Towards Efficient Server Architecture for Virtualized Network Function Deployment: Implications and Implementations

Yang Hu and Tao Li
IDEAL Lab, University of Florida
Gainesville, FL, USA
huyang.ece@ufl.edu, taoli@ece.ufl.edu

***Abstract*—Recent years have seen a revolution in network infrastructure brought on by the ever-increasing demands for data volume. One promising proposal to emerge from this revolution is Network Functions Virtualization (NFV), which has been widely adopted by service and cloud providers. The essence of NFV is to run network functions as virtualized workloads on commodity Standard High Volume Servers (SHVS), which is the industry standard.**

However, our experience using NFV when deployed on modern NUMA-based SHVS paints a frustrating picture. Due to the complexity in the NFV data plane and its service function chain feature, modern NFV deployment on SHVS exhibits a unique processing pattern—heterogeneous software pipeline (HSP), in which the NFV traffic flows must be processed by heterogeneous software components sequentially from the NIC to the end receiver. Since the end-to-end performance of flows is cooperatively determined by the performance of each processing stage, the resource allocation/mapping scheme in NUMA-based SHVS must consider a thread-dependence scheduling to tradeoff the impact of co-located contention and remote packet transmission.

In this paper, we develop a thread scheduling mechanism that collaboratively places threads of HSP to minimize the end-to-end performance slowdown for NFV traffic flow. It employs a dynamic programming-based method to search for the optimal thread mapping with negligible overhead. To serve this mechanism, we also develop a performance slowdown estimation model to accurately estimate the performance slowdown at each stage of HSP. We implement our collaborative thread scheduling mechanism on a real system and evaluate it using real workloads. On average, our algorithm outperforms state-of-the-art NUMA-aware and contention-aware scheduling policies by at least 7% on CPU utilization and 23% on traffic throughput with negligible computational overhead (less than 1 second).

Keywords—NFV; Networking; thread scheduling; NUMA

I. INTRODUCTION

Today's service providers need greater performance, flexibility, and adaptability from the network services that support them. Gartner forecasts that the number of devices connected to the Internet of Things (IoT) will reach 26 billion by 2020, and impose an unprecedented challenge to data transmission services and data center network infrastructures [1]. To meet the rapidly increasing volume of traffic and deliver both capital (Capex) and operational (Opex) expenditure advantages, thirteen of the world's largest service providers (AT&T, Verizon, China Mobile, CenturyLink, etc.) propose Network Functions Virtualization (NFV) [2]. NFV allows data center networking functions such as load balancing, firewalls, and switching to be implemented as software or virtual machine-based Virtualized Network Functions (VNF). The VNFs are consolidated on Standard High Volume Servers (SHVS) with software switching instead of fixed-function specialized hardware. By doing so, NFV creates highly flexible and adaptable network resources that can be deployed quickly to respond to changing demands at lower cost. To date, NFV has gained over 220 industry participants including the European Telecommunications Standards Institute (ETSI) [3], the Linux Foundation OPNFV [5], and Oracle [6]. According to a recent study, the global NFV market is expected to grow 52% from 2013-2018 [7].

Because SHVS are expected to continue to serve as the backbone for network infrastructure, their performance when running VNFs must be considered. Intel has proposed an initial x86-based reference server architecture, Intel Open Network Platform [8], to enable NFV deployment. However, we observe current SHVS architecture support for NFV deployment falls short on generality and flexibility, and is not fully prepared for NFV. For example, although existing hardware-based high performance I/O technologies such as Single Root I/O Virtualization (SR-IOV) [9] and Data Direct I/O (DDIO) [10] can achieve line rate VM-to-network throughput by bypassing the hypervisor layer, they do not support overlay-based network virtualization for multi-tenant and VM migration, making them less flexible in modern SDN/NFV deployment. More importantly, VM-to-VM traffic, which is dominant in NFV enabled environments, must traverse the PCI Express bus in SR-IOV and DDIO, leading to throughput that is inferior to the throughput in a software switch.

In this study, we characterize the architectural overhead of SHVS using real Telco and cloud NFV workloads. We observe that NFV deployment presents more performance demanding and complex processing patterns than typical IT workloads. In a NFV environment, a packet flow needs to traverse the end-to-end data path, namely the NFV data plane, which includes a variety of software components that reside within VNFs (e.g. virtual NICs and packet processing routines within VNFs) and hypervisor virtualization stacks (e.g. physical NICs, hypervisor I/O handler, virtual switch threads). A VNF may also process traffic flows in tandem with other VNFs in service chains. We term this packet processing style as a Heterogeneous Software Pipeline (HSP).

The software pipeline [11] is a parallel application that consists of several communicating stages that process streams of input data in tandem. This processing style demands end-to-

978-1-5090-3508-3/16/$31.00 ©2016 IEEE
* Authors with equal contribution.

RIGHTSLINK

end performance guarantees (either throughput or latency) for network flows. This indicates that the performance of an HSP is cooperatively determined by the performance of each processing stage throughout the processing path. In other words, the end-to-end performance slowdown of an NFV flow is the aggregated slowdown at each pipeline stage.

Since all software components in the pipeline are deployed on the shared computing resources of SHVS, finding efficient and effective resource allocation/mapping schemes for these software components, or threads, is of the utmost importance. However, existing SHVS hardware resource allocation schemes and performance estimation models lack support for software pipeline-style applications. Specifically, when a packet is passed between software components or threads that belong to neighboring pipeline stages, the performance slowdown for this packet at this stage can be decomposed into the slowdown caused by resource interference and the slowdown caused by inter-thread communication.

Though current thread and core allocation methods [12-14] can manage the former slowdown, they overlook the performance slowdown caused by inter-stage data transfer overheads. Considering that Non-Uniform Memory Access (NUMA) [15] architectures are ubiquitously adopted in contemporary SHVS, the message pass/packet transfer between stages in HSP causes the thread to access data from the memory of its predecessor thread, while its predecessor's memory may reside in remote NUMA nodes. Therefore, the inter-thread communication overheads must be factored in the performance slowdown estimation model in accordance with interference based model to provide thread-dependence scheduling in each stage of the software pipeline.

Designing the aggregated performance slowdown estimation model for end-to-end data path in NFV raises several questions. How can one quantify the performance slowdown caused by interference and communication at each stage? Moreover, how can these be combined into a comprehensive slowdown model? Finally, how can one compare the performance slowdown in the presence of varying resource sensitivities at each stage? To address these issues, we present a new performance model for estimating the end-to-end performance slowdown of flows in software pipeline processing environments such as NFV. Our model assesses the intra-stage performance slowdown caused by hardware resource contention and inter-thread/core communication overheads. It also estimates the end-to-end performance slowdown by summing the weighted inter-stage slowdown at each stage.

Leveraging our holistic performance estimation model, we design vFlowComb, a dynamic thread mapping mechanism that enables thread-dependence mapping for NFV service chains. To achieve this goal, vFlowComb features a Collaborative Thread Scheduling (CTS) mechanism that guarantees to minimize the end-to-end performance slowdown for each NIC hardware queue. CTS exploits a novel Dynamic Programming-Based Mechanism (DPBM) to find the thread-core mapping with the minimum aggregate performance slowdown, while considerably reducing the performance sampling and decision-making overheads.

This paper makes the following contributions:

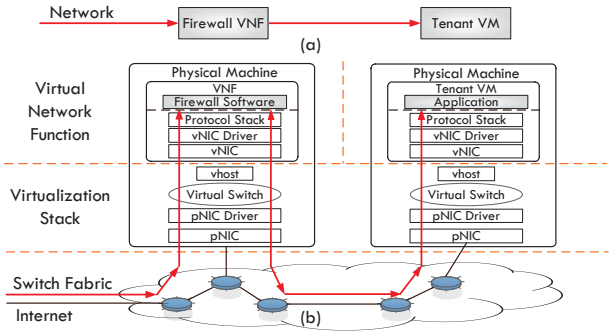

Figure 1. Heterogeneous software pipeline (HSP) in NFV data plane

- We explore the deployment of modern NFV workloads on current SHVS architectures. We observe that NFV adopts a heterogeneous software pipeline (HSP) processing style, which presents significant challenges for current thread mapping mechanisms and performance estimation tools.
- We explore the performance slowdown in the HSP on modern NUMA architectures. We propose a performance estimation model that evaluates the performance slowdown of each stage of HSP by considering hardware resource contention and inter-thread/core communication overheads.
- Based on our performance slowdown estimation model, we propose vFlowComb; a thread mapping mechanism that minimizes the end-to-end performance slowdown. We implement vFlowComb using Open vSwitch and OpenStack.

