SwitchNIC: An Hybrid Architecture for Network Functions with Fast and Consistent Shared State

YIRAN LEI, Carnegie Mellon University, USA
FRANCISCO PEREIRA, INESC-ID, Instituto Superior Técnico, University of Lisbon, Portugal
ARVIND KRISHNAMURTHY, University of Washington, USA
JUSTINE SHERRY, Carnegie Mellon University, USA

To support high-performance network functions, recent work has explored hybrid architectures that combine the flexibility of general-purpose processors with the performance of the PISA switch architecture. These designs partition code and state between switch platforms and general-purpose servers in pursuit of a "best of both worlds" design. However, integrating two separate platforms leads to the age-old challenge of ensuring high throughput and low latency while keeping distributed state consistent.

We propose a new, tightly-coupled architecture called SwitchNIC that guarantees consistent state between switch and general-purpose processor, while offering 99^{th} percentile packet latency of 6 μs. The key to SwitchNIC's performance is a novel, "on-path" state replication protocol, with latency further tightened by implementing general-purpose processing on locally-integrated ARM cores. We prototype SwitchNIC by deploying its state replication algorithms on a Tofino switch with Ethernet-attached ARM-based SmartNICs. SwitchNIC achieves up to 5.2× better energy efficiency and 6.4× better processing latency than server-based systems, none of which completely and correctly support shared state between platforms.

CCS Concepts: • **Networks** → **Middle boxes / network appliances**; **Programmable networks**; **Data center networks**.

Additional Key Words and Phrases: Network Functions, Middleboxes, Distributed States, Strong Consistency

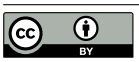
ACM Reference Format:
Yiran Lei, Francisco Pereira, Arvind Krishnamurthy, and Justine Sherry. 2025. SwitchNIC: An Hybrid Architecture for Network Functions with Fast and Consistent Shared State. *Proc. ACM Netw.* 3, CoNEXT4, Article 46 (December 2025), 21 pages. https://doi.org/10.1145/3768993

1 Introduction

Stateful network functions (NFs), such as caches and load balancers, play a vital role in modern networking infrastructure. However, NF developers have historically faced a trade-off between *expressivity* and *high performance* when selecting hardware platforms. Implementing NFs on general-purpose servers enables support for arbitrary computations — such as deep packet inspection — but often results in suboptimal performance. Conversely, programmable switches offer high throughput and low latency for certain NFs (e.g., load balancers), yet their limited computational and memory resources prevent them from efficiently supporting more complex or large-scale NFs.

Recent work has explored heterogeneous architectures that combine the programmability of general-purpose servers with the high performance and energy efficiency of the data plane [22–24, 31, 41]. However, integrating two distinct platforms reintroduces a long-standing challenge:

Authors' Contact Information: Yiran Lei, yiranlei@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Francisco Pereira, francisco.chamica.pereira@tecnico.ulisboa.pt, INESC-ID, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal; Arvind Krishnamurthy, arvind@cs.washington.edu, University of Washington, Seattle, USA; Justine Sherry, sherry@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, USA.



ACM 2834-5509/2025/12-ART46
https://doi.org/10.1145/3768993

achieving high throughput and low latency while maintaining *consistent* distributed flow state — something these systems have struggled to address effectively. Prior efforts have managed distributed flow state using three general approaches, each with inherent performance or correctness limitations: First, systems (like Gallium [41]) *statically partition* flow state, resulting in suboptimal performance when state is anchored at the server and must be processed there, even though it could have been updated more efficiently at the switch. Second, approaches (such as TEA [23] and NetCache [22]) enable state sharing but disallow write operations from the switch, forcing even trivial updates (e.g., counter increments) to be rerouted to the server, thereby limiting performance. Finally, more recent systems (like ExoPlane [24] and SwiSh [40]) allow independent writes on both platforms but do not enforce consistency guarantees, potentially causing the switch and server to observe divergent flow state values, leading to incorrect or conflicting NF behaviors.

We argue that achieving a "best of both worlds" design necessitates that flow states be both shared and updatable across the switch data plane and the server, while maintaining strong consistency. This capability first enables *efficient fast-path processing*, in which flow processing operations requiring only simple computations are directly updated within the high-performance switch data plane — completely bypassing the general-purpose processor. Simultaneously, this capability supports *flexible slow-path processing* for flow processing operations that demand complex operations beyond the switch's capabilities, delegating such tasks to general-purpose cores for arbitrary computation. As a result, stateful NFs that combine both simple and complex logic — such as a packet reassembler (§2.1) — can achieve competitive performance by leveraging the efficiency of the switch without sacrificing functionality. Strong consistency across distributed flow state guarantees that, irrespective of the update location, all state-dependent operations reflect the most recent value, thereby ensuring correctness and coherent NF behavior.

This paper presents SwitchNIC, a hybrid architecture for network functions that enables fast and consistent shared state. Our key insight to achieve strong consistency is straightforward: since the switch data plane lies *on the packet path* to the server — a common setting in data center networks, the system can *always* determine when state synchronization between the switch and the server is required. By embedding state synchronization within the data packets themselves, SwitchNIC eliminates race conditions entirely. To support efficient synchronization across platforms, we implement a fully mutable state table in the switch data plane, allowing flow states to be directly inserted, updated, or deleted — and carried in packet headers — without invoking expensive switch control plane operations. To further reduce processing latency and improve energy efficiency, we design a heterogeneous architecture that co-locates the switch with ARM cores rather than remote general-purpose servers.

We built a prototype of SwitchNIC and evaluated five stateful NFs across three platforms: SwitchNIC, traditional server-based middleboxes, and prior hybrid systems. SwitchNIC enables a range of applications that were previously unsupported or incorrectly handled by existing hybrid approaches, including packet reassemblers, deep packet inspectors, and key-value store caches that allow writes on the switch. Our evaluation demonstrates that SwitchNIC achieves up to 5.2×/1.5× higher energy efficiency and 6.4×/1.6× lower processing latency compared to server-based systems and prior hybrid systems, respectively. Furthermore, SwitchNIC sustains performance under traffic churns two orders of magnitude higher than those tolerated by prior hybrid systems.

Paper roadmap. The remainder of the paper is organized as follows. In §2, we motivate the need for performant, strongly consistent shared state by analyzing the limitations of prior work. We then present a system overview in §3. The strong consistency protocol is detailed in §4, followed by our data plane design in §5. We evaluate the system in §6 and discuss related work in §7.

2 Motivation

Network Functions are critical to today's data center networks, performing a wide range of tasks such as load balancing, statistic collection, and deep packet inspection. They range in complexity across two key dimensions: code complexity, and state complexity. Protocol-Independent Switch Architecture (PISA) [13] switches have proven to be successful at implementing NFs with low code and state complexity. For example, a L4 load balancer [29] maintains a flow-to-port mapping which is written on the first packet of a flow, but then only read by subsequent packets. The code is simple (state is written once and determined by a simple function), and the state is simple as well (flow and port). However, PISA switches fail to implement NFs with complex code or state. For example, they cannot perform many common cryptographic operations, preventing them from supporting many encapsulation protocols [14]. They also limit state usage in both size (providing only a very limited amount of memory at each PISA pipeline stage) and scope (confining state to a single stage and disallowing sequential read/write during packet processing).

This work aims to accelerate stateful NFs with both simple operations (e.g., with deterministic runtime, limited scope of state access, and small-scale flow states) and complex operations (e.g., with non-deterministic runtime, unrestricted state access, and large-scale flow states). A long line of work aims to develop "best of both worlds" designs that integrate PISA switching with server-based general-purpose processing to support both simple and complex operations in one unified design. However, the challenge of state management remains unresolved as, ideally, systems would provide the holy grail of *distributing* state between PISA and traditional compute while guaranteeing *strong consistency between platforms* at *low latency and high throughput*. No hybrid NF design today meets this lofty goal – and indeed, in many classical distributed systems environments, this goal is often impossible. However, as we demonstrate with SwitchNIC, in the context of switch-to-server integration, a distributed, consistent, and high-performance state is indeed possible.

