FAJITA: Stateful Packet Processing at 100 Million pps

HAMID GHASEMIRAHNI, KTH Royal Institute of Technology, Sweden
ALIREZA FARSHIN*, NVIDIA, Sweden
MARIANO SCAZZARIELLO, KTH Royal Institute of Technology, Sweden
GERALD Q. MAGUIRE JR., KTH Royal Institute of Technology, Sweden
DEJAN KOSTIĆ, KTH Royal Institute of Technology, Sweden
MARCO CHIESA, KTH Royal Institute of Technology, Sweden

Data centers increasingly utilize commodity servers to deploy low-latency Network Functions (NFs). However, the emergence of multi-hundred-gigabit-per-second network interface cards (NICs) has drastically increased the performance expected from commodity servers. Additionally, recently introduced systems that store packet payloads in temporary off-CPU locations (*e.g.*, programmable switches, NICs, and RDMA servers) further increase the load on NF servers, making packet processing even more challenging.

This paper demonstrates existing bottlenecks and challenges of state-of-the-art stateful packet processing frameworks and proposes a system, called FAJITA, to tackle these challenges & accelerate stateful packet processing on commodity hardware. FAJITA proposes an optimized processing pipeline for stateful network functions to minimize memory accesses and overcome the overheads of accessing shared data structures while ensuring efficient batch processing at every stage of the pipeline. Furthermore, FAJITA provides a performant architecture to deploy high-performance network functions service chains containing stateful elements with different state granularities. FAJITA improves the throughput and latency of high-speed stateful network functions by ~2.43× compared to the most performant state-of-the-art solutions, enabling commodity hardware to process up to ~178 Million 64-B packets per second (pps) using 16 cores.

CCS Concepts: • **Networks → Middle boxes / network appliances**; *Data center networks*; • **Computer systems organization → Multicore architectures**.

Additional Key Words and Phrases: Packet Processing Frameworks, Stateful Network Functions.

ACM Reference Format:
Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Gerald Q. Maguire Jr., Dejan Kostić, and Marco Chiesa. 2024. FAJITA: Stateful Packet Processing at 100 Million pps. *Proc. ACM Netw.* 2, CoNEXT3, Article 14 (September 2024), 22 pages. https://doi.org/10.1145/3676861

1 INTRODUCTION

Recent advances in networking hardware have boosted the speed of Network Interface Cards (NICs) and packet switching devices, facilitating faster Internet access [1, 2] and improving performance in data centers [3]. CPU core frequencies and memory access speeds have failed to follow the continuing growth in networking speeds [4, 5], and this has made packet processing challenging on commodity hardware, especially when realizing high-throughput NFs that process *hundreds*

*Work was done at KTH Royal Institute of Technology.

Authors' Contact Information: Hamid Ghasemirahni, hamidgr@kth.se, KTH Royal Institute of Technology, Stockholm, Sweden; Alireza Farshin, NVIDIA, Stockholm, Sweden; Mariano Scazzariello, KTH Royal Institute of Technology, Stockholm, Sweden; Gerald Q. Maguire Jr., KTH Royal Institute of Technology, Stockholm, Sweden; Dejan Kostić, KTH Royal Institute of Technology, Stockholm, Sweden; Marco Chiesa, KTH Royal Institute of Technology, Stockholm, Sweden.

ACM 2834-5509/2024/9-ART14
https://doi.org/10.1145/3676861

of millions of packets per second (pps). In addition, recently-introduced systems have achieved unparalleled throughput gains by storing packet payloads in temporary off-CPU locations, such as programmable switches [6], NICs [7], or RDMA servers [8]. These systems must process large amounts of packet headers (*e.g.,* 64 B) on CPUs, which further increases the load on commodity hardware and makes it even more challenging to build such systems. To process packets at today's high rates, recent works advocate: (*i*) offloading computationally expensive operations to programmable network devices & accelerators [9–12] and/or (*ii*) performing optimizations to maximize the benefits provided by the CPU's cache memories [13–18].

While offloading computation to external devices improves performance, stateful applications that have to maintain per-flow data structures (*e.g.,* load balancers, advanced traffic schedulers, and security-related NFs) still struggle to achieve high throughput and low latency for two main reasons. First, stateful NFs have a large memory footprint (*i.e.,* potentially up to gigabytes of state for millions of flows [8]). Unfortunately, today's high-speed accelerators have a constrained SRAM memory and registers (*e.g.,* O(10 MiB) that *cannot* hold these states within the ASIC switches [19]). For instance, Switcharoo [20] can store up to ~128-k flows (4-B per entry) in the data plane. Second, the logic of many advanced packet processing applications (*e.g.,* packet schedulers) *cannot* be realized on high-speed accelerators, such as ASIC switches [8], thus requiring external CPUs.

CPU-based packet processing does not suffer from the above constraints; thus, a CPU may realize arbitrary, advanced *chain of stateful NFs*. However, whenever the necessary state to process a packet is unavailable in the fast cache memories, the processing core must retrieve it from the slower DRAM memory. This retrieval pauses packet processing, significantly harming both throughput and latency if the packet processing pipeline is not optimized for stateful NFs. Processing hundreds of millions of packets per second is only possible when using multiple CPU cores, which requires careful attention to achieve high performance. To achieve linear scaling, most of the existing frameworks rely on *shared-nothing* architectures, where packets of the same flow are forwarded to the same core via a load balancing mechanism (*e.g.,* RSS, RSS++ [21], and Dyssect [22]). This architecture prevents concurrent access to memory locations, avoiding the performance drop caused by synchronization mechanisms.

Nevertheless, employing this architecture is *not* always feasible. For instance, when dealing with multiple stateful NFs with different exclusive definitions of flows, it is *impossible* to follow the shared-nothing architecture across the entire chain of NFs [23] (*e.g.,* a source IP counter tracking the number of distinct source IP addresses and a policer that relies on destination IP addresses have mutually exclusive definitions of flows). To tackle these challenges, this paper introduces FAJITA as an optimized stateful packet processing pipeline designed for commodity hardware. FAJITA (*i*) maximizes the efficiency of each CPU core by combining the benefits of existing processing pipelines and carefully leveraging software prefetching machinery available in modern CPUs. By doing so, FAJITA ensures that the essential data required for processing *a batch* of packets is available in the CPU's caches *before* the CPU needs it, thereby maximizing data locality. Furthermore, FAJITA (*ii*) presents a cache-friendly solution to minimize packet processing costs in scenarios where achieving a shared-nothing architecture is impossible. Figure 1 shows that FAJITA can improve the performance of stateful NF service

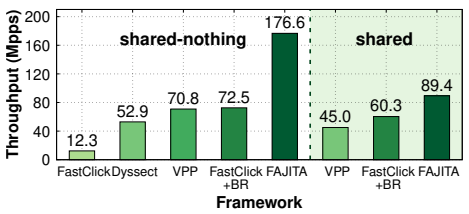


Fig. 1. FAJITA improves the throughput of NF service chains: (*i*) a Flow Statistics Counter followed by a rate limiter and a Load Balancer (shared-nothing region); and (*ii*) a source IP counter followed by a policer and a Load Balancer (shared region) when using 16 CPU cores and receiving 64-B packets & 2 million flows. BR stands for Batch Rebuilder, see § 2.2.

chains compared to existing state-of-the-art solutions (*e.g.,* Dyssect, FastClick, and VPP), by at least ~2.4× & ~1.5× when using shared-nothing & shared architectures, respectively. To the best of our knowledge, FAJITA is the first system that offers a performant solution for (*i*) deploying stateful NF service chains capable of processing ~178 Mpps (*i.e.,* ~1.4 Tbps with average 1-KiB packets)* and (*ii*) supporting chains of NFs with different flow granularities.

Contributions. In this paper, we:

- Evaluate the performance of state-of-the-art packet processing frameworks when dealing with stateful NFs and demonstrate bottlenecks that lead to considerable performance drops.
- Unveil challenges and crucial parameters that affect systems' performance when designing an efficient state-aware packet processing pipeline.
- Design, implement, and evaluate FAJITA to tackle these problems and maximize the performance of the system even with NFs with different or mutually exclusive flow definitions.†

2 BACKGROUND AND MOTIVATION

Stateful NFs present distinct challenges concerning memory utilization and processing speed, hindering existing frameworks from achieving very high packet processing rates (often on commodity servers). For example, Ribosome [8], a state-of-the-art architecture, sends only packets' header to the NFs deployed on commodity servers while storing payloads on external devices, resulting in server loads of potentially over a *hundred million* pps. In this section, we (*i*) identify three commonly-practiced principles for stateful packet processing that are essential to achieve high performance and (*ii*) demonstrate that overlooking any of them negatively impacts performance at high packet rates. To exemplify these principles, we run simple NFs deployed on several state-of-the-art packet processing frameworks (specifically: FastClick [24], Vector Packet Processing (VPP) [25], and Dyssect [22]) under various conditions. Moreover, we use a synthetic workload that offers a configurable number of flows with 64-B packets to maximize the packet rate for a given link bandwidth. Additionally, using 64-B packets emphasizes the importance of high throughput for small packets, to support recently-introduced high-performance systems that only deliver the packet header to NFs [6–8]. We utilize `perf` to monitor the performance of CPU counters (*e.g.,* cache misses) on the Device Under Test (DUT); this profiling has negligible overhead.

