2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

SpeedyBox: Low-Latency NFV Service Chains with Cross-NF Runtime Consolidation

Yimin Jiang[§], Yong Cui[§*], Wenfei Wu[§], Zhe Xu[§],
Jiahan Gu[§], K. K. Ramakrishnan[†], Yongchao He[§], Xuehai Qian[‡]
[§]Tsinghua University [†]University of California, Riverside [‡]University of Southern California
jymthu@gmail.com, cuiyong@tsinghua.edu.cn, wenfeiwu@tsinghua.edu.cn, xdb009@gmail.com
gjh14@mails.tsinghua.edu.cn, kk@cs.ucr.edu, heyongchaoSSS@gmail.com, xuehai.qian@usc.edu

***Abstract*—Software-based service chains in Network Function Virtualization (NFV) typically suffers high processing latency. This latency grows as chain lengths increase and possibly violates application requirements. Previous efforts focus on reducing latency while maintaining the perspective of each NF being an independent, isolated module. This results in processing redundancy that could eventually become the performance bottleneck.**

In this paper, we propose a low-latency NFV framework called SpeedyBox, that innovatively enables *cross-NF runtime optimizations* in a service chain to eliminate processing redundancy. SpeedyBox builds a fast data path for flows at runtime by consolidating the aggregate actions across diverse network functions (NFs) in a service chain. In SpeedyBox, each NF is instrumented with a stateful Local Match-Action Table (MAT), and leverages our easy-to-use APIs to record its per-flow behavior in the Local MAT. Next, SpeedyBox uses a Global MAT to build the fast data path by consolidating actions from each Local MAT, while providing the ability to express the stateful NF behaviors with an Event Table. We have implemented a prototype of SpeedyBox on the BESS and OpenNetVM NFV platforms. Our trace-driven evaluation on common NFs shows that SpeedyBox achieves significant latency reduction under real world scenarios.

***Keywords*-NFV; low-latency; service chain; consolidation**

I. Introduction

The recent trend in Network Function Virtualization (NFV) is to implement data-plane focused network functions (NFs) in software to achieve more elastic management and lower cost for networked systems [37], [35], [33]. NFs are often connected together to form a sequential *service chain*. The performance requirement for NFV is becoming increasingly stringent (e.g., latency targets of 0.5ms for edge cloud processing [5], [8]). However, software-based NFs (chain) suffer from high latency, which in some cases can become unacceptable when packets traverse a long chain (*e.g.*, up to ten NFs) [19], [27], [23].

A common practice is to co-locate all (or most) NFs in a chain on the same physical server, which reduces cross-server communication overhead [24], [32], [38], [19], [27], [23], [46]. Based on this practice, several solutions have been proposed to accelerate packet processing of NFs and service chains. We summarize two sets of previous work: (1) A set of approaches that focus on *accelerating a section of the data path*, including the use of special hardware (FPGA [25] and GPU [21], [41]), the use of software isolation (Netbricks [30]) to speed up the performance of a single NF, and introducing shared memory for packet delivery between NFs (NetVM [19]) to speed up the service chains. (2) Another body of work proposes to parallelize NFs execution to accelerate service chains (NFP [38] and Parabox [47]), leveraging the fact that some NF pairs do not have dependencies and can be executed in parallel.

Nevertheless, the above two broad NF (chain) processing optimizations hold a common assumption: NFs are modular, and *the boundary between neighboring NFs in a chain still remains*. Specifically, systems that hold this assumption can lead to redundant processing such as repeated parsing and classification, dropping packets late in the service chain (e.g., a packet that has been processed by an upstream NF is dropped by the downstream NF), packet overwriting and redundant IO. Thus, we argue that existing systems are far from optimal, and *there is an upper limit of how much the NF chains can be sped up*. Without a cross-NF optimization that consolidates the processing actions of different NFs, these redundancy still exist and could inevitably become the performance bottleneck as chain lengths increase.

The question now becomes: how can we achieve cross-NF optimization? We closely dissect and analyze the behaviors of modern NFs in enterprise networks [35], and observe that the per-flow behavior of an NF does not change unless an *event* that changes the NF behavior occurs. We can leverage this domain-specific insight to perform cross-NF optimizations. If there is no event happening, we can consolidate the aggregated actions across NFs as the initial packets of a flow traverse the chain, and can directly apply the consolidated action on subsequent packets, without having them traverse the original service chain all over again. Moreover, we analyze the behaviors of modern NFs and find that *events* do not happen frequently, which further enhances the motivation. We call this optimization *cross-NF runtime consolidation* in this paper. Our approach is orthogonal and complementary to the previous approaches we mentioned.

While cross-NF runtime consolidation is an ideal vision for improving the performance of service chains, realizing it in practice does involve two major challenges:

- *Challenge 1: How to collect the runtime behavior of different NFs at minimum cost?* NFs are usually heterogeneous from each other, containing diverse logic and processing actions on packets and internal states. To perform runtime consolidation, the primary concern is how to understand and collect NF behaviors. This requires (1) describing the behaviors of diverse NFs with a uniform abstraction and (2) extracting that behaviors at minimum cost.
- *Challenge 2: How to express the stateful behavior of a*

*Yong Cui is the corresponding author.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00016

service chain on a new, consolidated path? Most modern NFs are stateful [15], [16], [40] and so are the associated chains. Building a new data path needs to ensure that the state is properly handled. Furthermore, the packet output should be identical to that of the original chain. Some similar mechanisms in existing works such as [31], [35], [12], [14] are not applicable to express the stateful and complex behaviors in the context of NFV service chains.

To address the challenges, we propose a novel NFV framework named SpeedyBox that innovatively leverages *cross-NF runtime consolidation* to improve service chain performance. Overall, SpeedyBox has three logical components: the *Local Match-Action Table (MAT)*, a *Global MAT* and an *Event Table*. Each NF is associated with a Local MAT; as the initial packet of each flow traverses the chain, each NF uses SpeedyBox instrumentation APIs to record its processing behavior, including actions on packet and NF state, and populates a record in the Local MAT. Then, the *Global MAT* aggregates and consolidates the actions from each Local MAT to set up a fast data path; all subsequent packets of the flow would directly go through this fast path with all processing on the fast data path being optimized for faster execution. During the processing, the *Event Table* constantly checks if any condition is matched to trigger an associated *event* (*e.g.*, updates to the routing configuration) that changes NF behaviors at runtime, to guarantee the normal functionality of NFs.

SpeedyBox achieves high performance without causing laborious overhead for the NF developers. We provide easy-to-use APIs that minimize modifications on the code for an NF. Since the APIs seek to only record NF behaviors, the modifications do not change the original processing logic and are lightweight. For example, our modification on the Snort IDS [34] only adds up to 27 lines of code.

