Meeting SLOs in Cross-Platform NFV

Jane Yen*
University of Southern California
yeny@usc.edu

Jianfeng Wang*
University of Southern California
jianfenw@usc.edu

Sucha Supittayapornpong
Vidyasirimedhi Institute of Science and Technology
sucha.s@vistec.ac.th

Marcos A. M. Vieira
Universidade Federal de Minas Gerais
mmvieira@dcc.ufmg.br

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Barath Raghavan
University of Southern California
barathra@usc.edu

ABSTRACT

Network Functions (NFs) perform on-path processing of network traffic. ISPs are deploying NF Virtualization (NFV) with software NFs run on commodity servers. ISPs aim to ensure that NF chains, directed acyclic graphs of NFs, do not violate Service Level Objectives (SLOs) promised by the ISP to its customers. To meet SLOs, NFV systems sometimes leverage on-path hardware (such as programmable switches and smart NICs) to accelerate NF execution.

Lemur places and executes NF chains across heterogeneous hardware while meeting SLOs. Lemur's novel placement algorithm yields an SLO-satisfying NF placement while weighing many constraints: hardware memory and processing stages, server cores, link capacity, NF profiles, and NF chain interactions. Lemur's *meta-compiler* automatically generates code and rules (in P4, Python, eBPF, C++, and OpenFlow) to stitch cross-platform NF chain execution while also optimizing resource usage. Our experiments show that Lemur is alone among competing strategies in meeting SLOs for canonical NF chains while maximizing marginal throughput (the traffic rate in excess of the service-level objective).

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; **Network components**;

KEYWORDS

NFV, service chain, PISA switch

1 INTRODUCTION

Over the last few years network operators have begun to deploy virtualized network functions (NFs). These NFs typically perform packet processing in software on commodity servers. They replace specialized hardware middleboxes, leveraging cheaper commodity servers and cloud-like service management. They also permit flexible orchestration of the data plane by chaining together NFs (into *NF chains*) to meet operator needs. An important large-footprint use case is a *rack-scale deployment* of servers to run NFs for traffic ingressing or egressing a telecom central office, an ISP Point of Presence (PoP), or an enterprise border. *This is the setting we consider in this paper.*

Industry interest has prompted two threads of NFV work:

Software NFs. One thread has focused on programming and orchestrating NFs and achieving elastic scaling (*e.g.,* [12, 32, 33]), and improving their performance (*e.g.,* [2, 40, 41]). However, a key factor in practical deployments, the ability to meet service-level objectives (SLOs) for traffic processed by an NF chain, has received less attention [41]. Consider an ISP that serves residential or enterprise customers. It may want to apply security or isolation policies on traffic from these customers using NF chains. In doing so, however, the ISP runs the risk of *violating traffic SLOs* that it established with its customers because the NFs add processing overhead [9]. In discussions with ISPs, we have found that such SLOs usually have three required components: a *minimum rate* requirement on aggregate traffic processed by an NF chain, a *maximum rate* bound that limits bursts, and a *maximum delay* imposed by the NF chain. Accompanying these SLOs is a pricing model that sets a fixed price for the minimum rate, and a usage-sensitive price for traffic *above* the minimum rate.

Hardware acceleration. A second thread stems from the growing realization that custom-hardware middleboxes, while inflexible, delivered greater predictability and performance than software-based NFs. In response, researchers and industry alike have looked to leverage hardware acceleration (in the form of Protocol Independent Switch Architecture or PISA hardware [15, 27], Smart NICs [10, 24], GPUs [8, 36], and Network FPGAs [6, 26]). This line of work explores hardware acceleration on an NF-by-NF basis, yielding a suite of useful, but piecemeal, higher-performance implementations.

Hardware acceleration is particularly important in our setting as server scaling has its limits in a rack-scale deployment where space and power considerations are important. For example, the 32-port Barefoot Tofino-based PISA switch [4] we use, which has 3.2 Tbps of capacity, consumes about 450 W, comparable to a 1U two-socket Intel Xeon-based server. However, due to limitations of the P4 programming model and limited hardware resources, not all NFs can run on switches.