Experimental setup. Our testbed contains two commodity servers interconnected via a 32×400-Gbps Edgecore AS9516-32D switch equipped with an Intel® Tofino 2 ASIC [26]. One server acts as a traffic generator; the other is our DUT that runs a stateful round-robin load balancer unless specified otherwise. The traffic generator uses FastClick and measures different percentiles of end-to-end latency. The DUT is equipped with NVIDIA Mellanox ConnectX®-7 @ 200-Gbps NICs [2] and one Intel® Xeon® Gold 6444Y CPUs @ 3.60 GHz with 32-KiB per-core L1 instruction, 48-KiB L1 data, & 2-MiB per-core L2 caches, and a 45-MiB shared Last Level Cache (LLC). The Tofino switch makes clones of incoming packets with different source IP addresses to increase the offered load on the DUT. We plot average values with min/max error bars (although in many experiments, the range is small and almost invisible).

2.1 Principle 1: Minimize Memory Accesses

Stateful NFs typically store states per 5-tuple or with a coarser granularity, requiring a large amount of memory space for millions of flows [27]. This state information needs to be retrieved from potentially slow memory (*e.g.,* DRAM) into higher cache levels (*e.g.,* L1 cache) during packet processing. Typically, when an NF receives a packet, it (*i*) looks for the corresponding state

*Realizing this throughput requires much faster links/ports and/or payload trimming mechanisms.

†All source codes are available at: https://github.com/FAJITA-Packet-Processing-Framework.

and retrieves it, and (*ii*) traverses the state data structure that may contain additional levels of indirection, leading to more memory accesses. These memory accesses have a negative impact on performance, as the CPU must frequently wait for data [28]. Unfortunately, instruction-level parallelism and speculative execution (*e.g.,* hardware prefetching and branch prediction) cannot easily fill these waiting times, as the memory access pattern cannot be predicted.

To reduce the negative impact of managing state, stateful NFs typically employ optimized hash table data structures to ensure fast lookups (*e.g.,* Cuckoo hashing [29, 30]) or fast insertions (*e.g.,* chained hashing) [31]. However, hash tables remain one of the major bottlenecks in stateful NFs [32]. This issue is further exacerbated when the system operates at high packet rates with a large number of flows. To measure the correlation between the number of flows and the throughput of stateful NFs, we use FastClick to deploy a Load Balancer (LB) running on eight CPU cores and gradually increase the number of flows passing through it. Figure 2 demonstrates that an increasing number of flows causes great performance degradation in the LB due to the larger volume of data that must be stored in the system's memory. More specifically, beyond 128-k flows the system's performance rapidly decreases, primarily due to higher LLC misses.

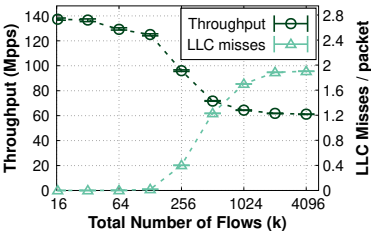


Fig. 2. Increasing the number of flows greatly reduces the performance of a load balancer due to an increase in the number of per-packet LLC misses. Note that the x-axis is logarithmic.

Additionally, more sophisticated NFs, such as advanced schedulers (*e.g.,* Reframer [17]) and security-related NFs (*e.g.,* stateful Deep Packet Inspections (DPIs) & Port Scan Detectors (PSDs)), store a large amount of per-flow data using advanced data structures with non-contiguous data. This potentially translates to multiple irregular memory access patterns, leading to the so-called *pointer chasing* problem. Since these indirect memory accesses are typically not predictable, in most cases, there are longer CPU waits due to the unavailability of data in CPU caches. To measure the impact of irregular memory accesses on the system's performance, we implement a synthetic NF on FastClick, with a configurable number of indirect memory accesses, to mimic the memory access overhead of various NFs with complex data-structures. Figure 3 shows the impact of each memory access on the system's performance when the NF is running on 8 CPU cores and receives 2-million flows.

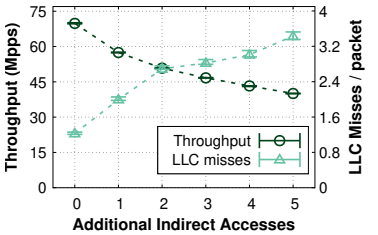
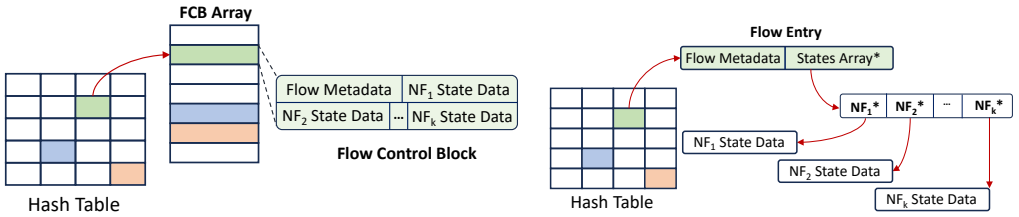


Fig. 3. Increasing the number of irregular memory accesses required for sophisticated NFs reduces the throughput of a stateful NF due to an increasing number of LLC misses.

How do existing systems address this issue? Packet processing frameworks typically take two approaches to implement a chain of stateful NFs. The first approach, as done by VPP [25], maintains a separate hash table to store states for each NF. In this model, each NF operates independently of other NFs, allowing VPP to easily extend a processing graph. However, this approach imposes additional processing overhead due to performing multiple hash table lookups/insertions and increased memory footprint.

The second approach, utilized by FastClick [24] and Dyssect [22], decouples flow management from the actual processing of each NF. By doing so, a single hash table stores the data required by all stateful NFs of a service chain and feeds each NF with the required data through the pipeline. More precisely, FastClick aggregates all required per-flow states into a data structure called Flow Control Block (FCB) and uses a single hash table to assign an FCB to each flow from a large FCB array



(a) FastClick state aggregation schema. (b) Dyssect state aggregation schema.

Fig. 4. FastClick vs. Dyssect state aggregation schema. Each red arrow denotes an irregular memory access.

(Figure 4a). In contrast, Dyssect maintains a data structure called *Flow Entry* that keeps pointers to the corresponding state location for each NF in the service chain.

Consequently, this model requires *at least* one additional memory access per NF compared to FastClick, reducing the benefits of aggregating states (see Figure 4b). Figure 5 shows the throughput of FastClick and VPP when running a round-robin LB and an Access Control List (ACL) individually and consecutively in a chain with 2-million flows with 64-B packets. These results demonstrate that VPP (*i.e.,* using the first approach) performs significantly better than FastClick (*i.e.,* using the second approach) *when running each NF individually* (due to better batching, see §2.2). However, chaining multiple NFs together highlights the overheads of using non-aggregated states. We only focus on VPP and FastClick to show the impact of state aggregation; Dyssect architecture is similar to FastClick, but it is slower (see §4.3).

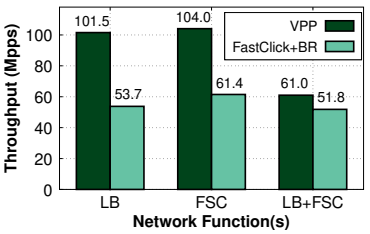


Fig. 5. VPP performs better than FastClick when deploying stateful NFs individually. The overheads of multiple separate hash tables are exacerbated with multiple NFs.

2.2 Principle 2: Perform Batch Processing at Every Stage

The importance of batch processing in achieving high performance is widely acknowledged, as it enhances CPU instruction locality and allows systems to fully utilize prefetching mechanisms available in modern processors. However, existing state-of-the-art systems, tailored for stateful packet processing, sacrifice batch processing at *some stages of the pipeline* to introduce other features (*e.g.,* state aggregation and load balancing). For instance, the state manager of FastClick breaks an input batch of packets into per-flow batches and forwards these *micro-batches* through the pipeline, preventing it from reaching the full potential of the underlying hardware. This behavior also imposes overhead on packet transmission due to multiple transmit calls. To reduce the transmission overhead, FastClick has recently introduced an element called *Batch Rebuilder (BR)* [33]; BR recreates the batch by buffering micro-batches before transmission at the cost of adding queuing delay. This behavior is also evident in Batchy [34] that rebuilds fragmented batches. Dyssect keeps batches intact through the pipeline. However, its dynamic load balancing feature prevents Dyssect from *fully* taking advantage of batch processing benefits. Dyssect assigns multiple shards (each encompassing a hash table and additional statistics) to each core. Consequently, the state of different packets in a batch may be in different hash tables, forcing Dyssect to process each packet independently in the state manager, imposing a noticeable drop in performance.

2.3 Principle 3: Minimize Shared Memory

Existing packet processing frameworks typically follow a run-to-completion model on multi-core servers, where each core instantiates the whole NF service chain and processes packets independently. To achieve a "shared-nothing" architecture with zero synchronization overhead, the majority of systems rely on RSS to dispatch packets among cores and guarantee flow-core affinity. However, we identified two scenarios that necessitate employing a shared architecture, where different cores operate on shared data structures, hence performing shared memory accesses.