Existing work for high-performance stateful NFs can be classified into four types, each of which fails at correct, distributed, and fast state in a different way:

- **Switch-only** (e.g., SilkRoad [29]): Flow states are entirely stored on programmable switches. These do not support complex state or computations.
- **Static state partitioning** (e.g., Gallium [41]): Flow states are partitioned between the programmable switch and servers but cannot be migrated or shared between platforms. This leads to poor performance, as many NFs update the same state in multiple code paths, some of which are simple while others complex. Since state can only be stored and accessed from one place (lest we have inconsistency), both simple and complex code paths must be mapped to the general purpose platform where the data is stored, removing any benefit from the switch platform.
- **Read-only state** (e.g., TEA [23], NetCache [22]): Flow states are shared between the programmable switch and the servers but are read-only on the switch, which avoids the challenge of ensuring state consistency. This leads to poor performance when simple writes must be routed to the server instead of implemented on the more performant and energy-efficient PISA switch.
- **Inconsistent state** (e.g., ExoPlane [24], SwiSh [40]): Flow states are shared and writable on both platforms, but weakly consistent, causing functionality errors under race conditions.

To illustrate the above challenges, consider the Packet Reassembler NF, which we now describe.

2.1 Packet Reassembler Example

A packet reassembler, illustrated in Figure 1, receives TCP packets that may arrive out-of-order (OOO) and buffers them until the missing packets have arrived. It then releases the packets in order such that they (barring more re-ordering or loss downstream) will arrive in order at the receiving server. Reassemblers can mitigate the overhead of (i) DoS-exploitable extra CPU and

(a) In-order arrival (simple) (b) Out-of-order arrival (complex)

Fig. 1. Packet reassembler has simple/complex operations for in-order/out-of-order packet respectively. Data packets with different sequence numbers are shown as the squares.

memory usage at the server to implement reordering [9, 19], (ii) bug-exploitable packet rewrite attacks [18]. As shown in Figure 1, the NF state has both simple and complex counterparts. The "next expected byte" counter is a simple integer, but the "outstanding packet buffer" of packets waiting to be released is typically implemented as a linked list [44]. Furthermore, the code paths are also varied: packets arriving in order simply update the "next byte expected" counter before being released, while packets arriving for a flow that is missing packets or out of order require the complex operation of checking the linked list of already-arrived packets.

Each of the four classes of switch-based NF designs fails to correctly implement the reassembler example. First, this NF cannot run on switch-only systems, because switches have neither adequate programmability to implement the sophisticated operations nor sufficient memory needed to buffer OOO packets for non-deterministic amounts of time. Second, static state partitioning (e.g., Gallium [41]) forces state to remain immovable and accessible only by a single platform, leading to suboptimal performance — for instance, in-order packets could be processed entirely on the switch, but the state is anchored at the server due to the complex code path required for handling out-of-order packets. Third, read-only state (e.g., TEA [23]) forbids the switch to update flow states, forcing every packet to use the server (as all packets update the state), overloading the servers and causing poor performance. Finally, weakly consistent states (such as Exoplane[24]) can lead to incorrect NF behaviors. For example, with simple operations on the PISA switch and complex operations on the general-purpose server, a partitioned design must share the "next expected byte" counter. Incorrect synchronization could lead to packets being released out of order, packets failing to be released at all, or erroneous connection resets once the device observes a corrupted state.

2.2 SwitchNIC: Achieving Strongly-Consistent Distributed States

Although it seems like a lofty goal, *we find that achieving strongly-consistent distributed state with high performance is indeed possible.* Our system relies on a simple insight: in packet processing with the switch pipeline *on-path* to the server — a common setting in data center networks, the system *always* knows when synchronization between switch and server is required. By piggybacking the state synchronization on the data itself, our system, SwitchNIC, avoids race conditions altogether.

Returning to our example of the reassembler, we now illustrate how SwitchNIC would implement the reassembler correctly with high performance and strongly consistent states. Figure 2 shows the step-by-step operations as SwitchNIC implements both slow- and fast-path updates. For all SwitchNIC systems, all packets arrive initially at the switch, but some may be forwarded by the switch to general-purpose processor cores for processing. For the reassembler, SYN packets are an example of packets that must be forwarded to the processor cores. The processor cores establish the initial connection state for SYN packets and then quickly migrate the state to the switch during $t_0 - t_3$. After setup, the switch efficiently updates the local state, i.e., incrementing the "next-expected-byte" counter, for in-order packets during $t_4 - t_7$. During this time, the processor core replicas become outdated, but this doesn't matter because no processing is ever directed to

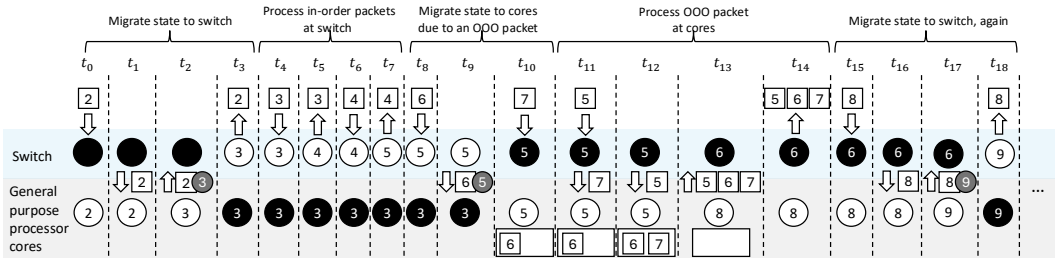


Fig. 2. Example of achieving strong-consistently distributed states in packet reassembler with SwitchNIC: SwitchNIC first migrates the flow states (initialized at general-purpose (GP) cores) to switch during $t_0 - t_3$, directly updates the states (e.g., incrementing "next-byte "counter) for in-order packets at switch during $t_4 - t_7$, piggybacks the latest state on the first OOO packets and overwrites the outdated state at GP cores during $t_8 - t_{10}$, uses the latest state to handle OOO packets (e.g., buffering) at GP cores during $t_{10} - t_{14}$, and finally resume migrating states to the switch for in-order packets from $t_{15} - t_{18}$. Square represents data packet with sequence number; white/black/grey circle represents active/inactive/migrating state.

the processor core. At t_8, an out-of-order (OOO) packet arrives, which cannot be handled on the switch-based fast path.

At this point, the switch knows that the packet must be forwarded to the processor core – *and that the state must be written back as well.* Hence, the arrival of an out-of-order packet triggers *immediate* state migration and sets an "invalid" flag in the switch dataplane. Once the OOO packet, along with the latest state, reaches the processor core at t_{10}, the outdated state is overwritten by the new one, and the processor core starts complex operations such as buffering the out-of-order packet. After the state is migrated to the processor core, subsequent arriving packets for the flow are also redirected to the processor core due to the presence of an invalid flag in the dataplane flow record; we observe this during $t_{10} - t_{14}$. After the gap in the in-order packet buffer is patched, the processor core is no longer required for processing. The buffered packets are released orderly, and the state is migrated back to the switch for fast-path updates during t_{15}–t_{18}, after which the process repeats upon the arrival of the next out-of-order packet.

SwitchNIC allows both the switch and general-purpose processor cores to update the flow states while maintaining strong consistency, because it is never the case that both platforms are updating the state at once. Instead, an explicit ownership handoff is piggybacked on the transferred data itself. We will discuss SwitchNIC's consistency protocol and guarantees further in §4.

It is important to note how the consistency algorithm is customized to the switch/processor setting under data center networks. This is *not* a general consistency protocol. The design works because the switch is always *on-path* for packets entering and exiting the processor cores – hence, the handoff of data ownership can be piggybacked on data transfer. Besides, the state stored on the switch is *necessarily limited* due to the hardware platform constraints, which also results in low state transfer overhead.

