A Closer Look at NFV Execution Models

Peng Zheng*
Xi'an Jiaotong University and
University of Minnesota
pzheng@umn.edu

Arvind Narayanan
University of Minnesota
arvind@cs.umn.edu

Zhi-Li Zhang
University of Minnesota
zhzhang@cs.umn.edu

ABSTRACT

Network Function Virtualization (NFV) advocates running service function chains (SFCs) on commodity servers as software, thereby providing a new level of flexibility to the deployment and management of network services. However, as we move from 10/40 Gbps to 100/400 Gbps line rates, it is challenging to build an NF execution framework that can deliver high performance at the maximum line speed using commodity servers, while providing scalability and flexibility afforded by software. In this paper, we investigate a fundamental problem of any NFV framework, *i.e.* how to execute SFCs on commodity servers by examining and comparing the performance of two execution models: the *pipeline* and *run-to-completion* models. In particular, we investigate how the multi-core server architecture affects the performance of SFC execution models by conducting extensive experiments on a testbed and shed new insights on the design and optimization of SFC execution models.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Networks** → **Network performance analysis**; *Middle boxes / network appliances*; • **Hardware** → *Networking hardware*;

KEYWORDS

NFV, NUMA, multi-core, execution model, run-to-completion, pipeline, hybrid

ACM Reference Format:
Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. 2019. A Closer Look at NFV Execution Models. In *3rd Asia-Pacific Workshop on Networking 2019 (APNet '19), August 17–18, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343180.3343188

*The work was done while Peng Zheng was a visiting Ph.D student at the University of Minnesota.

APNet '19, August 17–18, 2019, Beijing, China

ACM ISBN 978-1-4503-7635-8/19/08…$15.00
https://doi.org/10.1145/3343180.3343188

1 INTRODUCTION

Thanks to its software nature, network function virtualization (NFV) offers a great level of (potential) flexibility to network operators to provision and manage network services by dynamically scaling in or out instances of network functions (NFs) on demand. However, as packet processing on commodity multi-core servers is far slower than ASICs used in dedicated hardware middleboxes, attaining (near) line speed is a major challenge in NFV design, especially when going from 10/40 Gbps to 100Gbps and beyond. This is further compounded by the fact that various NFs are often strung together to form a service function chain (SFC) to meet a network service objective [21]. It is a daunting task to develop a scalable NFV framework that can continue to attain the maximal system throughput and minimal SFC processing latency, as the number of NFs in a SFC increases while constituent NFs become more complex.

In this paper we start by examining the NUMA memory hierarchy common in today's commodity servers and argue that memory access is a key bottleneck in increasing NFV performance. Given the memory hierarchy and access latencies of a typical commodity server (see §2 for more details), we see that *per-core packet processing* speed is fundamentally limited by the number of memory accesses: given that at least two memory accesses are needed per NF (one for reading/writing the packet header and one for reading/writing the NF state), if these memory accesses touch the DRAM (100ns), only about 5 millions packets can be processed per second (per NF), yielding a throughput of merely around 2.6 Gbps when the packet size is 64B (bytes). This speed increases 6-fold to 30 Mpps (*million packets per second*) when the memory access touches only the L3 cache (16.5ns). To attain a (near) line speed of 100 Gbps, it is imperative for each core to ensure most of its memory accesses are L1/L2 cache bound (1.2 or 4.1ns), while employing more cores for parallel packet processing. Therefore optimizing the operations of each NF to minimize L1/L2 cache misses is crucial.