Scenario 1: Efficient inter-core load balancing. In some conditions, RSS struggles to efficiently distribute packets among CPU cores, primarily due to having a small number of active flows in a short period of time and the uneven characteristics of flows (*e.g.,* skewed size and different rates) [21, 35]. To address this, state-of-the-art systems propose architectures to fairly balance load among cores by monitoring inter-core load imbalance for short periods and then dynamically migrating active flows to different cores. There is a trade-off between accuracy and migration overhead, as increasing the update frequency leads to a significant overhead of invoking the migration process while increasing the monitoring period leads to a coarser estimation of load imbalance and, consequently, lower load balancing accuracy. Additionally, existing solutions typically impose extra overhead on the processing pipeline due to either using (i) shared data structures for states [36] or (ii) shared variables to monitor per-core load and collect statistics [22]. We argue that the added overhead of existing load-balancing solutions prevents them from efficiently processing packets, and one should employ them only in scenarios when dealing with adversarial traffic patterns containing *less than a few tens of flows* per monitoring period. More specifically, §4.3 (i) shows that intra-server load balancing becomes unnecessary when the system processes more than hundreds of flows in a short period, and (ii) quantifies the overheads of load balancing for Dyssect.

Scenario 2: Stateful NFs with diverse flow definitions. The definition of state is not *exclusive* to the 5-tuple flow identifier. Some NFs use coarser granularity for flow definition (*e.g.,* source IP address for Port Scan Detector (PSD)); §A.1 gives examples of such NFs. In such cases, implementing a shared-nothing architecture requires additional considerations including manually configuring RSS, which is cumbersome. Maestro [37] and FlowMage [23] propose systems to simplify this task by automatically configuring RSS. However, there are many cases where achieving a shared-nothing architecture is *theoretically* impossible [23], regardless of the packet processing framework. Having multiple NFs with exclusive flow definitions is an example of such a case (as highlighted in §1). For instance, consider a chain of stateful NFs containing a policer followed by a PSD. In this scenario, no RSS configuration guarantees flow-core affinity for both NFs at the same time, as the NFs use exclusive flow definitions (*i.e.,* the policer uses destination IP addresses, whereas PSD relies on source IP addresses). Moreover, configuring RSS to dispatch packets based on either the source or destination IP address of packets to achieve a shared-nothing architecture for each of the NFs significantly increases the load imbalance among cores due to a reduction in the cardinality of RSS hash input. In addition to the overheads of shared data structures, having multiple NFs with different flow definitions prevents the packet processing framework from aggregating all states in a *single* hash table. Consequently, we have no option except to rely on separate hash tables per flow definition, which should be shared among cores. For instance, in the above example, aggregating states for the policer and PSD is impossible due to the different keys they use to access the state for a packet. §4.1.1 demonstrates the impact of having shared hash tables on performance.

We observed that existing packet processing frameworks (*i.e.,* FastClick, VPP, and Dyssect) overlook at least one of these principles, making them unable to process *hundreds of millions* of packets per second with many active flows. Unlike the existing state-of-the-art frameworks (see Table 1), FAJITA supports all three principles to accelerate stateful packet processing at 100 Mpps.

Table 1. Comparison of stateful packet processing frameworks respecting the three design principles.

Framework	Features: ✓indicates support, ✗indicates no support		
	Batch Processing	Minimized Mem. Access	Reduce Sync. Overhead
Dyssect	✓	✗	✗
VPP	✓	✗	✗
FastClick	✗	✓	✗
FAJITA	✓	✓	✓

3 FAJITA: STATEFUL PACKET PROCESSING AT 100 MILLION PPS

Efficient stateful packet processing requires an optimized architecture that effectively considers all three principles discussed in §2. This section presents our system, called FAJITA, which addresses the shortcomings of existing frameworks to realize high-performance stateful NF chains on commodity hardware by (*i*) optimizing the processing pipeline to *preserve batches at every stage* and enable batch state lookups; (*ii*) exploiting software prefetching to *accelerate state retrieval at each stage* & hiding memory access overheads; (*iii*) employing a performant architecture to *mitigate the synchronization overhead* when NFs with different flow definitions exist in the chain.

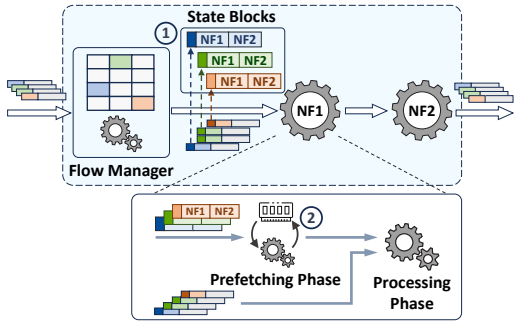


Fig. 6. FAJITA optimizes processing pipeline and performs accelerated state retrieval.

① **Optimized processing pipeline.** FAJITA adapts the most optimized state aggregation architecture (as done by FastClick, see §2.1 and Figure 4a) to take advantage of data locality by keeping all state information required to process a packet in a contiguous memory block. When receiving a batch of packets, FAJITA performs a bulk lookup in the hash table and retrieves all state blocks required for processing the batch. The system allocates a fresh block to packets with unsuccessful lookups and inserts it into the hash table. Next, FAJITA adds a small annotation to each packet containing the address of its corresponding state block, eliminating the need to break the input batch into smaller mini-batches (unlike FastClick). Consequently, *only one* memory access is sufficient to access 5-tuple state information for each packet throughout the pipeline, *regardless* of the number of stateful NFs in the chain.

② **Accelerated state retrieval.** To address the overheads associated with fetching state information and to provide efficient access to complex state data structures that may be used in various NFs, FAJITA exploits software prefetching techniques provided by modern CPUs to accelerate state retrieval and to maximize data locality. More specifically, the system utilizes software prefetching to (*i*) improve bulk lookup by prefetching buckets of the hash table, (*ii*) compensate for the memory footprint of a large aggregated state by loading the appropriate part(s) of state for each NF at the right time, and (*iii*) minimize the overheads of loading data structures with non-contiguous data. Note that this process is mostly done in FAJITA's architecture without relying on software developers' expertise to manually handle memory access overheads (as opposed to VPP), thus simplifying the development of high-performance stateful NFs. Figure 6 shows the processing pipeline in FAJITA when a chain of stateful NFs with the same flow granularity is deployed (*i.e.,* shared-nothing architecture).

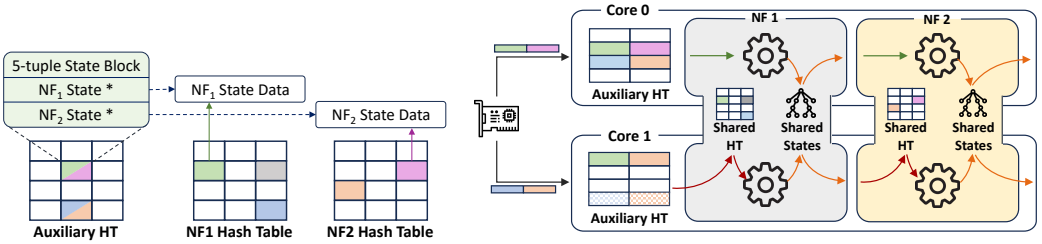


Fig. 7. (a) Auxiliary HTs' schema for supporting NFs with different flow granularities. (b) core 0 shows a scenario with a successful lookup at the auxiliary HT, whereas core 1 depicts the scenario with a failed lookup. As the <blue-orange> entry does not exist in the auxiliary HT, core 1 must perform two hash lookups in NF1 and NF2, while core 0 can retrieve state for the <green-pink> with a single hash lookup in the auxiliary HT.

③ **Mitigating synchronization overhead.** As discussed in §2.3, aggregating state information is impossible when deploying NFs with various flow granularities in a chain since each NF utilizes a different part of packet headers as hash tables' keys (*e.g.,* one NF uses destination IP addresses, while another one uses the source IP address of packets as the key). This condition leaves systems no choice but to maintain a separate hash table per NF, which hurts performance significantly (see Figure 5).* Additionally, in some scenarios when NFs have exclusive flow definitions (*e.g.,* a chain consisting of a policer followed by a PSD), NFs are forced to use shared hash tables and data structures among multiple cores regardless of RSS configuration and framework.

Inspired by the state aggregation architecture, FAJITA proposes a low-overhead solution to reduce shared memory accesses. It adds an additional per-core hash table, called *auxiliary HT*, to the beginning of the NF service chain. Auxiliary HTs use a 5-tuple as the key; this is aligned with the default RSS configuration and guarantees per-flow consistency of data as well as efficient load balancing of traffic among cores.† Auxiliary HTs enable FAJITA to perform only *a single expensive* lookup/insertion from/into shared hash tables. When the first packet of a flow arrives, FAJITA performs a lookup from shared hash tables and stores pointers to states in the auxiliary HT. Consequently, the remaining lookups can be performed directly from the auxiliary HT without any synchronization overhead. In cases where the chain contains NFs with 5-tuple flow granularity, FAJITA dynamically aggregates the state information for corresponding NF(s) & stores them into the auxiliary HT (avoiding having separate hash tables). As a result, for each packet, the state information for NFs with 5-tuple flow granularity and *pointers* to the state data of NFs with coarser flow definitions are aggregated in the same memory block. Figure 7 illustrates the state block structure used in the auxiliary HT when two NFs with coarse definitions of flows exist in the chain. Note that auxiliary HTs eliminate shared hash table lookups while CPU cores still need to access the shared state to read/update information per flow (see Figure 7b). Consequently, FAJITA utilizes locking mechanisms to update shared state information during the processing of a packet to ensure the integrity of data.