There are legitimate concerns regarding the isolation between NFs [27]. Additionally, approaches such as OpenNetVM [46] simplify deployment by having NFs in distinct containers so that NFs may be independently developed. However, this is an issue of tradeoffs - isolation, simplicity and flexibility in deployment often come at the cost of performance. Where performance is the dominant consideration, approaches such as [2], [30], [17] operate at the other end of the spectrum, having all the NFs integrated into a monolithic process. The approach in this paper seeks to strike a balance, by accommodating independent NFs in a service chain within containers. But by using a small set simple APIs we provide, developers can support SpeedyBox effectively. Network providers deploying NFV can work with NF vendors to extract the processing actions and enable consolidation across multiple NFs as we describe in this paper. We argue that it is worthwhile to *tradeoff the small amount of programming overhead and deployment effort for obtaining the high performance* of the entire service chain. SpeedyBox still seeks to support deployment of independently developed NFs without necessarily having to consolidate them in a single monolithic process.

This paper makes the following contributions:

- We design SpeedyBox, a novel low-latency NFV framework that exploits cross-NF runtime consolidation to reduce the processing redundancy in a service chain.
- We present a novel NF processing abstraction guided by the behaviors of modern NFs, and then describe how we build the Local MAT for each NF using SpeedyBox's APIs that are lightweight and easy-to-use. For example, our modification on Snort IDS only adds 27 lines of code. (§IV)
- Based on our NF processing abstraction, we propose a Global MAT that consolidates the processing actions from different NFs, while retaining the stateful behaviors of NFs with the Event Table. (§V)
- We have implemented the SpeedyBox prototype and five common NFs on both the BESS [2] and OpenNetVM platforms [46]. We have open-sourced our code [1]. (§VI)
- Our comprehensive evaluations show that SpeedyBox can significantly reduce latency, while strictly retaining the stateful behaviors of the original NFs. (§VII)

II. Context And Challenges

A. Background and Motivation

Low-latency NFV service chains are critical. There are increasing number of applications that demand low end-to-end latency, which put stringent requirement on the processing delay of in-network NFs (and chains) [38]. For instance, the per-packet processing time at the mobile edge cloud needs to be less than 0.5ms [5], [8]. If the requirement is violated, it can have a significantly negative impact on the quality of experience and normal functioning of applications [10], [39].

Redundancy is pervasive in NFV processing. Despite existing optimizations on NFV performance, we still observe redundant processing when a packet flow is being processed by a service chain. Consider a typical service chain derived from [24]: NAT→Load Balancer→Monitor→Firewall. We analyze four kinds of processing redundancy that this chain could incur:

- *R1: Repeated parsing and classification*. Each of the four NFs in the chain needs to perform the same parsing and classification steps on each packet, when ideally all we need is one parsing and one classification step [12];
- *R2: Late packet drop*. Packets that go through the NAT, Load Balancer and Monitor may then be dropped by the Firewall. The redundant and wasted processing inevitably degrades performance. Instead, it would be better to drop the packet at the beginning of the service chain [43], [23];
- *R3: Packet overwrite*. A NAT may modify the destination IP of the packet, but the downstream Load Balancer may further modify the same field, thus overwriting that header field. If we can merge the two modifications, processing and latency can be further reduced [14];
- *R4: Redundant IO caused by isolation*. The performance degradation in NFV brought about by isolation has been well studied [43], [46], [44], [18]. Even in our focused scenario where the entire service chain is put on a single machine, there still exists VM-based [27], [19] or container-based [23], [46], [18] isolation. This isolation inevitably incurs redundant IO and cross-core communication. Note that we

are not arguing against the isolation, but instead, we argue that a consolidated approach can mitigate such overhead by reducing wasteful communication.

All of these redundancy can significantly add to the processing latency. For example, according to our measurements (§VII), overwriting the packet (*R3*) twice can increases the latency by about 2x. Existing work only partially resolves these [12], [43], [23], [14]. Overcoming all the redundancy simultaneously is not possible unless we enable *cross-NF optimizations* across the entire service chain.

Root causes of redundancy. Typically, the intrinsic cause of processing redundancy is the trade-off between *modularity* and *performance*. NFs as the components of a service chain, are often developed independently, and naturally not optimized for performance when being used in a variety of different service chains. Without cross-NF optimizations, simultaneously eliminating *R1-R4* is not feasible. Of course, one can propose developing a highly customized and optimized "Hyper NF" that contains all the functionality and is equivalent to each service chain. However, this method is ad hoc and not generalizable. As service chains become more crucial and complex for modern applications and networked systems [24], [37], [35], we seek a framework that achieves cross-NF optimization without sacrificing the modularity of each NF.

Cross-NF runtime consolidation can build a fast path to eliminate redundancy. An NF usually has the same actions on packets from the same flow unless an *event* that changes NF behavior occurs (which we believe is infrequent). We can leverage this domain-specific insight and build a fast data path for service chain processing: once the initial packet of a flow traverses the service chain, we collect the actions of each NF and apply a consolidated action for subsequent packets directly. For NFs that can have events that change their behavior mid-stream for a flow, (*i.e.*, making the data path mutable in accordance with state changes), we also need a mechanism to inspect state changes and trigger subsequent packet action changes. Together, the cross-NF consolidation naturally eliminates *R1-R3*: (1) the system only needs to parse and classify the packet once; (2) the system can drop a packet early when it arrives at the chain, because the system knows after the initial packet that subsequent packets from the same flow could be dropped by downstream NFs; (3) the system can avoid overwriting packet fields, since it merges multiple actions into one. Further, the consolidation also mitigates the overhead of *R4* by reducing cross-NF communication.

B. Limitations of Existing Approaches

Different approaches for similar scenario. As already mentioned in §I, there are two sets of existing approaches that try to optimize NFV performance. One set of them aim at directly accelerating the data path, using specialized hardware [25], [41], [21], shared memory [19], [27] and abandoning VM isolation [30]. Another set of them aim at widening the data path, leveraging dependency context between NFs to exploit potential parallelism. Nevertheless, all of them still assume the boundary of processing between NFs should be preserved, which is not critical for scenarios that require extremely low latency. Different from these approaches, SpeedyBox enables cross-NF optimization by consolidating the processing of different NFs at runtime.

Similar approach for different scenarios. A similar insight that subsequent packets can be cheaply matched is also proposed in Open vSwitch (OVS) as it uses a Megaflow cache to consolidate forwarding rules [31]. However, the stateless forwarding nature and the intrinsically stateless design of OVS (compared with diverse NFs) makes this approach difficult to be applied directly in the NFV context. We summarize two fundamental limitations:

- *Stateless forwarding model*. OVS is *stateless* because its forwarding model is primarily based on OpenFlow [28]. The Megaflow cache assumes that packets from the same flow (i.e., same five tuple) can be forwarded in the same way. This is clearly not applicable for many stateful NFs [45] who decide their behaviors based on packet payload and internal state. For example, a Maglev Load Balancer [13] (discussed in §VI) that maintains per-flow state may update its forwarding behavior (e.g., change destination IP and port) at runtime if a backend server fails. Achieving this functionality in OVS is difficult.
- *Poor expressiveness for complex NF semantics*. OVS has *poor expressiveness* to support NFs that require complex computation. Many modern NFs require payload parsing/inspection or advanced functions to maintain internal state. For example, the Snort IDS requires regular matching to inspect packet payload [9], which is not supported in standard OVS.