With this basic understanding, we then take a closer look at two NFV *execution models* in the existing state-of-the-art NFV frameworks which differ in how cores are used for SFC processing. The *run-to-completion* (RTC) model [12, 20] composes and executes all NFs in an SFC on a single core;

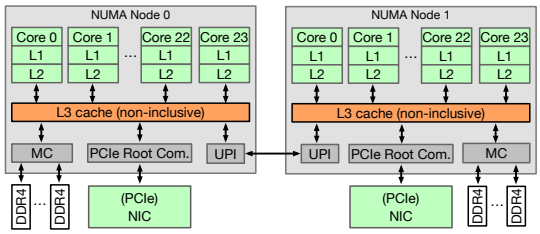


Figure 1: Architecture of muti-core server

whereas the *pipeline* (PL) model [9, 16, 19] runs one NF per core by processing packets across the cores in a pipeline fashion (see more details in §3.1). Clearly, the per-packet SFC processing latency of RTC increases with the number of NFs (and the number of memory accesses); it achieves high performance via scaling out, namely, running several instances of the same SFC on multiple cores. In comparison, PL staggers the per-packet processing latencies of different NFs in an SFC, but must pay the performance penalty of *inter-core transfer* latency: when switching from one core to another core, the packet being processed must be transferred via the shared L3 cache (worst case DRAM). In addition, due to more cores being used for processing each instance of an SFC, the number of cores available for scaling out is also reduced. Therefore, it is believed that RTC always outperforms PL. For example, it is shown in [20] that PL reduces the overall system throughput by as much as a third while increasing the SFC processing latency by up to 6 times.

Using the Intel VTune Amplifier tool [2], we conduct an in-depth performance analysis and benchmarking of NF/SFC packet processing. We compare the performance of the two execution models and investigate the factors affecting their performance. The story that emerges is far more complex. We find that with increasing packet sizes, the performance of PL catches up with that of RTC: both attain 100 Gbps line speed when the packet size reaches 1024B. The performance of RTC is limited by L1/L2 cache sizes: with more NFs in an SFC and more "complex" NFs involved (in terms of NF *state* size and the # of instructions), packing the entire SFC and the NF states in the L1/L2 (*d*- and *i*-)caches is no longer feasible; the performance of RTC thereby degrades quickly. Under RTC, scaling out is done for an entire SFC, the overall system throughput is not only constrained by the worst-performing NF in an SFC instance, but also hinges on our ability to properly load-balance the traffic among the SFC instances so as to keep all cores equally busy. This is however not an easy task, as it requires knowledge of both the types and scopes of the NF states in the chain. In contrast, PL offers more flexibility in optimizing and scaling out individual NFs in an SFC, e.g., one can provision different numbers of NF instances in an SFC to achieve better traffic load balancing while minimizing performance bottleneck.

Table 1: Memory hierarchy of a multi-core server

Type of Memory	Size	Type	Access Scope	# of Clock cycles	~ Access Delay (in ns)
L1 Cache	32KB	per core	dedicated	4	~1.2
L2 Cache	1MB	per core	dedicated	14	~4.1
L3 Cache	33MB	per socket	shared	44-70	13-20
Non-local L3	33MB	per socket	shared	100-150	29-44
Local DRAM	192GB	per socket	shared	250	70
Non-local DRAM	192GB	per socket	shared	420	125

Our results show that refactoring and decomposing NFs into modular functions with instructions and data localities that can fit into L1/L2 caches is important. We also highlight the need for intelligent NF state management. Our work calls for software/hardware co-design of an NFV runtime system that can optimize the NF/SFC performance at the compile time, appropriately provision the needed resources, and dynamically scale in/out NF & SFC instances on-demand.

