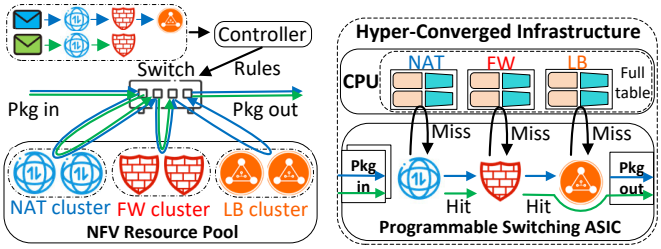
HyperSFC: State-Intensive Service Function Chaining on Hyper-Converged Edge Infrastructure

Yan Zou*, Tian Pan*‡, Lu Lu†, Zhiqiang Li†, Kehan Yao†, Yan Mu†, Ying Wan§, Tao Huang*‡, Yunjie Liu*‡
*Beijing University of Posts and Telecommunications, Beijing, China
†China Mobile Research Institute, Beijing, China
§China Mobile (Suzhou) Software Technology Co., Ltd, Suzhou, China
‡Purple Mountain Laboratories, Nanjing, China

***Abstract*—Service function chaining (SFC) enhances network efficiency and agility by enabling the orchestration of network functions (NFs) in a customized manner. In public cloud environments, SFC is implemented via sequential packet routing through a series of NFs, each hosted on an x86 server pool. However, such server pooling-based SFC provisioning is not easily deployable in the edge cloud due to constraints related to hardware budget and deployment footprints. In the edge cloud, hyper-converged infrastructure (*e.g.*, switch servers) offers a promising solution for cost and footprint reduction. Typically, a hyper-converged switch server comprises a programmable switch and a CPU, allowing NFs to be hosted on the CPU with packet routes controlled by the switch. To enhance performance, forwarding tables of NFs can be further offloaded to the switch. However, due to the limited on-chip memories of the switch, NFs with intensive states cannot be fully offloaded. To address this, we propose *busy submit* via packet circulation between the switch and CPU. While *busy submit* ensures packets experiencing table search misses in the switch are routed to the full tables in the CPU, it may lead to repeated circulations in the context of SFC processing, increasing both packet latency and CPU load. To mitigate this, we introduce *lazy submit*, reducing packet circulation overhead by postponing packet submission to the CPU until the end of the pipeline. Additionally, we use 2D Bloom filters to cache NF dependencies, reducing unnecessary circulations. The evaluation of an NF chain with 10 NFs shows that *busy submit* enables state-intensive SFC, while *lazy submit* reduces overall latency by 60%.**

I. Introduction

Service function chaining (SFC) [1] provides a flexible way for network operators to define and manage the flow of traffic through a series of network functions (NFs) such as firewalls (FWs), load balancers (LBs), network address translations (NATs), allowing for efficient and customizable network service delivery. In public clouds, which have rich computing resources (*e.g.*, servers) shared by a massive number of tenants, the SFC is usually implemented via sequential routing of tenants' packets through a series of NFs, each of which is hosted in a scalable x86 server pool (as shown in Fig. 1a). Specifically, different tenants can specify different SFCs with different packet routes through NF resource pools. The packet routes are determined by the forwarding rules in SDN switches installed by the controller. Even if the packet routes are different, two SFCs may still share the same path segment at the same resource pool which increases utilization. When traffic demand grows, the NF resource pool can easily be scaled out to handle peak traffic load by adding more server nodes [2]. With abundant server resources in data centers, the server pooling-based SFC provisioning runs well in public clouds. However, its deployment for edge clouds is not easily feasible due to the strict constraints of hardware budget and deployment footprints at the edge. First, as edge clouds are placed near the end-users, they will have a larger site number compared to public clouds. Therefore, the cost efficiency of each edge cloud must be considered and the server pooling for SFC appears quite expensive. Second, edge clouds usually have small footprints but full functionalities (including compute, storage, network, power supply, cooling system) for ease of deployment in anywhere near the customers. For example, AWS's edge cloud infrastructure is delivered as an industry-standard 42U rack [3]. If the network functionalities (*e.g.*, SFC) occupy too much rack space, the server payload for hosting tenant VMs will have to be compacted, which will reduce the total VM capacity as well as the cloud vendor's revenue. Overall, at the edge, we need a "high-density" implementation of SFC at affordable costs.



(a) SFC in public cloud (b) SFC in edge cloud

Fig. 1: Two approaches to implementing SFC in public cloud and edge cloud scenarios with different resource availabilities.

Fortunately, with the rise of edge computing demands, in the server market of 2023, we find such a device with, (1) small deployment footprints (*e.g.*, 2U), (2) affordable costs (price of a switch plus a server), and (3) both functions of

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62372053, Beijing University of Posts and Telecommunications-China Mobile Research Institute Joint Innovation Center, the research project (NO. Y2024CXT002) of China Mobile (Suzhou) Software Technology Co., Ltd.. Co-corresponding authors: Tian Pan (pan@bupt.edu.cn), Zhiqiang Li (lizhiqiangyjy@chinamobile.com), and Ying Wan (wanying2@cmss.chinamobile.com).

ISBN 978-3-903176-63-8

computing and networking, which is very suitable for SFC deployment in edge cloud. The device is a *hyper-converged switch server* (sometimes also called server switch) [4] which contains a programmable switch (*i.e.*, P4 switch) [5, 6] and a powerful x86 multi-core CPU, typically used for implementing cloud gateways and load balancers [7–9]. As the switch server has the full functions of computing and networking, the server pooling-based SFC architecture in public cloud can be migrated into the 2U box with the NFs implemented in multi-cores of the CPU and the packet routing controlled by the programmable switch as shown in Fig. 1b. At the edge, to handle the peak traffic load from major cloud customers with limited computing resources within the 2U box, we propose *HyperSFC* dedicated to SFC deployment for edge cloud based on the hyper-converged switch server, and consider fully leveraging the high performance and programmability of the P4 switch to maximally offload NFs to the data plane to lessen the CPU's burden. Specifically, the SFC can be implemented along the switch pipeline with multiple NF forwarding tables installed into the sequential pipeline stages.