The above limitations are hard to handle until the rise of NFV as a new solution in data path. Contrary to existing approaches, NFV opens the door to achieving complex state manipulation and computation directly in the data path. SpeedyBox leverages this new opportunity.

C. Addressing Design Challenges

- *Challenge 1: How to collect the runtime behavior of diverse NFs at minimum cost? (§IV)*

NFs are often developed by multi-vendors with various behavior. Consolidating their actions requires a common processing abstraction to describe them in an uniform way. We analyze some widely-deployed NFs in enterprise networks, and partition NF processing into (1) *header actions* that transform the packet header and (2) *state functions* that perform payload inspection or updating NF internal states. For each NF, we use an extended Match-Action Table (MAT) called *Local MAT* to record per-flow header actions and state functions. We standardize five types of header actions that cover most transformations on packet headers. To have uniform state functions, we collect the function handlers and invoke them at runtime to describe stateful behaviors. (§IV-A)

We design a set of NF instrumentation APIs following the proposed NF abstraction. Since the abstraction has paved a way for dissecting NF processing, our APIs are easy to use and can cover a body of commonly used NFs. The APIs

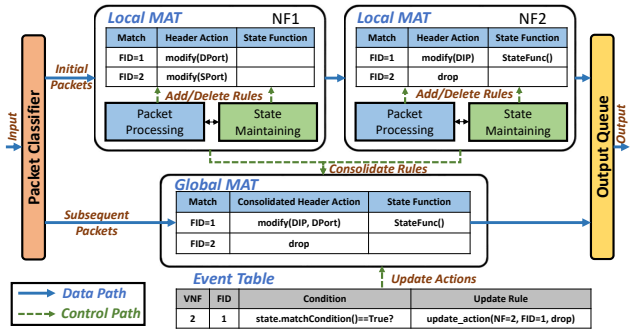


Fig. 1. SpeedyBox architecture.

only collect the behavior of NF codes and do not change the original processing logic, and the performance overhead can be neglected according to our measurement. The modification is also lightweight with negligible cost, *e.g.*, our modification on Snort IDS [34] only adds up to 27 lines of code. (§IV-B)

- *Challenge 2: How to express the stateful behavior of a service chain on a new consolidated path? (§V)*

Different from static switch rules, many modern NFs are stateful [15], [16], [40] — packet processing updates states and states decide packet data path. When consolidating these stateful NFs, we need to express the equivalent behaviors on the new data path. If packets traverse the original data path and trigger state updates, the new path must have its state updated in the same way. More importantly, if a state reaches certain conditions and causes future packet processing logic to be changed (we define it as an *event*), the new path must be updated to the new processing logic immediately. We carefully design the *Global MAT*, that enables a novel execution model for state functions (packet processing updates states), and also contains an *Event Table* to check the conditions for triggering an event (states decide packet processing). If a condition of an event is satisfied, the actions on packets and state would be modified accordingly. Thus, SpeedyBox is able to express stateful behaviors of a service chain. (§V)

III. Design Overview

SpeedyBox aims at building a fast data path for flows in service chains with the logic of the original NF service chain retained. For each flow, we define the initial packet as the first packet after a connection is established (e.g., after the 3-way TCP handshake). As the initial packet traverses a service chain, SpeedyBox collects how each NF in the chain operates on the packet and updates NF internal state, with an extend Local Match-Action Table (MAT). Each NF is associated with a Local MAT. SpeedyBox then aggregates all the actions/functions and consolidates them to form a new, logically-equivalent data path in a Global MAT. For subsequent packets of the same flow, they are directed to the new path for faster execution. However, some NFs may change their flow actions during runtime, e.g., when certain internal states reach certain conditions or thresholds. To this end, SpeedyBox proposes an Event Table to trigger *events* in a timely manner to update the behaviors of NFs.

Figure 1 shows the architecture of SpeedyBox, and also illustrates the workflow with an example of packet walkthrough. Once a packet arrives, the Packet Classifier first generates its FID by hashing the 5-tuple. The FID will remain consistent throughout the service chain (even if the 5-tuple is modified). Next, the Packet Classifier directs initial and subsequent packets to two different paths:

- For an initial packet, the Classifier sends it to the first NF of the original service chain. As the packet traverses each NF, the associated Local MAT records the processing rule of the flow, containing header actions and state functions . Additionally, NFs also register *events* that can update NF processing rules. The events can be triggered using the FID of the packet to match the pre-defined events in the Event Table at runtime. As soon as the service chain finishes processing the packet, SpeedyBox notifies the Global MAT to consolidate the rules for the FID from all Local MATs. In this example, NF_1 receives a packet with FID=1, and set the header action as `modify(DPort)`, which means modifying the destination port of packets with FID=1. Similarly, NF_2 set the header action of FID=1 as `modify(DIP)` that modifies the destination IP, and also specifies the state function (`StateFunc`) for packets from FID=1 according to its own processing rules (e.g., inspecting packet payload, incrementing flow counters, etc).
- For a subsequent packet, the Classifier sends it to the Global MAT. The Event Table first checks if an associated event has been triggered (`state.matchCondition`: a general callback handler that can be implemented with user-defined functions). If not, the Global MAT directly applies the cached consolidated header action (`modify(DIP,DPort)`) on the packet, and also executes the state functions by invoking the function handlers (`StateFunc`) as recorded by previous packets. If the conditions of an event are satisfied (e.g., a counter exceeds certain threshold), SpeedyBox updates the header actions or state functions of the associated flows (e.g., Maglev changes the destination IP of a flow [13]), so that subsequent packets can have updated processing rules.

IV. Local MAT

The Local MAT is responsible for recording NF actions or functions on packets and state. We first propose an unified NF processing abstraction based on our analysis of NFs in enterprise networks (§IV-A), and then discuss how to build the Local MAT with SpeedyBox's APIs (§IV-B).

A. NF Processing Abstraction

Since NFs can be diverse and complex, to effectively consolidate their actions, we propose a uniform abstraction for typical NFs. We examine and analyze the processing logic of several representative NFs in enterprise networks [35], such as Gateways (for conferencing/media/voice), Firewalls, VPNs, Load Balancers, NATs and IDSs. According to the survey in [35], these NFs are widely deployed, constituting about 82.7% of the total number of deployed middleboxes. We discuss the scope of our approach in §IV-A3.

Overall, we observe that NF processing can be partitioned into two parts: (1) *Header Action*: transformation of the packet header or packet drop (for basic routing), and (2) *State Function*: operations on NF internal state and inspection of packet payloads (for advanced middlebox functioning).