2 NFV EXECUTION TARGET

Multi-core Server Architecture. A typical commodity multi-core server architecture is shown in Figure 1. We take our testbed server as an example to introduce the architecture in detail. Our testbed consists of two high performance servers, each equipped with a dual-socket (24-core/socket) Intel(R) Xeon(R) Platinum 8168 CPU @2.7GHz (thus, a total of 48 physical cores per server) clocked at 3.4GHz. Each CPU-socket (i.e. a NUMA node) has its own set of CPU caches, DRAMs and PCIe slots. Each physical core has dedicated L1 and L2 caches of sizes 32KB and 1MB, respectively (for L1, one half is used as the *d*-cache, the other half for the *i*-cache). There is also a non-inclusive L3 cache that has a total capacity of 33MB for each socket, shared between all its 24 cores and non-local cores (with NUMA penalty). Cache line size for all three cache levels is 64 bytes. In modern day servers, different NUMA nodes are connected to each other using high-speed, low-latency, point-to-point system buses (e.g. Intel UPI - Ultra Path Interconnect). In each of our server, there are 3 such high-speed links that connect both the sockets with bus speed of 10.4 GT/s. A CPU can access data faster from components that are within its socket. There is a latency penalty when data objects travel across NUMA sockets which we refer to as *NUMA penalty* (due to packets arriving on a NIC connected to PCIe on NUMA node 0 being processed by a CPU core connected to NUMA node 1). We now use our server to dive deeper into the memory hierarchy and show why it matters to the performance and scalability of NF.

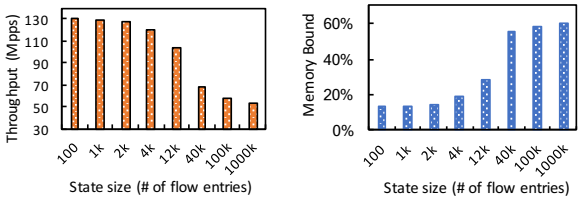
Memory Hierarchy and Access Latencies. The Intel architecture optimization reference manual [5] gives memory access latencies in *clock cycles* for caches specific to our servers. We further verify them and measure the DRAM access latencies using Intel's VTune Amplifier tool and present

the *empirically* observed results in Table 1. Getting *accurate* CPU wait times for data (*e.g.*, packets for NF processing) access is not trivial, and we rely on VTune to fill this gap. Intel claims that the metrics provided by VTune are *"precise"* when analyzed on Intel architectures, as it is able to tap into the exact instruction addresses [3]. We harness this tool to carefully and precisely investigate how NF-operator written software/code impacts CPU wait times caused by memory-bound data accesses. We assume this data to be present in either a CPU core's local L1 or L2 cache, the (per-socket) shared L3/LLC cache, or the local/remote DRAM.

We know that CPU wait times are significantly affected by two factors: i) location of the data in the memory hierarchy; and, ii) whether the accessed data was updated by some other core. In this paper, we focus on and dig deeper into how these two factors affect CPU wait-times, and what NFV frameworks can do to mitigate DRAM-bound latencies to achieve high-performance. It is known that the fastest time to access data by a CPU core is when the most updated copy is cached in its L1 memory – and for our servers, we find this time is merely 4 clock cycles (~1.2 ns). If the data is not in the L1 cache, the next fastest time is when the most updated copy is in the requesting core's L2 cache (~4.1ns). Accessing data from the L3 cache makes a CPU core wait between 44 to 70 clock cycles – due to the cache coherency issues and low level cache architectures [7]. There is a heavy penalty when data is accessed from the DRAM – 250 clock cycles when accessed from the local DRAM and 420 clock cycles from a non-local DRAM due to NUMA penalties.

Testbed and NF Implementation. Our testbed consists of two aforementioned Intel servers, each equipped with a (DPDK-capable) dual-port Mellanox ConnectX-5 EN 100Gbps NIC. We connect both the servers back-to-back, *i.e.*, `port 0` of server$_{tg}$ is connected to `port 0` of server$_{sfc}$, similarly, `port 1` of server$_{tg}$ is connected to `port 1` of server$_{sfc}$. server$_{tg}$ acts as a traffic generator. We use *TRex*[1] as it is one of the few software-based traffic generators capable of generating 100Gbps traffic[1].