However, such a design still faces several challenges. First, the P4 switch actually has very limited on-chip SRAM and TCAM [10] to hold the full tables of state-intensive NFs (*e.g.*, LB) [11]. To this end, we have to cache the active NF table entries in the switch while its full table still needs to be stored on the CPU. Note that we opt for cache-based table partitioning rather than assigning different NFs to the switch or CPU based on their table sizes [12] to fully leverage traffic locality to maximize forwarding performance and reduce CPU overhead. Following this design, we propose *busy submit* with packet circulation between the switch and CPU triggered on each table search miss at the fast path. Specifically, the busy submit needs to address (1) when a table search miss occurs, how to route the packet to the correct full table instance on the CPU, and (2) how to make efficient table bypass along the switch pipeline when the table search miss occurs and when the packet re-enters the switch after completing the full table search on the CPU. We add customized metadata to the packet header and create novel pipeline bypass logic to satisfy the above needs. Second, although busy submit can always route packets with table search misses in the switch to hit the full tables in the CPU, packets may experience repeated circulations when multiple table search misses occur during SFC processing, which will increase both packet processing latency and CPU overload. To this end, we propose *lazy submit* to reduce the packet circulation times and overhead by postponing the packet submission to the CPU until the end of the switch pipeline, and processing the full table search requests in parallel on the CPU. Moreover, we leverage 2D Bloom filters as the table search dependency encoder and table branch selector to handle the performance issue of lazy submit with SFC branches. Specifically, 2D Bloom filters are used to cache NF dependencies in the fast path, thereby reducing unnecessary packet circulations to the CPU for querying the SFC branch results. False positive handling of 2D Bloom filters is also discussed. With the step-by-step design of busy submit and lazy submit, HyperSFC becomes efficient and robust, promising for SFC deployment in the real world.

Our major contributions are summarized as follows:

- We propose HyperSFC, a novel SFC architecture based on hyper-converged switch servers for high-performance, cost-effective and small-footprint edge cloud deployment.
- To address the limited memory issue of the P4 switch for holding state-intensive NFs, we employ cache-based NF table partitioning and propose busy submit. This enables efficient packet circulation between the switch and CPU, triggered by table search misses in the switch pipeline.
- To address the repeated circulations due to multiple table search misses along the pipeline, we propose lazy submit to reduce the packet circulation overhead by postponing the packet submission to the CPU until the end of the SFC. Besides, we use 2D Bloom filters for fast SFC table dependency resolution when conducting lazy submit.
- We implement HyperSFC with the P4 switch software and Redis database, and the code is available on our git repository [13]. Our evaluation of an NF chain with 10 NFs shows that busy submit realizes state-intensive SFC efficiently with the CPU utilization below 3%, while lazy submit reduces the overall latency of busy submit by 60%.

II. Background and Motivation

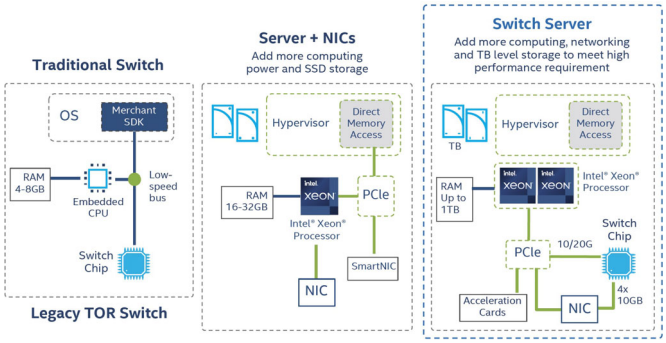


Fig. 2: The evolution of hyper-converged switch server [4].

Hyper-converged switch server. Fig. 2 depicts the architecture of Accton CSP-9550 [4], a hyper-converged switch server, and its evolutionary stages: from the original x86 server, to the smartNIC-accelerated x86 server, to the hyper-converged switch server. This evolution is fueled by the growing traffic loads in public and edge clouds, along with deployment footprint limitations. Actually, the use of programmable switches [6] for NF acceleration has been successfully demonstrated in various application scenarios [10, 14]. The P4 switch has 4 pipelines, and each pipeline has limited SRAM/TCAM on-chip memories (O(10MB)), distributed in 12 pipeline stages. For stateless NF implementation (*e.g.*, L3 forwarding), the forwarding tables of different NFs can be sequentially placed in the switch pipeline stages for high-throughput, low-latency packet processing. However, most stateful NFs have large memory footprints to maintain per-flow forwarding rules (*e.g.*, LB, NAT) and the P4 ASIC is

insufficient to accommodate the full tables of these NFs. The convergence of the CPU and P4 ASIC within a single box makes the switch server an excellent choice for implementing stateful NFs, and the switch pipeline is inherently suitable for accommodating the SFC. To take full advantage of traffic locality, we can use the P4 ASIC as the cache of the active entries, leaving the full tables maintained in the CPU as a backup. When table search misses occur in the switch, the packet needs to be routed to the CPU and when it completes the full table search, it will re-enter the fast path to get processed by the remaining NFs in the SFC. To implement the above interaction between the switch and CPU, some ports of the switch should be connected to the NICs attached to the CPU in the switch server. Packets must traverse the entire switch pipeline before being forwarded through the NICs and reaching the CPU. Upon completing the CPU processing, packets will be circulated back to the previous input port of the switch pipeline for remaining NF processing along the SFC.