3 System Overview

As illustrated in Figure 3, SwitchNIC introduces a tightly integrated heterogeneous architecture composed of a programmable switch and co-located ARM cores to support high-performance stateful NFs. SwitchNIC enables a fast path that handles simple operations and state updates directly in the switch data plane, bypassing the ARM cores, and a slow path that offloads complex processing to the ARM cores. Compared to remote servers, the co-located ARM cores reduce processing latency

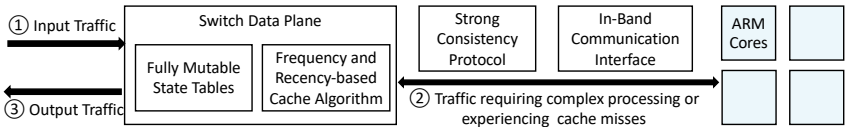


Fig. 3. SwitchNIC overview: it leverages a strong consistency protocol for consistent shared states, fully-mutable state tables for high-performance updates at switch, frequency and recency-based cache algorithm for more fast-path processing, and in-band communication interface for migrating the states via packet headers. Only traffic requiring complex processing or experiencing cache misses at switch needs to use ARM cores.

and improve energy efficiency. Despite distributing flow states across both platforms, SwitchNIC employs the following techniques to implement performant, strongly consistent shared states:
Strong Consistency Protocol (§4): We leverage the fact that the switch is always on-path to the ARM cores to perform just-in-time state synchronization. We implement our insight as a finite state machine on the switch that avoids race conditions altogether.
Fully Mutable State Table (§5.1): We implement a fully mutable state table that supports insertion, deletion, lookup, and update operations on the switch data plane. This data plane mutability enables data packets to efficiently migrate state — i.e., insert or delete entries from the table without invoking costly control plane functions or altering packet order.
Cache Algorithm Based on Recency and Frequency (§5.2): When the number of flow states exceeds the switch's limited memory capacity, the switch memory is treated as a cache managed by a novel replacement algorithm that accounts for both recency and frequency.

4 Strong Consistency Protocol

In this section, we begin by presenting a baseline design to illustrate key design idea. We then identify three correctness issues that can lead to inconsistencies and then present our mitigation strategies. Finally, we describe the complete design of our strong consistency protocol, which manages the flow state using a finite state machine implemented in the switch data plane.

4.1 Baseline Design

There are two key invariants that drive the strong consistency protocol.

- Packets are always first processed by the switch, which determines whether to forward them to the ARM cores or allow them to exit the system directly. This visit order is enforced by the underlying architecture.
- The switch holds the definitive view of where the valid flow state resides and serves as the master replica. The ARM cores do not independently track state; all state coordination and tracking are managed exclusively by the switch.

Together, these two invariants ensure that the switch always makes correct decisions regarding packet forwarding and the timing of flow state synchronization, thereby simplifying the enforcement of consistency guarantees. Next, we walk through a list of representative scenarios, illustrating how packets and states are handled in each case and explaining how consistency is preserved.
Scenario 1: simple processing, valid state at the switch. When the valid state resides at the switch and a packet arrives with processing requirements that can be satisfied locally, the switch directly updates the flow state and forwards the packet without involving the ARM cores, thereby enabling *efficient fast-path processing*. In this scenario, consistency is trivially preserved, as the flow state remains valid and up-to-date within the switch.

Scenario 2: complex processing, valid state at the switch. When the switch receives a packet requiring computation beyond its capabilities, it cannot process it locally — even if it holds the valid flow state. Instead, it piggybacks the current state onto the packet and forwards it to the ARM cores, ensuring the state replica accompanies the data. This enables *flexible slow-path processing*, where the ARM cores first import the latest state and then perform arbitrary computations not supported by the switch. Consistency is preserved, as the packet and its corresponding state travel together. After this state migration, the switch updates its knowledge to reflect that the ARM cores now hold the valid state and thus redirects subsequent packets of that flow to the ARM cores.

Scenario 3: invalid or absent state at the switch. When the flow state is invalid or absent at the switch, the switch detects this using its authoritative knowledge of state locations and forwards the packet to the ARM core. At the ARM core, two cases are possible:

- If the ARM core holds the valid flow state, it updates the state, processes the packet as needed, and returns it to the switch for exit.
- If the flow is entirely new and no valid state exists at the ARM core, a new state is initialized. The processed packet then exits the system after initialization.

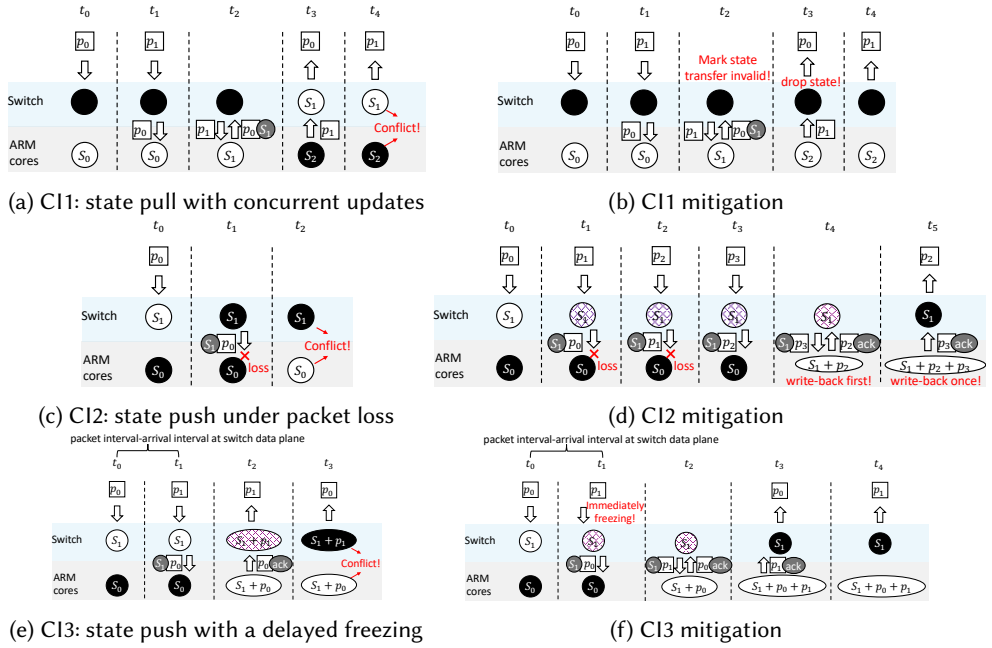
Initializing new flow states at the ARM cores enables complex operations during state creation — such as querying an external database to select a proper backend in a load balancer — which are beyond the switch's capabilities. Although the ARM cores create all new states, they do not track the location of the valid state; this responsibility lies solely with the switch. In this scenario, consistency is trivially maintained, as the valid and up-to-date state resides entirely within the ARM core.

Scenario 4: the switch pulls the state from ARM cores. When the flow state is invalid or absent at the switch, the switch may request the state from the ARM cores in addition to Scenario 3. In Scenario 3, packets are sent to the ARM cores for processing and returned to the switch to exit — *creating a natural round trip*. Scenario 4 leverages this round trip to migrate the state from the ARM cores to the switch along with the data. Specifically, for some Scenario 3 packets, the switch marks them as state requests by setting a flag in the packet header on their first pass. Upon receiving such a packet, the ARM core processes it (i.e., updates its local state) and optionally piggybacks the latest state onto the returning packet. If the ARM core chooses not to migrate the state (e.g., due to pending complex processing), it skips piggybacking. When the switch receives the packet back, it checks for the piggybacked state. If present, it activates the state and enables fast-path processing for future packets. This approach preserves consistency: the switch receives a valid and up-to-date state before use and updates its internal tracking if the pull succeeds.

Although the insight to ensure strong consistency — transferring the valid state along with the data packet prior to its usage — is intuitively straightforward, implementing it correctly presents significant challenges. These challenges emerge in the presence of packet loss between the switch and ARM cores, adversarial packet sequences, and inherent limitations of the switch data plane.