To illustrate the impact of multi-core NUMA architecture on NFV performance, we have implemented several simple yet highly optimized examples NFs[2] using DPDK libraries 18.11. These include: i) An access control (`ACL`) NF which performs a 5-tuple lookup using a set of configured rules to decide whether to block a flow; ii) a simple (*stateful*) layer-4 load balancer (`LB`) which assigns a flow id to each flow using a hash function performed on the packet's 5-tuple header fields, looks up a flow table using the flow id to map each flow to one of the several destination servers by rewriting the `dst_ip` and `dst_port` fields of the packet accordingly – note that the arrival of the first packet of a new flow creates a new entry in the flow table; and iii) a layer 3 forwarder (`L3FWD`) which performs exact matching by looking up a routing table using a hash of the destination IP address to find the output interface. For fast packet processing, we perform *zero-packet copy* and *zero-memory allocation* on the packet processing (fast) data path. To avoid memory allocation, we reserve large pools of memory (using hugepages) at the initialization and let our custom runtime manage the memory. We allocate 256 hugepages of 1GB size for the packet processing runtime system. To offload packet processing and steering logic to hardware/NIC, we use DPDK's `rte_flow` library. Depending upon the experimental settings, server$_{sfc}$ is used to run multiple instances of our NF/SFC. The server OS is Ubuntu 18.04.2. All the cores used by NFs are isolated from the kernel scheduler. Hyper-threading is disabled.



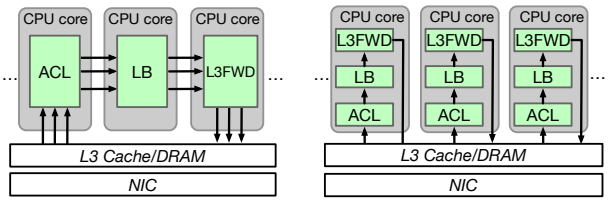
(a) Performance. **(b) DRAM bound**

Figure 2: NF LB performance vs. memory accesses

NF Packet Processing and Performance. To ensure fast NF packet processing, several optimization techniques are well-known. One is to design data structures that are cache-line friendly. Another is software prefetching which provides hints to CPU for prefetching the data into one of its hardware caches (especially L1/L2 caches) for faster data access. Packet batch/burst processing is yet another technique which allows the same set of read/write operations to be performed on a burst of packets by taking advantage of d/i-cache localities. DPDK libraries use these techniques extensively to avoid DRAM bound data accesses. For example, by default, DPDK uses a packet burst size of 32. Due to the space limitation, we will use `LB` NF only to illustrate why the multi-core NUMA architecture matters to NF packet processing performance, as LB involves *stateful* read/write memory accesses. In particular, we illustrate how the size of the NF state (as measured in terms of the # of flow entries in the flow table) affects the performance of `LB`. As shown in Figure 2b and 2a, LB performance degrades as the number of flow entries increases.

[1]TRex can generate traffic at the 100Gbps line rate with fixed frame sizes of 64, 128, 512 or larger bytes using 22 cores. However, it can only generate ~94Gbps at most when the frame size is set to 256 bytes due to a known TRex issue. For compatibility between the NIC and TRex, the traffic generator is installed with CentOS 7.4 distribution.

[2]While the NFs we implemented have limited functionality (compared to "commercial" NFs), they allow us to focus on the the basic NF operational logic, the state maintained and the induced NUMA memory access patterns when varying the packet and state sizes. They serve our goal to illustrate how the multi-core NUMA server architecture impacts the performance of NFs and NFV execution models.



(a) Pipeline execution model (b) RTC execution model

Figure 4: Two typical execution models

For instance, the DRAM-bound memory access remains less than 20% when the number of flow entries is less than 4K, allowing LB to process more than ~120 Mpps, achieving a throughput of ~81Gbps (the maximal achievable effective throughput with 64 bytes frame size after discounting the inter frame gap overheads is ~62Gbps). However, when flow entries reach more than 40K, packet processing rate drops to 69 Mpps or less. This is mainly because more than 55% of the data accesses are DRAM-bound.