Challenges of SFC provisioning on switch server. Following this hierarchical SFC design based on the switch server with packet circulation, there remain several challenges that need to be tackled. The first challenge involves correctly dispatching packets that encounter table search misses in the fast path to the corresponding full tables on the CPU, given the presence of numerous NFs and their associated tables on the CPU. The second challenge pertains to achieving a fast response after table search misses within the switch pipeline and implementing fast checkpoint recovery when packets circulate back to the pipeline after hitting the full table in the CPU. Specifically, it is expected that packets encountering table search misses can bypass the subsequent pipeline stages to reach the output port directly for fast CPU processing. It is also expected that when packets are circulated back to the pipeline, they can bypass the previously traversed pipeline stages to avoid redundant processing. The third challenge is about repeated circulation overhead reduction during one-pass SFC processing, since multiple packet circulations may occur if table search misses occur in multiple pipeline stages which will further incur longer packet processing latency and higher CPU utilization. The fourth challenge involves efficient table dependency resolution during packet circulation. In some SFC scenarios, the selection of the next table depends on the search result of the current table. Such inter-table dependencies may introduce complexity during packet circulation with or without the aforementioned optimizations. We provide our solutions to these challenges in the following sections.

III. Hierarchical SFC Processing with Packet Circulation between P4 ASIC and CPU

A. Busy Submit Model with Packet Circulations

Full table lookup based on fast-slow path coordination. To effectively store a significant number of match-action tables for state-intensive NFs, we retain the full tables in the CPU as a backup, and cache a subset of these full tables in the switch pipeline. In case of a table search miss in the switch pipeline, we ensure proper routing of the packet to the corresponding full table on the CPU by embedding metadata such as the *table ID* and *search key* into the packet header. The table ID records the exact table where the search miss occurs, and the search key carries the original search key for a full table lookup on the CPU. Upon arrival at the CPU, a key extraction logic extracts the key from the packet header and searches for it in the corresponding full table. In our design, we assign the task of action execution entirely to the switch to maximally lessen the burden on the CPU. Consequently, on the CPU, the full table search only returns a 1-bit result of either 1 (found) or 0 (not found), which is then added to the packet header. If the result is 1, the corresponding table entries will be dumped into the switch for cache replacement. When the packet is circulated back to the input port of the switch, if the result is 0, the default action will be executed (*e.g.*, packet drop); otherwise, the packet will search the previous table again. If the installed rule has taken effect, the packet will hit the rule and proceed to the next stage; otherwise, it will experience another table search miss and be forwarded to the CPU again. In our design, the packet does not need to be buffered in the queue until the table entry is updated from the CPU. Such a stateless design eliminates the need for deep buffering and complex buffer management logic in high-speed networks.

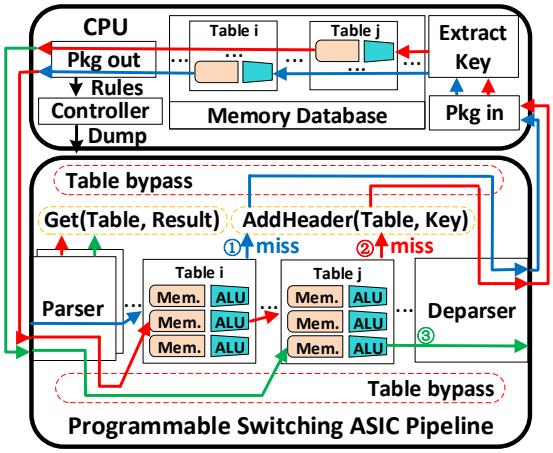


Fig. 3: Packet circulations between P4 ASIC and CPU for hierarchical SFC processing via the busy submit model.

A detailed example. Fig. 3 shows the packet flow of the busy submit model. After a packet (blue line) encounters the first search miss at table i in the pipeline, a header containing the table ID and search key is inserted into the original packet. It then proceeds directly to the end of the pipeline in the local match-action stage and is forwarded to the CPU. The CPU parses and recognizes that this is the packet designated for a full table lookup. Based on the parsed table ID and key, the CPU performs key matching on table i in the memory database. The lookup result is then encapsulated in the packet header and subsequently resubmitted to the switch. Simultaneously, the matched rules will be installed to the data plane by the controller running at the CPU. When the packet reaches the parser again (red line), the switch parses the lookup result and bypasses it to table i according to the table ID.

As the rules have been updated in the data plane, the packet can be correctly forwarded at table i. However, in Fig. 3, in the subsequent stages, it encounters another miss at table j. Similar to the previous match failure, the packet undergoes another search on the CPU and successfully proceeds through the pipeline after the second packet circulation (green line).

B. Table Bypass in the Data Plane

Table bypass before packet submit to CPU and after packet return. It is essential and beneficial to bypass irrelevant tables in the switch pipeline both before the packet is sent to the CPU and after the packet returns to the switch. On one hand, bypassing irrelevant match-action stages speeds up the processing of packets within the pipeline. On the other hand, certain P4 programs necessitate the sequential execution of table lookups. Preserving the matching order of the tables during hierarchical SFC processing ensures the correctness of the program's logic. By default, in the switch server, packets have to traverse the entire pipeline before being sent to the CPU, even if they encounter table search misses in the middle of the pipeline. However, continuing to search the tables in the subsequent stages after a table search miss occurs is neither necessary nor always correct. Therefore, it is recommended to terminate the current pipelined searches in advance when encountering a table miss and directly send the packet to the CPU for a full table search. The left part of Fig.4 illustrates the egress table bypass P4 logic we design to perform subsequent table bypass upon table search misses, before the packet is submitted to the CPU. Similarly, the right part of Fig.4 illustrates the ingress table bypass logic. When the packet returns from the CPU and is resubmitted to the pipeline, there is no need to repeat the search of the previously successfully searched table. Since we record the table ID of the most recent failed search in the packet, we can easily complete the table bypass after the packet returns to the pipeline.