*1) **Header Action**:* A header action denotes how an NF operates on the packet header or if it drops the packet. Based on the observations of the examined NFs and related work [30], [14], we define five standardized header actions for NFs: (1) `Forward`: NFs like Network Monitors only parse the packet for internal state update, and then forward it without modification; (2) `Drop`: NFs (e.g., Firewalls) may drop the packet, and set the associated packet descriptor to nil in our implementation; (3) `Modify`: NFs may modify the header fields to achieve basic routing. Examples include Gateways, Load Balancers and NATs; (4) `Encap` and `Decap`: Some NFs may add/remove headers for/from a packet. For instance, VPNs add an Authentication Header (AH) for each packet before forwarding (`encap`), and remove the AH when the other end receives the packet (`decap`) [42].

*2) **State Function**:* A state function denotes an advanced function that an NF invokes to update internal state or inspect the packet payload. Flows traversing an NF may be assigned different state functions, since they can be assigned to different conditional branches in the NF's code logic. Note that payload inspection is also cast as a state function, because the results of the inspection also results in state update in the NFs examined (e.g., in Snort, the inspection function will set the flags of malicious flows). Based on how state functions interact with payloads, we set three types for state functions: (1) read the payload (READ), (2) write the payload (WRITE) and (3) do not read or modify payload (IGNORE).

State functions are usually wrapped as callback functions in an NF's code, e.g., deep packet inspection in Snort IDSs [34] and per-flow packet counting in network monitors. Some NF programs may not wrap their logic in the form of callback functions, which will take more effort for us to carefully modify them. Fortunately, we find that popular NFs like Snort incorporate their main functionality in properly-decoupled callback functions, making it easy to integrate them into SpeedyBox (§VI-C). We use the handler of a callback function to represent the state function and store it in the Local MAT. During NF runtime, SpeedyBox executes the state function by invoking the associated handler.

Some state may be shared by a collection of flows [15], [36], and multiple flows may share a state function. In this case, we record the state function for all associated flows. In implementation, we carefully design the processing concurrency and avoid runtime conflicts or incorrect logic (§VI-A).

*3) **Applicable Scope for SpeedyBox**:* While we recognize that it would be ideal for SpeedyBox to support any general NF (and chain), NFs whose functionality largely relies on buffering a sequence of packets and operating on them are intrinsically not well-suited for our framework. Examples of such functions include caches and WAN optimizers [16], [35], which require executing a loop waiting for packets. The NFs would then perform operations on the aggregate of packets buffered. For these NFs, the performance bottleneck likely is in having to wait for the batch of packets to arrive rather than the packet processing overhead. SpeedyBox's gains are particularly significant with NFs that perform per-packet processing using a Match-Action primitive (*i.e.*, act on receiving each packet), such as a Firewall, Load-Balancer or NAT. SpeedyBox would be applicable to a large fraction of functions typically used in an enterprise network [35], with 82.7% NFs falling into this category. The rest of these NFs, while also able to be incorporated in SpeedyBox's framework, are actually inefficient with runtime consolidation.

```
// Extract FID from the packet
int nf_extract_fid(packet_descriptor*)

// Add header action for the flow
void localmat_add_HA(int FID, HA header_action,
        args* arg_list)

// Add the handler as a state function
// function type: (PAYLOAD) WRITE/READ/IGNORE
void localmat_add_SF(int FID, function_handler*,
       int function_type, args* arg_list)

// Register event with update action or function
void register_event(int FID, condition_handler*,
       args* arg_list, HA update_action,
       update_function_handler*)
```

Fig. 2. SpeedyBox APIs for network functions to build their Local MATs.

B. Building the Local MAT with Easy-to-Use APIs

The NF processing abstraction enlightens us to design easy-to-use APIs for NFs to build their Local MATs. Note that our goal is to minimize the modification to NF code, or at least not change the major processing logic. Specifically, SpeedyBox provides several interfaces to help programmers to specify NF header actions, state functions and events respectively, as shown in Figure 2. When adding header actions, we provide a set of standardized header actions, with additional arguments, such as which header field to modify. When adding a state function, we provide the handler of the state functions along with function arguments, and also the associated function type (payload read/write/ignore) (§IV-A2). For registering an event, we provide a `condition_handler` that checks whether certain conditions in the NF are matched. We also need to provide the action/function for updates.

At NF runtime, the Local MAT adds header actions and state functions for each flow *in sequence*. Maintaining the order of state functions is crucial to guarantee logic equivalence, otherwise code dependencies may be violated. We use a queue data structure to maintain the sequence in our implementation.

V. Global MAT

A. Guidelines for Consolidation

We first present two important observations on NF behaviors, which are crucial guidelines for the consolidation.

- *Observation 1: In most cases, per-flow header actions and state functions is deterministic by the initial packet.*

For instance, we manually inspect the source code of Snort [9] and observe that Snort assigns a rule matching function for each flow as initial packet arrives. For subsequent packets, the same function is invoked repeatedly. This is also true for other Layer-3 NFs, such as Load Balancers, NATs and Firewalls. For example, once a NAT allocates a header action with a new destination IP and port for a new flow, the same header action applies to all subsequent packets in the same flow.

- *Observation 2: In other cases, some NFs can trigger "events" that update their header actions or state functions during runtime. Specifically, an* event *is triggered when some internal state is updated to a certain condition.*

This point addresses the corner cases of the first observation, and is crucial for some NFs. For example, the Google Maglev load balancer [13] can reroute an *established* flow to a new backend server (with IP as `new_ip`) using consistent hashing, if the original backend server (with IP as `origin_ip`) fails. Suppose the original Maglev header action for a flow is `modify(DIP, origin_ip)`. After the rerouting, the header action needs to be updated as `modify(DIP, new_ip)`. This runtime action update during is called an *event* in SpeedyBox. Based on our observations of existing NFs, *an event is triggered only when some internal state is updated*. For common NFs, events do not occur frequently, but are crucial parts of NF stateful logic.

Overall, Observation #1 implies that the consolidated result can be reused and do not change in most cases, unless otherwise notified (by an *event*). Observation #2 addresses that we should first check if an *event* has been triggered, and then decide whether the consolidated result can be reused.

B. Consolidating Header Action

There are five heterogeneous header actions (§IV-A1) including `modify`, `encap`, `decap`, `forward` and `drop`. This heterogeneity imposes difficulties for consolidation. We seek to synthesize the header actions aggregated from NFs with an algorithm. The input of the algorithm is a list of header actions, and the expected output should be a consolidated header action. We omit the `forward` action in the following because we set it as the default action if no other action is provided. As SpeedyBox goes through the action list, we discuss how it consolidates different actions:

- **Drop**: As long as the list contains at least one `drop` action, the final action should be `drop`. In this case, we set the packet descriptor to null and release the packet memory.
- **Encap/Decap**: We use a stack to simulate the header encapsulation and decapsulation process. Encapsulation is pushing a new header to the (packet) stack, and decapsulation is popping a existing header from the stack. If two adjacent `encap` and `decap` actions operate on the same header, we eliminate them simultaneously.
- **Modify**: If two `modify` actions change the same field but with different values, we select the value of the latter `modify`. If they operates on different header fields, we consolidate the two `modify` actions into one using bit operations. Assume P_0 is the original packet, and denote P_1, P_2 as the output of *modify1* and *modify2*, respectively. Suppose *modify1* and *modify2* touch different fields, the output packet can be expressed as $P_0 \bigoplus[(P_0 \bigoplus P_1)|(P_0 \bigoplus P_2)]$. The $\bigoplus$ operator means XOR. We iterate the process incrementally and obtain the output.

