200 Gbps and toward 800 Gbps, these slow paths will face the same scaling wall that motivated this dissertation.

The right response is a domain-specific architecture (DSA) for the slow path offload—one that provides predictable tail latencies, high-bandwidth table update throughput, and large off-chip memory pools (e.g., HBMs) to complement the limited on-chip capacity of data-plane ASICs. Extending the match-action abstraction to a *match-compute* model—where a match is followed by a DAG of compute primitives capable of multi-cycle, stateful, and iterative operations—would provide the expressive power needed to program such a DSA across all of these domains.

Gigaflow and Kairo offer a first step in this direction: by fundamentally rearchitecting vSwitch cache generation and eviction through pipeline-aware locality and incremental computation, they provide a concrete reference point for what slow-path acceleration can look like in practice. Distilling these insights, alongside analogous advances across other domains, into a common set of programmable primitives is the path toward a general DSA for the slow path—and one of the most important open problems in networked systems today.

5.4 Closing Remarks

The central bet of this dissertation is that the right response to the terabit-per-second era is not to add more CPU cores or to buy larger TCAM chips, but to reconsider, from first principles, what the vSwitch cache should look like when SmartNICs are the primary forwarding substrate.

Gigaflow's reception—at ASPLOS 2025, at the Google Networking Summit, at the P4 Workshop, in discussions with chip manufacturers, and through a Google Summer of Code implementation—suggests that the community has been waiting for exactly this reframing.

Kairo's core insight—that the database community's decades of work on materialized views and incremental computation maps naturally onto the problem of cache correctness in programmable packet pipelines—is one that surprised even researchers who have worked extensively on both vSwitches and IVM. We hope it signals that the boundary between database systems and network data planes is more permeable than has been assumed, and that the tools developed in one community can be productively transferred into the other.

As link rates continue to climb and CPU performance continues to stagnate, the pressure to eliminate both cache misses and stale-cache processing will only intensify. The systems presented in this dissertation demonstrate that both problems are tractable when approached with the right abstractions: pipeline-aware locality for caching, and incremental computation for eviction. Together, Gigaflow and Kairo redesign both the fast path and update path of the modern vSwitch as a SmartNIC-native architecture—eliminating reliance on faster CPUs, and rearchitecting the end host network for the terabit-per-second era.

APPENDIX A

Gigaflow Artifact Appendix

A.1 Abstract

The artifact includes the source code of the GIGAFLOW cache integrated into Open vSwitch, a traffic generator, and the real-world vSwitch pipelines used in the evaluation. The archived version is publicly available on figshare at https://doi.org/10.6084/m9.figshare.28319711, and we maintain an active repository on GitHub at https://github.com/gigaflow-vswitch. We also actively maintain GIGAFLOW's project website and detailed documentation at https://gigaflow-vswitch.github.io/.

A.2 Artifact Repositories

- **OVS with GIGAFLOW Cache** (gvs): The GIGAFLOW cache implemented in Open vSwitch as a new fast path, available at https://github.com/gigaflow-vswitch/gvs.
- **Traffic Generator** (tgen): A traffic generator used to send and receive packets against the device under test, available at https://github.com/gigaflow-vswitch/tgen.
- **vSwitch Pipelines** (slowpath-pipelines): The real-world vSwitch pipeline rulesets and traffic traces used in the evaluation, available on figshare.
- **Experiment Orchestrator** (Gigaflow-Artifact-ASPLOS2025): An Ansible-based orchestrator that automates testbed setup and experiment execution across the three machines, available at https://github.com/gigaflow-vswitch/Gigaflow-Artifact-ASPLOS2025.

A.3 Dependencies

- **Software**: Autoconf, Automake, libtool, GNU Make, Clang (v3.4 or later), Python (v3.7

or later). All software dependencies for gvs and tgen are installed automatically via Ansible. The only manual prerequisite is installing Docker on the orchestrating machine, which is used to run the Ansible container.

- **Hardware**: Three Linux-based servers are required: a Collector server for storing rulesets, traffic traces, and experiment logs; an OVS Device-under-Test for running gvs; and a Traffic Generator node for running tgen. Each server should be equipped with a 16-core x86_64 CPU, 16 GB RAM, and a DPDK-supported SmartNIC.