A detailed example. Fig. 4 shows the process of table bypass in detail. The parser classifies the normal packet (Protocol type = IP) and the resubmitted packet (Protocol type = Resubmit) returned from the CPU according to the Protocol type identifier. For the normal IP packet, the packet header vector (PHV) undergoes matching in multiple sequential stages. Upon finding a match, the PHV executes the corresponding action. In scenarios where a search miss occurs (*e.g.*, in MAU1), the switch halts the execution of the predefined actions of the current stage. Instead, it will add the table ID and search key into the packet header, and update the identifier to Resubmit so that the CPU can identify the packet type and perform a full table lookup. After the identifier is replaced, subsequent stages can choose to skip their local stage processing based on the identifier until the end of the pipeline, and finally the packet is sent to the CPU. After full table searches in the CPU, the resubmitted packet will return to the switch and undergo sequential matching of MAUs, skipping those that have been processed based on the carried table ID, until it reaches the MAU where the table search miss occurs previously (*i.e.*, MAU1 in this example). If the result is a hit (*i.e.*, the rules have been installed), then execute the corresponding action. Otherwise, the default action of the P4 program (usually packet drop) is executed. The current bypass logic for the hierarchical SFC processing is loosely integrated into the original P4 programs. It can also be easily applied to other P4 programs that require the similar fast-slow path coordination.

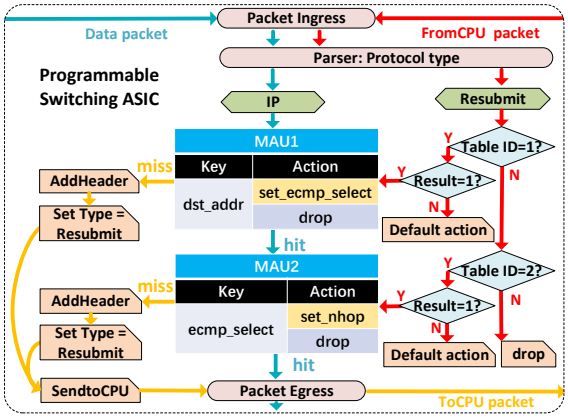


Fig. 4: Table bypass logic inside P4 ASIC before packet submit to CPU and after packet return.

C. Rule Caching Strategies

Cache update considering rule dependencies. For exact match tables, selecting the rules that need to be installed to the fast path is straightforward; simply choose the exact rules that are hit in the full table on the CPU. However, for longest prefix match tables, such as the routing table, the situation is different due to the prefix dependencies between rules. Directly selecting the rules hit on the CPU to install may lead to inconsistency in the lookup results. For example, if the prefix 10.0.*.* has been hit and updated to the fast path previously, while a longer prefix like 10.0.0.1 is still on the CPU. A packet arriving at the fast path with a DIP of 10.0.0.1 will match the 10.0.*.* directly at the fast path, leading to incorrect forwarding. Because the true longest prefix 10.0.0.1 is on the CPU, and a full table lookup would certainly match 10.0.0.1. Similar issues are also discussed in [15]. To ensure the correctness of longest prefix matching on the fast path, in such cases, the rule caching strategy to adopt should update the rules that are hit along with their dependent longer-prefix rules to the fast path.

Cache eviction based on rule frequencies. In general, we can use a frequency-based cache replacement strategy to evict inactive table entries. To reduce the counter memory usage for recording entry frequencies within a programmable switch, we can further utilize succinct data structures such as sketches. It is worth noting that maintaining consistency in table lookup results is also necessary during cache eviction. Therefore, for longest prefix match tables, when evicting a low-frequency entry, it's necessary to evict other prefix-dependent entries along with it. To ensure roughly the same quantity between the total number of evictions and updates, it's necessary to conduct precalculation based on the frequency of entries and the dependency between entries before each cache eviction.

D. Limitations of Busy Submit

Although busy submit can always route packets with table search misses in the switch to hit the full tables in the CPU, packets may undergo repeated circulations when multiple table search misses occur during SFC processing. This will lead to four issues. First, the packet processing latency will be increased by manyfold. Second, the CPU will be occupied repeatedly by the circulated packets. Third, the switch bandwidth and the southbound interface will be consumed repeatedly by the circulated packets. Finally, the CPU has to process the full table search requests one by one. Compared to batch processing, which utilizes the concurrent capabilities of the multi-core CPU, this one-by-one processing is very inefficient.

IV. Packet Circulation Reduction with Lazy Packet Submit

Delayed packet submit until end of fast path pipeline. If an SFC has a single table search pipeline without any table search branches, the packet route within the switch will follow a deterministic path. Consequently, there is no need to upload the packet each time it encounters a table search miss. Instead, we can store all the table IDs which experience search misses and their corresponding search keys along the pipeline and upload them as a batch at the end of the pipeline. If there are no table branches, the result will be the same as the per-stage packet submit model (*i.e.*, the busy submit model). When the lazily submitted packet returns to the switch with a group of table search results, we need to parse them into a header stack in one pass at the switch parser. It is important to note that when pushing the results into the header stack, they should be pushed in a reverse order. This ensures that when the packet begins to traverse the pipeline and perform table searches, the results popped out will align with the table sequence.

As shown in the left part of Fig. 5, the packet sequentially incorporates the IDs of the tables not hit during the pipeline processing (table i, table k) into the packet header. Unlike busy submit, lazy submit does not execute egress table bypass. After being resubmitted to the switch, it is parsed into the header stack. Each element within the stack comprises the table ID and search result, popping out subsequently during the match process and facilitating a jump to the corresponding table via ingress table bypass. The delayed submission on the data plane reduces the number of packet loops to a minimum of 1.

