POSTER: Kairo – Incremental View Maintenance for Scalable Virtual Switch Caching

Annus Zulfiqar, Ben Pfaff[†], Gianni Antichi[‡*], Muhammad Shahbaz
University of Michigan [†]*Feldera* [‡]*Queen Mary University of London* [*]*Politecnico di Milano*

1 MOTIVATION & GOALS

Data centers manage complexity by delegating simple, high-speed packet forwarding to the network fabric and rely on virtual switches (vSwitches) at the end hosts to enforce complex policies—managing connectivity across physical interfaces, containers, and virtual machines (VMs). Since their inception, several key optimizations, including wildcard caches [14], faster lookups using learned index models [15, 16], and high hit-rate SmartNIC offloads [24, 25], have significantly improved vSwitch forwarding performance. Yet, supporting fast vSwitch policy updates has largely been overlooked as they were not deemed performance-critical.

In this paper, we argue that, owing to the architectural evolution of modern vSwitches for performance (from N-table policy to single-table caching [14, 16, 19]) and infrastructure scaling driven by increased link rates and the frequent, diverse update patterns introduced by emerging tenant workloads (e.g., distributed training in the cloud [9, 12, 20, 23] and fast inference at the edge [6, 21])—the unwieldy (bottom-up) vSwitch update mechanism (Figure 1a) has become the primary bottleneck in scaling vSwitch cache sizes to support larger flow rule sets while sustaining high performance.

To address this, we present Kairo. We make the case for framing the problem of maintaining vSwitch cache state as an instance of the Incremental View Maintenance (IVM) problem [4, 10], where cached lookups are efficiently updated by reacting only to changes in vSwitch rules in a top-down manner (Figure 1b), rather than recomputing from scratch. We also outline the key challenges of applying IVM techniques in this setting.

1.1 Cache Evolution in Virtual Switches

To support emerging network virtualization use cases [8, 11, 18], vSwitches have evolved into multi-table packet pipeline processors [14, 16, 19], supporting intricate control flows to implement complex network policies [14, 25]. To cater for such flexibility and deliver high performance at the end host, the Open vSwitch (OVS) introduced Microflow [14, 19] as a fast exact-match cache and relegated the full pipeline to a slow-path for cache misses. Later, wildcarded Megaflow cache [14, 16, 19] significantly reduced cache misses while NuevomatchUP [16] optimized the cache lookup speed by replacing the Tuple Space Search classifier with Machine Learning based learned index models [15]. More recently, Gigaflow [24, 25] proposed a multi-table cache for SmartNIC offloads that leverages pipeline-aware locality, further improving hit rates beyond Megaflow. With these optimizations, vSwitch caching has become indispensable for delivering high throughput and low latency, but maintaining a consistent (update-to-date) view of the vSwitch pipeline poses a significant challenge while scaling to larger cache sizes (e.g., tens of millions of rules per vSwitch) [16, 25, 26].

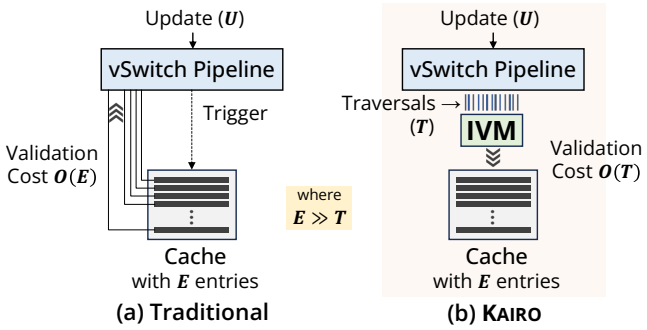

Figure 1: Comparison of (a) traditional (bottom-up) OVS updates versus (b) our proposal, Kairo, (top-down)—with update cost independent of cache size (E).

• ***Updating vSwitch Caches: The Traditional Approach.*** The vSwitch cache [14, 19] captures *traversals* [25] (i.e., a complete sequence of table lookups through the vSwitch pipeline) as single cache entries. However, when vSwitch rules are updated with insertions or deletions, there is no straightforward way to determine which cache entries must change—and how they must change. This challenge arises because the pipeline stores wildcarded rules with priorities, meaning that any rule update can potentially invalidate an arbitrary cache entry. As a result, vSwitches resort to re-evaluating the entire cache [14] by re-executing complete traversals of each entry through the pipeline[1] in order to evict invalidated entries and update others in place, resulting in $O(E)$ complexity, where E is the cache size (Figure 1a).