4.2 CI1: state migration from ARM cores to switch with concurrent updates

The first correctness issue (CI) arises when state migration from the ARM cores to the switch overlaps with a concurrent state update, leading to a *stale* state at the switch. Figure 4a illustrates this scenario. Assume the flow state S_0 is initialized at the ARM cores. Packet p_0 arrives and updates the state to S_1, initiating the migration of S_1 to the switch during the interval from t_0 to t_3. Before this migration completes, another packet p_1, belonging to the same flow, arrives at the switch at time t_1. As the switch has not yet received the migrating state S_1, it is unable to process p_1 and instead forwards it to the ARM cores. There, the state is further updated from S_1 to S_2 during interval $t_2 - t_3$, making the in-flight state S_1 obsolete. When p_0 returns to the switch and installs



(a) CI1: state pull with concurrent updates

(b) CI1 mitigation

(c) CI2: state push under packet loss

(d) CI2 mitigation

(e) CI3: state push with a delayed freezing

(f) CI3 mitigation

Fig. 4. Correctness issues (CI) that lead to state inconsistencies during state migration and corresponding mitigation strategies. Square represents data packets; black/shaded/white/grey circle represents inactive/frozen/active/migrating state.

the now-outdated state S_1, all the subsequent packets are erroneously processed based on the stale state S_1, rather than the correct and most recent state S_2.

This correctness issue could be avoided by preventing overlap between state migration and concurrent updates. However, fully eliminating this overlap is infeasible, as it would require either instantaneous state transfer between platforms or the ability for the switch to pause and resume flows undergoing migration — both of which are impractical with current hardware.

Mitigation strategy for CI1. To address this issue, SwitchNIC cancels the state migration when concurrent updates are detected. When the switch initiates a state migration, it also tracks which flow's state is being migrated and monitors for the arrival of subsequent packets from that flow before the migration completes. If such packets are observed, indicating that the state at the ARM cores will be further updated and the in-flight migrating state will be stale, the switch invalidates the migration by discarding the incoming state rather than activating it.

Returning to our example, illustrated in Figure 4b, when the switch begins pulling a flow's state at time t_1, it simultaneously monitors for subsequent packets from the same flow. Upon detecting a new packet at time t_2, it flags the ongoing migration as invalid. As a result, when the migrated (but now outdated) state S_1 arrives at the switch at time t_3, it is discarded. This ensures that subsequent packets are processed using the most recent state S_2 at ARM cores, preserving consistency.

Canceling state migration upon detecting concurrent updates ensures that only the latest state is installed at the switch. However, this also implies that a state may undergo multiple canceled migrations before a success. SwitchNIC optimizes this process by minimizing the time window for concurrent updates through co-location of the switch and ARM cores, thereby reducing the likelihood of cancellation. With a round-trip time (RTT) of ~4 μs between the switch and ARM

cores, any flow below 3 Gbps ensures a sufficient inter-packet gap for successful migration. Even higher-rate flows experience few cancellations due to the natural burstiness of packet arrivals.

4.3 CI2: state migration from switch to ARM cores with packet loss

The second correctness issue occurs when a state updated at the switch fails to be written back to the ARM cores due to packet loss, as shown in Figure 4c. Assume the switch has maintained and updated the flow state for some time, while the corresponding state at the ARM cores S_0 has become stale. At time t_0, a packet p_0 requiring complex processing arrives and triggers a state migration from the switch to the ARM cores. The switch piggybacks the latest state S_1 onto p_0's header, invalidates the local copy, and redirects subsequent packets of the flow to the ARM cores. However, if p_0 is dropped — due to, for example, link failure or ARM core overload, which happens rarely but is nonetheless necessary to consider for correctness — at time t_1, the state S_1 will never reach the ARM cores. As a result, all subsequent packets are processed based on the stale state S_0 at ARM cores, rather than the correct and most recent state S_1, leading to inconsistency.

Note that packet loss in the reverse direction — from the ARM core to the switch during state migration — does not cause inconsistencies, as the ARM core retains the only copy of the state both before and after the loss.

Mitigation strategy for CI2. To address this issue, SwitchNIC employs a continuous write-back strategy. Upon initiating state migration, the switch freezes the local state — making it readable but immutable — and continues piggybacking this frozen state onto packets and sends them to ARM cores, treating them as write-back requests until an acknowledgment (ACK) is received from the ARM cores. Meanwhile, the ARM cores apply only the first successfully received write-back request and ignore the rest redundant ones, thereby ensuring idempotency despite the presence of multiple in-flight write-back packets.

Returning to our example, as illustrated in Figure 4d, when packet p_0 triggers a migration at time t_1, the switch marks the latest state S_1 as frozen and piggybacks it onto p_0 and subsequent packets p_1, p_2, and p_3. Due to packet loss during the interval $t_2 - t_3$, packets p_0 and p_1 fail to reach the ARM cores. However, packet p_2 arrives successfully at t_4. The ARM core then replaces the stale state S_0 with S_1 and updates it to $S_1 + p_2$, subsequently sending an ACK back to the switch. This ACK may be piggybacked on p_2 or sent as a standalone message if buffering occurs. Later, packet p_3, still carrying the now outdated state S_1, arrives at the ARM cores. Since the write-back has already been applied, p_3 skips the write-back and directly updates the state to $S_1 + p_2 + p_3$, also sending an ACK. When the switch receives the ACK from p_2 at t_5, it safely invalidates and deletes S_1. Any subsequent ACKs, such as the one from p_3, are redundant and discarded — though they serve as a fallback in case earlier ACKs are lost.

Our write-back strategy guarantees that state updates occur in the same order as if the state had always resided at the ARM cores. Consider the scenario illustrated in Figure 4d. Suppose the system never migrated state to the switch and instead processed all packets at the ARM cores. In that case, the state would begin at S_1, remain unchanged while packets p_0 and p_1 are lost, and be sequentially updated to $S_1 + p_2$ and then $S_1 + p_2 + p_3$ upon the arrival of packets p_2 and p_3. This matches the final state under our strategy.

4.4 CI3: state migration from switch to ARM cores with delayed state freezing

The final correctness issue arises when the switch *fails to immediately* freeze a state during migration to the ARM cores, allowing subsequent packets to mistakenly use the "frozen" state at the switch and bypass the ARM cores. This can happen if developers do not carefully consider the idiosyncrasies of the PISA switch. Figure 4e illustrates this scenario. Assume the switch holds the latest state, S_1, which must be written back to the ARM cores. At time t_0, packet p_0 triggers the migration, and

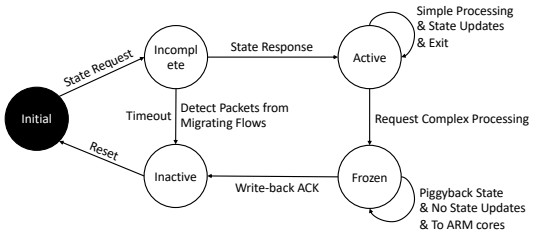


Fig. 5. Flow state lifecycle at switch expressed as a finite state machine.

at t_2, the ARM cores receive p_0 and update the state to $S_1 + p_0$. However, if there is a slight delay between the initiation of migration and the freezing of the state at the switch, a subsequent packet p_1 — say the next packet on the switch data plane — may still treat the switch state as active and update it to $S_1 + p_1$, then exit the system. As a result, by time t_3, both the switch and ARM cores do partial updates and hold diverging state versions, leading to state inconsistencies.

This freezing delay primarily stems from the sequential memory access limitations of PISA switches. The baseline data plane design experiences this delay because it relies on an indicator — placed in earlier pipeline stages than the flow state — to denote the status of the flow state. A packet first reads this indicator; if it says that the flow state is *active*, the packet proceeds to perform a simple update and accesses the flow state. However, in some cases, a packet may only determine that it requires complex processing — and thus triggers state migration — *after* accessing an active flow state. For example, a packet reassembler may realize that this is an OOO packet after comparing and finding the packet's sequence number is larger than the flow state's "next expected byte". At this point, there is no direct mechanism for the packet to update the indicator from *active* to *frozen*, nor to re-access the flow state for piggybacking, as the packet has already passed those pipeline stages. One potential solution is to recirculate the packet, allowing it to re-enter the switch data plane and revisit the relevant stages to update the indicator. However, this approach introduces delays in freezing the state and compromises consistency, as previously discussed.