Concurrent multi-table lookup in the slow path. In the lazy submit model, when a packet carrying multiple unsuccessful table IDs and their corresponding search keys arrives at the CPU, the CPU can efficiently perform packet lookups in a batch. In a commercial switch server, the CPU is typically a powerful multi-core processor, allowing us to distribute the tables of multiple NFs across multiple CPU cores and perform parallel lookups on these cores to enhance CPU processing efficiency and reduce overall packet processing latency. As shown in the right part of Fig. 5, the CPU utilizes multi-threading to perform key lookups simultaneously on different cores, and installs corresponding rules to table i and table k in the switch, thus greatly improving processing efficiency.

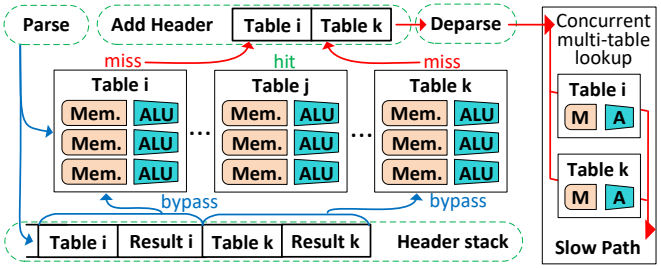


Fig. 5: Lazy submit for packet circulation overhead reduction.

A. Bloom Filter-aided Lazy Submit to Handle SFC Branch

The issue of lazy submit with SFC branch. The previous lazy submit model relies on the absence of SFC table branches. Once an SFC has a table branch, the subsequent search of the NF table will depend on the search result of the preceding NF table. In this case, we must obtain the correct search result of the previous table in advance in order to determine the exact table to search at the next stage. However, if we do not cache the table entry denoting the branch in the fast path, we have to go to the CPU to query the full table to obtain the branch information. This will disrupt the continuous table query process in the lazy submit model. That is, we cannot postpone the packet submit to the CPU until the end of the pipeline according to the lazy submit model, which causes SFC table branch performance penalty. For instance, in the upper left corner of Fig. 6, the search result of table i significantly affects whether the branch result of the next table is table j or table k. If we can cache the next-hop branch information for each table in a compact manner within the fast path, then there is no need to search the full table on the CPU to obtain this information, thus still ensuring the efficiency of delayed packet submission. Considering that hash calculations are very convenient to perform on programmable switches at the fast path, we can use succinct data structures such as Bloom filters [16] to store branch information, without consuming too much of the switch's memory resources.

Encoding table search dependency with 2D Bloom filters. A Bloom filter is a space-efficient probabilistic data structure containing a bit vector with all bits initialized to 0. When an item is added to the Bloom filter, it undergoes hashing through multiple hash functions, and the corresponding bits in the Bloom filter are set to 1 according to the hashing results. While a Bloom filter can determine whether a key "may belong to the set" or "definitely does not belong to the set", it cannot further narrow down the scope when dealing with multiple sets. In other words, it cannot specify which set the key belongs to from multiple sets. To overcome the limitation of a single Bloom filter, here, we propose 2D Bloom filters, represented by bit vectors in the form of a matrix (as shown in the upper right corner of Fig. 6). The 2D Bloom filters actually contain f Bloom filters, each has h hash functions (in Fig. 6, $f = 5$, $h = 4$). To determine the table branch for a packet, we compute based on the 2D Bloom filters following a two-step

process, as shown in Algorithm 1. The first step is to encode the relationship between a key and its next table ID into the 2D Bloom filters (Algorithm Line 2-8), and the second step is to search the 2D Bloom filters to decode the next table ID for an incoming key (Algorithm Line 9-15). With the 2D Bloom filters maintaining the relationship between keys and their next table IDs, when we experience a table search miss, we do not need to query the remote CPU for branch information.

Specifically, for the encoding process (*i.e.*, Function Generator() in Algorithm 1), for each key, we encode its next table ID into an f-bit code (the f is the number of Bloom filters in the 2D Bloom filters). Then, we check each bit of the f-bit code in a for loop. If the checked bit is 1, we insert the key into the corresponding Bloom filter using the hash functions along with that Bloom filter. If the checked bit is 0, we do nothing. The process needs to be done for all the keys in that table to encode their next table IDs into the fast path. For the entire fast path, we need to maintain one unit of 2D Bloom filters for each table to encode the relationship between the keys in that table and their next table IDs. For the decoding process (*i.e.*, Function Selector() in Algorithm 1), for an incoming key, we compute its next table ID based on the 2D Bloom filters. The computation is straightforward: we just search the key in each Bloom filter of the f Bloom filters. Then, we will get an f-bit result with 1 denoting the search success and 0 denoting the search fail. With the f-bit search result, we can decode the next table ID for the incoming key in the fast path.

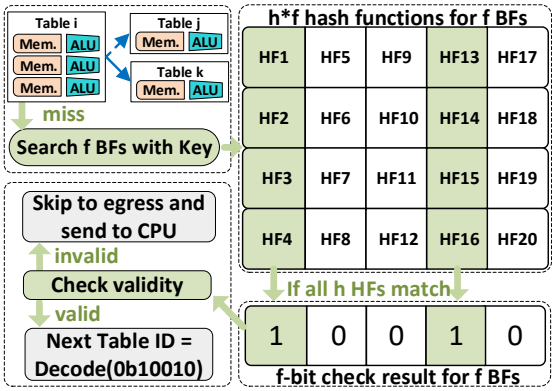


Fig. 6: Next table selection with pre-encoded 2D Bloom filters.