In addition to IP and Port fields, we may also need to modify the remaining fields of packets, such as checksum, TTL, MAC address and packet length. Since these fields are unlikely to be part of the main processing logic of NFs according to our observations, we modify these fields at the end of the consolidation, ensuring that SpeedyBox outputs valid packets.

C. Consolidating State Function

Consolidating state functions requires executing the functions aggregated from different Local MATs. The key challenge here is maintaining the stateful logic of the original chain in the new data path. We first describe how we we execute the state functions (§V-C1), and then introduce how we optimize the execution with parallelism (§V-C2).

*1) **Executing State Function with the Event Table**:* SpeedyBox executes state functions in the order that they are added to the Local MATs, so that the NF processing logic is retained. We define all state functions of a rule as a *state function batch*, and all state functions in a batch should be executed in sequence. When executing state function batches, SpeedyBox triggers events when conditions are matched. Based on Observation #2 (§V-A), the Global MAT *checks whether certain conditions are matched as soon as the associated states have been updated*. This motivates us to design the *Event Table*.

The Event Table supports triggering NF-registered events and enables updating NF actions/functions at runtime. SpeedyBox lets NFs specify *under what conditions an event should be triggered* and also *the associateed update action/function*, leveraging the `register_event` interface provided by SpeedyBox exposed APIs (Figure 2). When registering an event, a `condition_handler` is needed that specifies how to check whether an event is triggered and what the exact condition is. In addition, we also need to provide the associated header action/state function handler for updates.

Figure 3 demonstrates the workflow of the Event Table using an example of a DOS Prevention NF for illustration. The DOS Prevention NF detects a DOS attack by monitoring the number of TCP_SYN flag on a per-flow basis. The top left table is the original global MAT before consolation, and the bottom right table is its consolidated global MAT. If the number of SYN flags seen exceeds a threshold (`flow1_cnt>100`), the Event Table (bottom left) triggers an event to replace the `modify` action with a `drop` action (top middle), and a new consolidated global MAT is computed (top right).

*2) **Optimization: Combining State Function Execution with Parallelism**:* To further reduce the latency, we incorporate parallel execution [47], [38] into SpeedyBox with our customizations. While the state functions in a batch should be executed sequentially to guarantee the equivalence of NF internal logic, we find that some state function batches across NFs can be executed in parallel. We analyze the dependency

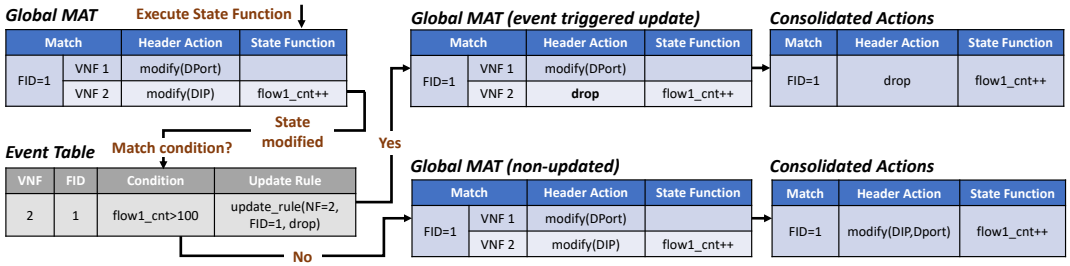


Fig. 3. State function execution with the Event Table to express stateful behaviors.

TABLE I. DEPENDENCY ANALYSIS OF STATE FUNCTION PARALLELISM. FOR CHAIN $NF_1 \rightarrow NF_2$, SF_batch_1 AND SF_batch_2 ARE THE STATE FUNCTION BATCH OF NF_1 AND NF_2, RESPECTIVELY.

SF_batch_1 \ SF_batch_2	Payload Write	Payload Read	Payload Ignore
Payload Write	N	N	Y
Payload Read	Y	Y	Y
Payload Ignore	Y	Y	Y

(Y: parallelizable, N: not parallelizable)

between two batches in Table I. The way that $batch_1$ affects the processing of $batch_2$ comes from packet payload dependency. Note that in our problem space, there is no packet header dependency because such dependency is already eliminated by the Global MAT, which aggregates the header actions belonging to the same flow. Each state function has different actions on the payload: write/read/ignore (§IV-A2). Since each batch contains multiple state functions, we determine the action property of an entire batch as: the action of the state function that has the highest priority in the batch (priority: WRITE>READ>IGNORE). For example, a batch with {read, read, write} is determined as *write*. If $batch_1$ and $batch_2$ both read the payload, we can pass the packet payload descriptor to them simultaneously, enabling parallel execution. However, if $batch_1$ writes the payload, they cannot be parallelized unless $batch_2$ ignores the payload.

VI. IMPLEMENTATION

A. Execution Environment

We have implemented our SpeedyBox prototype on top of BESS [2] and OpenNetVM [46]. Both of them are well-known NFV platforms and are already leveraged by some academic researches [29], [47], [38], [23]. We briefly introduce our customizations on these two platforms.

BESS: BESS ([17]) typically implements an entire service chain as a single process on a dedicated core . We implement the Global MAT as a global array that can be accessed by all Local MATs. We develop the packet classifier using the `Task` class, the Global MAT executor as a new BESS module, and construct a service graph with two branches: one branch for initial packets that traverse the original service chain, and the other for subsequent packets that traverse the Global MAT. The entire customization adds about 1900 LOC to BESS.

OpenNetVM: OpenNetVM [46] runs each NF on one dedicated core, and interconnects NFs leveraging RX/TX queues that deliver shared memory packet descriptors. We develop the Global MAT at the NF Manager, and the packet classifier at the Manager's RX queue thread. The consolidation of Local MATs rules involves inter-core communication. We leverage the existing inter-core message queues (implemented as ring buffers in OpenNetVM) to achieve this. Our extensions to OpenNetVM have around 2800 LOC.

TABLE II. NFS IMPLEMENTED FOR EVALUATION AND ADDITIONAL LOC TO INTEGRATE THEM INTO SPEEDYBOX.