3 EVALUATING EXECUTION MODELS

A typical SFC. Figure 3 depicts a simple SFC consisting of 3 network functions discussed before (§2). For simplicity, we consider this SFC to be a sequential chain of NFs, *i.e.*, incoming traffic is first processed by ACL, then LB and finally by L3FWD.[3]

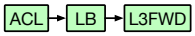


Figure 3: SFC of 3 NFs connected sequentially

3.1 Existing Execution Models

The following two execution models for SFC processing on a single multi-core server[4] have been used in the literature.

Pipeline (PL). This model assigns one core per NF in an SFC, with traffic steered to each NF in the chain one by one. Figure 4 (a) illustrates how the 3-NF SFC is executed under the pipeline model. NFV frameworks such as NetVM [9], ClickOS [16] and E2 [19] employ this model. Under the PL model, packets processed by different cores must be transferred via the L3 cache, thus incurring the *inter-core transfer* performance penalty [20].

Run-to-Completion (RTC). Under RTC, all NFs in an SFC are executed in a single core, thereby avoiding transferring packets across multiple cores. Figure 4 (b) depicts the same 3-NF SFC deployed using RTC as the execution model. NFV frameworks using this model include NetBricks [20] and Metron [12]. We note that accesses to the L3 cache or DRAM may be necessary if the NF instructions and data (state and packets) exceed the L1/L2 cache size.

Given a set of execution target servers and traffic workload characteristics (often dynamic in nature), a natural question arises: which execution model is capable of attaining the maximum line speed in SFC packet processing – so as to achieve the server-level objectives (SLOs)? In other words, *which execution model is better in providing high performance, scalability and flexibility for the given SFCs?* To answer this question, we provide a comprehensive in-depth analysis of two existing SFC execution models.

[3]While an SFC is more generally represented as a directed-acyclic graph (DAG), we focus on a simple sequential SFC. However, our discussions can easily be extended and applied to NFV frameworks that employ parallelism in SFC processing.

[4]Execution models can also be expand to include multiple servers. In this short paper, we however focus on a single server. Nonetheless, our proposed approach can easily be extended across multiple servers.

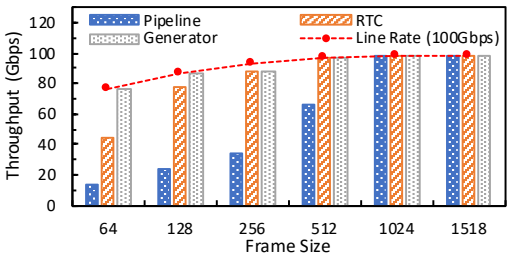


Figure 5: Throughput of RTC and Pipeline

3.2 Performance Evaluation

We primarily use the simple SFC in Figure 3 as an example to analyze the performance of the PL and RTC models. Specifically, we quantify the packet inter-core transfer overheads by measuring the *wait time* of a CPU core when fetching instructions and data. This wait time is measured using the number of CPU clock cycles. In the following, we run the example SFC shown in Figure 3 using 18 cores for both execution models. Under the PL model, 18 cores are used to run 6 SFC instances, whereas under the RTC model, 18 SFC instances are executed, one per core. We generate 90 flows using *TRex*[1] at the maximal line speed, and the flows are steered evenly to each SFC instance. We compare the performance of PL versus RTC under varying frame/packet sizes – from 64 bytes to 1518 bytes.

Throughput. Figure 5 shows the throughput of both RTC and PL models. The bar with the label *'Generator'* in Figure 5 corresponds to the throughput of the traffic generator, giving the physical limit achievable. We summarize our key observations: i) For frame sizes between 64 and 512 bytes, the RTC model is 1.4-3.3× faster than the PL model. However, as we further increase the frame size, the throughput of PL gets closer to that of RTC. ii) When the frame size reaches 1024 bytes or larger, both RTC and PL reach the 100Gbps line rate for our three-NF SFC. iii) Note that all the throughputs in Figure 5 are the data rate, *e.g.*, the maximum throughput of 64 byte-sized frames is 76.0 Gbps with the corresponding raw bit rate of 99.8 Gbps due to the 20 bytes inter-frame gap of the Ethernet frame overhead. RTC can achieve the maximum throughput (thus the line-rate packet processing) for frame sizes larger than 256 bytes.