The rest of this paper is organized as follows. Section 2 gives a brief introduction of the NFV data plane and network I/O NUMA issue. Section 3 characterizes the NFV deployment on NUMA-based SHVS and proposes performance slowdown estimation model. Section 4 presents the collaborative thread scheduling mechanism for heterogeneous software pipelines. Section 5 evaluates our design. Section 6 discusses related work and Section 7 presents our conclusions.

II. Background and Motivation

A. Network Functions Virtualization

1) Control/Data Plane

In software-defined networks, the network environment can be split into three planes: the application plane, the control plane, and the data plane. Tenants interact with the application plane, requesting deployment of their virtual private networks. The control plane responds to these requests and instantiates virtual links between tenant VMs and VNFs using tunneling techniques [16] or encapsulation policies. The data plane instantiates configurations furnished by the control plane and provides a network traffic backbone for each tenant's virtual private network. The data plane consists of all tenant VMs and VNFs, and the virtual switches by which they are connected. All components are consolidated on a SHVS architecture and are allocated on shared computing resources. Figure 1(a) shows a simple tenant virtual network with a VM and a firewall. In this setup, all incoming traffic must pass through a firewall before entering the tenant VM. Each packet will traverse the

RIGHTSLINK

software components before being processed in the VNF and the VM.

2) Software Pipeline in Data Plane

Here, we describe the detailed processing patterns of the software pipeline in NFV data plane and illustrate a software pipeline implementation based on the Linux kernel with NAPI, a virtual machine, and an Open vSwitch-based software switch. Each software component receives packets from its predecessor, processes them based on its functionality, and sends them to the successor components, as shown in Figure 1(b).

When an incoming packet arrives at the input buffer of a physical NIC, it will be DMA'd to the kernel DMA RX-buffer, sk_buff, which is allocated in main memory. In multi-10G networks, this buffer allocation/de-allocation could significantly stress the memory subsystem (tens of millions of allocations per second). Once in the buffer, a hardware interrupt is triggered. An interrupt handler associated with one of the processor cores is called and schedules a softIRQ context to its local core or another CPU core. All CPU cores examine their poll queue using the poll method and process the queued softIRQ context. Modern NICs support multiple receive and transmit descriptor queues (multi-queue) technique. The NIC controller computes a hash value for each incoming packet. Based on these hash values, the NIC assigns packets of the same data flow to a single queue and distributes traffic flows evenly across queues. To maximize the network transmission performance in multi-core server systems, Receive-side Scaling (RSS) [17] and Receive Packet Steering (RPS) are used. RSS enables multiple NIC queues to have their own associated CPU core while RPS assigns a specific core for a softIRQ context. These core assignments should be carefully designed to avoid unbalanced CPU loads.

Virtual Switch: In virtual machines, the hypervisor provides intra-server networking connectivity for virtual machines. In this virtual network, the hypervisor creates one or more virtual NICs (vNICs) for each VM to connect to physical NICs (pNICs) of the host server and facilitates network connection between the VM network stack and hypervisor network stack through virtualized switches (e.g. Linux Net bridge and Open vSwitch) [18]. When using virtual switches, the intra-server network connection is no longer limited by network speed but memory bandwidth since no packet must pass through PCI-E links to special purpose hardware. This enables high-performance communication among VMs. More importantly, virtual switches enable cross-server bridging in a way that makes the underlying server architecture transparent. A virtual switch within one server can transparently join with another virtual switch in another server, simplifying VM migration.

B. NFV Workloads

In this paper, we use Clearwater [19] as our NFV plat-form. Clearwater is a cloud-based Telco-grade IP Multimedia Subsystem (IMS). IMS is widely adopted by large Telcos to provide IP-based voice, video, and messaging services based on soft-switching. Clearwater consists of a series of typical function components with various resource utilization patterns

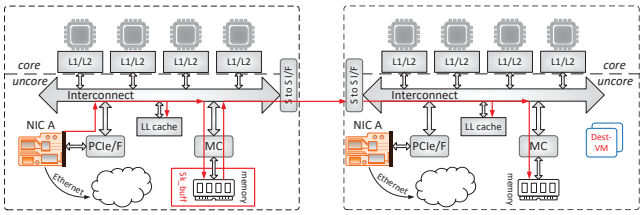

Figure 2. I/O NUMA. We interchangeably use NUMA node, socket and processor in this paper.

in a Telco data center, and could be easily deployed as VNFs in NFV environment.

1) Workloads Description

Bono is a scalable edge proxy in the NFV environment. It serves as a gateway and provides connections to the Clearwater system for clients. ***Sprout*** processes the incoming requests from Bono, acting as a registrar and authoritative routing proxy. The Sprout cluster includes a memcached cluster to store client registration data. ***Homestead*** provides web services interface to Sprout for retrieving authentication credentials and user profile information; providing a subscriber server and employs Cassandra as the backing store for its managed data.

2) Testing Methodology for Clearwater

We deploy Clearwater as virtual machines in our characterization and evaluation (the detailed configuration is described in Section 3.1). In this paper, we use SIPp [20] to generate real world Telco NFV traffic. It is a performance-testing tool for Telco infrastructure and can establish and release multiple calls to an NFV cluster. We choose user registration and deregistration (reg-dereg) calls for the traffic flow in this paper. A reg-dereg call consists of three requests: one for registration, one for authentication, and one for deregistration. SIPp initiates each call with an initiated call rate. If a response to a request times out (10s), the call will be tagged as failed. SIPp initiates call with an initiated call rate. Each round of experiment runs for 300s. We run 5 trials and take the average results. We use the Successful Call Rate (SCR), which is used as an indicator of the service quality of the NFV system. SCR gives the ratio of the successful call rate to the initiated call rate. The maximum SCR is 1.

C. Network I/O NUMA

With NUMA architectures, each socket (i.e. processor node) is associated with a local memory node through the memory controller. Multiple cores in one socket share the last level cache, memory controller, and PCI-e interface (e.g. NIC) through the intra-socket interconnect. Inter-socket communications are enabled through point-to-point high-speed interconnects (e.g. Intel's QPI). In a multi-socket server with NUMA enabled, the PCI devices are associated with a subgroup of NUMA nodes, as shown in Figure 2.

III. Characterizing NFV Workloads in SHVS

In this section, we characterize the performance of NFV flow to identify inefficiencies in current NUMA-based SHVS from the viewpoint of architecture level.

RIGHTSLINK

TABLE I. PLATFORM CONFIGURATIONS

Item	Value
SHVS system	IBM x3850 X5, 8-socket NUMA
Processor	Intel Xeon X7550, 2.0GHz (Nehalem) 8 physical cores (16 with Hyper-Threading)/socket 18MB L3 cache for each socket 64KB L1 cache and 256KB L2 cache for each core
Memory	64GB, DDR3 for each socket, 512GB in total
Interconnection	Intel QuickPath Interconnect, 6.4GT/s
NIC	Intel X520 10GB, Mellanox 40GB Associate with socket 0 and 4

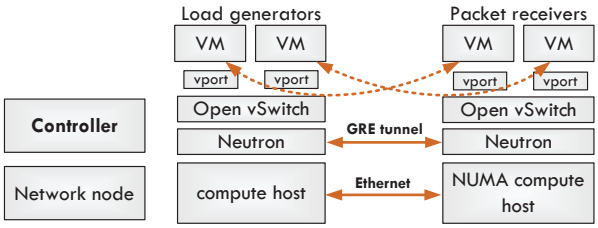

Figure 3. Framework setup

A. Characterization Setup

Hardware Platform: Our physical platform configuration is shown in Figure 3. The system uses four Intel X520 SPF+ 10 Gigabit Ethernet NICs divided into two groups and are associated with two NUMA nodes respectively. To clearly expose the bottlenecks, we configured the IBM x3850 system as 2 sockets and only use one NIC in the characterization.