Network Function	LOC for Core Functionalities	Added LOC
Snort	1129	27 (+2.4%)
Maglev	141	23 (+16.3%)
IPFilter	110	20 (+18.2%)
Monitor	223	19 (+8.5%)
MazuNAT	358	20 (+5.6%)

B. Packet Classifier

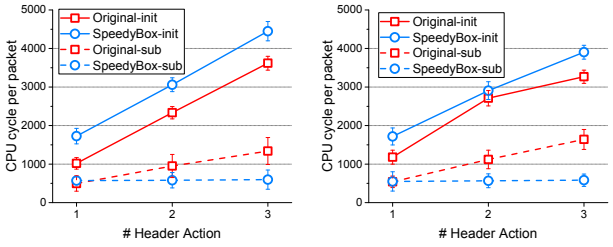
FID Consistency: To guarantee FID consistency across Local MATs, the Packet Classifier hashes the five tuple of a packet header to a 20 bits FID, and attaches it directly to the packet as a meta-data. This 20 bits sequence can represent more than 1 million concurrent flows (enough for our evaluations). We can extend the FID length to accommodate more flows. Once assigned, the meta-data remains consistent along the service chain in the data path, so that every NF in the chain can use a consistent FID. Since both BESS and OpenNetVM deliver packets across service chains with lightweight packet descriptors, our approach does not incur high packet copying overhead. When the packet leaves the service chain, SpeedyBox detaches the meta-data from the packet.

Tracking Flow State: In order to clean up stale rules for flows whose connections have been closed, the Packet Classifier also monitors the TCP_FIN and TCP_RST flag. Once the final packet of a flow (with FIN or RST flag) arrives, we delete the corresponding rule from the Global MAT and all Local MATs and free the associated memory space.

C. Network Functions

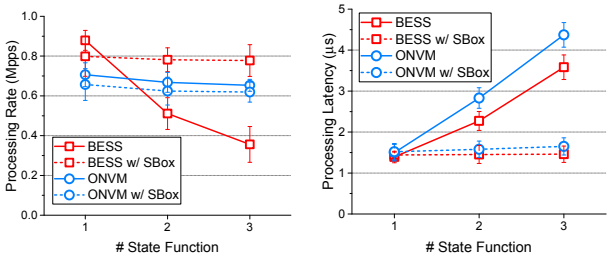
We implement five popular network functions and integrate them into SpeedyBox with our APIs, and the additional lines of code (LOC) for core functionalities is shown in Table II.

Snort [34] is an IDS that inspects traffic against the rule lists to classify the packets. We derive the source code of Snort from [9] and migrate it to our system. We first modify the libpcap-based Snort into DPDK-compatible, and then cast the handlers of the packet inspection functions as the state



(a) BESS (b) OpenNetVM

Fig. 4. Effect of header action consolidation. (*init* and *sub* for initial and subsequent packets)



(a) Processing Rate (b) Processing Latency

Fig. 5. Effect of state function parallelism.

functions. And since Snort does not modify packets, we set Snort with `forward` as the header action.

Maglev [13] is Google's software Load Balancer for networking. A Maglev NF distributes flows to their destinations like normal Load Balancer, while also tolerates network faults (e.g., a destination machine fails unexpectedly) leveraging consistent hashing and connection tracking. Since Maglev is not open-sourced, we implement our Maglev NF logic by closely following the consistent hashing algorithm presented in Section 3.4 of Maglev's paper [13]. We set per-flow events for Maglev as updating the destination IP for existing flows when failure occurs, and emulate unpredictable failure events by setting the condition handlers with random trigger functions.

IPFilter [3] is a Firewall prototype that parses flow headers and checks against a header blacklist with linear scanning. For flows that match the blacklist, we set them with `drop` actions, or otherwise with `forward` actions.

Monitor is a network monitor that is commonly used in academia [30], [38], [35]. It maintains packet counters for each each flows, and sets each flow with a `forward` action and a state function to maintain the associated counter.

MazuNAT [6] closely resembles the NAT module in Click [22] that translates the IP and port for flows. We omit irrelevant functionalities in [6] such as ICMP packets handling. MazuNAT sets each flow with a `modify` action.

VII. Evaluation

Our evaluation is performed on a testbed system having an Intel Xeon E5-2660 v4 CPU (2.00GHz) containing 14 physical cores, with 32GB of RAM and an Intel Corporation 82599ES 10-Gigabit NIC. The OS runs Linux kernel 4.4.0-31. To test the performance, we deploy a DPDK-based Packet Generator [4] on another server (also equipped with a 10G NIC) and directly connect it to the testbed. In the following figures and tables, SBox denotes SpeedyBox, and ONVM denotes OpenNetVM.

A. Performance Microbenchmarking

We created several micro-benchmarks to evaluate the basic performance of two optimizations in SpeedyBox. Specifically, we evaluate how header action consolidation and state function parallelism improve service chain performance. All experiments in this section are performed using 64B packets.

*1) **How does header action consolidation improve performance**:* We vary the number of header actions to evaluate how consolidating them saves CPU cycles for processing. We use a chain with 1-3 IPFilter NFs. The results are representative, and comparable with other NFs, so we only provide IPFilter results here, while the evaluation results of other NFs are in [7].

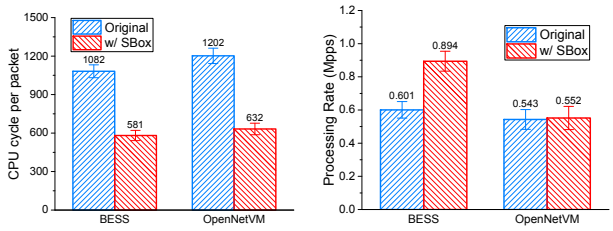
Figure 4(a) and 4(b) show the results with BESS and OpenNetVM respectively. For comparisons, we plot the CPU processing cycles of the chain w/ and w/o SpeedyBox for initial and subsequent packets. The results are similar on both frameworks. Thus we focus on BESS for analysis. As shown in Figure 4(a), for both chains w/ and w/o SpeedyBox, the initial packet needs to spend much more processing cycles than subsequent packets due to the initialization processes (e.g., linear matching of ACL lists for new flows). For subsequent packets, when there is only one header action, SpeedyBox costs more processing cycles than the original chain because of the extra overhead for recording the processing rules into the Local MAT; when the number of header actions increases to 2 and 3, consolidation reduces 40.9% and 57.7% CPU cycles than original chain. Theoretically, this reduction can be as high as $\frac{N-1}{N}$ for N header actions.

As a special case, we next demonstrate that consolidating header action can enable early packet drops, thus reducing CPU cycles. We use a chain with three IPFilters (NF1, NF2, NF3) and set the corresponding actions as {`forward`, `forward`, `drop`} for all flows, respectively. Originally packets in a flow need to traverse the entire chain and are dropped at NF3. With SpeedyBox, however, subsequent packets can be dropped early when they arrive at the chain because of header action consolidation. Table III shows that SpeedyBox enables early packet drop and saves ∼65% CPU cycles.

*2) **How does state function parallelism improve performance**:* Parallelizing state functions can increase processing rate and reduce overall execution latency. We use a chain of 1-3 identical synthetic NFs for evaluation. The synthetic NF has no header action, and has one state function that is

TABLE III. Early packet drop saves CPU cycles.