Mitigation strategy for CI3. SwitchNIC achieves immediate state freezing by introducing a second status indicator and a duplicated state, both placed at a *deeper* pipeline stage than the original state. The second indicator explicitly signals whether the flow state is *frozen*. If not, the packet updates the duplicated flow state in the same manner as the original. If frozen, the packet reads the duplicated state, piggybacks it, and proceeds to the ARM cores. Setting the second indicator ensures the *immediacy* of state freezing — even for the *next* packet in the switch data plane. Although the *next* packet may still update and potentially pollute the original state — since the first indicator, located at an earlier stage, continues to mark the state as *active* — it will correctly detect the state as *frozen* upon reaching the second indicator. This detection triggers the appropriate behavior, preventing further updates at the switch and forwarding the state to the ARM cores.

Returning to the example in Figure 4f, when packet p_0 initiates state migration at time t_1, SwitchNIC instantly marks the associated state as "frozen" by using the proposed technique. Consequently, the next packet p_1 at the switch is aware that the state is immutable and being written back, and thus is forwarded — along with the frozen state — to the ARM cores, thereby preventing partial updates and ensuring strong consistency.

4.5 Put everything together: strong consistency protocol as a finite state machine

We introduce how switch and ARM cores collectively manage the distributed flow state to achieve strong consistency.

Flow state management at switch. We use an independent finite state machine (FSM) to manage the lifecycle of *every* flow state in the switch data plane. This FSM is designed to address the correctness issues mentioned earlier.

Recall that state is always initialized at the ARM cores and later transferred to the switch, which involves three FSM states as shown in Figure 5. Initially, the flow is in the `initial` state, where no flow state is stored at the switch. Migration begins when the switch sends a state request — containing a flow ID and an empty state value — to the ARM cores. At this point, the FSM transitions to `incomplete`, where the switch knows which flow is being migrated but has not yet received its state value. This ARM core-to-switch migration may exit early under two conditions:

- **Concurrent updates**: If the switch observes subsequent packets of the migrating flow before receiving the response from ARM cores, it cancels the migration by transitioning the FSM to `inactive` and discarding the state from ARM cores, avoiding state inconsistency in §4.2.
- **Timeouts**: If no response arrives for a long time, suggesting either the state-carrying packet was lost or the ARM core declined the state transfer request due to pending complex computation, the FSM also transitions to `inactive`. This acts as a self-cleaning mechanism that prevents memory from being occupied by incomplete or pending migrations.

This migration *succeeds* only if the switch receives a valid response from the ARM cores before any subsequent packet from the migrating flow arrives, which transitions the FSM to the `active`. The `active` state enables fast-path execution, allowing local updates and bypassing the ARM cores.

When a packet requiring complex processing enters the switch, the switch immediately freezes the corresponding flow state using the technique described in §4.4, transitioning the FSM from `active` to `frozen`. In the `frozen` state, the switch continuously piggybacks the frozen flow state onto every packet of the flow and forwards them to the ARM cores for write-backs, thereby avoiding the inconsistency discussed in §4.3. This process continues until the switch receives an acknowledgment (ACK) from the ARM cores, indicating a successful write-back and confirming that it is safe to clean the state. At this point, the FSM transitions from `frozen` to `inactive`. Finally, the switch releases all resources associated with the flow (e.g., resetting data plane registers), transitioning the FSM from `inactive` to `initial`, ready to accommodate a new flow state.

Flow state management at ARM cores. The state management at the ARM cores complements the operations in the switch data plane. Upon receiving a request to migrate a state from the ARM core to the switch, the ARM core may:

- Forward the requested state to the switch by piggybacking it onto a response packet; or
- Choose not to respond if the requested state has pending complex processing. In this case, the lack of response causes the FSM at switch transition to `inactive` after a timeout.

Upon receiving a write-back request from the switch, the ARM core first imports the state from the switch before performing any further updates related to that state. It then sends an acknowledgment (ACK) back to the switch to signal a successful write-back. In the presence of continuous write-back requests, the ARM core accepts only the first request and discards the redundant ones by leveraging a dedicated write-back bit:

- The bit is *reset to zero* when the flow state is requested by the switch and migrated to it, indicating that a future write-back will be necessary.
- The bit is *set to one* upon receiving the first write-back request, indicating completion. Subsequent redundant write-backs are discarded, as the bit is already set when they arrive.

5 Switch Data Plane

This section presents the design of a switch data plane tailored for a high-performance caching system along with strong consistency. SwitchNIC selects cached flows based on both frequency and recency, and manages their flow states — such as insertion, deletion, and updates — directly via data packets within the data plane. We introduce the key components in the following subsections.

5.1 Fully Mutable State Table

We implement a fully mutable state table in the switch data plane — supporting packet-driven read, write, insert, and delete operations with deterministic runtime — without relying on match-action tables that require expensive control-plane intervention for updates, as seen in prior hybrid systems [22–24]. This full mutability within the data plane achieves orders-of-magnitude higher insertion and deletion throughput compared to control-plane-based approaches, making it well-suited for handling dynamic traffic churns in data center environments [12, 31]. Such data plane mutability is made possible through the exclusive use of registers for state storage and stateful ALUs — computing circuits with deterministic execution time implemented directly in the switch ASIC — for register manipulation.

The state table is logically a single table structure distributed across multiple pipeline stages, with different columns implemented at different stages in the pipeline. Five columns serve as one-bit control flags, encoding FSM states and updated carefully to ensure correct transitions. The remaining columns store two key–value pairs that maintain both the original and duplicated flow states (as explained in §4.4). Each table entry may either track a single flow — using the flow's ID as the entry key, as in a load balancer [39] — or represent multiple flows that share the same value after hashing their IDs, by storing a common hash value, as in sketch-based NFs [17, 38, 43].

The state table supports packet-driven insertion, deletion, and updates within the data plane:

- **Update:** If a new packet entering the switch encounters a table entry that has already cached its flow state, it directly updates the cached state and exits, when it requires only simple processing. Simple processing with deterministic runtime supported by the switch includes: (i) basic register read/write operations; (ii) arithmetic and logical operations on register values supported by stateful ALUs (e.g., addition); and (iii) computations efficiently approximated via lookup tables, e.g., computing α^n using a precomputed table where the i-th entry stores α^i [32].
- **Insertion:** If a new packet entering the switch maps to an empty table entry, it marks its customized header to request the flow state and is forwarded to the ARM cores. The ARM core processes the packet, piggybacks the requested state, and returns the packet back to the switch. On its second pass, the packet *inserts* the retrieved state into the table entry and then exits.
- **Deletion:** If a new packet entering the switch finds the table entry has already cached its flow state and it requires complex processing, it initiates a write-back request by setting a flag and piggybacking the frozen state in the customized header and is forwarded to the ARM cores. The ARM core imports the state, marks the packet as an acknowledgment (ACK) by setting a header flag (or using a standalone ACK packet if needed), and returns the packet (ACK) to the switch, where the packet *deletes* the state on its second pass and exits.

Due to the limited programmability of registers, the state table is implemented as a hash table. Moreover, the restricted number of pipeline stages in the switch data plane allows only one state table to be supported. These constraints stem from current hardware limitations, which we expect can be alleviated with improved designs. Even under these restrictions, SwitchNIC resolves hash collisions by offloading flows to the ARM cores when their mapped entries are already occupied. In addition, multiple NFs can be fused into a composite NF to accommodate the one-table limitation.

5.2 Cache Algorithm Based on Both Recency and Frequency

We implement a novel cache algorithm that incorporates both frequency and recency to match the traffic characteristics of modern data centers. As observed in prior work [12, 31], data center traffic exhibits high heterogeneity in flow sizes — with 90^{th} percentile flow being orders of magnitude larger than the median — and extreme dynamism, with over 10 million new flows arriving per second. To adapt to these patterns, our cache design prioritizes frequency, by favoring heavy-hitter flows over short-lived "mice" flows, *as well as* recency, by focusing on currently active flows rather than stale ones.