To understand the cost of this bottom-up revalidation approach as cache sizes grow, we set up the Cord OFDPA pipeline (OFD, 10 tables) pipeline [13] in OVS and generate a workload of 1M unique flows. We then apply rule updates at runtime and measure the time-to-update (TTU) of the cache. As shown in Figure 2, larger caches significantly improve hit rates and reduce cache misses. However, TTU increases

[1]This process, known as cache revalidation, also translates cache statistics (e.g., hits) into flow stats for the pipeline; not on the critical path.

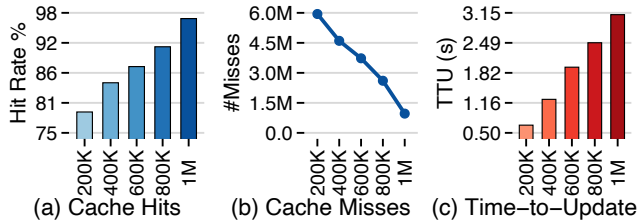

Figure 2: Cache hits (%), cache misses, and time-to-update vSwitch (TTU) with increasing Megaflow cache size using the Cord OFDPA pipeline (OFD) [13].

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 during revalidation. For instance, revalidating 500K entries takes 1.62 s in Cord OFDPA (OFD, 10 tables) [13], compared to 3.6 s in Antrea OVS (ANT, 22 tables) [1–3].

These inefficiencies become even more pronounced at higher link rates (e.g., 400 Gbps) and under emerging update patterns [16, 25], where slow cache updates can prolong policy inconsistencies [7, 14], exacerbate routing errors due to stale flows [14, 17], waste CPU and bandwidth, and delay responses to network events [17, 22].

To mitigate these issues and ensure correct packet processing under frequent or critical updates, OVS thus limits its Megaflow cache size to 200K entries [14]—a trade-off that balances the performance benefits of caching with the overhead of managing invalid entries during delayed updates.

2 CACHE UPDATES AS AN IVM PROBLEM

In this paper, we introduce Kairo, which reimagines cache maintenance as an instance of the Incremental View Maintenance (IVM) problem [4, 10]. Rather than reprocessing the entire cache on every rule update, Kairo incrementally updates only the affected parts of the cache using a top-down strategy (Figure 1b). This approach enables faster responses to rule changes and decouples update cost from cache size.

In database systems, IVM maintains a view $V = Q(DB)$ over a frequently updated database DB by computing the delta ΔV from changes in the underlying database ΔDB—without re-evaluating the full query Q. Drawing on this model, we treat the vSwitch rule tables as the base database and the cache as a collection of materialized views, each representing a unique traversal (T) through the pipeline. This abstraction is especially effective in vSwitches, where rule updates typically affect a small subset of entries (on the order of tens to hundreds), while the cache may represent millions of flows [14, 16, 25]. By focusing only on incremental changes, Kairo enables faster, update-bound maintenance that scales more efficiently than traditional full-cache revalidation.

Update Scheme	Time-to-Update (TTU)		
	100K	150K	200K
Traditional	335.00 ms	503.00 ms	670.00 ms
Kairo	2.72 ms	3.01 ms	3.20 ms

Table 1: Time-to-update (TTU): Kairo vs traditional (bottom-up) with increasing Megaflow cache sizes with Cord OFDPA (OFD, 10 tables) [13] pipeline.

• ***Traversals as Relational Queries for IVM.*** At the core of Kairo is the concept of maintaining traversals—linear, unrolled paths through the vSwitch pipeline—as first-class queries. Each traversal captures the decision logic for a particular control flow based on the current rule set. Traversals are constructed on demand during cache misses and maintained as partial views of the pipeline. When a rule is updated, Kairo uses IVM to adjust only the affected traversals. This targeted update strategy avoids the combinatorial explosion that would result from materializing all possible rule combinations across tables, and ensures that the update cost scales with the number of impacted traversals, $O(T)$, not with the total cache size, $O(E)$.

When a new packet arrives, Kairo checks whether it matches an existing traversal: (1) New Traversal: If no match is found, a new query is generated to represent the packet's traversal, added to the cache, and incrementally maintained with future rule updates. (2) Existing Traversal: If the packet matches an existing traversal, it is recorded in a packet table associated with that traversal. Any future rule updates affecting this traversal trigger incremental updates to the corresponding cache entry. This per-traversal, traffic-driven update model ensures cache consistency without reprocessing unaffected flows. It also aligns naturally with demand-driven caching, where only active parts of the rule space are materialized.

3 PRELIMINARY EVALUATION

We use DBSP [4, 5] as our IVM engine to evaluate Kairo. We implement the Cord OFDPA (OFD) pipeline using DBSP along with a set of incremental queries, each representing a traversal corresponding to a segment of the Megaflow cache. We then populate the pipeline's tables and apply rule updates in a manner that mirrors how Open vSwitch (OVS) performs them at runtime.