False positive handling. Using Bloom filters inevitably introduces false positives, where packets may be directed to the wrong next branch tables. We handle false positives by using fixed-weight codes (the number of 1s in the f-bit is fixed). When encoding in Generator(), we fix the number of '1's in the code to n (Algorithm Line 4). Correspondingly, we check the weight before decoding in Selector() (Algorithm Line 13-14). If the count is less than n, it indicates that the key does not exist in the entire table. If the count exceeds n, it means it needs to be sent to the CPU. Only when the count equals n, the code will be decoded to retrieve the correct table ID for packet forwarding.

Algorithm 1: BF-aided Branch Select Algorithm

```
1  BloomFilterSet→BFS, Next Table ID→ntid
2  Function Generator(key − ntid map):
3      for each key in key-ntid map do
4          code = encode(ntid) // encode to f-bit
5          for i = 0 to f do
6              if code[i] = 1 then
7                  Insert key into BFS[i]
8      return BFS
9  Function Selector(key, BFS):
10     for i = 0 to f do
11         Search key in BFS[i]
12         code[i] = (Search BFS[i] success)  ? 1 : 0
13     if check(code) = True then
14         ntid = decode(code)
15     return ntid
```

V. Evaluation

A. Experimental Setup

We develop an emulation system to evaluate the performance of HyperSFC on an x86 server with an Intel Xeon Gold 5220 2.20GHz CPU and 16GB RAM, running Ubuntu Linux 18.04. We use Mininet [17] to construct a topology and implement a virtual programmable switch based on the simple switch model of BMv2 [18] in our testbed. We connect a virtual host with a port of the P4 switch to emulate a hyper-converged infrastructure. When a table search miss occurs in the switch, the packet is forwarded to the host with the modified header. We use an agent on the host to process the header and match the keys in the memory database. When completing the full table search, the agent connects to the BMv2 switch, and uses *simple_switch_cli* to dump the corresponding table entries. We implement both busy submit (461 lines of P4 code) and lazy submit (676 lines of P4 code) on the P4 switch, and develop an agent based on rawsocket and Redis (206 lines of C++ code) on the host, available on our git repository [13]. We conduct experiments to measure the latency, bandwidth, and associated costs of different models. Additionally, we optimize a P4-based load balancer [19] using HyperSFC and compare its performance with that of a load balancer with the same logic implemented purely in software.

B. Latency Overhead

Different models in HyperSFC exhibit different packet processing latencies. We implement SFC with the BMV2 switch containing different number of tables and measure the packet latency of the busy submit (busy), lazy submit (lazy), lazy submit with parallel table search (lazy-p), and pure P4 switch forward (base). As shown in Fig. 7, the busy submit has the highest latency because every table search miss results in a packet circulation. While the lazy submit

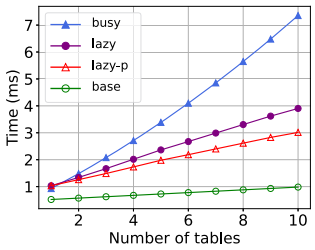


Fig. 7: Comparison of packet processing latency of busy, lazy, lazy-p models and pure BMv2 forwarding.

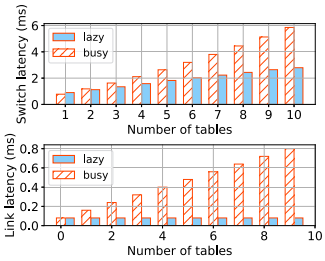


Fig. 8: The lazy submit reduces the switch processing latency and link transmission latency of the busy submit.

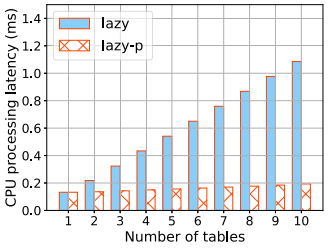


Fig. 9: The lazy-p model reduces the CPU processing latency of the lazy by searching multiple tables in parallel.

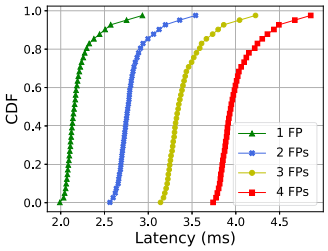


Fig. 10: The false positives during table dependency resolution in the switch pipeline lead to increased latency.

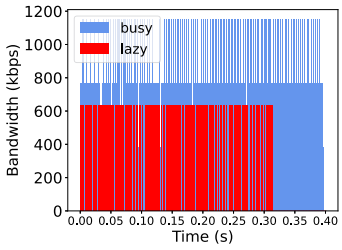


Fig. 11: The lazy submit reduces the southbound bandwidth consumption between the switch and the CPU (4 tables).

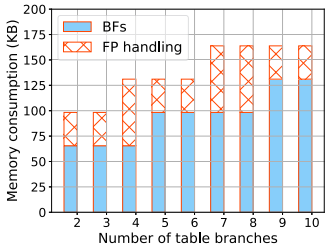


Fig. 12: The memory required to deploy 2D Bloom filter-based table branch selector and to handle false positives.

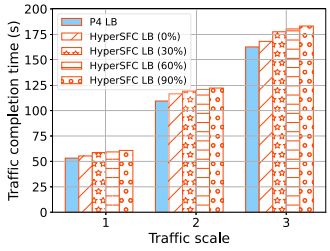


Fig. 13: The traffic completion time of a load balancer reference design in pure BMv2 and HyperSFC, respectively.