Software Platform: We use the open source cloud platform OpenStack Kilo [21] to build a full-fledged cloud environment for NFV deployment. Our test cloud consists of three compute nodes, one cloud controller node, and one networking node. All compute nodes run RHEL 6.4. The network service Neutron helps tenants to build their own private software defined networks and Open vSwitch based virtual switches. All VNFs and tenant VMs are deployed as virtual machines with 2 vCPU and 4GB memory. The VMs are consolidated on NUMA-based SHVS. They communicate with each other using GRE [16]. The networking hierarchy is shown in Figure 3.

Workloads: Our real world NFV workloads are introduced in Section 2. To clearly identify the bottlenecks, we also use the network intensive micro-benchmark Netperf [22] to generate UDP STREAM and TCP STREAM as stable and controllable traffic loads. As shown in Figure 3, we deploy NFV workloads or simple networking workloads on VNFs/tenant VMs as packet receivers. We deploy client VMs on other machines as load generators.

B. Characterization of Heterogeneous Software Pipeline on NUMA based SHVS

We investigate the performance and architectural behaviors of current NUMA-based SHVS when executing heterogeneous software component pipeline in NFV deployment. We vary the thread-to-socket/core mapping and co-located contentions to examine the performance trade-offs in HSP. These results indicate that new performance modeling tools are needed.

1) Methodology

As we described in Section 2, the flow path in the NFV data plane can be seen as a packet traversing the software components. The software components in the NFV data plane are the `ksoftirq` kernel thread that handles the NAPI routine and the virtual switch routing process in which the packets are written to TAP's socket buffer, and the `vhost-net` thread that copies the packets from the socket buffer to the VM's vNIC buffer. In this characterization, we focus on the packet receiving process (i.e. incoming flow processing) since it contains all of the critical software components in the NFV data plane. We collect the architectural statistics using Intel's Performance Counter Monitor tools [23].

2) Impacts of Inter-socket Communications

We first vary the thread-to-socket mappings to investigate the impact of inter-socket communication (CPU-to-CPU and CPU-to-RAM) on the performance of HSP.

In this experiment, we study seven different thread-to-socket mappings with different socket affinities for the `ksoftirq` kernel thread, the `vhost-net` thread, and the VM thread. The different configurations, A-G, are shown in Figure 4(a). A local node consists of a NUMA node and a NIC. Different threads on the same socket are mapped onto different cores. We consider three network traffic loads. We use Netperf to generate 1400B UDP packets and 64KB TCP packets. We also use SIPp to generate a traffic flow at a rate of 300 calls/second. We report the *cache miss per packet* and *received packet throughput* (packet per second) in Figure 5(a). For the NFV loads, we report the *successful call rate* as defined in Section 2.

Observations: For the UDP flow we can observe approximately 30 LLC misses per packet in the `ksoftirq` in configuration G. This is caused by the inter-socket DMA transmission overhead since the incoming packets should be brought into LLC for `ksoftirq` processing on the remote NUMA node. In configuration E, we can observe there are around 5 LLC misses per packet in the `vhost-net` and VM threads, and nearly no LLC misses at `ksoftirq`. This is because `vhost-net` needs to access the vNIC of the target VM across the sockets. We can also observe the traffic throughput increases from 1.2Mpps to 1.45Mpps, while the collocated `ksoftirq`, `vhost-net`, and VM gain the high-est throughput at 1.71Mpps.

To examine a real NFV deployment scenario, we increase the VM consolidation (5 VMs) and traffic flow and re-run the tests. We present the results in Figure 5(b). We can observe the performance in configuration E experiences a severe drop. The LLC misses per packet for the VM and `vhost-net` threads increase to around 20 misses per packet and the traffic throughput drops from 1.4Mpps to 1.02Mpps. This is because intensive inter-socket communication occurs between the `vhost-net` thread and the vNIC buffers.

Finding 1: *In this experiment, we observe that the inter-socket communication overheads caused by asymmetry in NUMA-based SHVS significantly impact the performance of heterogeneous software pipeline workloads like NFV.*

RIGHTSLINK

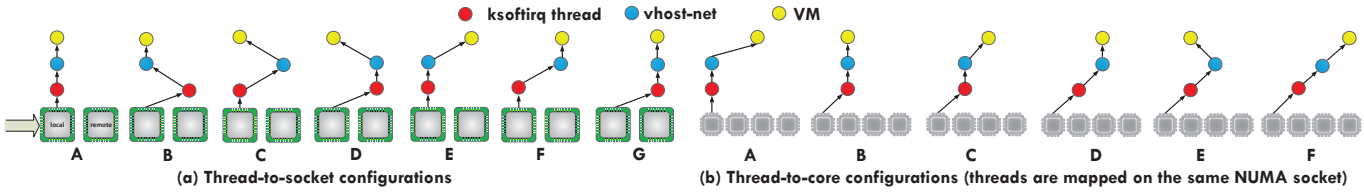

Figure 4. Thread-to-socket/core mapping configuration

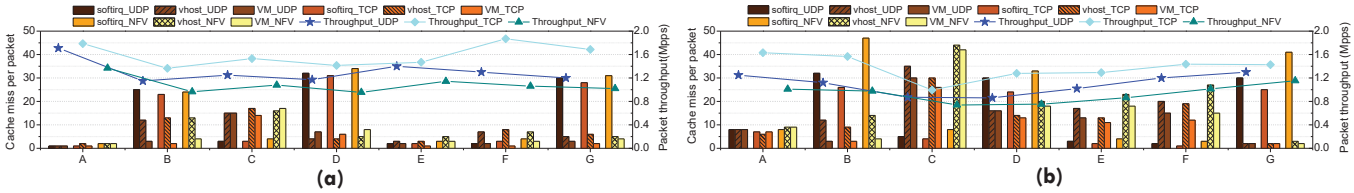

Figure 5. Performance and architectural behaviors of Inter-socket scenarios

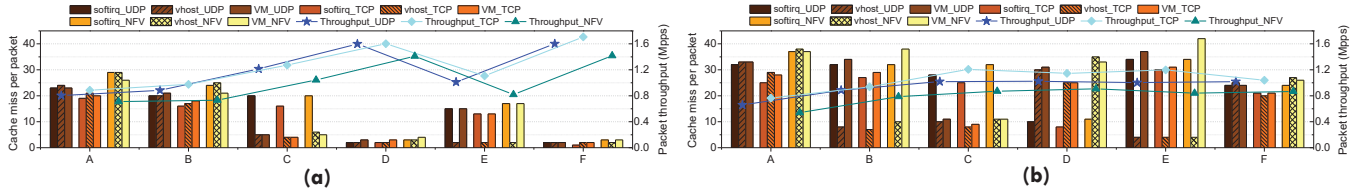

Figure 6. Performance and architectural behaviors of Intra-socket scenarios

Finding 2: *The thread heterogeneity of each stage in a software pipeline exhibits sensitivity to hardware resources and inter-socket communication overheads, while also being related to workload intensity.* For example, we can observe the inter-thread communication between `ksoftirq` and `vhost-net` is more sensitive to inter-socket access than inter-thread communication between the NIC driver and `ksoftirq` when more VMs are consolidated.

Finding 3. *The thread heterogeneity of each stage incurs heterogeneous performance slowdown at each stage. However, the low-performance slowdown at an earlier stage does not necessarily result in a low end-to-end performance slowdown for the whole pipeline.* In Figure 5(b), it is clear that configuration E exhibits lower slowdown at the NIC-`ksoftirq` stage than configuration G does, even though its end-to-end throughput is lower than G.