When does the auxiliary HT reduce shared memory accesses? When at least one NF with a 5-tuple flow definition exists in the chain, the auxiliary HT does **not** introduce any memory access overhead since the system aggregates the state of those NFs in the auxiliary HT; hence, the auxiliary HT always leads to fewer hash table lookups. In contrast, consider a scenario where

*NFs with a similar granularity can still aggregate their states in a single hash table.
†It is possible to use different keys and RSS configurations.

h NFs with various flow definitions are deployed on a server, each having a separate hash table with a flow granularity other than 5-tuples. If we assume that the network traffic in a given time window contains f flows with the average size $\overline{p}_f$, then without the auxiliary HT the total number of accesses to the shared hash tables is $N_{no\_aux} = f * \overline{p}_f * h$. With auxiliary HT, when the first packet of a 5-tuple flow arrives, since no record exists for the corresponding flow in the auxiliary HT, it causes an extra unsuccessful lookup and stores a pointer to the state in the auxiliary HT. For the rest of the packets belonging to this flow, FAJITA can retrieve the state information directly from the auxiliary HT thus avoiding additional lookups in the rest of the hash tables, thus the total number of hash table accesses $N_{aux} = f * (h+1) + f * (\overline{p}_f - 1)$. The auxiliary HT reduces hash table lookups when $N_{no\_aux} \geq N_{aux}$ *i.e.,* when $\overline{p}_f \geq \frac{h}{h-1}$. The maximum value of $\frac{h}{h-1} (h \in \mathbb{N}\, and\, h > 1)$ is 2. Therefore, in the worst-case scenario, auxiliary HT leads to fewer shared hash table lookups when the average flow size of traffic is greater than 2, which is satisfied even for highly skewed network traffic. To put this value into perspective, we consider CAIDA to be a traffic trace with a highly skewed flow size distribution [22]. Our analysis shows that the average number of packets per flow in CAIDA is ~19.7. Increasing the number of NFs in the chain or the average number of packets per flow further increases the benefits of auxiliary HT.

The combination of ①, ②, and ③ enables FAJITA to take full advantage of all three principles essential for stateful packet processing at high rates. FAJITA reduces wasted CPU cycles & shared memory accesses, and improves the performance of *every* stateful NF in the chain by making the most out of caches, which is essential for high-speed packet processing. To avoid overheads of active load balancing, FAJITA relies on RSS to dispatch packets among cores; see §4.3 for detailed analysis. Moreover, FAJITA is the first packet processing framework that presents a solution for supporting NF service chains with multiple flow granularities.

3.1 Implementation

We implemented FAJITA as an extension to FastClick. By doing so, we enable network developers to benefit from *both* FAJITA and other optimizations previously integrated into FastClick. Next, we explain the detailed implementation of our proposed optimizations into FastClick.

① To realize an optimized packet processing pipeline, FAJITA (*i*) leverages FastClick's aggregated per-flow data structure (*i.e.,* FCB*) to store all necessary data required by stateful NFs in a single hash table, (*ii*) exploits DPDK's Cuckoo hashing API (*i.e.,* `rte_hash`) to perform bulk lookups, and (*iii*) extends the packet metadata of FastClick to store 8 bytes of annotation per packet containing a pointer to the corresponding FCB throughout the processing path.

② To accelerate state retrieval, FAJITA uses x86 prefetching instructions to prefetch (*i*) primary & secondary buckets of the hash table during the bulk lookup†, (*ii*) the right cache line of the FCB that stores the state belonging to each NF, and (*iii*) proactively load the states stored in advanced data structures (*i.e.,* containing indirect accesses). For the latter, FAJITA offers a customizable function that allows developers to tailor the prefetching process to suit the specific data structures used by each NF, offering flexibility and simplifying the development process.

Prefetching considerations. The use of software prefetching should be approached with care, as its effectiveness depends on various factors, such as the prefetching time and memory access patterns. Stateful NFs are particularly memory intensive, so inefficient use of prefetching can cause cache pollution and waste memory bandwidth, ultimately negatively impacting performance. Moreover, modern CPUs often include hardware prefetching capabilities, and the use of software prefetching should be avoided when it could potentially conflict with hardware prefetching. For

*As explained in §2.1; more details available in [38].

†This is inherently supported by `rte_hash_lookup_bulk_data`.

example, the hardware prefetching mechanism typically prefetches the next cache line(s) after touching/loading a cache line. Therefore, FAJITA only utilizes software prefetching to fetch the *first cache line* of large per-flow states and relies on hardware prefetching to do the rest.
③ During the initialization, FAJITA creates a pool of empty state blocks (*i.e.,* FCB list). When FAJITA uses shared-nothing architecture, it allocates memory in such a way that all states of a flow can be packed back-to-back to maximize data locality. Conversely, when FAJITA needs to share states among multiple cores, it separates states belonging to different flows by a cache line to prevent the overhead of false sharing and avoid unnecessary cache line invalidations & cache trashing due to hardware prefetching when different cores access different flow states. In the latter case, when FAJITA receives a batch, it performs a bulk lookup in the auxiliary HT to fetch the list of state blocks for the batch. FAJITA keeps a mask denoting unsuccessful lookups. For such cases, FAJITA uses the provided mask to perform the required insertion/lookup in the shared hash tables. At the end of processing a batch, FAJITA updates the missing state blocks in the auxiliary HT to have a successful lookup for the further packets of the same flows. FAJITA also implements efficient state removal for the latter case. To do so, FAJITA maintains a counter for each coarse-grained state data to keep track of the number of flows with finer-grained granularities sharing this state. For example, multiple 5-tuple flows share the same state in a source IP rate limiter, where removing the rate limiter state is only acceptable when all 5-tuple flows have either timed out or ended.

4 FAJITA EVALUATION

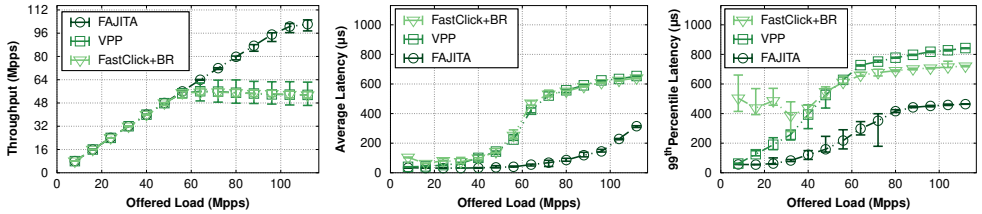
This section demonstrates the effectiveness of FAJITA in improving the performance of stateful NFs. To do so, we use the same testbed described in §2 and compare FAJITA's performance with FastClick, Dyssect, and VPP as three state-of-the-art frameworks capable of deploying stateful NF chains. Furthermore, we consider FastClick+BR as the baseline for FastClick, as Figure 1 highlighted the overheads of FastClick without BR.

NF configurations. To ensure a fair comparison among all frameworks, we developed stateful NFs with identical behavior across them. More specifically, we use chains of stateful NFs that include a stateful round-robin load balancer, a Flow Statistics Counter (FSC), and a flow rate limiter. Additionally, in experiments involving NFs with coarse flow definitions, we incorporate a policer and a source IP tracker. The composition of the NF chain is stated for each experiment.

Traffic workload. We use both (*i*) synthetic 64-B traces to show the full potential of FAJITA when processing small-sized packets with the least spatial locality (*i.e.,* consecutive packets belong to different flows) and (*ii*) realistic traces (*e.g.,* CAIDA and a university campus trace) to demonstrate the applicability of our optimizations to other workloads. We chose two different realistic traces to examine the effect of traffic locality. In particular, our campus trace is more bursty with fewer flows, which represents a worst-case scenario for showcasing FAJITA's improvements when dealing with very skewed flow size distribution. As the realistic traces have a low packet rate, we split the trace into smaller windows and then replay multiple windows in parallel to increase the offered load without affecting the distribution and properties of flows. Additionally, we truncate packets to 64 B to report the maximum possible throughput in terms of pps without saturating the link. We use CAIDA traffic traces in the experiments unless stated otherwise.

4.1 Does FAJITA Improve Performance?

Figure 1 already showed the benefits of FAJITA, where it could achieve up to ~2.4× higher throughput compared to the most performant state-of-the-art packet processing frameworks. This section further evaluates the effectiveness of FAJITA in improving performance by looking at the average and tail latency of stateful NFs. To do so, we measure the end-to-end latency of packets processed by a stateful load balancer followed by a FSC deployed on the DUT with eight



(a) Throughput. (b) Average end-to-end latency. (c) Tail (P99) latency.

Fig. 8. FAJITA significantly improves the DUT's ability to process packets at various offered loads when running a chain of stateful NFs consisting of a load balancer and a FSC on eight cores.

allocated CPU cores while changing the offered load. We change the offered load to have a better understanding of FAJITA's ability to process packets under different loads, especially when the system is *not* overloaded. Figure 8 shows the achieved throughput (y-axis, in Mpps) along with average and 99^{th} percentile end-to-end latency (y-axis, in µs) with respect to an increasing offered load using FastClick, VPP, and FAJITA when running the chain.