We implement this cache algorithm directly in the data plane to identify and maintain the *most recent, heavy hitter* flows. Our approach is two-fold:

- **Frequency:** SwitchNIC uses a set of Count-Min Sketches [17] to estimate the access frequency (i.e., packet count) of each flow, classifying a flow as a heavy hitter if its frequency exceeds a configurable threshold.
- **Recency:** Only *data packets* can trigger state transfers into the cache. This ensures that inactive flows — those with no incoming packets — even heavy hitters in history will not be cached. Additionally, each cached entry is associated with a time-to-live (TTL) timer, which periodically cleans previously active flows that have become inactive.

Our evaluation shows this frequency-recency-aware caching strategy achieves strong performance under highly skewed and dynamic traffic conditions.

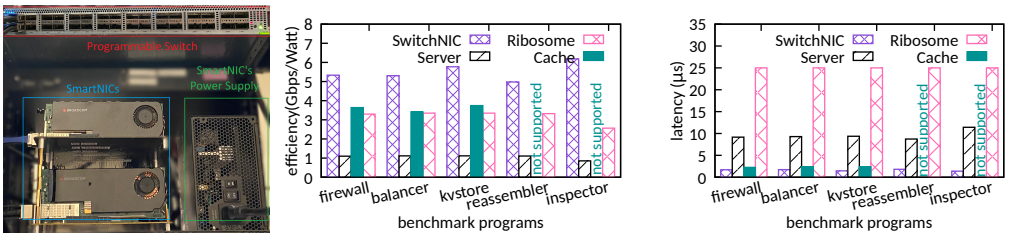
6 Evaluation

Our evaluation seeks to answer the following questions:

- How does SwitchNIC compare to prior server-only systems and hybrid systems in terms of energy efficiency and latency (§6.1)?
- How does the physical distance between heterogeneous platforms impact performance (§6.2)?
- How does the full state mutability at the switch data plane impact performance (§6.3)?
- How do workload characteristics affect system performance (§6.4)? Specifically, the impact of the ratio of complex computation (§6.4.1), ratio of simple write updates (§6.4.2), and the number of heavy hitter flows (§6.4.3)?
- What is the resource overhead of SwitchNIC (§6.5)?

SwitchNIC Implementation & Testbed. Recall that SwitchNIC co-locates a programmable switch with ARM cores. We construct a hardware prototype using standalone commodity components: four Broadcom Stingray PS1100R SmartNICs [1] with ARM cores and one Intel Tofino programmable switch [3]. Each SmartNIC connects to the switch via a 100 Gbps Ethernet link. The SmartNICs are equipped with 8-core ARM Cortex-A72 processors fabricated in 16 nm, operating at 3 GHz, and are paired with dual-channel 8 GB DRAM. Importantly, these SmartNICs are self-powered and operate independently of host servers, representing a minimal yet energy-efficient general-purpose packet processing platform. Figure 6a shows a photograph of the assembled prototype. To compare SwitchNIC against conventional server-based platforms, we also attach two commercial off-the-shelf (COTS) servers to the switch via 100 Gbps Ethernet links. Each server is a dual-socket machine equipped with two 16-core Intel Xeon Gold 5218 CPUs (14 nm, 2.30 GHz) [5].

We implement SwitchNIC strong consistency protocol with approximately 2,700 lines of code at the switch — comprising P4 code for the data plane and C++ for the control plane — and around 1,500 lines of code on the SmartNIC as a DPDK-based userspace application. The full codebase is open-sourced at [25]. We have implemented the consistency protocol — including state migration and custom header parsing — as a callable API on both the switch and SmartNIC/host sides. To



(a) hardware (b) energy efficiency (c) latency

Fig. 6. The SwitchNIC hardware prototype with ARM cores co-located with a programmable switch, and comparisons of its energy efficiency and average processing latency with prior systems.

ease development, we provide a template function that requires the developer only to (i) define the NF's flow states and (ii) specify the computation logic for manipulating those states.

Benchmark Network Functions. We implement five network functions — packet reassembler, packet inspector, key-value store, load balancer, and firewall — as benchmark programs. The first three require strong consistency and involve both simple and complex state updates, while the latter two represent traditional NFs with only simple processing. Detailed descriptions of these benchmarks are provided in Appendix A.1.

Traffic Workload. We use both *uniform* and *skewed* workloads for evaluation. Flows in the uniform workload have an equal number of packets, while the packet count per flow in the skewed workload follows a Zipf distribution with a skewness parameter of 0.99, consistent with prior studies [22, 27]. We first generate PCAP traces on the host and replay these traces using a DPDK-based traffic generator. The workload is characterized by three configurable, orthogonal properties:

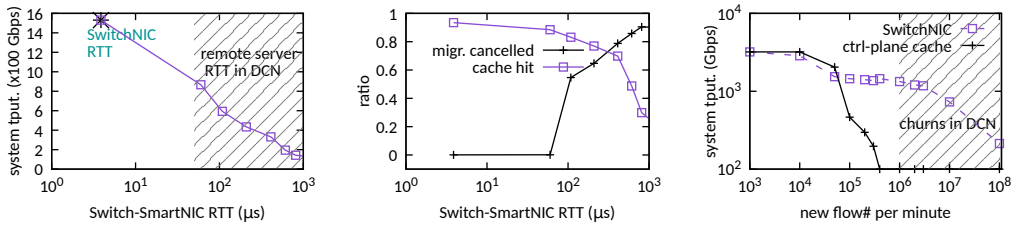
- the number of concurrent flows, ranging from 1,000 to 1 million
- the ratio of packets requiring complex processing, varying from 10^{-5} to 10^{-1}
- the churn size, i.e., the number of new flows per minute, ranging from 1,000 to 100 million

Performance Metric and Measurement. The performance metrics include throughput, energy efficiency, and processing latency. Energy efficiency is calculated by dividing the throughput by the power consumption. We use different methods to measure power consumption across platforms:

- For server, we temporarily disable CPU cores not used by NFs and use Intel SoC Watch [4] to profile the power consumption of the CPU and DRAM.
- For SmartNIC, we employ a WattsUp power meter [8] to measure the comprehensive power consumption of the entire device.
- For switch, we estimate per-port power consumption to be 7 Watts based on a prior study [21].

We use different methods for the SmartNIC and server to ensure fairness: SmartNIC is solely dedicated to NF processing, with no additional cores or memory allocated to other tasks, whereas the server has significantly more resources and is capable of running multiple workloads concurrently.

The latency metric is defined as the time interval between a packet's initial arrival at the switch and the time when its processing completes. For fast-path processing, this corresponds to the packet forwarding latency within the switch data plane pipeline. For slow-path processing that involves the server or SmartNIC, latency is computed by subtracting the timestamp recorded at the packet's first entry into the switch from the timestamp recorded when it re-enters the switch after completing processing on the external platform.



(a) throughput versus distance (b) cache perf. versus distance (c) data-plane state mutability

Fig. 7. Impact of Switch–SmartNIC distance and flow state's data-plane mutability on cache performance.

6.1 SwitchNIC versus Prior Systems

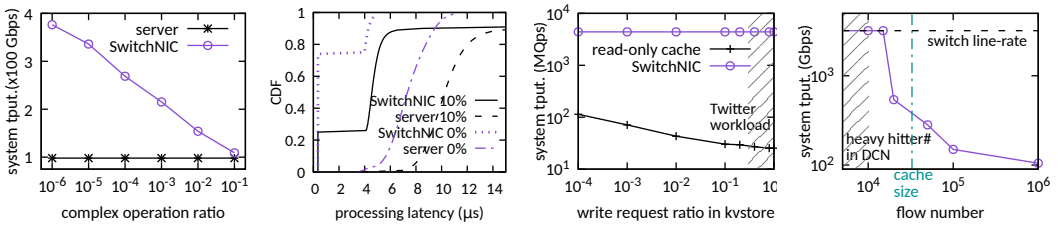
In this section, we evaluate SwitchNIC against three categories of prior systems. The first category includes caching systems (e.g., ExoPlane [24] and NetCache [22]) that store flow states on a programmable switch and offload excess states to an external device. These systems are implemented using the same hardware as SwitchNIC. The second category consists of server-only middlebox systems. The third category is the state-of-the-art server-based accelerator, Ribosome [31], which leverages a programmable switch to separate packet headers from payloads, forwarding only headers to the server for processing. We evaluate all systems using a skewed workload with 100k concurrent flows, 100k new flows per minute (traffic churn), and a 0.01% complex processing ratio, reflecting typical data center conditions [31].