3) Impacts of Intra-socket Co-located Contentions

We further investigate the performance of a heterogeneous software pipeline in the presence of intra-socket co-located contention. We study the performance slowdown caused by the contention of co-located software components and other user workloads to derive the performance implications.

We first investigate the impact of thread-to-core mappings. We design six thread-to-core mappings as shown in Figure 4(b). In each mapping, all threads (`ksoftirq` kernel thread, `vhost-net` thread, and VM thread) are mapped onto the *same NUMA socket* and may use SMT sharing. We repeat the procedure from Section 3.2.2, using 4VMs in this case due to the limits of available cores, and report the cache miss per packet and traffic throughput in Figure 6.

We observe that the throughput in configurations A, B and E is significantly lower to the other configurations in the single VM scenario. In addition, the LLC misses per packet at `ksoftirq` and VM are very high (around 35 misses per packet). This is because the VM is running in user space while the `ksoftirq` is running in kernel space. The frequent context switching leads to severe performance degradation.

Finding 4: *The `ksoftirq` and VM threads are very contentious. It would be better to co-locate them on separate CMP cores, not on single core with SMT.*

Finding 5: *All scenarios will come across performance bottlenecks when more VMs are consolidated. Intensive resource sharing causes very high cache misses at all threads, limiting throughput due to resource contention.*

C. Performance Slowdown Model

Motivated by our characterizations of the HSP inter-socket communication overheads and intra-socket co-located contention overheads, we observe the necessity to find the best trade-off between them. Though many prior studies have explored application slowdown caused by resource sharing or asymmetric communication overheads [13], [14], [24-35], they are designed for evaluating the overall slowdown for applications. There is not an explicit model to measure the inter-stage performance slowdown that coordinately considers both the communication overhead and application slowdown at each thread core in HSP. We develop an enhanced estimation model to solve this problem.

We partition an HSP into several stages; each stage consists of a thread and its communication path. The intra-stage performance slowdown is caused by the inter-socket communication overhead (potential) and application contention slowdown at current stage thread, as depicted in Figure 7.

RIGHTSLINK

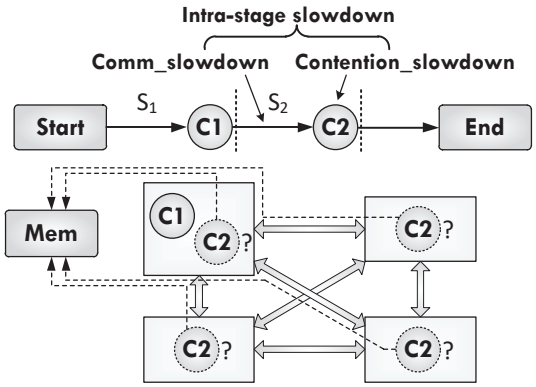

Figure 7 Intra-stage performance slowdown

We build our estimation model based on mature NUMA-aware performance analysis models [15], [36] and our empirical studies. When a thread accesses remote memory (e.g. C2 accesses memory of C1), the performance degradation is caused by four factors: memory controller contention (MC), interconnect contention (IC), last-level cache (LC) and remote access latency (RL). To estimate the inter-stage overheads caused by inter-socket access, we explore how these factors contribute to the HSP inter-stage performance degradation. We conduct a series of experiments where the socket affinities of `ksoftirq`, `vhost-net`, and VM threads (configurations E, F, and G in Section 3.2) are gradually altered. We then vary the contention flow number for each. We present the *overheads contribution factors* for all three stages, as shown in Figure 8. We can observe that, though the four factors' contributions to performance degradation change slightly when input flow numbers vary, different stages still manifest typical distributions. For example, the `ksoftirq` is more sensitive to LC and `vhost-net` is more sensitive to MC, which can be expected based on the characterizations in Section 3.2.

Prior work [15] proposes to use Performance Monitoring Units (PMU) to quantify the four factors. In this solution, the reciprocal of last level cache hit rate (L3_hit) is used for evaluating the LC. The cycle loss due to L3 misses (cycle_loss) is used to evaluate the MC. The cycle_loss at the remote node is used to measure IC. The RL is expressed by the ratio between local IPC to remote IPC. The correlation coefficients between some PMU readings and the corresponding NUMA overheads are around 0.9. However, [15] estimates the NUMA overheads by naively adding up all four PMU readings. This may lead to less accurate evaluation results since different NUMA performance factors have various contributions to the inter-socket NUMA overhead at a stage. This is unacceptable in HSP, because the accumulated error estimation at each stage may lead to large deviations.

We present an enhanced model that estimates the intra-stage overheads as a weighted sum of PMU readings. Take the intra-stage overheads calculation between C1 and C2 for example. There are four socket candidates for thread C2. For each socket candidate, the required PMU metrics are: the $L3\_hit|_{candidate\_socket}$, the $cycle\_loss|_{C1\_socket}$, the $cycle\_loss|_{candidate\_socket}$, and the $IPC|_{C1\_socket}$ / $IPC|_{candidate\_socket}$. The required PMU metrics are weighted to the corresponding overhead contribution factor at this stage. The weighted sum indicates the inter-socket overhead between the C1 socket and the candidate sockets at this stage. We call this sum the *performance slowdown index*.

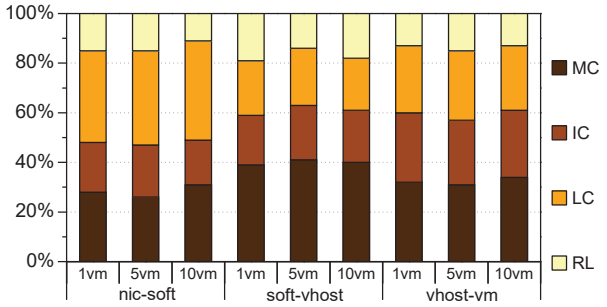

Figure 8. Architectural overheads contribution factors

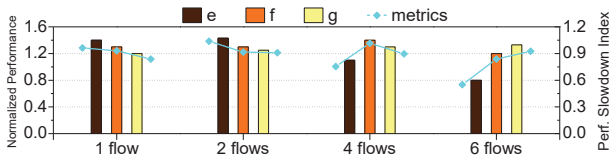

Figure 9. Validation of performance slowdown index

Now we possess a performance estimation model that is capable of evaluating the performance slowdown of a stage of HSP. We validate this model by examining the end-to-end performance slowdown in a series of experiments. In the validation, we test three scenarios, configurations E, F, and G as shown in Figure 4(a), since they are the typical scenarios with inter-socket communication at different stages in an HSP. We vary the input flow number and present the corresponding performance slowdown index, as shown in Figure 9. The results illustrate that our revised architectural metrics can accurately identify performance differences among different stages in HSP. Given that, we can use it as an indicator for our thread scheduling scheme that is tailored to HSP.

IV. vFlowComb-Efficient Thread Scheduling for NFV Deployment

In this section, we present vFlowComb, an architecture support thread mapping framework based on our HSP overheads estimation model. vFlowComb facilitates traffic load-aware and priority-aware data plane hardware resource scheduling and provides a guarantee on the end-to-end NFV flow performance on SHVS architecture. We introduce vFlowComb based on the Open vSwitch implementation. To achieve this goal, vFlowComb exploits a Collaborative Thread Mapping Scheme (CTM). It features a dynamic programming-based thread-mapping scheme (DPBM) to coordinate thread scheduling in the NFV service chain.

A. Preliminary

Flexible and accurate NFV delivery requires that more data, such as core affinity of VNFs, flow throughput, and NIC hardware queue-flow mappings, be exposed to the NFV data plane. Current NICs equipped with hardware-based packet classification support this capability. Although beyond the scope of this

RIGHTSLINK

paper, we can also implement software-/hardware-based packet classification schemes for specific protocols to improve flexibility and reduce Capex. In this study we assume that we can obtain NIC hardware queue-flow mapping, thread affinities and target VNF information for incoming flows, flow priorities, and throughputs.