The result demonstrates that FAJITA significantly improves the ability of the system to perform stateful packet processing. More specifically, Figure 8a shows the ability of FAJITA to scale linearly until the saturation point that happens at ~2× higher offered load compared to FastClick and VPP (*i.e.,* ~102 Mpps vs. ~53 Mpps). Additionally, comparing the average and tail latency values in Figures 8b and 8c demonstrates the benefits of FAJITA even at lower rates when none of the frameworks have reached their saturation points mostly due to the principles overlooked by FastClick and VPP discussed in §2. More specifically, VPP requires multiple hash lookups to retrieve state information per NF, while FastClick fails to process packets in a batch throughout the pipeline. Finally, as shown in Figure 8c, FastClick suffers from significantly high tail latency even at very low rates. This issue is mainly due to the added latency from the Batch Rebuilder element at the end of the chain, whereas both VPP and FAJITA perform efficiently using adaptive batch sizes.

4.1.1 Auxiliary HT minimizes the shared memory overheads. To demonstrate the benefits of FAJITA in scenarios where implementing the shared-nothing architecture is *not* possible, we deploy a chain of NFs consisting of a source IP counter, followed by a policer, and a load balancer. The load balancer uses 5-tuple flow identifiers and does not require sharing states among cores; however, the other two NFs should share their states, as RSS dispatches packets according to 5-tuple flow identifiers. Figure 9 shows the maximum throughput achieved by VPP, FastClick, and FAJITA (with & without auxiliary HTs), while changing the number of cores to quantify the overheads of state sharing. These results show two main takeaways. First, even *without* the auxiliary HT, FAJITA still achieves higher throughput than the other frameworks despite sharing states among cores. Second, introducing auxiliary HTs enables FAJITA to overcome the synchronization overheads of sharing data structure thanks to minimized shared lookups, resulting in up to 50% higher throughput when using 16 CPU cores.

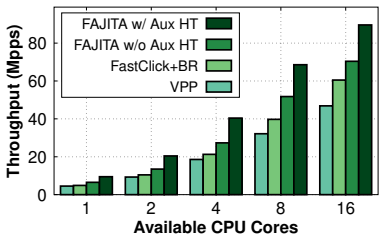
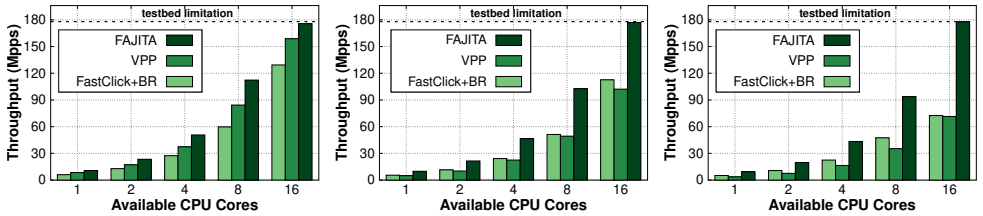


Fig. 9. Auxiliary HTs enables FAJITA to minimize synchronization overheads for an NF chain composed of a source IP counter, a policer, and a load balancer.



(a) FSC. (b) FSC + LB. (c) FSC + LB + Rate Limiter.

Fig. 10. The maximum throughput each framework can tolerate at high rates when receiving the scaled CAIDA traffic and deploying NF chains with different numbers of NFs.

4.2 Does FAJITA Scale?

To examine the scalability of FAJITA, we conduct experiments with various numbers of CPU cores allocated to the system when running chains with different numbers of stateful NFs. As shown in Figure 10, the throughput achieved by FAJITA scales almost linearly in all experiments regardless of the number of NFs in the chain, going up to ~178 Mpps with 16 available CPU cores where we hit the testbed bottleneck. Note that VPP's performance is significantly higher when only one stateful NF is running on the server. Increasing the number of stateful NFs causes performance to drop in VPP because of multiple hash lookups and insertions while FAJITA tolerates the added processing overhead due to the optimized processing pipeline and aggregated states. As discussed earlier, packets were truncated to 64-B in experiments with high rates to report the maximum possible throughput in terms of pps.

FAJITA retains its benefits regardless of the packet size. We conduct an experiment with various packet sizes using synthetic traces to ensure that larger packets do not affect FAJITA's performance benefit. Figure 11 shows the achieved throughput (y-axis, in Gbps) with respect to different packet lengths in bytes using FastClick, VPP, and FAJITA when using four cores and running a chain of three NFs (*i.e.,* similar to the Figure 10c). We report the throughput in Gbps to see the link saturation point. Increasing the packet size makes FAJITA's benefits more visible with the same ratio, reaching the link's full capacity at 512-B. As expected, FAJITA can process packets with various sizes efficiently and with the same improvement ratio.

4.3 Is RSS Sufficient for FAJITA?

Section 2.3 discussed the overheads of using advanced inter-core load balancing mechanisms, which typically involve actively keeping track of per-core load and sharing data structures. In this section, we examine the impact of load imbalance introduced by RSS on FAJITA when receiving traffic with highly skewed flow size distribution. To achieve this, we generate synthetic traffic traces with various numbers of active flows and measure the maximum throughput that packet processing frameworks achieve when having less than 1% packet drop. The generated traffic traces contain only elephant flows to introduce a high load *imbalance* among CPU cores on the DUT. Additionally, the DUT monitors the number of packets passing through each core every 100 ms (*i.e.,* the minimum suggested monitoring period in RSS++ [21] and Dyssect [22])

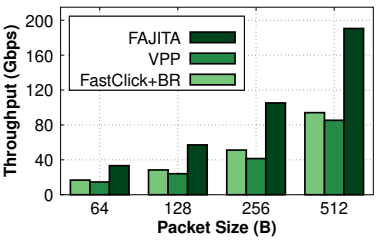


Fig. 11. FAJITA's benefit is consistent when receiving various packet sizes. Note that the y-axis is in Gbps.

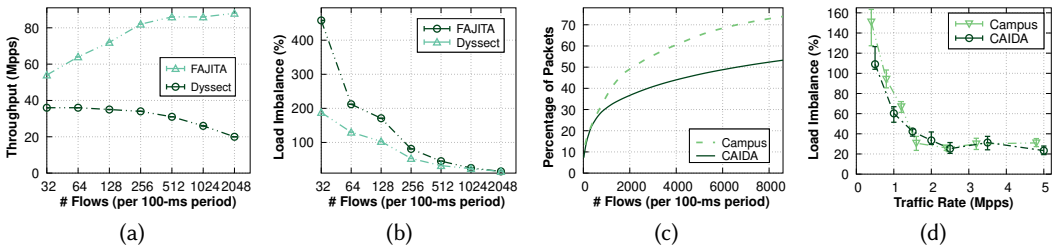
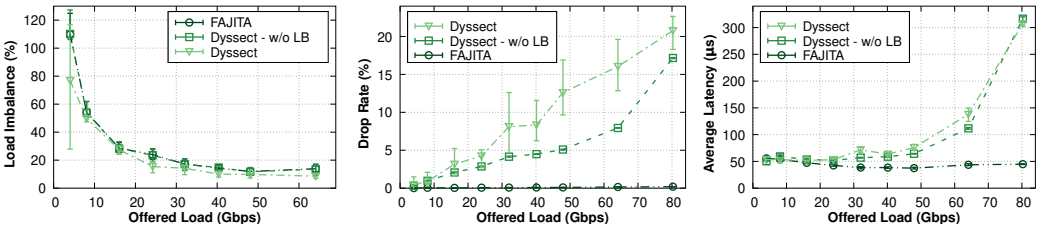


Fig. 12. Impact of inter-core load imbalance caused by RSS. Our synthetic analysis (figures a and b with logarithmic x-axis) shows the throughput and inter-core load imbalance with a varying number of active flows in a 100-ms period. We show a similar trend in realistic traffic traces with skewed flow sizes in (c) and (d). Note that (c) shows the CDF of packets per flow when replaying each trace at 5 Mpps rate.

and computes the traffic imbalance factor, defined as the percentage difference between the highest and lowest number of packets received by each core. We run a stateful NF (*i.e.,* FSC) on FAJITA and Dyssect as the state-of-the-art framework with an advanced load balancing mechanism. Moreover, we ensure that during the experiment Dyssect reaches the desired load balance by invoking the migration process *only once* as frequent migration of flows introduces a noticeable overhead on Dyssect.

Figures 12a and 12b demonstrate how varying number of elephant flows seen in a 100 ms period affects the load imbalance and consequently the throughput of FAJITA and Dyssect when 8 CPU cores are allocated to the DUT. As shown in these figures, when receiving only tens of flows in a processing window, the total throughput of FAJITA drops by ~40% due to the high load imbalance among cores but recovers rapidly when the number of flows is beyond 300. Moreover, the figures demonstrate that FAJITA achieves 50% higher throughput than Dyssect even when receiving only 32 active flows with a high load imbalance among CPU cores. This is primarily due to the overheads discussed in §2. Additionally, we examined CAIDA and our campus trace as two realistic traffic traces with highly skewed flow size distribution to measure the expected number of received flows and load imbalance in 100 ms periods. Figure 12c demonstrates the CDF of packets seen in the monitoring period with respect to the number of flows when replaying the trace files at 5 Mpps. Note that the number of flows increases linearly when increasing the offered load as more packets appear during the monitoring period. We also measure the introduced load imbalance when replaying our traffic traces at various rates when running FAJITA on 8 CPU cores (see Figure 12d). Our results show that the introduced load imbalance drops to less than 30% when replaying traces at 2 Mpps rate, confirming the rationale behind FAJITA's choice to rely on RSS at high packet rates with possibly millions of active flows [8], instead of incurring the overheads in the processing pipeline to implement a better load balancing mechanism.