* Both authors contributed equally to this paper.

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431292

Leveraging hardware acceleration through manual configuration has always been possible, but ISPs that have taken this approach have suffered from the high management overhead such a manual approach imposes.

In addition, we found when working on NFV in industry that application of generic cloud computing platforms (e.g., OpenStack, Kubernetes, etc.) aided in automation but yielded abysmal performance. On the other hand, hand-tuned NFV deployments were difficult to optimize and maintain. Neither provided the requisite SLO guarantees. Lemur aims to get the best of both worlds, automation and highly-tuned performance, while meeting SLOs.

Goal. Given multiple NF chains and their associated SLOs, we seek to *automatically* place, configure, and execute multiple chains *across heterogeneous hardware* such that: (a) each NF chain receives at least its minimum rate, and (b) the total *marginal* rate (the rate above the minimum which each chain can burst) is maximized, which maximizes revenue for the ISP. Automatic configuration means: (a) decide where an NF should run (in software, on a PISA or OpenFlow switch, or in a SmartNIC), and the degree of NF scale-out required (using multiple cores) to achieve the SLOs, and (b) execute each NF chain across hardware with little operator intervention. In doing so, we must leverage all hardware made available on-path, thereby providing operational flexibility.

Lemur's goal is *not* to provide a unified programming language for heterogeneous platforms. Instead, it respects existing hardware offload efforts, and embodies a practical approach to generating appropriate NF placement.

Automatic configuration poses several challenges. How do we specify NF chains in a manner amenable to analysis for acceleration and scaling decisions, and compilation for execution? How do we determine which hardware element (CPU, PISA switch, smart NIC, OpenFlow switch) each NF in the chain should run on? How do we scale partial NF chains by replicating them across multiple cores in order to meet SLOs? How do we respect constraints imposed by hardware accelerators (*e.g.,* pipeline stages on PISA switches)? How do we accommodate link capacity constraints between switches and servers? How do we work around limitations in the programming and execution models used by hardware accelerators?

Lemur: Approach and Contributions. In this paper, we present Lemur, a system to address these challenges. Lemur takes as input a high-level description of multiple NF chain DAGs and their associated SLOs. Lemur's output is a placement configuration for each NF chain along with *coordination code* that ensures that the NF executes on the appropriate hardware element specified by the placement.

Contributions. First, Lemur provides *Placer*, which determines NF placement and provisioning, and addresses the several competing challenges identified above: determining hardware acceleration, deciding the scale out for NFs on multi-core systems, respecting link capacities, and hardware-specific constraints (§2). Our work includes an oft-ignored element of performance prediction, run-to-completion NF execution, as opposed to cross-core execution. A MILP formulation can address a scalable run-to-completion formulation while meeting SLO requirements and link-capacity constraints, but off-the-shelf solvers cannot determine if a set of NF chains respects hardware constraints, since that requires actually invoking the hardware-specific compiler. An alternative, optimal approach (§3) leverages the structure of the problem to (a) enumerate *placements* of NFs on different hardware elements, (b) use resource and performance profiles of each NF on each hardware element to determine how to scale out server-placed NFs to multiple cores in each placement, (c) determine which placements maximize the aggregate marginal throughput while satisfying link capacity constraints, and (d) select a placement that respects hardware limitations. Enumerating placements is computationally expensive, so we develop (§3) a heuristic capable of near-optimal performance with very low placement delay. Our placement algorithm addresses the issue that today's PISA switches do not expose an inexpensive API to check the feasibility of placements.