(CPU cycle)	NF1	NF2	NF3	Aggregate
BESS	530	582	577	1689
BESS w/ SBox	—	—	—	591 **(-65.0%)**
ONVM	510	570	540	1620
ONVM w/ SBox	—	—	—	570 **(-64.8%)**



(a) CPU cycle (b) Processing rate

Fig. 6. Consolidation and parallelism improves the performance of the Snort + Monitor chain.

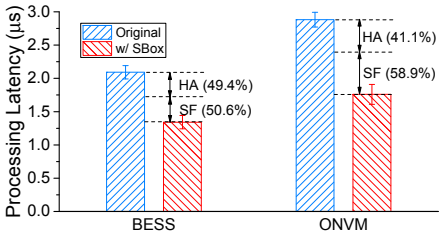


Fig. 7. Latency reduction of the Snort+Monitor chain and the contributions of two optimizations (HA denotes header action consolidation, SF denotes state function parallelism).

equivalent to the Snort packet inspection (does not modify payload). According to the analysis in Table I, two of these state functions can be parallelized. We then vary the number of the synthetic NFs (# of state function) in the chain and measure the processing rate and per-packet latency.

Figure 5(a) shows the processing rate results. For BESS, the processing rate of the original chain decreases as number of state function increases. Nevertheless, SpeedyBox prevents the processing rate from decreasing a lot by carving out a new parallel data path. For instance, compared to the original BESS (sequential), BESS with SpeedyBox achieves 2.1x processing rate. For OpenNetVM, however, the processing rate stays relatively stable. This is because the original OpenNetVM uses pipelined processing and will not sacrifice processing rate when the number of state functions increases.

Figure 5(b) shows the latency benefits of SpeedyBox. For example, SpeedyBox reduces the latency by 59% for BESS when there are three state functions. Suppose we have N identical state functions, the optimal latency reduction can be $\frac{N-1}{N}$. Note that when there is only one state function, there exists a little performance degradation due to extra overhead caused by collecting NF behaviors.

B. Service Chain Performance

1) ***How does each optimization contribute to the overall performance improvement****:* Next, we show how the header action consolidation and state function parallelism contribute to the overall improvement by using a chain with a Snort NF followed by a Monitor. Both of them have header actions and state functions, and thus will benefit from the two optimizations simultaneously. Figure 6 shows the CPU cycle reduction and processing rate improvement of the Snort+Monitor chain. SpeedyBox reduces CPU cycles of per packet processing by 46.3% and 47.4% for BESS and OpenNetVM, respectively. The reduction is due to the header action consolidation. Moreover, SpeedyBox improves the processing rate of BESS by 32.1% by incorporating parallelism. Note that SpeedyBox does not improve the processing rate of OpenNetVM. This is because the original OpenNetVM already uses pipelined processing, and the processing rate will not drop significantly when chaining two NFs. This observation is identical with the results provided in OpenNetVM's paper (Figure 2 in [46]).

Next, Figure 7 analyzes how SpeedyBox reduces the processing latency and how much each optimization contributes to the reduction. For BESS, the overall processing latency is reduced by 35.9%; of this reduction, we measure that 49.4% is contributed by header action consolidation while the remaining 50.6% by state function parallelism. The result on OpenNetVM is similar, except that parallelism makes up a larger portion (58.9%) than that in BESS. This is due to the inter-core communication overhead in OpenNetVM, which introduces extra CPU cycles and partially reduces the benefit of header action consolidation.

2) ***Can SpeedyBox support long chains****:* Next, we demonstrate how SpeedyBox supports long chains without violating performance. We use a chain with 1-9 IPFilters. The ACL rules of the IPFilters are carefully modified to avoid packet drops. Note that in OpenNetVM, we can only support a maximum chain length of 5, limited by the number of cores on our testbed; for BESS, there is no such limit as all NFs are part of one process. The processing rate and latency results are shown in Figure 8. The latency performance of SpeedyBox is nearly irrelevant to the chain length, indicating that SpeedyBox can significantly reduce the latency for long chains thanks to cross-NF consolidation. Besides, SpeedyBox can maintain high processing rate for BESS when running long chains. And again, SpeedyBox does not improve the processing rate of OpenNetVM, since the original OpenNetVM adopts a pipelined model that already ensures high processing rate regardless of the chain length [19], [46].

3) ***How does SpeedyBox perform in real-world service chains****:* Finally, to understand how SpeedyBox performs under real world setups, we derive two service chains from [24], [26] with some customization: we replace the general notion of "IDS" in [26] with our Snort IDS. Similarly, we also replace "NAT" with MazuNAT, "Load Balancer" with Maglev, and "Firewall" with IPFilter.. We measure the *flow processing time* as the aggregated time spent processing all packets in a flow, demonstrated in Figure 9. We use the popular datacenter trace as the input traffic [11]. Since the payloads in the trace are null for anonymization, we synthesize the testing traffic with customized payloads according to the inspection rules in Snort.

Chain 1: MazuNAT+Maglev+Monitor+IPFilter. This chain is identical with the example in Motivation (§II). We do not set events for Maglev in this experiment. SpeedyBox can consolidate the header actions of all four NFs in the chain. The processing time of all flows in the trace is shown in Figure 9(a). For BESS, SpeedyBox reduces the flow processing time

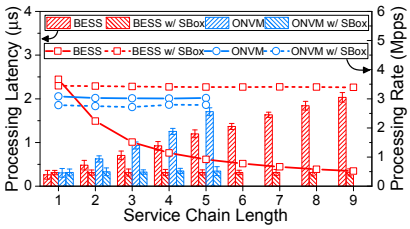


Fig. 8. Performance of SpeedyBox in supporting a service chain with different lengths (The maximum length in OpenNetVM is 5 due to core number limits).

at 50^{th} percentile by 39.6%. For OpenNetVM, the reduction of flow processing time at 50^{th} percentile is 40.2%.

Chain 2: IPFilter+Snort+Monitor. Figure 9(b) shows the results. SpeedyBox parallelizes the state function execution of Snort and Monitor, and consolidates the header actions of all three NFs. For BESS, SpeedyBox reduces the flow processing time at 50^{th} percentile by 41.3%. For OpenNetVM, SpeedyBox reduces the flow processing time at 50^{th} percentile by 34.2%. The results demonstrate that SpeedyBox can significantly reduce the processing latency under real world workload.

C. Empirical Tests on Equivalence

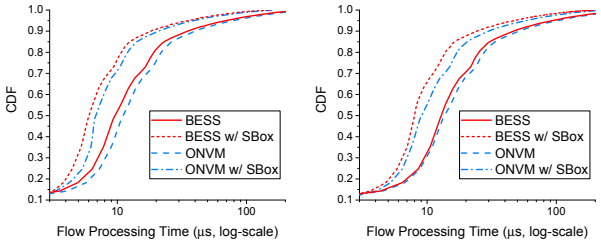
SpeedyBox is designed by strictly retaining the logic equivalence of the original NF and service chain. To test the equivalence, we sample and test some chains. Generally, the methodology is to inject various packets into the system to cover different conditional branches in the code. If the system generates identical packet outputs and state, we are confident that SpeedyBox guarantees equivalence. We show three representative case studies:

*1) **Testing Snort (different conditional branches)**:* We inject three sets of flows containing suspicious payloads that match all the three types of inspection rules (Pass/Alert/Log) of Snort to cover the conditional branches sufficiently. We examine and find the log outputs are identical.