Comparison against Dyssect. To better understand the benefits/overheads of inter-core load balancing at high packet rates, we compare FAJITA with Dyssect, which uses a dynamic inter-core load balancing mechanism. More specifically, we measure load imbalance factor, throughput, and latency of FAJITA and Dyssect when changing the offered load. Additionally, to distinguish the overheads of load balancing vs. potential pipeline inefficiencies, we also measure the aforementioned metrics when Dyssect does not perform load balancing. The results of our experiments have the following takeaways. First, Figure 13a shows that employing load balancing mechanisms is more beneficial at lower rates, which directly translates to fewer active flows (compare the load imbalance percentage of FAJITA and Dyssect at ~5 Gbps). Second, comparing to Dyssect with and without LB demonstrates the overheads of load balancing (see post-10-Gbps regions in Figure 13b), causing



(a) Inter-core load imbalance. (b) Drop rate. (c) Average end-to-end latency.

Fig. 13. FAJITA vs. Dyssect when running a load balancer on eight cores and receiving CAIDA traffic. FAJITA can process packets efficiently at line rate by simply relying on RSS while Dyssect experiences packet drops despite improving the inter-core load imbalance.

significant packet drops. Third, FAJITA's throughput can efficiently scale without introducing any packet drops or increased latency despite not having any load balancing mechanisms (see Figure 13c), further highlighting the importance of having an optimized processing pipeline. Note that FAJITA's performance persists up to line rate; not shown for better visibility.

4.4 What is the Impact of Each of FAJITA's Optimizations?

Section 4.1.1 showed the benefits of FAJITA's third optimization (*i.e.,* mitigating synchronization overhead). This section aims to measure the individual contribution of FAJITA's other optimizations (*i.e.,* optimized pipeline and accelerated state retrieval) to performance gains. Figure 14 shows the throughput of a FSC while processing 64-B packets in four different scenarios: (*i*) the first column (*FastClick*) emphasizes the overheads of transmission when sending small per-flow batches, (*ii*) the second column (*FastClick+BR*) represents the selected baseline for FastClick where transmission overhead is eliminated by re-batching packets at the end of the processing pipeline, (*iii*) the third column (*FAJITA w/o Prefetch*) shows the impact of FAJITA's optimized processing pipeline that avoids creating per-flow mini-batches while keeping pointers to the aggregated state block per flow, and (*iv*) the last column (*FAJITA w/ Prefetch*) demonstrates the entire benefits of FAJITA, which combines accelerated state retrieval with an optimized processing pipeline. This result shows that solely relying on an optimized processing pipeline can help FAJITA achieve 5.5× and ~15% higher throughput compared to FastClick without and with BR, respectively. Note that unlike BR, FAJITA does not introduce any latency overheads (see §4.1). Moreover, utilizing accelerated state retrieval further increases FAJITA's improvement by ~60%, showing the importance of employing both FAJITA's optimizations to utilize the underlying hardware more efficiently.

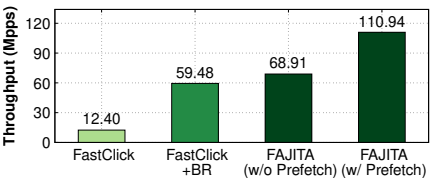
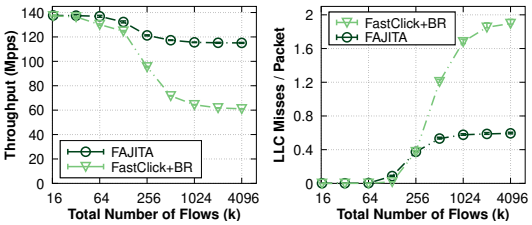


Fig. 14. FAJITA is *most* effective when utilizing an optimized processing pipeline with accelerated state retrieval (*i.e.,* FAJITA w/ Prefetch); it runs a FSC with eight cores.

4.5 Does FAJITA Change the Impact of Statefulness?

Section 2.1 showed the detrimental effect of statefulness on performance when increasing the number of flows due to an increase in the system's memory footprint and higher cache miss rate. This section repeats the same experiment to demonstrate the impact of FAJITA on memory overheads of NFs when dealing with various numbers of flows.

Figure 15 shows the throughput and average per-packet LLC misses for both FAJITA and FastClick when processing different numbers of flows containing 64-B packets. This result further highlights the benefits of employing FAJITA's processing path optimizations, which significantly reduces the overheads of statefulness, *i.e.,* the throughput is affected by less than 20%, even with 4-M flows, compared to a 60% throughput drop in the case of FastClick, which shows the impact of 3× fewer per-packet LLC misses.



(a) Throughput. (b) Avg. LLC misses.

Fig. 15. FAJITA improves the tolerance of the system for statefulness; it achieves higher throughput with a larger number of active flows, correlated with fewer LLC misses.

4.6 How Does Different Workloads Affect the Level of Performance Improvement?

We investigate the adaptability of FAJITA's enhancements across diverse workloads, as changing the traffic distribution can affect the FAJITA's benefits. Specifically, different traces may have a different spatial locality [17], thereby influencing cache locality. This directly affects the benefits coming from the batch-breaking avoidance done by FAJITA. Second, different traces have a different flow size distribution as shown in §4.3, which can directly influence hash tables' access pattern. Figure 16 compares the throughput of FastClick, VPP, and FAJITA when using various workloads for a stateful NF chain composed of a LB, a FSC, and a flow rate limiter. The results suggest that different workloads affect performance. More specifically, systems achieve higher throughput with our campus trace in comparison with CAIDA and synthetic traces (*i.e.,* with minimal traffic locality). Our analysis reveals that the higher throughput is correlated with having a higher spatial locality in the campus trace, resulting in fewer cache misses. However, FAJITA is still capable of improving throughput by ~22% and ~77% compared to FastClick and VPP, respectively. This shows that the benefits of FAJITA are still present, even for traffic workloads with a higher spatial locality.

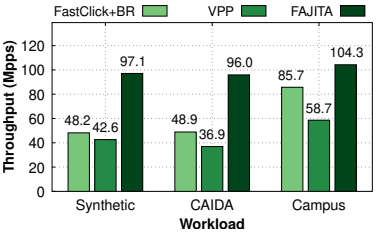


Fig. 16. FAJITA improves the throughput of various traffic workloads when processing a stateful NF service chain. Packets are truncated to 64-B.

4.7 Is FAJITA Beneficial for Advanced Data Structures?

As mentioned in Section 3, FAJITA's second principle relies on software prefetching with proactively loading the state for each NF right before processing a packet. Additionally, FAJITA provides a customizable function for developers to tailor the prefetching process to suit the specific data structures used by an NF. To show the benefits of this optimization, we deploy a chain of synthetic stateful NFs, where each NF performs *only one* indirect access to a memory location. This emulates scenarios where multiple stateful NFs with advanced state data-structure are deployed on the server.

Figure 17 demonstrates the throughput of FAJITA when a variable number of synthetic NFs run on the server with and

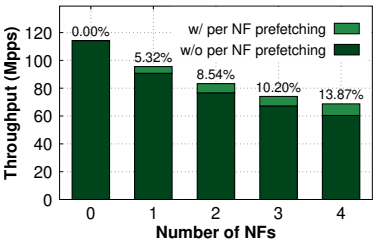


Fig. 17. Benefits of prefetching non-contiguous states increase with a higher number of indirect accesses - in this case, induced by chains of NFs.

without utilizing the provided function for prefetching state data with indirect accesses. The server utilizes 8 CPU cores and receives 64-B packets. As shown in the figure, increasing the number of NFs causes a noticeable performance reduction due to more processing and memory accesses. However, utilizing the prefetching function leads to more speedup (see the numbers on top of each bar) as the system can prefetch required data in advance and reduce the data retrieval overhead.

5 FREQUENTLY ASKED QUESTIONS

Does the auxiliary HT's size affect FAJITA's performance? The auxiliary HT uses the 5-tuple as the key in the hash table, and increasing the number of flows can potentially lead to higher memory allocation and increase the memory footprint of the packet processing framework.

More specifically, each auxiliary HT needs to store 8-B pointers per NF in the chain and ~15 B of metadata for each entry. This requires allocating a few hundred MB of memory on the commodity server, which increases the memory footprint of FAJITA compared to running the chain without an auxiliary HT. We argue that, despite the elevated memory footprint of the packet processing framework, auxiliary HT leads to less cache footprint and fewer LLC misses per packet during the packet processing, as auxiliary HT enables the system to bypass multiple *expensive* hash table accesses. Note that allocating more memory on a commodity server does not introduce an issue. Figure 7 already shows the benefit of using auxiliary HTs in terms of throughput. Additionally, Figure 18 signifies the impact of using auxiliary HT on the average number of LLC misses per packet in the same setup as §4.1.1

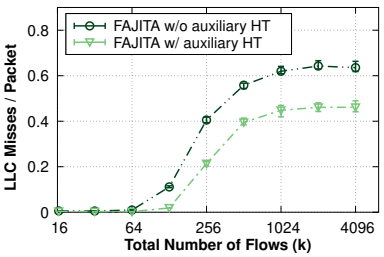


Fig. 18. Using per-core auxiliary HTs in FAJITA leads to fewer LLC misses when running a chain of 3 NFs with various flow granularities. Note that the x-axis is logarithmic.