Second, Lemur provides a *meta-compiler* that, given NF implementations, produces low-overhead coordination code and tables to ensure that the NF chain executes as determined by Placer. The meta-compiler automatically (§4) reasons about DAGs in NF chains, and generates code for function chaining. A key architectural novelty in Lemur's meta-compiler is the use of a top-of-the-rack (ToR) PISA switch as a coordinator (in addition to acting as an NF accelerator), which shares fate and improves performance. Additionally, our meta-compiler overcomes another limitation of PISA switches: the programming model of these switches does not permit reasoning about modular NFs. Recent work [37] has explored language support for modular and composable P4 programs; in contrast, Lemur targets minimal changes to P4 to support NF composition.

In our evaluations, we use canonical NF chains [21], which in our experience both in industry and research reflect actual deployments. For these, Lemur outperforms alternative approaches. It finds feasible placements in all our experiments while other approaches find feasible placements in about 17-76% of the cases. Lemur also obtains a maximum marginal throughput difference over competing approaches of more than 50% of the link capacity across our experiments. We demonstrate that Lemur's meta-compiler can reduce manual labor: nearly 30% of code in Lemur is auto-generated. Finally, Lemur's heuristic placement algorithm can generate near-optimal placements in a little over three seconds. We have open sourced our implementation and MILP formulation.[1]

2 LEMUR: OVERVIEW

Overview. The input to Lemur (Figure 1) is an *NF chain specification* that, for each chain, describes which traffic aggregates to apply, the DAG of NFs, and the corresponding SLO. The Placer consumes the specs and determines, for each NF in every NF chain, whether Lemur should execute that NF in on-path hardware, and if so, on which element. If Placer decides to run the NF on a server it also determines how many cores to allocate to the NF. The resulting *placement configuration* of the NF chains is guaranteed to satisfy the specified SLOs.

Given the specification, and the Placer's placement configuration, the *meta-compiler* parses the specification, selects the NF implementation for the hardware target identified by the configuration, and automatically generates code to route traffic between NFs in the NF chain. For example, traffic may first ingress the ISP at the

[1] https://github.com/USC-NSL/Lemur

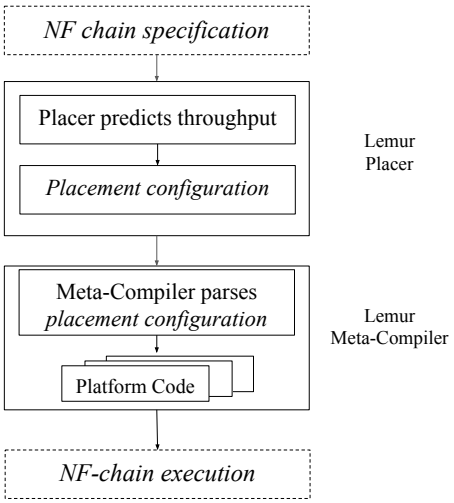


Figure 1: Overview of Lemur's design.

PISA switch, then traverse NFs on a server and a smart NIC. Traffic may bounce back to an NF on the PISA switch before returning to a server (which may be necessary to satisfy SLOs as we discuss later), and return again to the switch before egressing the ISP.

Specifying NF chains. Lemur provides a natural and abstract means for operators to specify NF chains. Inspired by BESS [2], our specification is not novel, but is critical for enabling automated placement and execution. To use hardware middleboxes typically required an operator to manually configure pipelines, and a core NFV aim was to automate such work. However, Tier-1 operators, even for non-hardware accelerated settings (such as VM-based NFs using SR-IOV) do manual setup, and we have learned from them that they frequently leave stranded resources due to configuration complexity. In Lemur, even though it supports multiple hardware platforms, NF chain configuration is as straightforward as with software-only NFV.

A Lemur user (*e.g.,* an ISP operator) specifies NF chains using a dataflow language, where the nodes represent NFs and the edges represent packet flow between NFs. For example, `ACL -> Encryption -> Forward` specifies an NF chain in which `ACL`, `Encryption`, and `Forward` are all NF names. This NF chain represents incoming packets filtered by an access control list (ACL) NF, encrypted by an encryption NF, and finally emitted out an appropriate port based on MAC address-based forwarding.