B. Collaborative Threads Scheduling

Contrasted to conventional thread scheduling mechanisms that only focus on thread co-location contention mitigation [14], we present a Collaborative Threads Scheduling (CTS) mechanism that guarantees end-to-end performance in the HSP. Specifically, we extend the existing contention-aware thread scheduling mechanism by establishing a new dimension that also considers inter-thread communication overhead between predecessor and successor threads in a service chain.

1) Problem Definition

We begin by introducing the system models. All software components in an application constitute a software pipeline (e.g. NFV service chains and virtual switch threads). Each service chain, k, forms a pipeline, P_k, with N_k stages. All service chains share the same source node, s, but terminate on a different node, v_d. Every stage, S_j, is characterized by its communication overhead, o_j, with the predecessor stage, S_{j-1}. For example, packets belonging to network flow F_k will be handled by pipeline P_k and will be processed by a corresponding thread at each stage. This thread will be mapped to core v_m.

The problem can be defined as: given a set of K weighted flows, $F = \{F_1, F_2, \ldots, F_K\}$, with weights, $W = \{w_1, w_2, \ldots, w_K\}$, where the weights express priority levels and each flow has fixed source and destination nodes, optimize the overall weighted system throughput/latency by finding the optimal thread-core mapping for the individual flows. The problem is analogous to a single source shortest path problem where the communication overheads are used as distances.

2) Dynamic Programming Based Mapping (DPBM)

To find the optimal thread-core mapping, the thread scheduler must consider all possible thread-socket/core mapping combinations for a given flow. However, an exhaustive search would result in exponential complexity, $O(c^n)$, which cannot meet the requirement of adaptive mappings at runtime. We propose an algorithm based on **dynamic programming (DP)** that derives optimal solutions for minimizing the end-to-end performance slowdown using M cores to execute flow F_k. We define a recursive function, $\delta_j(s, v_m)$, for each core candidate, v_m, in stage S_j to store the thread-core mapping configuration that achieves the minimized aggregated slowdown at stage S_j, where $o(u, v)$ is the performance slowdown between u and v. Giving:

$$\delta_j(s, v) = \min\{\delta(s, u) + o(u, v) \mid u \notin S_j, v \in S_j\} \quad (1)$$

Let $F_{k,j}$ be a sub-flow of flow F_k that only includes stages S_1 to S_j of F_k. The goal is to find the optimal thread-core mapping that achieves the minimized aggregated slowdown for flow $F_{k,j}$. In this scenario, the aggregated slowdown indicated by $\delta_j(s, v_m)$ at stage S_j only depends on the aggregated slowdown indicated by $\delta_j(s, v_l)$ at previous stage S_{j-1} and the intra-stage performance slowdown index, $o(v_l, v_m)$, between v_m and v_l. we can rewrite the function as:

$$\delta_j(s, v_m) = \min\{\delta_{j-1}(s, v_l) + o(v_l, v_m) \mid v_m \in S_j, v_l \in S_{j-1}\} \quad (2)$$

Algorithm 1. Aggregated Slowdown Minimization

Input: K weighted flows F with N_k stages, M_j cores in stage S_j, the aggregated slowdown $o(v_l, v_m)$ at core v_m,
Output: The thread-core mapping table R that achieves the minimized aggregated slowdown for the flow F.

```
Initialize R, δ(s, v)
for j=1 to N_k do
  for v_m ∈ S_j do
    if j=1 then
      δ_1(s, v_m) = o(v_m, s)
    else
      δ_j(s, v_m)= min{δ_{j-1}(s, v_l) + o(v_l, v_m) | v_l ∈ S_{j-1}}
    endif
    R.append(v_m)
end
return R
```

The dynamic programming starts by computing the aggregated slowdown at each core in stage S_1. The DP continues to compute the aggregated slowdown at each core in stage S_2. Since the programming already stored the minimized aggregated slowdown path from the source node to the cores at the first stage, the minimized aggregated slowdown at a core in stage S_2 can be easily calculated by choosing the minimal sum of the aggregated slowdown at a given core in stage S_1 and the slowdown index between the stage S_2 core and stage S_1 core. Thus, iteratively, an optimal solution is achieved because all combinations of thread-core mappings are considered. However, the complexity is reduced since optimal solutions are stored in tables and do not need to be recomputed. Since vFlowComb schedules thread mappings based on NIC queue, the space/time complexity is $O(MN_k)$ for mapping one NIC hardware queue.

3) CTS in vFlowComb

We incorporate the CTS mechanism into vFlowComb to handle thread-chain scheduling in a real NFV environment. Figure 10 illustrates a typical workflow of CTS in vFlowComb. We assume we can obtain the NIC queue-flow mapping and destination VNFs for each flow by leveraging current NIC hardware. Based on this initial mapping information, CTS aims to efficiently map the software components (`ksoftirq` and `vhost-net` threads) in NFV HSP to improve the performance of NFV flows and resource utilization of SHVS.

The typical workflow of vFlowComb is as follows.

Dominant VNF identification: In this stage, vFlowComb analyzes a snapshot of system flow patterns to identify the priority VNFs and critical flows. To achieve practical efficiency, vFlowComb schedules threads for dominant flows of each NIC queue instead of for a single flow. vFlowComb oversees the flow distribution at each NIC queue and identifies a target VNF with the highest incoming flow traffic (aggregated packets per second) from this NIC queue. vFlowComb then defines this VNF as the dominant target VNF for this NIC queue and schedules a thread chain between them. If vFlowComb detects a dominant VNF is a priority VNF, the corresponding NIC queue will be tagged as a priority queue. Finally, vFlowComb

RIGHTSLINK

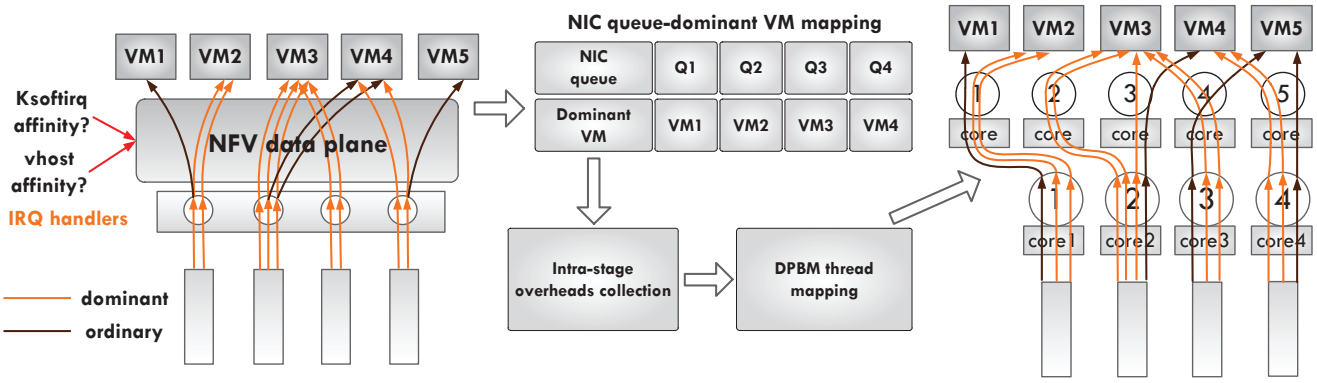

Figure 10. An overview of vFlowComb workflow

obtains a queue group, $\{Q_1, Q_2, Q_3 \ldots, Q_m\}$, which is sorted by priority and flow traffic.

Core pool allocation: vFlowComb collects the available core information and divides the cores into different groups; cores will either be in the `ksoftirq` core group, $\{kC_1, kC_2, kC_3, \ldots, kC_n\}$, or in the `vhost-net` core group, $\{vC_1, vC_2, vC_3, \ldots, vC_n\}$. The benefits of a differentiated core pool are twofold. First, it helps the DP by greatly reducing the search space, which increases the efficiency of CTS. Second, since cores in certain stage only need to sample intra-stage overheads to the cores/sockets in the next stage, it reduces the amount of data collection in the performance slowdown model. When allocating a core to the core pool, vFlowComb prefers the core with the lowest utilization. This provides more performance headroom for threads in early stages so that the global $\delta(s, v_m)$ for each core will not be refreshed frequently.