Is it possible to implement the auxiliary HT on external devices? FAJITA automatically deploys the auxiliary HT as a component of the processing pipeline on the CPU since this approach (*i*) makes the framework independent of existing hardware and makes the deployment of NF chains easier, and (*ii*) demonstrates the benefit of the auxiliary HT *without extra hardware resources*. Introducing specialized accelerators (*e.g.,* SmartNICs and FPGAs) to keep meta-information on a per-flow basis will further reduce the memory footprints of the system with the cost of adding complexity to the network configuration as well as increasing operational costs.

Does FAJITA make a distinction between processing batch size and I/O batch size? In the current implementation of FAJITA, the processing batch size is always equal to the I/O batch size (*e.g.,* 64 packets), *i.e.,* all received packets from the NIC are processed as a single batch. However, FAJITA's design makes it possible to have a large I/O batch size and then split this batch into multiple smaller processing batches. By doing so, an application can fine-tune the size of I/O and processing batches to minimize I/O overhead and maximize instruction & data locality for packet processing. We briefly investigated this and noticed that tuning I/O and processing batches could result in up to 5 % improvements on top of FAJITA.

Do other optimizations affect FAJITA improvements? This paper mainly focused on addressing the specific challenges of stateful packet processing in existing frameworks. However, there are other bottlenecks (*e.g.,* code inefficiency, unoptimized metadata management, and software IP lookups) that could affect the performance of both stateless and stateful packet processing. Several works perform other optimizations to address these challenges (*e.g.,* PacketMill [14]); these efforts are complementary to FAJITA's optimizations and can potentially result in higher performance improvements; see §6. Moreover, integrating an overhead-free load balancing mechanism in FAJITA

could potentially further improve performance. For instance, Maestro [37] and FlowMage [23] can be used to configure RSS to ensure shared-nothing architecture, which is orthogonal to FAJITA. Systems such as Metron [12] offer dynamic scaling with low-overhead and could be integrated into FAJITA to scale the number of CPU cores depending on the received load.

6 RELATED WORK

To be best of our knowledge, FAJITA is the first to propose an optimized processing pipeline that operates at >100 Mpps and supports stateful NF service chains with different flow definitions. This section summarizes some related works that were not previously mentioned.

Optimized packet processing. Efforts to optimize packet processing can be divided into two main categories. The first category exploits programmable network devices to accelerate packet processing. More specifically, Tiara [39], Cheetah [9], Faild [40], Silkroad [27], and Beamer [41] focus on improving *inter-server* load balancing. TEA [42], PayloadPark [6], NFSlicer [43], nicmem [7], and Ribosome [8] utilize network devices to temporarily store some parts of the packet and only deliver the relevant parts of the packet required for processing (*e.g.,* the packet header) to the servers to more efficiently utilize the available link bandwidth. Switcharoo [20] enables ASIC switches to implement high-performance key-value stores entirely in the data plane, which could be beneficial for stateful packet processing. Sirius [44], FlowBlaze [45], and Pigasus [46] use programmable devices (*e.g.,* ARM cores and FPGAs) to perform stateful packet processing. The second category utilizes software optimizations to improve packet processing. In particular, PacketMill [14] and Morpheus [18] perform low-level optimizations (*e.g.,* metadata management, run-time code instrumentation, and compiler optimizations) to address software inefficiencies. CacheDirector [15] and DDIOTune [16] tune Direct Cache Access (DCA) to reduce the tail latency of high-speed NFs. Clara [47], Gallium [48], FlightPlan [49], and ExoPlane [50] propose systems to automatically analyze and offload parts of packet processing into programmable hardware. Furthermore, LemonNFV [51] consolidates heterogeneous NFs without code modifications. Our work is orthogonal to these efforts.

Hash tables and state management. Hash tables are extensively used in networking applications for storing per-connection states. To achieve high performance, such systems commonly rely on open addressing hash schemes [29, 52–54], which eliminate pointer chasing problem. Among these, Cuckoo hashing [29] is popular in high-speed packet processing by providing worst-case constant lookup time, which is the predominant operation in NFs. Hence, Cuckoo hash tables are the foundation for a plethora of high-performance applications [55–57]. Girondi *et al.,* [31] studied the impact of using different hash tables on the performance of connection tracking for high-speed NFs, highlighting the negative impact of shared hash tables on the performance for a stateful load balancer. Moreover, CuckooSwitch [58] presents a high-performance hash table for software-based Ethernet switches. CuckooSwitch could achieve 92 Mpps on a 16-core server with 8×10-Gbps NICs.

Prefetching. Software prefetching has been available for many years in advanced CPUs, and many works have exploited this feature to improve the performance of their applications by hiding memory access latency [59]. Here, we briefly discuss some of the most relevant works done on prefetching. As discussed in §3.1, it is important to pay careful attention to the applications' workflow when using software prefetching to avoid polluting cache memories. More specifically, applications should know when *to use and not to use* prefetching. Lee *et al.,* [60] thoroughly investigated this topic and tried to address these questions. Software prefetching has been used to (*i*) improve access to remote memory [61], (*ii*) accelerate flash-based storage systems [62], and (*iii*) boost indirect memory accesses [63, 64]. Wang *et al.,* [65] studied different workloads in the cloud and explained the correlation between memory access patterns & the benefits of prefetching. Furthermore, Seer [66] proposes a system that provides hints about the upcoming packets which

enables networking applications to implement high-performance caching by prefetching to-be-used state data in advance and evicting unused data, and Nostradamus [28] examines the potential benefits of prefetching on the performance of stateful NFs when receiving such hints.

7 CONCLUSIONS

The unavailability of data in CPU caches can significantly affect performance when processing packets at high link data rates. In particular, the performance of stateful NFs (with a higher memory footprint) can be greatly affected by high wastage of CPU cycles when their data is unavailable in cache memories. FAJITA proposes an optimized processing pipeline with accelerated hash table lookups to maximize the benefits of cache memories for stateful packet processing. More specifically, FAJITA (i) minimizes memory access overheads by exploiting batching & software prefetching machinery available in modern CPUs, and (ii) alleviates the overheads of accessing shared data structures by introducing auxiliary hash tables. Our work once again emphasizes the importance of cache optimizations for high-speed packet processing.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions on this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889). This work has been partially supported by Vinnova (the Sweden's Innovation Agency), the Swedish Research Council (agreement No. 2021-04212), and KTH Digital Futures.

REFERENCES

[1] Intel Barefoot Networks. Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs, 2020. https://www.barefootnetworks.com/products/brief-tofino-2/.

[2] NVIDIA Mellanox. ConnectX-7 400G Adapters, 2024. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[3] Zhiping Yao, Jasmeet Bagga, Hany Morsy. Introducing Backpack: Our second-generation modular open switch, November 2016. https://engineering.fb.com/data-center-engineering/introducing-backpack-our-second-generation-modular-open-switch/.

[4] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. Dark Packets and the End of Network Scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3230718.3230727.

[5] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association. https://www.usenix.org/system/files/conference/hotcloud18/hotcloud18-paper-thomas.pdf.

[6] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. *Parking Packet Payload with P4*, page 274–281. Association for Computing Machinery, New York, NY, USA, 2020. https://doi.org/10.1145/3386367.3431295.

[7] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. *The Benefits of General-Purpose on-NIC Memory*, page 1130–1147. Association for Computing Machinery, New York, NY, USA, 2022. https://doi.org/10.1145/3503222.3507711.

[8] 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. https://www.usenix.org/conference/nsdi23/presentation/scazzariello.

[9] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/barbette.

[10] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020. https://www.usenix.org/conference/osdi20/presentation/brunella.

[11] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. *Offloading Load Balancers onto SmartNICs*, page 56–62. Association for Computing Machinery, New York, NY, USA, 2021. https://doi.org/10.1145/3476886.3477505.

[12] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 171–186, Renton, WA, April 2018. USENIX Association. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf.

[13] Bangwen Deng, Wenfei Wu, and Linhai Song. Redundant Logic Elimination in Network Functions. In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 34–40, New York, NY, USA, March 2020. Association for Computing Machinery. https://doi.org/10.1145/3373360.3380832.

[14] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-core 100-Gbps Networking. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, New York, NY, USA, March 2021. Association for Computing Machinery. https://doi.org/10.1145/3445814.3446724.

[15] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, March 2019. ACM. http://doi.acm.org/10.1145/3302424.3303977.

[16] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020. https://www.usenix.org/conference/atc20/presentation/farshin.

[17] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 807–827, Renton, WA, April 2022. USENIX Association. https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni.

[18] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Domain Specific Run Time Optimization for Software Data Planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 1148–1164, New York, NY, USA, February 2022. Association for Computing Machinery. https://doi.org/10.1145/3503222.3507769.

[19] 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, August 2013. https://doi.org/10.1145/2486001.2486011.

[20] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. Millions of Low-Latency State Insertions on ASIC Switches. *Proc. ACM Netw.*, 1(CoNEXT3), November 2023. https://doi.org/10.1145/3629144.