As shown in Figure 6b, SwitchNIC achieves up to ~5.2×, ~1.5×, and ~2.4× higher energy efficiency compared to the prior server-only system, caching system, and Ribosome, respectively. The improvement over the server-only baseline stems from both higher throughput — since most traffic is processed directly at the switch, bypassing the SmartNICs — and the use of energy-efficient ARM cores. Compared to prior caching systems that caches the top heavy hitters, SwitchNIC benefits from greater adaptability to traffic churn. SwitchNIC maintains flow states using a fully mutable state table in the data plane, enabling rapid adaptation to dynamic traffic and achieving a higher cache hit ratio. In contrast, prior caching systems rely on match-action tables, where insertion and deletion are costly operations, making it difficult to keep the cache in sync with newly arriving flows. Note prior caching systems do not support some of the NFs due to lack of strong consistency. Compared to Ribosome, SwitchNIC achieves better energy efficiency through its in-switch caching mechanism, which enables the switch to handle most traffic locally. In contrast, Ribosome lacks a caching layer at the switch and instead separates packet headers from payloads, forwarding only the headers to the server. This design offers better performance than a server-only approach but is less efficient than SwitchNIC due to the absence of local state processing.

As shown in Figure 6c, SwitchNIC achieves up to ~6.4×, ~1.6×, and ~18× lower processing latency compared to the server-only baseline, prior caching system, and Ribosome, respectively. The latency improvement over the server-only baseline is due to SwitchNIC's fast-path processing in the switch, achieving latencies as low as ~350 ns. Compared to prior caching systems, SwitchNIC benefits from a higher cache hit ratio under dynamic traffic conditions. Ribosome exhibits the highest latency, as it must wait for separately delivered packet headers and payloads to arrive at the switch before reassembling them into a complete packet for processing.

6.2 Impact of Platform Proximity

We study the impact of the distance between the switch and the SmartNIC on system performance. To emulate increased separation, we introduce additional packet processing delay at the SmartNIC,



(a) tput. vs. complex ops (b) lat. vs. complex ops (c) tput. vs. write (d) tput. vs. flow#

Fig. 8. Impact of key workload characteristics (ratio of complex operation, ratio of write update, flow number) on the performance of SwitchNIC and prior system.

simulating the latency associated with longer communication paths. The experiment uses the packet reassembler NF under a skewed workload comprising 10k concurrent flows (with a cache size of 30k entries), 100k new flows per minute, and 1024-byte packets.

As shown in Figure 7a, SwitchNIC's throughput decreases as the distance between the switch and SmartNIC increases. This degradation is primarily due to a higher likelihood of canceled state transfers caused by concurrent state updates — a consistency mechanism described in §4.2. The increased distance leads to more frequent cancellations of state transfers, reducing cache occupancy and ultimately lowering the cache hit ratio, as shown in Figure 7b. These findings validate our design choice of co-locating the ARM cores with the switch, achieving an RTT of approximately 4 μs, which maintains a low cancellation rate. By contrast, prior works that connect the switch to remote general-purpose servers can incur latencies exceeding 50 μs [10, 16], risking significant performance degradation under this architecture.

6.3 Impact of Full State Mutability at Switch Data Plane

We compare SwitchNIC's full data plane mutability with prior caching systems' control plane updates across varying traffic churn levels (measured by #new flows per minute). The experiment employs a simple load balancer as the network function, using a skewed workload of 10k flow keys, 1024-byte packets, and a switch cache size of 32k entries.

Figure 7c shows that the throughput of both systems decreases as traffic churn increases. This reduction occurs because higher churn leads to more frequent cache insertions and evictions, which shortens the effective active duration of cache entries. Control-plane-based caching systems, in particular, struggle to keep pace with new flow arrivals once churn exceeds 10^6 flows per minute. This lag causes the cache to fall behind the set of currently active flows, failing to promptly insert new active flows and thereby rapidly reducing the cache hit rate toward zero. In contrast, SwitchNIC can sustain churn rates up to 10^8 flows per minute by handling all state update operations entirely within the data plane. Given that typical churn rates in contemporary data center networks range from 10^6 to 10^8 flows per minute [12, 31], traditional caching solutions risk losing their caching benefits under high churn due to frequent state insertions and deletions.

6.4 Impact of Workload Characteristics

We investigate the impact of key workload characteristics as follows.

6.4.1 Vary Ratio of Complex Computation. We evaluate the impact of complex operations by varying the proportion of packets that require complex computation, which are uniformly distributed throughout the traffic. This experiment uses a packet reassembler, with out-of-order packet arrivals triggering complex buffering operations. The workload follows a skewed distribution and consists

of 100k flow keys, a cache size of 30k entries, and 1024-byte packets. We compare SwitchNIC against a baseline server-only system. Prior caching systems are excluded from this comparison, as they lack the strong consistency to support this complex NFs.

As shown in Figure 8a, the system throughput decreases as the proportion of complex computations increases. This is expected because a larger fraction of packets now require processing by the ARM cores, leading to more frequent state migrations and reduced cache residency time. Nevertheless, SwitchNIC maintains line-rate performance (i.e., 100 Gbps) even under an extreme scenario where 10% of packets arrive out of order — a condition rarely observed in real data center traffic — demonstrating the robustness and efficiency of our design. In terms of latency, SwitchNIC consistently outperforms the server-based system. Figure 8b shows that when no complex computation is needed, over 70% of packets are handled entirely in the switch data plane, achieving fast-path latency of approximately 350 ns. As the complexity ratio increases, more packets are offloaded to the SmartNIC, but SwitchNIC still maintains lower latency than the server due to the SmartNIC's compact design and streamlined interconnects.

6.4.2 Vary Ratio of Simple Write Updates. Supporting simple write operations directly at the cache with strong consistency is also critical for performance. To demonstrate this, we evaluate a key-value store workload with varying ratios of write operations. We compare SwitchNIC— which supports direct state updates in the switch data plane — with a prior caching system, NetCache [22], which does not. In NetCache, every write operation triggers a cache invalidation and redirects the update to the SmartNIC. The workload consists of requests over 10k unique keys, and the switch cache is provisioned with 32k entries, sufficient to theoretically accommodate all keys.

Figure 8c shows that SwitchNIC achieves throughput — measured in queries per second — that is two to three orders of magnitude higher than that of the prior caching system with a read-only cache. In SwitchNIC, all state is cached and updated directly within the switch data plane, allowing throughput to scale up to the switch's native packet forwarding capacity. In contrast, prior system must frequently synchronize with the SmartNIC due to its lack of in-switch mutability, resulting in significant performance degradation as the write ratio increases. This limitation becomes especially problematic in real-world key-value store workloads, such as those observed in Twitter's cache clusters [37], where write-heavy traffic (i.e., write ratios exceeding 30%) is common.

6.4.3 Vary Number of Heavy Hitter Flows. We evaluate the scalability of our hash table design by varying the number of active flows in the system. This experiment uses the packet reassembler with 1024-byte packets and a uniform workload, representing a worst-case scenario for caching since all flows are of equal size and performance is purely constrained by cache capacity. As shown in Figure 8d, SwitchNIC achieves switch line-rate throughput when the number of flows is small, as all the states are cached and processed entirely within the data plane. Throughput begins to degrade before reaching the cache capacity due to inherent hash collisions. As the flow count continues to grow beyond the cache size, performance degrades further because of a declining cache hit rate. Nevertheless, prior work [33] indicates that contemporary data center workloads typically consist of approximately 10^4 heavy-hitter flows — a scale well within SwitchNIC 's capabilities.