A.4 Setup

The experiment orchestrator uses Ansible, launched inside a Docker container, to manage all three machines. Credentials and IP addresses for the Collector, OVS, and Tgen nodes are configured in the inventory.ini file. The Ansible container is started with `make ansible`, after which connectivity to all three machines can be verified with `make ping`.

A.5 Running the Experiments

All experiments are driven via `make` targets executed from within the Ansible Docker container. To run all end-to-end and microbenchmark experiments, the following three commands are used in sequence: `make setup-gvs-experiment` retrieves the rulesets and traffic from the Collector and installs gvs and tgen with their dependencies; `make run-gvs-experiment` loops over all available pipelines, locality settings, and GIGAFLOW table configurations, sends traffic, and collects logs on the Collector; `make teardown-gvs-experiment` uninstalls gvs and tgen and clears logs from the local machines while preserving them on the Collector. End-to-end and microbenchmark experiments can also be run independently using `make run-gvs-ee-experiment` and `make run-gvs-bm-experiment`, respectively. To run a single experiment for a specific pipeline, locality, and GIGAFLOW table configuration, the relevant parameters are set in vars/main.yml and the fine-grained `make` targets—`install-dataset`, `install-gvs`, `install-tgen`, `start-switch-gvs`, `install-rules`, `start-tgen`, `collect-logs`, and their corresponding teardown targets—are executed in order.

APPENDIX B

Kairo Artifact Appendix

B.1 Abstract

The artifact includes the source code of KAIRO, a system for real-time rule updates in Open vSwitch using Incremental View Maintenance (IVM). It is publicly available on GitHub at https://github.com/kairo-cache-eviction. We also actively maintain KAIRO's project website and detailed documentation at https://kairo-cache-eviction.github.io/.

B.2 Artifact Repositories

- KAIRO (kairo): The main repository containing the KAIRO implementation, built on top of DBSP for IVM-based real-time network rule processing.
- **Rulesets** (rulesets): A collection of multi-table OpenFlow-compatible rulesets used in the evaluation, including pipelines from the Gigaflow ASPLOS'25 paper available under `rulesets/gigaflow/`, along with generated update rulesets under `rulesets/updates/` and ruleset configurations under `rulesets/configs/`.

B.3 Dependencies

- **Software**: Rust (for building and running the KAIRO circuits), Python (for ruleset analysis and generation utilities), DBSP (the Rust crate providing the IVM math operations, must be installed locally and its path set in `Cargo.toml`).
- **Hardware**: A Linux-based server equipped with an x86_64 CPU and sufficient RAM to hold the rulesets and DBSP circuit state in memory.

B.4 Setup, Running, and Profiling

Detailed step-by-step instructions for setting up Kairo, preparing rulesets, generating circuit code, running experiments, and collecting timing profiles are documented in the repository's `README.md` at https://github.com/kairo-cache-eviction/.

BIBLIOGRAPHY

[1] Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michael Rusinowitch, Adel Bouhoula, and Mondher Ayadi. Automatically Distributing and Updating In-Network Management Rules for Software Defined Networks. In IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS), 2022.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. VLDB, 2012.

[3] MIT CSAIL Alliances. The Death of Moore's Law: What it means and what might fill the gap going forward. https://cap.csail.mit.edu/death-moores-law-what-it-means-and-what-might-fill-gap-going-forward, last accessed: 05/18/2025.

[4] Amazon. AWS Nitro System. https://aws.amazon.com/ec2/nitro/, last accessed: 05/18/2025.

[5] AMD. AMD PENSANDO DSC3-400 DISTRIBUTED SERVICES CARD. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-dsc3-product-brief.pdf, last accessed: 02/05/2025.

[6] AMD. AMD Vivado Design Suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html, last accessed: 02/05/2025.

[7] AMD. Pensando. https://www.amd.com/en/products/data-processing-units/pensando.html, last accessed: 02/06/2026.

[8] AMD Xilinx. AMD OpenNIC Project. https://github.com/Xilinx/open-nic, last accessed: 02/05/2025.

[9] AMD Xilinx. Vitis Networking P4. https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html, last accessed: 04/11/2024.

[10] Antrea. Antrea: Enhance pod networking and enforce network policies for Kubernetes clusters. https://antrea.io/, last accessed: 02/06/2026.