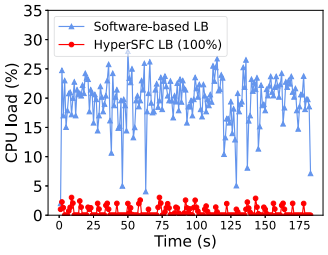


Fig. 14: Compared with the pure software-based NF, HyperSFC is less CPU-intensive in its slow path.

reduces latency by minimizing repeated circulations, achieving a 47% reduction when SFC contains ten NF tables, with even larger reduction for more tables. The lazy submit can further improve latency by simultaneously transmitting multiple table information to the CPU. This enables parallel processing of table search requests, leading to an overall reduction of 60%. In our setup, all packets are sent to the CPU due to table misses, resulting in significant performance differences between HyperSFC models and the base case. In reality, only packets that experience a cache miss in the fast path are sent up. When the cache hit rate is high, the average packet processing latency will be close to that of the base case.

In order to better distinguish the contributions of lazy submit in the pipeline and parallel lookups on the CPU to the packet latency, we conduct a deeper analysis of the two steps. In lazy submit, the optimization of latency primarily comes from two aspects. First, lazy submit reduces the frequency of resubmitting packets through switch processing. Second, it minimizes packet circulation as much as possible, thereby reducing the transmission latency over the link between the CPU and the switch. As shown in Fig. 8, the absolute value of the impact of switch processing is more significant. For parallel lookups on the CPU, even if there are many table misses in the SFC, lazy-p can minimize the latency overhead to the time of a single circulation through parallel lookups. Therefore, the advantage of parallel table lookups is positively correlated with the number of tables (as shown in Fig. 9).

In the lazy submit model, we use a branch selector based on 2D Bloom filters to address the SFC table branch performance penalty. However, false positives in branch selection during pipeline processing can disrupt continuous table processing, resulting in immediate submission of packets to the CPU. As depicted in Fig. 10, an increase in false positives leads to a higher number of circulations, resulting in bloated latency. In the worst case, every missed branch selection can cause a false positive, making it similar to busy submit. Each occurrence of a false positive causes a certain amount of latency in lazy submit, so it is important to minimize false positives. One solution is to use larger Bloom filters to reduce false positives or optimize caching strategies to reduce table search misses.

C. Bandwidth Overhead and Memory Consumption

Apart from latency, the circulation of packets between the switch and CPU will also incur additional bandwidth consumption. In busy submit, although less additional information is attached each time the packet is sent to CPU, the increased frequency of submit leads to greater bandwidth consumption. Conversely, while lazy submit involves attaching a lot of additional information in its submission, the reduced frequency of submit results in overall more efficient bandwidth usage. As shown in Fig. 11 , in the case of an SFC with 4 match-action tables, busy submit consumes more bandwidth, reaching up to 1120kbps. In comparison, lazy submit consumes less bandwidth, peaking at 624kbps — a 44% reduction, accompanied by a 20% reduction in total traffic processing time.

To address the pipeline branch issue, we introduce the 2D Bloom filters to select the next table for a packet in the fast

path. This requires additional memory consumption of the switch, which will increase as the number of branches grows. Moreover, to address false positives, we need even more Bloom filters with additional memory usage. Typically, the number of NF table branches is below 10. Fig. 12 illustrates that each table with branches requires memory proportional to the number of branches, and the memory required to encode branches is on the order of a few hundred KB. Overall, such memory consumption is within an acceptable range.

D. HyperSFC-based L4 Load Balancer

Finally, we evaluate the performance benefits that HyperSFC can bring with a widely deployed NF (L4 load balancer). Here, we compare three LB instances: firstly, a pure switch-based LB with the P4 official implementation [19]; secondly, our version of the LB based on the HyperSFC architecture, which enhances the performance of the LB implemented by the P4 official version; and finally, a pure software LB implemented in C++ with the same logic. The P4 version of the LB has the fastest forwarding speed but limited memories, while the software LB has a larger memory space but lower performance due to forwarding based on CPU cycles. Our HyperSFC architecture can simultaneously leverage the advantages of both: achieving nice forwarding performance on the fast path and maintaining a sufficient number of table entries on the CPU on the slow path. Additionally, by caching table entries on the fast path, we can minimize CPU usage.

In HyperSFC, the number of cached table entries on the fast path affects the table lookup hit rate, which in turn affects packet processing latency and the CPU load on the slow path. In Fig. 13, we cache different proportions of table entries to the fast path, resulting in the proportion of table entries only on the CPU ranging from 0% to 90% (where 0% represents all traffic hitting the fast path, with no traffic submitted to the CPU), and then observe the completion time of traffic processing. Due to table lookup misses on the fast path, HyperSFC LB takes longer to process traffic. However, even when 90% entries are only on the CPU, there is not a significant difference in the final completion time of the traffic. This is because for stateful services like LB, each flow only experiences one table lookup miss for the first packet, and subsequent packets from the same flow can hit the updated entries in the fast path. Therefore, HyperSFC's hierarchical model has a significant advantage in processing state-intensive NFs. In Fig. 14, we compare the performance differences between LB implemented purely in software and LB based on HyperSFC. It can be seen that all processing in the purely software-based LB relies on CPU cycles, consuming an average of 20.86% of CPU resource, while HyperSFC LB consumes very little CPU resource because it only submits the first packet of each flow that misses to the CPU. Even in the most extreme case, where there are no cached table entries in the fast path initially, HyperSFC only consumes 3% of CPU resource due to the submission of the first packet. In contrast, the software-based LB needs to consume CPU resource to process each arriving packet.

VI. Related Work

Due to their ultra-high performance, small form factor, and pipeline-based forwarding architecture, programmable switches [6] are considered an ideal platform for implementing SFC in resource-constrained edge clouds. For example, P4SC [20] offers high-level primitives to build SFCs and compiles the SFC representations into the programmable switch pipeline. Dejavu [14] composes multiple sequential NFs into a single programmable switch, leveraging its pipeline recirculation capability to handle long SFCs that cannot be accommodated within a single switch pipeline. P4Visor [21] consolidates multiple SFCs into a single programmable switch by merging different P4 programs with code redundancy to improve resource efficiency. However, these works do not consider the issue of insufficient switch memories to accommodate state-intensive NFs such as SNAT [10].