Table 2: NFs processing cycles of two models

NFs	ACL	LB	L3FWD
Pipeline	35.1	75.8	59.2
RTC	10.2	8.3	7.5

Latency. As shown earlier, PL has lower throughput compared to RTC. This is mainly due to the inter-core transfer overheads. To further examine such overheads, we empirically analyze the per-NF process latency under each execution model for the three NFs (ACL, LB & L3FWD) in the service function chain. We use VTune [2] to measure the number of clock cycles CPU takes to process a function. This metric implicitly accounts for the time CPU waits for data to be fetched from the memory (*i.e.*, L1/L2/L3/DRAM). The results are shown in Table 2. We see that the average processing latencies of RTC for ACL, LB, L3FWD are 10.2, 8.3, 7.5 cycles, *resp.* While the processing latencies of the same three NFs using PL increase by 3-9×. We remark that RTC latencies are less than 10.2 cycles because data is available in the L1/L2 cache.

The key observation, revealed by the Table 2, is that the inter-core transfer latency is about 52-68 cycles which is close to clock cycles required to access data from L3 cache (44-70 cycles as shown in Table 1). Note that the clock cycles of ACL-processing in both RTC and PL models are less than the L3 cache access latency; however, the processing cycles of both LB and L3FWD are larger than minimal L3 cache latency. Specifically, the inter-core transfer from ACL to LB is ~68 cycles (we can approximately calculate it $75.8 - 8.3$), and ~52 cycles from LB to L3FWD ($59.2 - 7.5$). LB requires 16 more cycles on average than L3FWD.

3.3 Inter-core Transfer Overhead

We summarize the inter-core transfer overheads by empirically evaluating them under different experiment settings. Table 3 shows various inter-core transfers overheads, depending on where the most updated data object being accessed is present. From the table we can find that the inter-core transfer is bound by the hierarchy of the memory system. The fastest inter-core transfer between two cores happens when both lie in the same NUMA node. The inter-core transfer overhead is at least one L3 cache access (l_{L3}), which is ~44 cycles for *clean* data; otherwise it is 70 clock cycles if the data transferred is modified by the first core. For transfers between two cores on different NUMA nodes, an extra NUMA penalty (due to UPI latency as discussed earlier) is added, incurring 100-150 extra clock cycles. However, if the data is not cached in the LLC/L3 cache, then the inter-core transfer is cloned through the DRAM, the access-latency is about 250 cycles for the local DRAM (l_{DRAM}) and 420 for non-local

Table 3: Minimal inter-core transfer overhead

Data source	Latency
Local L3 - clean	l_{L3}
Local L3 - dirty	l_{L3} + cache coherency penalty
Remote L3	l_{L3} + (cache coherency penalty) + *NUMA* penalty
Local DRAM	l_{DRAM}
Remote DRAM	l_{DRAM} + *NUMA* penalty

memory due to NUMA penalties. Thus, the multi-core memory hierarchy has significant impact on performance of SFC execution models.

4 DOES RTC ALWAYS WIN?

Despite the performance advantages of the RTC execution model, we find that RTC suffers several shortcomings. In this section, we perform experiments to empirically demonstrate these shortcomings.

4.1 NF States

Due to the complexity of real world SFCs, the size of the *state* maintained by NFs in a SFC can be large. The length of the SFC can also grow longer. For example, an LB NF may maintain millions of flow mapping entries between the public IP and private IP address; while another NF such as L3FWD in the same chain may also need to perform an exact hash table lookup with millions of entries. On the other hand, the memory (L1/L2 cache) resources for each core is limited. If one or a subset of NFs require more resources than the local cache sizes, RTC cannot scale up to support a large number of flows with high performance.