[11] Antrea. Antrea OVS Pipeline. https://antrea.io/docs/main/docs/design/ovs-pipeline/, last accessed: 02/06/2026.

[12] Antrea-IO. Antrea OVS Pipeline. https://github.com/antrea-io/antrea/blob/main/docs/design/ovs-pipeline.md, last accessed: 02/06/2026.

[13] Josh Bailey and Stephen Stuart. Faucet: Deploying SDN in the Enterprise. https://queue.acm.org/detail.cfm?id=3015763, last accessed: 05/18/2025.

[14] Sorav Bansal, Dharmendra S Modha, et al. Car: Clock With Adaptive Replacement. In USENIX FAST, 2004.

[15] BizTech. Firewall Rules Are Meant to Be Managed, Not Broken. https://biztechmagazine.com/article/2012/08/firewall-rules-are-meant-be-managed-not-broken, last accessed: 01/11/2026.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. In ACM SIGCOMM CCR, 2014.

[17] 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 ACM SIGCOMM, 2013.

[18] BROADCOM. Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800, last accessed: 02/05/2025.

[19] BROADCOM. Trident4 / BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, last accessed: 05/18/2025.

[20] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. VLDB, 2023.

[21] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. Enhancing Global Network Monitoring with Magnifier. In USENIX NSDI, 2023.

[22] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. Enhancing Global Network Monitoring with Magnifier. In USENIX NSDI, 2023.

[23] CAIDA. The CAIDA UCSD anonymized internet traces. https://www.caida.org/catalog/datasets/passive_dataset/, last accessed: 05/18/2025.

[24] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. https://www.rfc-editor.org/info/rfc3376, last accessed: 05/18/2025.

[25] Pei Cao and Sandy Irani. Cost-Aware Proxy Caching Algorithms. In USENIX Symposium on Internet Technologies and Systems (USITS), 1997.

[26] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Ming Liu, and Zeke Wang. Demystifying Datapath Accelerator Enhanced Off-path SmartNIC. arXiv preprint arXiv:2402.03041, 2024.

[27] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In ACM SIGCOMM, 2017.

[28] Corporate Compliance Insights. FireMon's Annual State of the Firewall Report Finds Chronic Deficiencies in Basic Firewall Hygiene. https://www.corporatecomplianceinsights.com/firemons-annual-state-of-the-firewall-report-finds-chronic-deficiencies-in-basic-firewall-hygiene/, last accessed: 01/11/2026.

[29] 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 ACM CoNEXT, 2019.

[30] Jia Cui, Yan Jiang, Lei Song, and Xuewen Zeng. Optimizing Flow Entries Update in SDN Switch with Batch Update Mechanism. International Journal of Innovative Computing, Information and Control, 2021.

[31] Alexis de Talhouët. The evolution of SDN: What service mesh offers telco. https://www.redhat.com/en/blog/evolution-sdn-what-service-mesh-offers-telco, last accessed: 05/18/2025.

[32] DPDK. DPDK. https://www.dpdk.org/, last accessed: 05/18/2025.

[33] Raphael Durner and Wolfgang Kellerer. Network Function Offloading Through Classification of Elephant Flows. IEEE Transactions on Network and Service Management, 2020.

[34] Feldera. The incremental compute platform for data-intensive products. https://www.feldera.com/, last accessed: 01/21/2026.

[35] Firemon. 60% of Enterprise Firewalls Fail Critical Compliance Checks, According to FireMon Insights. https://www.firemon.com/press-room/press-releases/insights-firewall-failure-report/, last accessed: 01/11/2026.

[36] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In USENIX NSDI, 2017.

[37] 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 USENIX NSDI, 2018.

[38] Open Networking Foundation. Stratum - Enabling the Era of Next Generation SDN. https://opennetworking.org/stratum/, last accessed: 05/18/2025.

[39] The Linux Foundation. Linux Bridge. https://wiki.linuxfoundation.org/networking/bridge, last accessed: 05/18/2025.

[40] Github. DPDK. https://github.com/DPDK/dpdk, last accessed: 01/21/2026.

[41] Github. Feldera. https://github.com/feldera/feldera, last accessed: 01/21/2026.