Intra-stage performance slowdown collection: Obtaining intra-stage performance slowdown statistics is critical for implementing CTS. During the data collection period, each core in a stage (e.g. C1) traverses the core pool of the next stage (e.g. C2) and queries the PMU readings. The data that is returned will be preserved in a table that is maintained by the core. In fact, for each PMU query, all returned PMU readings are socket-based, which reduces the number of inter-processor interrupts between queryer and queryee. From this, the intra-stage performance slowdown between cores in neighboring pools can be established.

Threads scheduling: We apply dynamic programming based mapping (DPBM) to CTS to conduct thread-core scheduling. As shown in Figure 11, CTS assigns the `ksoftirq` core and `vhost-net` core using DPBM for each queue in the sequence. vFlowComb maintains the global $\delta(s, v_m)$ for each core, v_m. After a queue is mapped, the intra-stage performance slowdown and $\delta(s, v_m)$ of all impacted cores will be refreshed to avoid inaccurate scheduling for the next queue. We present the overheads of the dynamic programming-based mapping scheme in Section 5.

Our implementation is based on Receive Packet Steering (RPS) feature provided by the Linux kernel. First, RPS selects CPU cores to execute SoftIRQ based on hash values (`skb->rxhash`) calculated from received packet headers. Next, the `rps_get_cpu` function selects a SoftIRQ core based on the

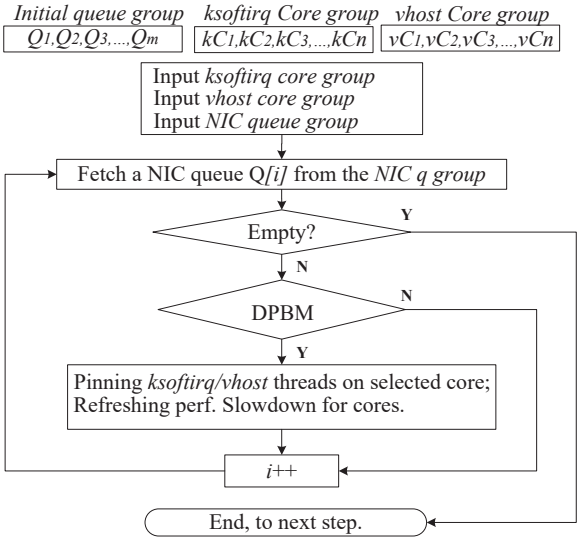

Figure 11. Flowchart of decision making loop

`sock_flow_table` that contains hash/core relation pairs. Finally, the `enqueue_to_backlog` function executes SoftIRQ on the core.

V. Evaluation

vFlowComb provides end-to-end performance guarantees for NFV workloads by cooperatively scheduling the critical threads in the heterogeneous software pipeline. In this section, we evaluate the effectiveness of leveraging vFlowComb to improve the performance of heterogeneous software pipeline on current NUMA-based SHVS platform. We also discuss the design space in vFlowComb.

A. Effectiveness of Collaborative Threads Scheduling

We evaluate the effectiveness of CTS using various traffic loads and co-located contention intensities.

1) Methodology

NFV environment: The test scenarios are designed to mimic real service chains in NFV deployment, as shown in Figure 12. Each service chain consists of no more than 3 VNFs. Each VNF uses 2 vCPU with 4GB memory. All VNFs are deployed on NUMA machine.

RIGHTSLINK

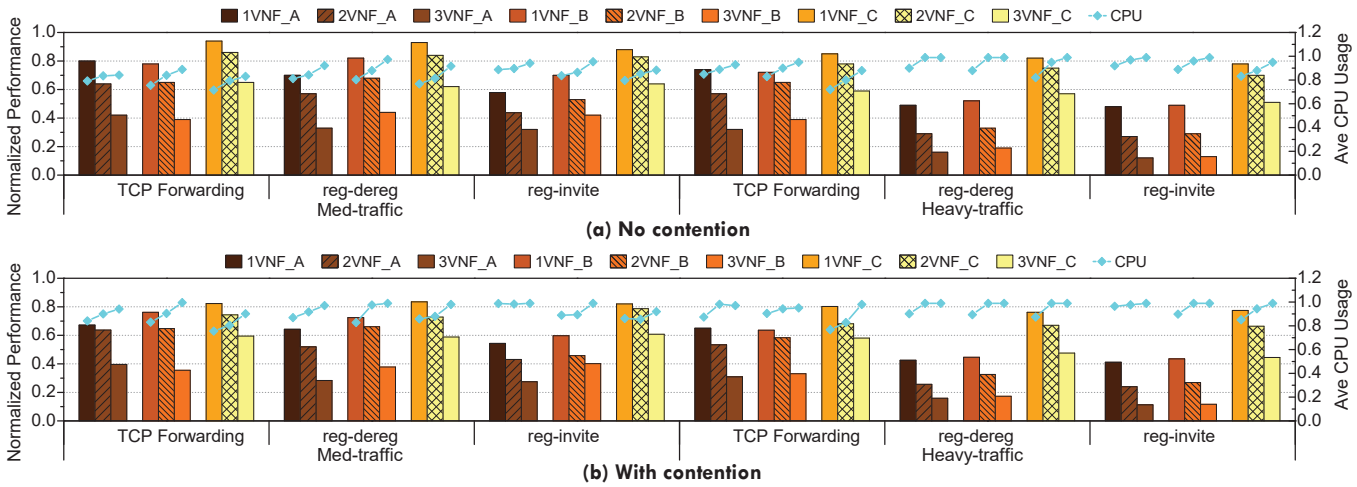

Figure 13. Average end-to-end traffic flow performance and average system CPU utilization

We set up 4 dedicated servers (2 clients and 2 receivers). Each server is connected to a 10GB NIC. We deploy 32 VMs on the clients and receivers as traffic generators and traffic sinks, respectively. They can generate a maximum of 32 traffic flows, which can be processed by 8 service chains. The traffic flows are processed by VNFs in the service chain in tandem through the virtual switches.

NFV workloads: We evaluate vFlowComb using TCP traffic and Telco NFV traffic. For TCP traffic, we set all VNFs as packet forwarding components. For real NFV traffic, the VNFs are deployed as Clearwater components, as described in Section 2. We generate heavy and medium traffic by tuning the packet generation rate (packet per second) at the generators. At each client VM, we use Netperf to generate 512B TCP_STREAM traffic at 24Kpps (medium) and 32Kpps (heavy). We use a small packet size to maximize the system stress [37]. For the NFV workload *reg-dereg*, we set the input call rate as 300 call/s (medium) and 500 call/s (heavy) and for *reg-invite*, we set the input call rate as 100 call/s (medium) and 150 calls/s (heavy). The gateway of the NFV deployment employs a load balancer so that the traffic flows are balanced among service chains based on VNF loads.

Test cases: We test all three traffic loads (TCP STREAM, *reg-dereg*, and *reg-invite*) using input traffic capacity, VNF number in the service chain, and co-located contention. First, we vary the input traffic capacity. Second, we vary the different VNF numbers in a service chain. For TCP traffic, we change the length of the service chain by adding or reducing the number of packet forwarding VMs. For the Telco traffics, we change the VNF number by consolidating the Clearwater function modules on the remaining VNFs. Finally, we vary the co-located contention on the NUMA machine. In the contention scenario we co-locate eight VMs running a single instance of GraphAnalytic from CloudSuite [38]. Each test case lasts 300s. We report the average end-to-end traffic flow performance and average system CPU utilization in Figure 13. Note that we report the *normalized performance* of different traffic loads. For TCP traffic, we report the packet receive rate, which is the ratio of received packets to the sent packets, with a maximum value of 1. For the Telco traffics, we report the successful call rate as defined in section 2.2.2.