Table 1 compares the time-to-update (TTU) performance of Kairo with the traditional bottom-up OVS update scheme across varying Megaflow cache sizes using the OFD pipeline. The conventional approach scales poorly, with TTU reaching 670ms at 200K entries. In contrast, Kairo can achieve up to two orders of magnitude lower TTU, maintaining sub-4ms update times even as cache size increases from 100K to 200K entries, demonstrating the potential of IVM to decouple updates from cache size.

REFERENCES

[1] Antrea. Antrea: Enhance pod networking and enforce network policies for Kubernetes clusters. https://antrea.io/, last accessed: 05/18/2025.

[2] Antrea. Antrea OVS Pipeline. https://antrea.io/docs/main/docs/design/ovs-pipeline/, last accessed: 05/18/2025.

[3] Antrea-IO. Antrea OVS Pipeline. https://github.com/antrea-io/antrea/blob/main/docs/design/ovs-pipeline.md, last accessed: 05/18/2025.

[4] Budiu, M., Chajed, T., McSherry, F., Ryzhyk, L., and Tannen, V. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *VLDB* (2023).

[5] Feldera. Batch jobs waste 99.9% of their time reprocessing unchanged data. https://www.feldera.com/, last accessed: 05/21/2025.

[6] Gobieski, G., Lucia, B., and Beckmann, N. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *ACM ASPLOS* (2019).

[7] Joe Stringer. Revaliwhat? Keeping Kernel Flows Fresh. https://www.openvswitch.org/support/ovscon2014/18/1230-revaliwhat.pdf, last accessed: 05/18/2025.

[8] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., and Zhang, R. Network Virtualization in Multi-tenant Datacenters. In *USENIX NSDI* (2014).

[9] Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A., and Swift, M. ATP: In-network aggregation for multi-tenant learning. In *USENIX NSDI* (2021).

[10] Materialized View. Everything You Need to Know About Incremental View Maintenance. https://materializedview.io/p/everything-to-know-incremental-view-maintenance, last accessed: 05/18/2025.

[11] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. In *ACM SIGCOMM CCR* (2008).

[12] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: generalized pipeline parallelism for DNN training. In *ACM SOSP* (2019).

[13] OF-DPA, C. OpenSwitch OF-DPA User Guide. https://netbergtw.com/wp-content/uploads/Files/OPS_of_dpa.pdf, last accessed: 05/18/2025.

[14] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The design and implementation of open vSwitch. In *USENIX NSDI* (2015).

[15] Rashelbach, A., Rottenstreich, O., and Silberstein, M. A Computational Approach to Packet Classification. In *ACM SIGCOMM* (2020).

[16] Rashelbach, A., Rottenstreich, O., and Silberstein, M. Scaling Open vSwitch with a Computational Cache. In *USENIX NSDI* (2022).

[17] Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. Taurus: A Data Plane Architecture for per-Packet ML. In *ASPLOS* (2022).

[18] Tourrilhes, J., Pettit, J., 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: 05/18/2025.

[19] Tu, W., Wei, Y.-H., Antichi, G., and Pfaff, B. Revisiting the open vswitch dataplane ten years later. In *ACM SIGCOMM* (2021).

[20] Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: ternary gradients to reduce communication in distributed deep learning. In *NeurIPS* (2017).

[21] Xu, X., Ding, Y., Hu, S. X., Niemier, M., Cong, J., Hu, Y., and Shi, Y. Scaling for edge inference of deep neural networks. In *Nature Electronics* (2018).

[22] Yu, L., Sonchack, J., and Liu, V. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM* (2020).

[23] Zinkevich, M., Weimer, M., Li, L., and Smola, A. Parallelized Stochastic Gradient Descent. In *NeurIPS* (2010).

[24] Zulfiqar, A., Imran, A., Kunaparaju, V., Pfaff, B., Antichi, G., and Shahbaz, M. A Smart Cache for a SmartNIC! Scaling End-Host Networking to 400Gbps and Beyond. In *IEEE Hot Chips* (2024).

[25] Zulfiqar, A., Imran, A., Kunaparaju, V., Pfaff, B., Antichi, G., and Shahbaz, M. Gigaflow: Pipeline-Aware Sub-Traversal Caching for Modern SmartNICs. In *ACM ASPLOS* (2025).

[26] Zulfiqar, A., Pfaff, B., Tu, W., Antichi, G., and Shahbaz, M. The Slow Path Needs an Accelerator Too! In *ACM SIGCOMM CCR* (2023).