[42] Github. Gigaflow vSwitch. https://github.com/gigaflow-vswitch, last accessed: 01/21/2026.

[43] Github. Open vSwitch. https://github.com/openvswitch/ovs, last accessed: 01/21/2026.

[44] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In ACM ASPLOS, 2019.

[45] Google Summer of Code - P4 Language Consortium. Accelerating OVS with Gigaflow: A Smart Cache for SmartNICs. https://summerofcode.withgoogle.com/archive/2025/projects/YD415t5R, last accessed: 03/08/2026.

[46] Malvika Gupta. Open vSwitch offload by SmartNICs on Arm. https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/open-vswitch-offload-by-smartnics-on-arm, last accessed: 05/18/2025.

[47] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav H. Dave, Andrew W. Moore, and Peter G. Neumann. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In ACM ANCS, 2015.

[48] Hennessy, John L. and Patterson, David A. Computer Architecture: A Quantitative Approach. 6th edition, 2017.

[49] Tom Herbert. The Fast Path/Slow Path Mirage. https://medium.com/@tom_84912/the-fast-path-slow-path-mirage-bb1546358543, last accessed: 03/18/2026.

[50] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In USENIX NSDI, 2019.

[51] IBM. Equi-join. https://www.ibm.com/docs/en/informix-servers/14.10.0?topic=join-equi, last accessed: 02/6/2026.

[52] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In ACM SIGMOD, 2017.

[53] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Efficient Query Processing for Dynamically Changing Datasets. ACM SIGMOD Record, 2019.

[54] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Nina Vortmeier, and Wolfgang Lehner. Conjunctive Queries with Inequalities Under Updates. VLDB, 2018.

[55] IEEE. IEEE Standard for Information Technology - Local and Metropolitan Area Networks - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications-Aggregation of Multiple Link Segments. https://standards.ieee.org/ieee/802.3ad/1088/, last accessed: 05/18/2025.

[56] IEEE. IEEE Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management. http://standards.ieee.org/getieee802/download/802.1ag-2007.pdf, last accessed: 05/18/2025.

[57] Intel. Intel Infrastructure Processing Units (IPUs) and Smart-NICs. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, last accessed: 03/12/2024.

[58] Intel. Intel Ethernet Controller 700 Series - Open vSwitch Hardware Acceleration Application Note. https://builders.intel.com/docs/networkbuilders/intel-ethernet-controller-700-series-open-vswitch-hardware-acceleration-application-note.pdf, last accessed: 05/18/2025.

[59] Intel. Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, last accessed: 05/18/2025.

[60] Intel. Tofino2: Second-generation P4-programmable Ethernet Switch ASIC that Continues to Deliver Programmability without Compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, last accessed: 05/18/2025.

[61] Istio. Simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io/, last accessed: 05/18/2025.

[62] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. Softflow: a middlebox architecture for open vSwitch. In USENIX ATC, 2016.

[63] Anjali Singhai Jain, Mrittika Ganguli, Valas Valancius, and Nupur Jain. Service Mesh P4 Data Plane. https://opennetworking.org/2022-p4-workshop-gated/, last accessed: 05/18/2025.

[64] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In ACM SIGCOMM, 2013.

[65] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In ACM SIGCOMM, 2013.

[66] Joe Stringer. Revaliwhat? Keeping Kernel Flows Fresh. https://www.openvswitch.org/support/ovscon2014/18/1230-revaliwhat.pdf, last accessed: 02/06/2026.

[67] Juniper Networks. Longest Prefix Matching in Networking Chips. https://community.juniper.net/blogs/sharada-yeluri/2023/01/02/longest-prefix-matching-in-networking-chips, last accessed: 02/05/2025.

[68] Minlan Kang, Anirudh Gember-Jacobson, Albert Greenberg, Changhoon Kim, Mohammad Al-Fares, Sangjin Han, Jennifer Rexford, and David Walker. Scalable Rule Management for Data Centers. In USENIX NSDI, 2013.

[69] Georgios Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Maguire. What You Need to Know About (Smart) Network Interface Cards. In International Conference on Passive and Active Network Measurement (PAM), 2021.

[70] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). https://www.rfc-editor.org/info/rfc5880, last accessed: 05/18/2025.