*2) **Testing Maglev (containing events)**:* We inject a flow with 10 packets into Maglev, and set the associated event condition as "change the destination IP from ip_1 to ip_2, from the sixth packet". Denote the sequence of the 10 packets as $pkt1$, $pkt2$, ... and $pkt10$. We check the packet outputs and find the destination IP of $pkt1$-$pkt5$ is ip_1, and the destination IP of $pkt6$-$pkt10$ is ip_2. The remaining headers and packet payloads going to ip_2 are verified to be true. Thus, the event has been triggered correctly.

*3) **Testing real world chains (comprehensive test)**:* We also test the equivalence of SpeedyBox in real world service chains in §VII-B3. In the first chain's Maglev NF, we set events for 20% flows during mid-stream. We find that there is no difference between the packet output for both chains. Further, we compare the per-flow counters of the Monitor and the log outputs of Snort. Results show that the value of all counters and the Snort logs are all identical with and without SpeedyBox. And the events of Maglev have been triggered correctly for all associated flows.

Our empirical tests show that NFs consolidated in SpeedyBox have equivalent logic with the original NFs and chains.



(a) Chain 1 (b) Chain 2

Fig. 9. CDF of flow processing time of datacenter traces in different real world service chains.

VIII. Other Related Work

Consolidation in NFV: To our best knowledge, CoMb [35] is the first to introduce the notion of consolidation into NFV literature. However, CoMb focus on consolidating network functions on single physical machine to enable better resource management. We inherit the scenario that multiple NFs are densely packed on one machine, but we focus on reducing the processing redundancy in service chains.

Redundancy elimination. OpenBox [12] proposes to eliminate redundant logic across NFs, by dissecting NF into basic elements, reorganizing the element graph and identifying the redundant operations. However, OpenBox does not enable packet early drops and parallel execution. SpeedyBox enables these two optimizations through consolidation. SNF [20] also proposes redundancy elimination in NFV by synthesizing the service chains, while the authors do not mention how to implement complex NFs such as Snort in SNF's framework. SpeedyBox provides flexible framework for state management and also enables state functions parallel execution.

Optimizations in software switches: VFP [14] extends OVS's idea by introducing a transposition engine that records actions of the first packet and apply the transposition directly to subsequent packets, similar to OVS's methodology. However, existing solutions of OVS and VFP are not applicable in NFV, since NFs are too diverse to consolidate and some NFs may require runtime logic update that makes the data path mutable. SpeedyBox overcomes the two challenges with stateful MATs and the Event Table.

IX. Conclusion

There are substantial processing redundancies across NFs in a service chain because NFs are typically developed independently. In this paper, we propose SpeedyBox, a low latency NFV framework to eliminate the redundancy by consolidating functionality in service chains, without sacrificing modularization. Evaluation shows that SpeedyBox can significantly reduce processing latency in real world service chains.

X. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and advice. This work is supported by National Key R&D Program of China under Grant 2017YFB1010002, NSFC (No. 61872211), NSFC (No. 61802225), and the US NSF Grant CRI-1823270.

REFERENCES

[1] Anonymous repository of speedybox. https://github.com/FastPathNFV.

[2] Bess. https://github.com/NetSys/bess.

[3] Click ipfilter. http://read.cs.ucla.edu/click/elements/ipfilter.

[4] Dpdk packet generator. http://dpdk.org/browse/apps/pktgen-dpdk/.

[5] A low-latency nfv infrastructure for performance-critical applications. https://software.intel.com/en-us/articles/low-latency-nfv-infrastructure-for-performance-critical-applications.

[6] Mazunat. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[7] Microbenchmark results of speedybox. https://github.com/FastPathNFV/Microbenchmark.

[8] Qos challenges in the nfv/5g networks. http://www.mycom-osi.com/blog/qos-challenges-in-the-nfv-5g-networks.

[9] Snort. https://www.snort.org/.

[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[11] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[12] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 511–524. ACM, 2016.

[13] 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)*, pages 523–535, Santa Clara, CA, 2016. USENIX Association.

[14] D. Firestone. Vfp: A virtual switch platform for host sdn in the public cloud. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, pages 315–328. USENIX Association, 2017.

[15] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 7–12. ACM, 2012.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155*, 2015.

[18] D. Hong, J. Shin, S. Woo, and S. Moon. Considerations on deploying high-performance container-based NFV. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking, CAN@CoNEXT 2017, Incheon, Republic of Korea, December 12, 2017*, pages 1–6, 2017.

[19] J. Hwang, K. Ramakrishnan, and T. Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 445–458. USENIX Association, 2014.

[20] G. P. Katsikas, M. Enguehard, M. Kuzniar, G. Q. Maguire Jr, and D. Kostic. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2016.

[21] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, page 22. ACM, 2015.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[23] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 71–84. ACM, 2017.

[24] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma. Service Function Chaining Use Cases In Data Centers. Internet-Draft draft-ietf-sfc-dc-use-cases-06, Internet Engineering Task Force, Feb. 2017. Work in Progress.

[25] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 1–14. ACM, 2016.

[26] W. S. LIU, H. Li, and O. Huang. Service Chaining Use Cases. Internet-Draft draft-liu-service-chaining-use-cases-00, Internet Engineering Task Force. Work in Progress.

[27] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[29] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[30] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *OSDI*, pages 203–216, 2016.

[31] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[32] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM computer communication review*, 43(4):27–38, 2013.

[33] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, volume 13, pages 227–240, 2013.

[34] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[35] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 24–24. USENIX Association, 2012.

[36] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 227–240. ACM, 2015.

[37] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[38] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 43–56. ACM, 2017.

[39] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[40] W. Wu, Y. Zhang, and S. Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 29–35. ACM, 2016.

[41] X. Yi, J. Duan, and C. Wu. Gpunfv: a gpu-accelerated nfv system. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 85–91. ACM, 2017.

[42] J. Yonan. Openvpn, 2007.

[43] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 3–17. ACM, 2016.

[44] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood. Performance management challenges for virtual network functions. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 20–23. IEEE, 2016.

[45] W. Zhang, G. Liu, A. Mohammadkhan, J. Hwang, K. Ramakrishnan, and T. Wood. Sdnfv: flexible and dynamic software defined control of an application-and flow-aware data plane. In *Proceedings of the 17th International Middleware Conference*, page 2. ACM, 2016.

[46] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood. Opennetvm: a platform for high performance network service chains. In *Proceedings of the 2016 workshop*

on Hot topics in Middleboxes and Network Function Virtualization, pages 26–31. ACM, 2016.

[47] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, pages 143–149. ACM, 2017.