[21] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, pages 318–333, New York, NY, USA, December 2019. ACM. http://doi.acm.org/10.1145/3359989.3365412.

[22] Fabrício B Carvalho, Ronaldo A Ferreira, Ítalo Cunha, Marcos AM Vieira, and Murali K Ramanathan. Dyssect: Dynamic scaling of stateful network functions. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1529–1538. IEEE, May 2022. https://doi.org/10.1109/INFOCOM48880.2022.9796848.

[23] Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Marco Chiesa, and Dejan Kostić. Deploying Stateful Network Functions Efficiently using Large Language Models. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 28–38, New York, NY, USA, April 2024. Association for Computing Machinery. https://doi.org/10.1145/3642970.3655836.

[24] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, May 2015. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2772722.2772727.

[25] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed data plane and network functions virtualization by vectorizing packet processing. *Computer Networks*, 149:187–199, February 2019. https://www.sciencedirect.com/science/article/pii/S1389128618312957.

[26] Intel. Tofino®2, 2023. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[27] 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, August 2017. Association for Computing Machinery. https://doi.org/10.1145/3098822.3098824.

[28] Hamid Ghasemirahni, Alireza Farshin, Dejan Kostic, and Marco Chiesa. Just-in-Time Packet State Prefetching, July 2024. https://arxiv.org/abs/2407.04344.

[29] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2):122–144, May 2004. https://doi.org/10.1016/j.jalgor.2003.12.002.

[30] Nicolas Le Scouarnec. Cuckoo++ hash tables: High-performance hash tables for networking applications. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 41–54, July 2018. https://doi.org/10.1145/3230718.3232629.

[31] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed Connection Tracking in Modern Servers. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, June 2021.

[32] Yipeng Wang, Sameh Gobriel, Ren Wang, Tsung-Yuan Charlie Tai, and Cristian Dumitrescu. Hash table design and optimization for software virtual switches. In *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*, pages 22–28. Association for Computing Machinery, August 2018. https://doi.org/10.1145/3229538.3229542.

[33] FastClick. MinBatch Element. https://github.com/tbarbette/fastclick/blob/main/elements/standard/minbatch.hh, 2023.

[34] Tamás Lévai, Felición Németh, Barath Raghavan, and Gabor Retvari. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/levai.

[35] Hugo Sadok, Miguel Elias M Campista, and Luís Henrique MK Costa. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 127–133, New York, NY, USA, November 2018. ACM. http://doi.acm.org/10.1145/3286062.3286081.

[36] Tom Barbette. Public repository with all the experiments conducted in the course of the RSS++ paper, 2019.

[37] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. Automatic Parallelization of Software Network Functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1531–1550, Santa Clara, CA, April 2024. USENIX Association. https://www.usenix.org/conference/nsdi24/presentation/pereira.

[38] Tom Barbette, Cyril Soldani, and Laurent Mathy. Combined Stateful Classification and Session Splicing for High-Speed NFV Service Chaining. *IEEE/ACM Trans. Netw.*, 29(6):2560–2573, December 2021. https://doi.org/10.1109/TNET.2021.3099240.

[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. https://www.usenix.org/conference/nsdi22/presentation/zeng.

[40] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the Edge: Transport Affinity without Network State. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 111–124, Renton, WA, April 2018. USENIX Association. http://dl.acm.org/citation.cfm?id=3307441.3307452.

[41] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, Renton, WA, April 2018. USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/olteanu.

[42] 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, July 2020. Association for Computing Machinery. https://doi.org/10.1145/3387514.3405855.

[43] Anirudh Sarma, Hamed Seyedroudbari, Harshit Gupta, Umakishore Ramachandran, and Alexandros Daglis. NFSlicer: Data Movement Optimization for Shallow Network Functions, 2022. https://doi.org/10.48550/arXiv.2203.02585.

[44] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, Boston, MA, April 2023. USENIX Association. https://www.usenix.org/conference/nsdi23/presentation/bansal.

[45] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/pontarelli.

[46] 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. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[47] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 772–787, New York, NY, USA, October 2021. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483583.

[48] 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, July 2020. Association for Computing Machinery. https://doi.org/10.1145/3387514.3405869.

[49] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592. USENIX Association, April 2021. https://www.usenix.org/conference/nsdi21/presentation/sultana.

[50] 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. https://www.usenix.org/conference/nsdi23/presentation/kim-daehyeok.

[51] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1451–1468, Boston, MA, April 2023. USENIX Association. https://www.usenix.org/conference/nsdi23/presentation/li-hao.

[52] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing. *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pages 281–288, October 1985. https://doi.org/10.1109/SFCS.1985.48.

[53] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC '08, page 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. https://doi.org/10.1007/978-3-540-87779-0_24.

[54] Rina Panigrahy. Efficient Hashing with Lookups in Two Memory Accesses. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, page 830–839, USA, 2005. Society for Industrial and Applied Mathematics. https://doi.org/10.48550/arXiv.cs/0407023.

[55] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 281–294, Denver, CO, June 2016. USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/breslow.

[56] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, April 2013. USENIX Association. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[57] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, April 2014. Association for Computing Machinery. https://doi.org/10.1145/2592798.2592820.

[58] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 97–108, New York, NY, USA, December 2013. Association for Computing Machinery. https://doi.org/10.1145/2535372.2535379.

[59] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.*, 49(2), August 2016. https://doi.org/10.1145/2907071.

[60] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.*, 9(1), March 2012. https://doi.org/10.1145/2133382.2133384.

[61] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020. https://www.usenix.org/conference/atc20/presentation/al-maruf.

[62] Han Wang, Longfei Luo, Liang Shi, Changlong Li, Chun Jason Xue, Qingfeng Zhuge, and Edwin H.-M. Sha. SFP: Smart File-Aware Prefetching for Flash Based Storage Systems. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, GLSVLSI '21, page 45–50, New York, NY, USA, June 2021. Association for Computing Machinery.

[63] Sam Ainsworth and Timothy M. Jones. Software Prefetching for Indirect Memory Accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 305–317. IEEE Press, February 2017.
[64] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. Informed Prefetching for Indirect Memory Accesses. *ACM Trans. Archit. Code Optim.*, 17(1), March 2020. https://doi.org/10.1145/3374216.
[65] Jiajun Wang, Reena Panda, and Lizy Kurian John. Prefetching for cloud workloads: An analysis based on address patterns. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–172, July 2017. https://doi.org/10.1109/ISPASS.2017.7975288.
[66] Jason Lei and Vishal Shrivastav. Seer: Enabling Future-Aware Online Caching in Networked Systems. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 635–649, Santa Clara, CA, April 2024. USENIX Association. https://www.usenix.org/conference/nsdi24/presentation/lei.

A SUPPLEMENTARY MATERIAL

A.1 Details of Evaluated NFs

We evaluate the performance of FAJITA for the most common stateful NFs that are widely used in realistic networks, as done by other works [12, 14, 22, 24, 37]. This section provides more details about each NF and the per-flow state information used by them to process packets.

Load Balancer (LB). A stateful LB distributes incoming network traffic among multiple servers while maintaining the state of each flow (*i.e.,* it uses 5-tuple flow identifiers as the key). This ensures consistent dispatching of packets belonging to the same flow. We utilize a LB as the main NF in our experiments, as it is widely used by other works to measure different performance metrics. This NF selects the destination IP address for the first packet of each flow using a specified algorithm (*i.e.,* round-robin in our case) from an IP pool and maintains this decision as the flow's state. Subsequent packets of the same flow use this state, ensuring continued consistency in traffic distribution.

Access Control List (ACL). A stateful ACL regulates network traffic based on the state of connections. Unlike traditional ACLs that only consider individual packets, a stateful ACL keeps track of the state of active connections. This enables more nuanced control by allowing or denying traffic based on the context of the entire connection rather than isolated packets. To ensure no packet drop and consequently incorrect measurement deployed ACLs in our experiments simply accept all packets and store only an integer value as the state of a flow (*i.e.,* it uses 5-tuple flow identifiers as the key).

Flow Statistics Counter (FSC). A FSC is a stateful NF that monitors and records data for each individual flow, including information about the number of packets, bytes, or other relevant metrics associated with each flow (*i.e.,* it uses 5-tuple flow identifiers as the key).

Source IP Statistics Counter. This NF acts similarly to the FSC but keeps the data per source IP address instead of a 5-tuple flow identifier. We use this NF as a sample application that has a flow definition with a coarse granularity. Additionally, this NF could potentially cause false sharing, as multiple cores share and update a state for each source IP address.

Policer. This NF is designed to restrict the download rate for individual servers and keeps state per destination IP address. Similar to the Source IP Statistics Counter, this NF may impose an overhead on the system due to having shared states among cores.

Port Scan Detector (PSD). This is a security NF that identifies port scan activities. To do so, a PSDs typically store the list of destination ports accessed per source IP address (or source and destination IP addresses). Source IPs with a high number of accessed destination ports are considered potentially malicious.

Synthetic NF. To mimic the behavior of stateful NFs advanced state data structures, we developed a synthetic NF with a configurable number of memory accesses per packet. More specifically, the NF stores an array of pointers as the state with a provided length (*i.e.,* number of memory accesses per packet). To process a packet the NF increments the value of each state entry with no further processing overhead.