[71] Kinvolk.io. Performance Benchmarks Analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd, last accessed: 05/18/2025.

[72] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2010.

[73] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In USENIX NSDI, 2014.

[74] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In ACM SIGMOD, 2018.

[75] Patricia Kummrow. The IPU: A New, Strategic Resource for Cloud Service Providers. https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/The-IPU-A-New-Strategic-Resource-for-Cloud-Service-Providers/post/1335081, last accessed: 05/18/2025.

[76] Maciej Kuzniar, Peter Peresini, Marco Canini, Dejan Kostic, and Jennifer Rexford. What You Need to Know About SDN Control and Data Planes. Technical Report EPFL-REPORT-199497, EPFL, Lausanne, Switzerland, 2014.

[77] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In USENIX NSDI, 2021.

[78] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. IEEE Transactions on Computers, 2001.

[79] Mingjie Lin, Jianying Luo, and Yaling Ma. A Low-Power Monolithically Stacked 3D-TCAM. In 2008 IEEE International Symposium on Circuits and Systems (ISCAS), 2008.

[80] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic Engineering with Forward Fault Correction. In ACM SIGCOMM, 2014.

[81] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-Based 5G User Plane Function. In SOSR, 2021.

[82] Akbar Majidi, Xiong Gao, Shengkai Zhu, Neda Jahanbakhsh, Jian Zheng, and Guoqiang Chen. MiFi: Bounded Update to Optimize Network Performance in Software-Defined Data Centers. IEEE Transactions on Networking, 2022.

[83] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In SOSP, 2019.

[84] Marvell. Data Processing Units (DPU). https://www.marvell.com/products/data-processing-units.html, last accessed: 05/18/2025.

[85] Marvell. Marvell LiquidIO III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, last accessed: 05/18/2025.

[86] Materialized View. Everything You Need to Know About Incremental View Maintenance. https://materializedview.io/p/everything-to-know-incremental-view-maintenance, last accessed: 02/06/2026.

[87] Ilya Maximets. [ovs-discuss] bond members flapping in state up and down. https://mail.openvswitch.org/pipermail/ovs-discuss/2021-March/050991.html, last accessed: 03/08/2026.

[88] Ilya Maximets. The Morning Paper. https://blog.acolyer.org/2015/06/17/differential-dataflow/, last accessed: 03/09/2026.

[89] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In ACM SIGCOMM CCR, 2008.

[90] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In CIDR, 2013.

[91] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017.

[92] Microsoft. Microsoft announces acquisition of Fungible to accelerate datacenter innovation. https://blogs.microsoft.com/blog/2023/01/09/microsoft-announces-acquisition-of-fungible-to-accelerate-datacenter-innovation/, last accessed: 05/18/2025.

[93] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Hardware-Accelerated Network Control Planes. In HotNets, 2018.

[94] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In ACM SOSP, 2019.

[95] Felicián Németh, Gábor Rétvári, and Marco Chiesa. Normal Forms for Match-Action Programs. In ACM CoNEXT, 2019.

[96] NSC by orhanergun.net. Exploring TCAM Memory. https://netseccloud.com/tcam-memory, last accessed: 02/05/2025.

[97] Nvidia. Nvidia Bluefield DPU BSP v3.8.5. https://docs.nvidia.com/networking/display/bluefielddpuosv385/virtual+switch+on+bluefield+dpu, last accessed: 02/05/2025.

[98] Nvidia. OVS-DOCA Hardware Offloads. https://docs.nvidia.com/doca/sdk/ovs-doca+hardware+offloads/index.html, last accessed: 02/05/2025.

[99] Nvidia. NVIDIA BlueField Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, last accessed: 02/06/2026.

[100] Nvidia. CONNECTX-6 DX. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/, last accessed: 05/18/2025.

[101] NVIDIA. OVS offload using ASAP2 Direct. https://docs.nvidia.com/networking/display/MLNXENv531001/OVS+Offload+Using+ASAP2+Direct, last accessed: 05/18/2025.

[102] Nvidia. Single Root IO Virtualization - SR-IOV. https://docs.nvidia.com/networking/display/mlnxofedv461000/single+root+io+virtualization+(sr-iov), last accessed: 05/18/2025.