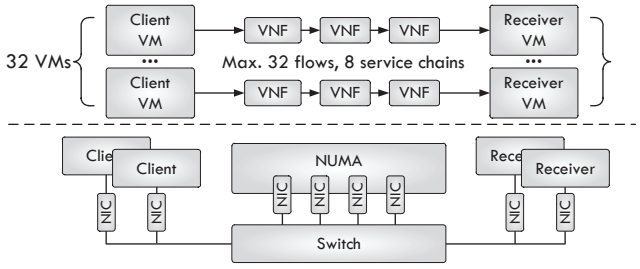

Figure 12. NFV environment setup for evaluation.

Baseline scheduling polices: We use two baseline scheduling policies. The first policy is a NUMA node-aware scheduling policy (A). This policy schedules all threads in the flow data path onto the destination VNF socket. The second policy is a contention-aware scheduling policy (B). It places threads on the core with the least performance slowdown, but ignores the inter-socket communication overheads.

2) Overall Performance Improvement Analysis

To From Figure 13 we can draw several key insights that show the benefits of vFlowComb.

Overall benefits: In all scenarios, we observe that vFlowComb outperforms other methods in terms of average end-to-end performance and global CPU usage. Though the end-to-end performance improvement may be not significantly improved in some scenarios, we can always expect at least 7% global CPU usage savings brought by vFlowComb.

Benefits for the long service chain: When there are 3 VNFs in the service chain, both A and B suffer significant performance degradation. The contention-aware policy performs slightly better than NUMA-aware policy since global contention increases with the more consolidated VNFs. Nevertheless, vFlowComb is still able to avoid performance degradation by finding the best flow data path with the least end-to-end performance slowdown, while saving global CPU resources.

Benefits for the latency sensitive traffic loads: The successful call rate is determined by the request response time.

RIGHTSLINK

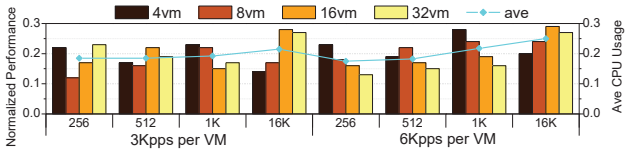

Figure 14. Performance improvements over NUMA-aware policy under extremely high throughput traffics.

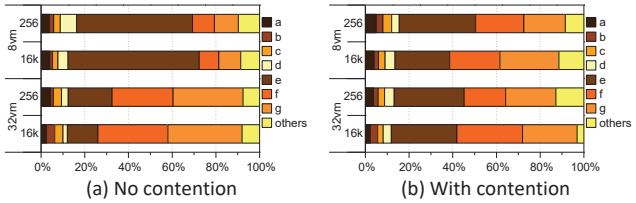

Figure 15. Thread mapping decision distribution under different traffic loads, patterns are defined in Section 3.

Both reg-dereg and reg-invite are TCP traffics with mixed packet sizes. In medium traffic scenarios, vFlowComb demonstrates about 20% more successful call rate than A and B. When the input call rate is increased, vFlowComb can average 50% more successful calls than A and B. Compared to stable stream traffic, vFlowComb performs a little worse when running Telco workloads (about 14% performance degradation). This is because the packet size and flow direction vary frequently in Telco traffic, while vFlowComb works under a low PMU sampling rate (1 sample/s) to reduce the computation overheads. Nevertheless, vFlowComb still outperforms the baseline policies with performance improvements from 15% to 100%.

3) Zoomed Performance Improvement Analysis

To We further examine the effectiveness of vFlowComb when processing extremely high throughput traffic flows. In this test, we zoom in the packet receiving process. We employ a client machine and a receiver machine to establish a peer-to-peer network. We opt to use the Mellanox 40 GB NICs to completely remove any NIC bottleneck. We deploy various numbers of VMs (8, 16, 24, and 32) on the client machine and deploy the corresponding number of destination VMs on NUMA machine. For each client VM, we choose 3Kpps and 6Kpps TCP STREAMs for packet generation rate of medium and heavy traffic, respectively. We report the packet receive rate improvements over the baseline NUMA node-aware policy, as shown in Figure 14. We can observe that vFlowComb achieves an average of 20% more packet receive rate than the NUMA node-aware scheduling policy. The high-speed NIC can clearly reveal the bottlenecks in the NFV data plane. Under these circumstances, vFlowComb can intelligently place threads in the flow data path that minimizes the end-to-end performance slowdown. Interestingly, in the scenario with heavy traffic and small packet size (6Kpps, 256B), the performance improvement of vFlowComb drops when increasing the number of VMs. This is because this traffic poses a significant challenge to the I/O subsystem, creating a CPU time slice discontinuity.

We explore the thread mapping decision distributions under varying system traffic loads and co-located contentions to further demonstrate the correlation between thread mapping

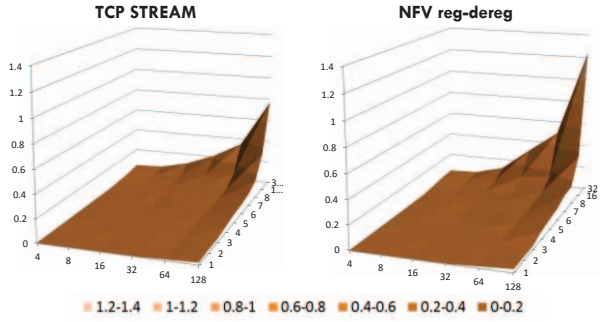

Figure 16. Computational overheads of our DPBM with various NFV

decisions and system traffic/contention status. We collect the scheduling decisions statistics for each mapping pattern listed in Figure 4. As shown in Figure 15a, we notice that the distribution of mapping E is mainly adopted in the medium traffic scenarios where `ksoftirq` is placed on a NIC node. The mappings F and G are adopted when the traffic load increases. In these situations, vFlowComb frequently schedules `ksoftirq` and `vhost-net` to destination VM nodes to avoid the expensive inter-socket communication at `ksoftirq-vhost` and `vhost`-VM stages.

We then introduce co-located contention delve deeper into the thread scheduling trends of vFlowComb. We deploy 4 single-node GraphAnalytic VMs on the NUMA machine to stress the memory subsystem. To create an irregular memory access pattern, we do not pin vCPUs to specific pCPU cores. As shown in Figure 15(b), the mapping decisions are mainly balanced among configuration E, F, and G, since contention occurs irregularly on different NUMA sockets.

B. *Overheads Analysis*

In this section, we evaluate the overheads of DPBM based CTS mechanism. We first evaluate the computation overhead of our DPBM algorithm with various core and NIC hardware queue numbers. In this case, we assume that the `ksoftirq` core pool and `vhost` core pool are identical in size. We explore the performance of DPBM using TCP STREAM (3Kpps) and reg-dereg (250 call/s) respectively.

As shown in Figure 16, we can observe that the DPBM-based CTS can achieve optimal thread mappings within 0.77s (TCP STREAM) and 1.22s (NFV workloads) in a standard SHVS configuration (128 cores, 32 NIC queues).

VI. Related Work

Recent studies highlight various opportunities for enhancing networking performance. While all of the prior studies either focus on optimizing the I/O performance on NUMA system or improving the virtual switch performance, our research bridges the gap between networking function virtualization and NUMA-based server systems to provide more flexible data plane flow management.

Thread mapping on multicore and NUMA system. Several research efforts have developed mechanisms that mitigate hardware resource interference and improve the throughput on

RIGHTSLINK

multicore and multi-socket NUMA systems. Chadha et al. proposed a run-time system to determine the optimal thread count, processor voltage, and frequency [39]. Sirdharan et al. proposed a model to predict optimal thread count [40]. Tang et al. [25] develop an adaptive approach to achieve optimal thread-to-core mappings in a data center to reduce the co-located interference. Blagodurov et al. [36] observe the limitation of contention-aware algorithms designed for UMA systems and present new contention management algorithms for NUMA systems. Liu et al. [15] characterize the impact of architecture-level NUMA access overhead on cloud workload consolidation and incorporate the overhead into the hypervisor's virtual machine memory allocation and page fault handling routines.