To demonstrate this problem, we measure the throughput of RTC under different flow table sizes. We configure the number of state entries of LB and L3FWD in our SFC from 100 to 100K, and then send the same number of flows – randomly generated – to the SFC instances. We use 18 cores and test with the frame sizes of 64 and 128 bytes. The results shown in Figure 6 suggest that when the state size (as measured by the number of flow entries) of LB and L3FWD grow from 1K to 100K, the throughput drops by 2.2-2.3×. Further analysis shows that the performance degradation is mainly caused by cache misses in the flow or hash table look-ups. Recall that each CPU core of our server contains about 1MB dedicated L1 and L2 caches. If a core suffers a L2 cache miss, it has to fetch the data from the LLC/L3-cache or DRAM, which is similar to incurring an inter-core transfer penalty (§3.2).

4.2 Cache Locality

With increasingly complex NF behavior and diverse access patterns of NF states, coupled with a large number of NFs in a SFC, the RTC model can suffer poor cache locality (both *i*-cache and *d*-cache). To demonstrate the *d*-cache locality problem, we measure the percentage of *DRAM-bound* memory accesses under RTC for varying NF state sizes. Figure 7 shows that the DRAM-bound data accesses of RTC increase

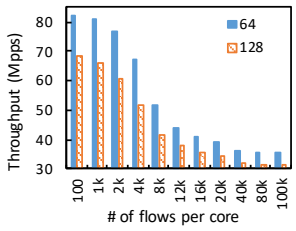


Figure 6: RTC throughput with NFs states size

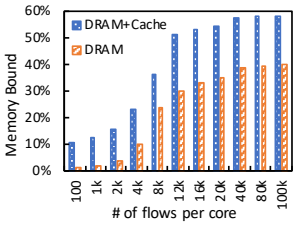


Figure 7: Memory bound of RTC execution model

significantly as the flow table size grows. The memory bound % measures the fraction of clock cycles (or time) when a CPU core is stalled due to load or store instructions: the bar labeled "DRAM" represents the amount of time CPU is stalled due to load/store instructions bound to the DRAM, while the bar labeled "DRAM+Cache" represents the time CPU is stalled due to the same instructions bound to the DRAM or caches. When all the NF states and instructions can fit into the L1/L2 caches, RTC performs very well. However, when the total state size in an SFC grows significantly larger than the cache sizes, RTC suffers L1/L2 cache misses, resulting in significant performance degradation. Hence RTC does not scale well as the length or complexity of SFC grows.

4.3 Flexibility

The RTC model also falls short in terms of providing flexibility in SFC scaling: the entire SFC must be replicated, whereas under the PL model, individual NFs can be replicated and scaled out/in. The RTC model must allocate resources for all the NFs even though only one NF in the chain is the bottleneck. In contrast, under the PL model, one can scale out only the worst performing NFs in the chain. This leads to cost-savings and better efficiency in resource utilization. Lastly, the RTC model cannot exploit the benefits of NF level parallelism [22, 25], which can significantly reduce the overall SFC processing latency. In contrast, NF level parallelism can be easily supported by the PL model.

5 RELATED WORKS

NFV execution model and NUMA architecture. Existing NFV systems either deploy RTC [12, 20] or PL [9, 16, 19] model. None of them provide in-depth analysis and comparison of these execution models under the NUMA architecture. Routebricks [6] is perhaps the first to design software routers on multi-core servers, comparing the pipeline versus parallel ("RTC") execution models for packet processing. But the routing NFs under consideration are stateless only.

NFV/SFC Optimization. Various approaches have been developed for improving NFV/SFC performance, *e.g.*, parallelizing NF chains [22, 25], reducing redundant operations [4, 15], improving CPU scheduling efficiency [14, 20, 23], providing faster I/O [8, 9, 16], profiling and bottleneck detection of NFs [10, 17, 18] and offloading certain NF operations to NICs/switches [12]. These studies complement our work, which focus on NFV execution models.