Other works propose hierarchical SFC provisioning to address the issue of switch memory insufficiency through different table partition approaches to different memory hierarchies. For example, P4SFC [22] mentions partially offloaded NFs, which are similar to our proposal; however, it does not discuss how to efficiently conduct packet circulation between the switch and CPU on per-stage table search misses. P4NFV [12] mentions multiple packet traversals between the switch and CPU during sequential NF table searches. However, P4NFV adopts a totally different table partition approach: it places stateful NFs on the CPU and other NFs with small tables in the switch. In comparison, our HyperSFC caches popular entries of each table in the switch and leaves their full tables on the CPU. Our cache-based table partitioning leverages traffic locality, resulting in improved SFC performance (*e.g.*, with fewer circulations). Similarly, Metron [23] divides the SFC into stateless and stateful operations. It instructs all available programmable hardware, including switches and NICs, to implement the stateless operations, while dispatching incoming packets to CPU cores that execute their stateful operations. TEA [24] allows to extend the limited switch memories to large virtual tables built on external DRAM. TEA also does not discuss the efficient packet circulation behaviors when per-stage table search misses occur in the switch pipeline. Tiara [25] proposes to fully use heterogeneous hardware to divide the matching tables of the stateful NF into memory-intensive tasks and throughput-intensive tasks, and map them into the most appropriate hardware, respectively (*e.g.*, programmable switch, FPGA, CPU). Tiara does not extend the architecture from NF processing to SFC processing. Microsoft's Sirius [26] offloads NF processing from hosts [27] to a remote shared DPU pool. However, it also does not elaborate on how to conduct SFC with the collaboration of the hosts and the remote DPU pool. LuoShen [7] leverages similar switch server hardware developed by Alibaba Cloud to accommodate NFs for the edge cloud. However, LuoShen leaves the entire stateful NFs (such as LB) to the CPU and FPGA, without caching their popular entries to the switch pipeline for further performance acceleration.

VII. Conclusion

Achieving state-intensive SFC in the edge cloud under the stringent constraints of hardware cost and deployment space is more challenging than in public cloud environments. In this work, we propose *HyperSFC*, a service function chaining architecture dedicated to the edge cloud, based on the recently proposed hyper-converged switch server. Specifically, to address the performance limitation of CPU-based NF processing, we offload the forwarding tables of NFs to the programmable switch and propose the busy submit model with packet circulation to address the fast path table search miss issue due to the limited on-chip memories of the switch. Furthermore, to address repeated circulations in the SFC context, which increase the packet latency and CPU utilization, we propose the lazy submit model, delaying packet submission to the CPU until the end of the switch pipeline and leveraging 2D Bloom filters to reduce unnecessary circulations due to NF table search dependencies.

References

[1] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," Tech. Rep., 2015.

[2] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[3] "Aws outposts," https://aws.amazon.com/outposts/, 2022.

[4] "Accton tests programmable switch server for virtualized networks," https://networkbuilders.intel.com/docs/networkbuilders/accton-tests-programmable-switch-server-for-virtualized-networks.pdf, 2021.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[7] T. Pan, K. Liu, X. Wei, Y. Qiao, J. Hu, Z. Li, J. Liang, T. Cheng, W. Su, J. Lu *et al.*, "{LuoShen}: A {Hyper-Converged} programmable gateway for {Multi-Tenant}{Multi-Service} edge clouds," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 877–892.

[8] X. Shi, Y. Li, C. Jia, X. Hu, and J. Li, "L7lb: High performance layer-7 load balancing on heterogeneous programmable platforms," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2023, pp. 1–2.

[9] Y. Zou, T. Pan, L. Lu, Z. Li, K. Yao, T. Huang, and Y. Liu, "P4rss: Load-aware intra-server load balancing with programmable switching asics," in *ICC 2023-IEEE International Conference on Communications*. IEEE, 2023, pp. 1893–1898.

[10] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 194–206.

[11] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 15–28.

[12] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. Ramakrishnan, and L. N. Bhuyan, "P4nfv: P4 enabled nfv systems with smartnics," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–7.

[13] "Hypersfc repository," https://github.com/graytower/HyperSFC, 2023.

[14] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch asic," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, pp. 141–149.

[15] T. Pan, T. Zhang, J. Shi, Y. Li, L. Jin, F. Li, J. Yang, B. Zhang, X. Yang, M. Zhang *et al.*, "Towards zero-time wakeup of line cards in power-aware routers," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1448–1461, 2015.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[17] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[18] "behavioral model," https://github.com/p4lang/behavioral-model, 2015.

[19] "P4 load balancer," https://github.com/p4lang/tutorials, 2015.

[20] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 1–9.

[21] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 98–111.

[22] J. Ma, S. Xie, and J. Zhao, "P4sfc: Service function chain offloading with programmable switches," in *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2020, pp. 1–6.

[23] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron:{NFV} service chains at the true speed of the underlying hardware," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 171–186.

[24] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. 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*, 2020, pp. 90–106.

[25] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding *et al.*, "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)*, 2022, pp. 1345–1358.

[26] D. Bansal, G. DeGrace, R. Tewari, M. Zygmunt, J. Grantham, S. Gai, M. Baldi, K. Doddapaneni, A. Selvarajan, A. Arumugam *et al.*, "Disaggregating stateful network functions," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1469–1487.

[27] D. Firestone, "{VFP}: A virtual switch platform for host {SDN} in the public cloud," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 315–328.