[103] Cord OF-DPA. OpenSwitch OF-DPA User Guide. https://netbergtw.com/wp-content/uploads/Files/OPS_of_dpa.pdf, last accessed: 02/06/2026.

[104] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. ACM SIGMOD Record, 1993.

[105] ONF. OpenFlow Table Type Patterns. https://opennetworking.org/wp-content/uploads/2013/04/OpenFlowTableTypePatternsv1.0.pdf, last accessed: 05/18/2025.

[106] ONOS: Open network operating system. https://opennetworking.org/onos/, last accessed: 05/18/2025.

[107] Open Daylight: modular open platform for customizing and automating networks of any size and scale. https://www.opendaylight.org/, last accessed: 05/18/2025.

[108] Open-vSwitch. ofproto-dpif-upcall.c. https://github.com/openvswitch/ovs/blob/master/ofproto/ofproto-dpif-upcall.c, last accessed: 02/06/2026.

[109] Open-vSwitch. Open vSwitch with DPDK. https://docs.openvswitch.org/en/latest/intro/install/dpdk/, last accessed: 02/06/2026.

[110] Openstack. Open vSwitch with DPDK Datapath. https://docs.openstack.org/neutron/latest/admin/config-ovs-dpdk.html, last accessed: 02/05/2025.

[111] Oracle. Datacenter and Server Thermal Trends and Challenges. https://www.meptec.org/Resources/6%20-%20Oracle.pdf, last accessed: 02/07/2025.

[112] OVN. Open Virtual Network. https://www.ovn.org/en/, last accessed: 02/06/2026.

[113] P4 Language Consortium. 2025 P4 Workshop. https://p4.org/event/2025-p4-workshop/, last accessed: 03/08/2026.

[114] P4 Language Consortium. DEMO - Gigaflow: Pipeline-Aware Sub-Traversal Caching for Modern SmartNICs. https://www.youtube.com/watch?v=matjyRjsBbk, last accessed: 03/08/2026.

[115] P4 Language Consortium. Gigaflow - Pipeline-Aware Sub-Traversal Caching for Modern SmartNICs. https://gigaflow-vswitch.github.io/, last accessed: 03/08/2026.

[116] Francesco Paolucci, Davide Scano, Filippo Cugini, Andrea Sgambelluri, Luca Valcarenghi, Carlo Cavazzoni, Giuseppe Ferraris, and Piero Castoldi. User Plane Function Offloading in P4 switches for enhanced 5G Mobile Edge Computing. In International Conference on the Design of Reliable Communication Networks (DRCN), 2021.

[117] Larry L. Peterson, Carmelo Cascone, Brian O'Connor, Thomas Vachuska, and Bruce Davie. Software-Defined Networks: A Systems Approach. Systems Approach LLC, 2021.

[118] Larry L. Peterson and Bruce S. Davie. Computer Networks: A Systems Approach. Morgan Kaufmann, 9th edition, 2021.

[119] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. https://www.rfc-editor.org/info/rfc7047, last accessed: 05/18/2025.

[120] Ben Pfaff and Jesse Gross. Open vSwitch datapath developer documentation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/openvswitch.rst, last accessed: 05/18/2025.

[121] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In USENIX NSDI, 2015.

[122] Diana Andreea Popescu. Latency-driven performance in data centers. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-937.pdf, last accessed: 05/18/2025.

[123] Proxmox. Proxmox. https://www.proxmox.com/en/, last accessed: 02/06/2026.

[124] ACM Queue. A Conversation with Alan Kay. Big Talk With the Creator of Smalltalk—and Much More. https://queue.acm.org/detail.cfm?id=1039523, last accessed: 03/09/2026.

[125] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A Computational Approach to Packet Classification. In ACM SIGCOMM, 2020.

[126] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. Scaling Open vSwitch with a Computational Cache. In USENIX NSDI, 2022.

[127] Anders Rasmussen, A Kragelund, M Berger, Henrik Wessing, and Sarah Ruepp. TCAM-based high speed Longest prefix matching with fast incremental table updates. In IEEE International Conference on High Performance Switching and Routing (HPSR), 2013.

[128] Redhat. What's a service mesh? https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh, last accessed: 05/18/2025.