Networking I/O optimization. Several prior works have been proposed to reduce the networking I/O overhead in the operating system, either in kernel or user space. Among those, Affinity-accept [41] and FastSocket [42] explore TCP connection traffic and improves the packet processing efficiency in Linux kernel by affinitizing the incoming flows to one core. However, they only address conventional network I/O issues in the operating system and avoid providing analysis on NUMA deployment and virtual switches.

NUMA-aware I/O optimization. Some recent works have addressed the networking I/O performance on NUMA-based hardware. Hyper-switch [43] employs a dynamic offloading scheme to distribute packet processing to idle processor cores; this scheme takes into account the impact of CPU cache locality and NUMA systems. NetVM [44] proposes a NUMA-aware queue/thread management technique that keeps the consistency of core-thread affinity of each flow on each NUMA node. However, they did not consider thread dependencies in the NFV service chain scenario and did not design for HSP-based NFV deployment.

VII. Conclusion

NFV plays an important role in current Telco and cloud data centers. It relies heavily on the performance of commodity standard high volume servers. We observe the processing style of these workloads manifests as a heterogeneous software pipeline, in which the traffic flows are sequentially processed by heterogeneous software components. We conduct intensive and extensive characterizations of NFV deployment on current NUMA-based SHVS using real world Telco workloads. Our experimental results indicate that HSP introduces heterogeneous performance slowdown at different stages (intra-stage performance slowdown). The intra-stage performance slowdown is jointly determined by inter-socket communication overheads and co-located contention. We build a performance slowdown estimation model that accurately evaluates the intra-stage and end-to-end performance slowdowns. We then design a collaborative thread scheduling mechanism that is tailored to thread mapping in HSP. It exploits a dynamic programming-based end-to-end performance slowdown estimation method that accurately maps threads in the NFV data plane to improve traffic throughput (on average 23%) and increase the CPU utilization (7%) with negligible overhead (decision making time less than 1s).

Acknowledgment

We would like to thank Mingcong Song for his extraordinary efforts on this work. We also would like to thank Clay Hughes, James Poe, Can Duan, and Huixiang Chen for making this paper better. We appreciate our shepherd Nam Sung Kim and anonymous reviewers for their valuable suggestions. This work is supported in part by NSF grants 1527535, 1423090, 1320100,1117261, 0937869, 0916384, 0845721(CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

References

[1] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," Proc. 42nd Annu. Int. Symp. Comput. Archit. - ISCA '15, pp. 14–26, 2015.

[2] C. Cui, H. Deng, D. Telekom, U. Michel, and H. Damker, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action," Netw. Funct. Virtualisation – Introd. White Pap., no. 1, pp. 1–16, 2012.

[3] ETSI ISG NFV, "Network Functions Virtualisation (NFV): Architectural Framework," 2013.

[4] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-tenant Datacenters," Proc. 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), pp. 203–216, 2014.

[5] L. Foundation, "Open Platform for NFV (OPNFV)." [Online]. Available: https://www.opnfv.org/.

[6] Oracle, "The Road to NFV Success Is Paved with Intelligent Orchestration," 2015.

[7] TechNavio, "Global Network Function Virtualization Market 2014-2018," 2014.

[8] W. Paper and S. Infrastructure, "Intel® Open Network Platform Server Reference Architecture: SDN and NFV for Carrier-Grade Infrastructure and Cloud Data Centers," 2014.

[9] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," High Perform. Comput. Archit. (HPCA), 2010 IEEE 16th Int. Symp., pp. 1–10, 2010.

[10] Intel, "Intel Data Direct I/O Technology (Intel DDIO): A Primer."

[11] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT, pp. 22–32, 2011.

[12] W. Wang, J. W. Davidson, and M. Lou Soffa, "Predicting the Memory Bandwidth and Optimal Core Allocations for Multi-threaded Applications on Large-scale NUMA Machines," pp. 419–431, 2016.

[13] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," Int. Symp. Microarchitecture, 2015.

[14] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers," 2014.

[15] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," Proc. - Int. Symp. Comput. Archit., pp. 325–336, 2014.

[16] P. Garg and Y.-S. Wang, "NVGRE: Network Virtualization using Generic Routing Encapsulation," 2014.

[17] "Scaling in the Linux Networking Stack." [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt.

RIGHTSLINK

[18] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, A. Networks, and M. Casado, "The Design and Implementation of Open vSwitch," 12th USENIX Symp. Networked Syst. Des. Implement., pp. 117–130, 2015.

[19] "Project Clearwater." http://www.projectclearwater.org/about-clearwater/.

[20] "Welcome to SIPp." [Online]. Available: http://sipp.sourceforge.net/.

[21] "OpenStack Cloud Software." [Online]. Available: www.openstack.org.

[22] R. Jones, "NetPerf: a network performance benchmark," Inf. Networks Div. Hewlett-Packard Co., 1996.

[23] R. S. Roman Dementiev, Thomas Willhalm, Otto Bruggeman, Patrick Fay, Patrick Ungerer, Austen Ott, Patrick Lu, James Harris, Phil Kerly, Patrick Konsor, Andrey Semin, Michael Kanaly, Ryan Brazones, "Intel® Performance Counter Monitor - A better way to measure CPU utilization." .

[24] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multicore Memory Systems," ACM Trans. Comput. Syst., vol. 30, no. 2, pp. 1–35, 2012.

[25] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. Lou Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in ACM SIGARCH Computer Architecture News, 2011, vol. 39, no. 3, p. 283.

[26] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Lou Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture - MICRO-44 '11, p. 248, 2011.

[27] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2007, pp. 146–158.

[28] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in Proceedings - International Symposium on Computer Architecture, 2008, pp. 63–74.

[29] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu, "Going vertical in memory management: Handling multiplicity by multi-policy," Proc. - Int. Symp. Comput. Archit., no. 1, pp. 169–180, 2014.

[30] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu, "Rethinking Memory Management in Modern Operating System : Horizontal , Vertical or Random ?," vol. 65, no. 6, pp. 1–14, 2016.

[31] M. Song, Y. Hu, Y. Xu, C. Li, H. Chen, J. Yuan, and T. Li, "Bridging the Semantic Gaps of GPU Acceleration for Scale-out CNN-based Big Data Processing: Think Big, See Small," 25th Int. Conf. Parallel Archit. Compil. Tech., 2016.

[32] Y. Hu, C. Li, L. Liu, and T. Li, "HOPE: Enabling Efficient Service Orchestration in Software-Defined Data Centers," in Proceedings of the 2016 International Conference on Supercomputing, 2016, pp. 10:1–10:12.

[33] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters.," USENIX Annu. Tech. Conf., pp. 277–289, 2015.

[34] Q. Wang and B. C. Lee, "Modeling Communication Costs in Blade Servers," SIGOPS Oper. Syst. Rev., vol. 49, no. 2, pp. 75–79, Jan. 2016.

[35] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 367–376.

[36] S. Blagodurov, S. Zhuravlev, A. Fedorova, and M. Dashti, "A Case for NUMA-aware Contention Management on Multicore Systems," Proc. 19th Int. Conf. Parallel Archit. Compil. Tech., no. Llc, pp. 557–558, 2010.

[37] Intel, "Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers," 2003.

[38] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, 2012, pp. 37–48.

[39] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (LIMO): controlled parallelism forimproved efficiency," Int. Conf. Compil. Archit. Synth. Embed. Syst., pp. 141–150, 2012.

[40] S. Sridharan, G. Gupta, and G. Sohi, "Adaptive, efficient, parallel execution of parallel programs," … SIGPLAN Conf. Program. …, pp. 169–180, 2014.

[41] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," EuroSys'12, p. 337, 2012.

[42] X. Lin and Y. Chen, "Scalable Kernel TCP Design and Implementation for Short-Lived Connections," Asplos, pp. 339–352, 2016.

[43] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner, "Hyper-Switch: A Scalable Software Virtual Switching Architecture," Present. as part 2013 USENIX Annu. Tech. Conf. (USENIX ATC 13), pp. 13–24, 2013.

[44] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," Proc. 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), vol. 12, no. 1, pp. 445–458, 2014.

RIGHTSLINK