6.5 Resource Utilization

SwitchNIC supports up to 32k cache entries at the switch, surpassing the capacity of prior caching systems [22, 24], which typically support 10k entries. This improvement stems from a key architectural difference: prior work mainly uses match-action table — resources that are inherently limited — and SwitchNIC utilizes less constrained switch components. The cache size in SwitchNIC is primarily bounded by the utilization of the Hash Distance Unit (73.6%) and the Stateful ALU

(64.6%), while register usage accounts for 14.17% of available SRAM capacity for 32k entries. At the SmartNIC, our system occupies the only 100 GbE port and employs 2–8 cores and 5 GB of DRAM.

7 Related Work

To optimize the performance of high-speed stateful NFs, prior work has explored homogeneous architectures based on either programmable switches [15, 16, 22, 26, 28–30, 38], which offer high efficiency, or general-purpose servers [2, 6, 7, 11, 20, 36, 42], which provide portability and support for complex functionality. However, switch-based systems are fundamentally constrained by limited memory and programmability, while server-based systems suffer from lower energy efficiency and performance.

To address these limitations, many systems have proposed hybrid architectures that attempt to combine the efficiency of the data plane with the flexibility of general-purpose compute. For instance, Gallium [41] partitions NF code paths, offloading simple codes to the switch and leaving complex logic on the server, thereby improving server efficiency. Other systems such as TEA [23] and ExoPlane [24] overcome switch memory limitations by integrating external memory from servers. A recent system, Ribosome [31], improves server processing efficiency by using the switch to disassemble packets and only forwarding packet headers to servers. Despite these advances, prior hybrid approaches often suffer from poor flow state management, which can degrade performance or lead to incorrect NF behavior. Common issues include read-only state at the switch [22, 23], static state partitioning without sharing [41], or weak consistency in shared state [24, 40].

This work achieves the "best-of-world" design by support performant strongly consistent shared states between platforms, which combines efficient fast-path and flexible slow-path processing.

8 Discussion & Conclusion

Our system can be abstracted as a consistency protocol between two computing nodes that share the same computational states, with one node lying on the data path of the other. This abstraction applies not only to the switch–NIC setting discussed earlier, but also to the NIC–host setting [34], where the NIC provides a high-performance yet constrained processing pipeline and the host acts as a flexible fallback. We argue that achieving strong state consistency in the NIC–host setting is equally important and leave a deeper exploration of this space to future work.

In this paper, we propose SwitchNIC, a tightly integrated heterogeneous architecture that combines a programmable switch with co-located ARM cores to support high-performance stateful NFs. SwitchNIC combines efficient fast-path with flexible slow-path processing through a high-performance strong consistency protocol. We implement SwitchNIC on a hardware prototype and demonstrate that it outperforms prior systems in both energy efficiency and processing latency.

9 Acknowledgment

We are grateful to the anonymous reviewers for their constructive feedback, and to Isabel Suizo, Xiangfeng Zhu, Chenxingyu Zhao, Jaehong Min, and Ming Liu for their support and thoughtful suggestions. This work was supported by ONR Award N000142412059, the European Union (ACES project, 101093126), and INESC-ID (UIDB/50021/2020). Additional support was provided in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF grants CNS-2212193, CNS-2213387, and CNS-2504469. Justine Sherry holds concurrent appointments as an Associate Professor at Carnegie Mellon University and as an Amazon Scholar. This paper describes work performed at CMU and is not associated with Amazon. Francisco Pereira was supported by the FCT scholarship PRT/BD/152195/2021.

References

[1] Broadcom stingray ps1100r. https://gtmteknoloji.com/wp-content/uploads/2020/08/PS1100R-PB100.pdf.

[2] Dpvs. https://github.com/iqiyi/dpvs.

[3] Intel open-tofino. Website. https://github.com/barefootnetworks/Open-Tofino.

[4] Intel soc watch. https://www.intel.com/content/www/us/en/docs/socwatch/get-started-guide/2023-1/overview.html.

[5] Intel xeon gold 5218. https://www.intel.com/content/www/us/en/products/sku/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz/specifications.html.

[6] Katran. https://github.com/facebookincubator/katran.

[7] Openbox. https://openboxproject.github.io/.

[8] Watts up pro. https://arcb.csc.ncsu.edu/~mueller/cluster/arc/wattsup/metertools-1.0.0/docs/meters/wattsup/manual.pdf.

[9] Imad Aad, Jean-Pierre Hubaux, and Edward W. Knightly. Impact of denial of service attacks on ad hoc networks. *IEEE/ACM Transactions on Networking*, 16(4):791–802, 2008.

[10] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of ANCS 2015*. F.R.S.-FNRS - Fonds de la Recherche Scientifique, 07 May 2015.

[12] Theophilus Benson, Aditya Akella, and Dave Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference*. Association for Computing Machinery, Inc., November 2010.

[13] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[14] 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. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[15] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. Millions of low-latency state insertions on asic switches. *Proc. ACM Netw.*, 1(CoNEXT3), nov 2023.

[16] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[18] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, USA, 2001. USENIX Association.

[19] Amir Herzberg and Haya Schulmann. Stealth dos attacks on secure channels. In *Network and Distributed System Security Symposium*, 2010.

[20] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Romain Jacob, Jackie Lim, and Laurent Vanbever. Does rate adaptation at daily timescales make sense? In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 90–106, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. ExoPlane: An operating system for On-Rack switch resource augmentation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1257–1272,

Boston, MA, April 2023. USENIX Association.

[25] Yiran Lei and Francisco Pereira. Switchnic artifact. https://github.com/A-Dying-Pig/SwitchNIC.

[26] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: Performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 516–529, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, March 2016. USENIX Association.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, March 2016. USENIX Association.

[29] 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*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[31] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A High-Speed stateful packet processing approach for tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, Boston, MA, April 2023. USENIX Association.

[32] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, March 2017. USENIX Association.

[33] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and flexible host networking for accelerated computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 405–423, Santa Clara, CA, July 2024. USENIX Association.

[35] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, February 2019. USENIX Association.

[36] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, April 2018. USENIX Association.

[37] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[38] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 561–575, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358, Renton, WA, April 2022. USENIX Association.

[40] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, Renton, WA, April 2022. USENIX Association.

[41] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, page 26–31, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 207–222, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

A Appendix

A.1 Description on benchmark programs

We use the following five stateful NFs as benchmark program

- Packet Reassembler: This NF ensures in-order delivery of TCP packets. Its state tracks the next expected TCP sequence number. Simple computation involves incrementing this state upon receipt of in-order packets, whereas complex computation entails buffering out-of-order packets and releasing them once all prior packets have arrived.
- Packet Inspector: This NF implements an intrusion detection system that scans packets for known patterns using regular expressions. Packets matching certain patterns result in banning the corresponding flow. The NF distinguishes between shallow and deep inspection: shallow inspection examines only the first 64 bytes of the packet, while deep inspection analyzes the remaining payload. Shallow inspection yields three possible outcomes: (1) a full match triggers immediate flow banning, (2) a partial match escalates the packet to deep inspection, and (3) no match results in forwarding or dropping the packet based on the current flow state. Shallow inspection, classified as simple computation, is implemented using the switch TCAM, whereas deep inspection, considered complex computation, is performed on the SmartNIC using Hyperscan [35].
- Key-Value Store: This NF handles read and write requests for a key-value store application. Its state consists of key-value pairs, supporting simple read and write operations that return or update the current value associated with a requested key.
- L4 Load Balancer: This NF distributes connections among a pool of backend servers. Its state consists of backend server IP addresses and TCP ports. It involves no sophisticated computations, performing only simple packet header lookups and modifications.
- Firewall: This NF filters traffic based on a state bit indicating whether the connection is allowed. It only needs operations such as packet dropping.

Received June 2025; accepted September 2025