[129] RedHat Research. An Open vSwitch Security Feature Causes a Security Problem. Here's How To Prevent It. https://research.redhat.com/blog/2024/01/05/an-open-vswitch-security-feature-causes-a-security-problem-heres-how-to-prevent-it/, last accessed: 01/21/2026.

[130] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In ACM SIGCOMM, 2012.

[131] Gerald Rogers and Pravin Shelar. Using Open vSwitch with DPDK. https://github.com/openvswitch/ovs/blob/master/Documentation/howto/dpdk.rst, last accessed: 05/18/2025.

[132] Richard Sanger, Brad Cowie, Matthew Luckie, and Richard Nelson. Characterising the Limits of the OpenFlow Slow-Path. In IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2018.

[133] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's law for traffic offloading. ACM SIGCOMM CCR, 2012.

[134] Scott Schweitzer, CISSP. Power, Heat, Space, and the Move to Double-Wide SmartNICs. https://www.linkedin.com/pulse/power-heat-space-move-double-wide-smartnics-scott-schweitzer-cissp/, last accessed: 02/07/2025.

[135] Serve The Home. Intel X710 OCP NIC 3.0 Power Consumption Specs. https://www.servethehome.com/intel-x710-da2-ocp-nic-3-0-review-10gbe-for-the-form-factor/intel-x710-ocp-nic-3-0-power-consumption-specs/, last accessed: 05/18/2025.

[136] Serve The Home. Pensando Distributed Services Architecture SmartNIC. https://www.servethehome.com/pensando-distributed-services-architecture-smartnic/, last accessed: 05/18/2025.

[137] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In ACM SIGCOMM, 2016.

[138] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source Routed Multicast for Public Clouds. In ACM SIGCOMM, 2019.

[139] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In ACM SIGCOMM, 2020.

[140] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. ACM SIGCOMM CCR, 1999.

[141] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for per-Packet ML. In ACM ASPLOS, 2022.

[142] Synopsys. An Introduction to TCAMs. https://www.synopsys.com/designware-ip/technical-bulletin/introduction-to-tcam.html, last accessed: 02/05/2025.

[143] David E. Taylor and Jonathan S. Turner. ClassBench: A Packet Classification Benchmark. IEEE/ACM Transations on Networking, 2007.

[144] The Economist. Jensen Huang says Moore's law is dead. Not quite yet. https://www.economist.com/science-and-technology/2023/12/13/jensen-huang-says-moores-law-is-dead-not-quite-yet, last accessed: 02/05/2025.

[145] The Netmap Project. netmap - the fast packet I/O framework. http://info.iet.unipi.it/~luigi/netmap/, last accessed: 02/05/2025.

[146] Tigera. Project Calico. https://www.tigera.io/project-calico/, last accessed: 05/18/2025.

[147] Yuta Tokusashi. NetFPGA-PLUS. https://github.com/NetFPGA/NetFPGA-PLUS, last accessed: 04/11/2024.

[148] Jean Tourrilhes, Justin Pettit, et al. OpenFlow switch specification, version 1.5.1 (protocol version 0x06). https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, last accessed: 02/06/2026.

[149] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open VSwitch Dataplane Ten Years Later. In ACM SIGCOMM, 2021.

[150] Tufin. The State of Firewall Security. https://lp.tufin.com/rs/769-ICF-145/images/F_DR_Tufin_State%20of%20Firewall%20Research%20Report_Dec2024.pdf, last accessed: 01/11/2026.

[151] Zahid Ullah, Manish K Jaiswal, Ray CC Cheung, and Hayden KH So. UE-TCAM: An Ultra Efficient SRAM-Based TCAM. In TENCON 2015-2015 IEEE Region 10 Conference, 2015.

[152] VMWare. vSphere Distributed Switch. https://www.vmware.com/products/vsphere/distributed-switch.html, last accessed: 02/05/2025.

[153] VMware. VMware NSX. https://www.vmware.com/products/nsx.html, last accessed: 05/18/2025.

[154] VMware. VMware NSX: Network virtualization platform. https://www.vmware.com/products/nsx.html, last accessed: 05/18/2025.

[155] VMware. VMware's Per-CPU Pricing Model. https://news.vmware.com/company/cpu-pricing-model-update-feb-2020, last accessed: 05/18/2025.