SFC Scalability. NFV state management is key to scalability and fault tolerance of SFC which has been widely explored [11, 13, 24]. However, the impact of NUMA memory architecture and cache resources on SFC scaling is seldomly discussed. Our work sheds new insights on further directions in NFV/SFC state management.

6 ON-GOING WORK & CONCLUSION

Our initial evaluation results show that both PL and RTC models have their own pros and cons when it comes to performance, scalability and flexibility. To the best of our knowledge, no existing work has systematically examined the SFC execution models under multi-core server target execution environments, in particular how the NUMA memory hierarchy affects their performance in depth. Our work also suggests why a NFV framework with a *hybrid* execution model can potentially be advantageous. Such a model can combine the respective strengths of RTC and PL, while mitigating their shortcomings. We outline several challenges and on-going directions toward the envisioned hybrid model we are pursuing.

- *How to profile NFs automatically.* Understanding the behavior of each NF in a SFC is essential to choose the best execution model. To do this, we can either use a precise profiling tool such as Intel vTune [2] to characterize the NF performance on specific hardware architectures, or perform static program analysis, *e.g.*, symbolic execution [10], to generate NF performance behavior profiles.
- *How to scale NFs elastically.* The key is to manage the NF state in a scalable manner, *i.e.*, how to refactor a large NF state to smaller components that fit into different CPU caches, perform NF *state-aware* traffic partition to leverage traffic parallelism, and steer the traffic appropriately to be executed by each SFC instance so as to reduce CPU wait times and increase its utilization.
- *How to schedule NF execution dynamically.* We need a SFC runtime system to schedule NFs dynamically to improve the efficiency and meet SLOs of various SFCs. Toward this end, we are building a new NFV framework that supports rapid NF development with ease-of-orchestration in a multi-core server architecture. This can be further extended to multi-server/cluster environments.

ACKNOWLEDGMENTS

The research is supported in part by US NSF under Grants CNS-1411636, CNS-1618339, CNS-1617729, CNS-1814322, and CNS-1836772. Peng Zheng gratefully acknowledges financial support from National Key R&D Program of China (2017YFB0801703) and China Scholarship Council.

REFERENCES

[1] 2019. Cisco T-Rex: Realistic traffic generator. (2019). https://trex-tgn.cisco.com

[2] 2019. Intel VTune Amplifier. https://software.intel.com/en-us/vtune. (2019).

[3] 2019. Memory Access Analysis for Cache Misses and High Bandwidth Issues (Intel VTune Amplifier 2019 User Guide). https://software.intel.com/en-us/vtune-amplifier-help-memory-access-analysis. (2019).

[4] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/2934872.2934875

[5] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Optimization Reference Manual. (2019). https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[6] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/1629575.1629578

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

[8] Massimo Gallo and Rafael Laufer. 2018. ClickNF: a Modular Stack for Custom Network Functions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 745–757. https://www.usenix.org/conference/atc18/presentation/gallo

[9] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[10] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 517–530. https://www.usenix.org/conference/nsdi19/presentation/iyer

[11] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

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

[13] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi19/presentation/khalid

[14] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/3098822.3098828

[15] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. 2018. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 504–517. https://doi.org/10.1145/3230543.3230563

[16] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[17] P. Naik, D. K. Shaw, and M. Vutukuru. 2016. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 154–160. https://doi.org/10.1109/NFV-SDN.2016.7919491

[18] Jaehyun Nam, Junsik Seo, and Seungwon Shin. 2018. Probius: Automated Approach for VNF and Service Chain Analysis in Software-Defined NFV. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/3185467.3185495

[19] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[20] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[21] Carlos Pignataro and Joel Halpern. 2015. Service function chaining (SFC) architecture, RFC 7665. (2015).

[22] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/3098822.3098826

[23] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 283–297. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

[24] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312. https://www.usenix.org/conference/nsdi18/presentation/woo

[25] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. 2017. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 143–149. https://doi.org/10.1145/3050220.3050236