[156] Open vSwitch. Open vSwitch Manual. http://www.openvswitch.org/support/dist-docs/ovs-vswitchd.conf.db.5.pdf, last accessed: 05/18/2025.

[157] Yanshu Wang, Dan Li, Yuanwei Lu, Jianping Wu, Hua Shao, and Yutian Wang. Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness. In USENIX NSDI, 2022.

[158] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, Zhentao Zhang, Zikang Chen, Zeke Wang, Zihui Zhang, Shunmin Zhu, and Wenzhi Chen. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In ACM SIGCOMM, 2023.

[159] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. TernGrad: ternary gradients to reduce communication in distributed deep learning. In NeurIPS, 2017.

[160] Wikipedia. iptables. https://en.wikipedia.org/wiki/Iptables, last accessed: 02/05/2025.

[161] Wikipedia. Moore's law. https://en.wikipedia.org/wiki/Moores_law, last accessed: 02/05/2025.

[162] Wikipedia. Virtual Extensible LAN. https://en.wikipedia.org/wiki/Virtual_Extensible_LAN, last accessed: 02/05/2025.

[163] Wikipedia. VLAN. https://en.wikipedia.org/wiki/VLAN, last accessed: 02/05/2025.

[164] Wikipedia. von Neumann architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture, last accessed: 02/05/2025.

[165] Wikipedia. Loop unrolling. https://en.wikipedia.org/wiki/Loop_unrolling, last accessed: 02/6/2026.

[166] Wikipedia. Strongly connected component. https://en.wikipedia.org/wiki/Strongly_connected_component, last accessed: 02/6/2026.

[167] Wired. Is Moore's Law Really Dead? https://www.wired.com/story/moores-law-really-dead/, last accessed: 02/05/2025.

[168] Xilinx. Vivado Design Suite User Guide - Power Analysis and Optimization. https://www.xilinx.com/support/documents/sw_manuals/xilinx2021_2/ug907-vivado-power-analysis-optimization.pdf, last accessed: 03/02/2025.

[169] Xilinx. Alveo SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/sn1000-product-brief.pdf, last accessed: 05/18/2025.

[170] Xilinx. Alveo U25 SmartNIC. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html, last accessed: 05/18/2025.

[171] Xilinx. Ovs offload. https://www.xilinx.com/publications/solution-briefs/partner/vvdn-ovs-solution-brief.pdf, last accessed: 05/18/2025.

[172] AMD Xilinx. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html, last accessed: 05/18/2025.

[173] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. In Nature Electronics, 2018.

[174] Jialun Yang, Tao Li, Jinli Yan, Junnan Li, Chenglong Li, and Baosheng Wang. Pipecache: High hit rate rule-caching scheme based on multi-stage cache tables. Electronics, 2020.

[175] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In ACM SOSP, 2023.

[176] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive Programmable Switches. In ACM SIGCOMM, 2020.

[177] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In USENIX NSDI, 2024.

[178] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: An Improved LIRS Replacement Algorithm. In Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR), 2021.

[179] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized Stochastic Gradient Descent. In NeurIPS, 2010.

[180] Annus Zulfiqar, Ali Imran, Venkat Kunaparaju, Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. A Smart Cache for a SmartNIC! Scaling End-Host Networking to 400Gbps and Beyond. In IEEE Hot Chips, 2024.

[181] Annus Zulfiqar, Ali Imran, Venkat Kunaparaju, Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. A Smart Cache for a SmartNIC! Rethinking Caching, Locality, & Revalidation for Modern Virtual Switches. In USENIX NSDI Posters, 2025.

[182] Annus Zulfiqar, Ali Imran, Venkat Kunaparaju, Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. Gigaflow: Pipeline-Aware Sub-Traversal Caching for Modern SmartNICs. In ACM ASPLOS, 2025.

[183] Annus Zulfiqar, Ben Pfaff, Gianni Antichi, and Muhammad Shahbaz. Kairo - Incremental View Maintenance for Scalable Virtual Switch Caching. In ACM SIGCOMM Posters and Demos, 2025.

[184] Annus Zulfiqar, Ben Pfaff, William Tu, Gianni Antichi, and Muhammad Shahbaz. The Slow Path Needs an Accelerator Too! In ACM SIGCOMM CCR, 2023.