An NF can have parameters. For example, the ACL can have an associated rule:

```
ACL(rules=[{'dst_ip':'10.0.0.0/8','drop': False}])
```

to drop packets other than those destined to `10.0.0.0/8`. Also, an NF chain may specify conditional execution through branching, like:

```
ACL -> [{'vlan_tag': 0x1, Encryption}] -> Forward
```

which encrypts packets matching a specific vlan tag.

This specification is high-level and declarative. NFs in NF chain specifications use a predefined but extensible vocabulary. They do not specify *where* or *how* an NF executes. For example, Lemur users would not need to know whether an `ACL` NF runs on hardware accelerators or on x86 servers. Indeed, for one NF chain, Lemur might decide to configure an `ACL` on a PISA switch, but for another in software.

Use Case	$t_{\min}$	$t_{\max}$	Description
Bulk	0	∞	Best effort
Metered bulk	0	α	Best effort, capped at α
Virtual pipe	α	α	Exactly α guaranteed
Elastic pipe	α	β	At least α w/ bursts up to β
Infinite pipe	α	∞	At least α

Table 1: Lemur's SLOs capture key operator use cases.

Each chain processes traffic from one or more traffic aggregates. An aggregate specifies a combination of flow 5-tuple values (source and destination IP addresses and port numbers, as well as protocol number); in our setting, an aggregate may represent traffic from a customer, for example.

Specifying performance objectives. Finally, for each traffic aggregate, the operator specifies the SLO that must be satisfied by the associated NF chain. We have derived Lemur's SLO specifications from discussions with operators; the specifications are simple yet capture important use cases. For each NF chain and traffic aggregate, the operator can specify: a min throughput $t_{\min}$, a max throughput $t_{\max}$, and a max delay $d_{\max}$. Lemur must provision for the NF chain to achieve *at least* $t_{\min}$ throughput with at most $d_{\max}$ delay. Operators also permit traffic to burst up to $t_{\max}$. These bounds also determine pricing: operators often charge a fixed price for $t_{\min}$, with use-based pricing above that rate; this is contractual with customers and must not be violated by NF chains that it applies to customer traffic in order, for example, to enforce its own security policies. Finally, because traffic usage beyond $t_{\min}$ generates revenue, Lemur attempts to maximize the aggregate *marginal throughput* (the traffic rate in excess of $t_{\min}$).

Our simple SLO spec can capture several key use cases (Table 1). Large carriers often sell enterprises and smaller operators virtual and/or elastic pipes. Residential traffic, on the other hand, is typically advertised as an elastic pipe but given metered bulk in reality. Bulk is used for low-priority traffic that consumes excess resources after other demands are met. Finally, customers with bursty demand and a willingness to pay for any level of usage can select infinite pipes (which are of course limited by hardware and interconnects).

3 THE PLACER

Placement of NFs is a key problem in NFV. Prior work aimed for unified orchestration and execution within one platform (*e.g.,* software), and on monolithic NFs. Lemur's Placer, faces harder challenges: it must not only perform NF placement with limited resources, but do so while avoiding SLO violations, and while accommodating multiple hardware categories (*e.g.,* PISA switches, smart NICs) which have not only different resources but different *types of* resources.

3.1 The Placement Problem

The input to Placer is a collection of NF chains, and associated SLOs (*e.g.,* $t_{\min}$ and $t_{\max}$ for each chain). In addition, Placer is also given the underlying topology consisting of a single PISA switch connected to several servers each of which may have one or more attached smart NICs.

Placer produces a *placement* that specifies whether each NF in an NF chain should run on the PISA switch, a smart NIC (and which smart NIC), OpenFlow switch, or a server (and which server and config). If it places an NF on a server, Placer also specifies NF core allocation. Given NUMA in modern servers, and CPU socket-NIC association, Placer also specifies the NIC to which the chain is assigned to the chain.

A placement is to be *feasible* if the following conditions hold: (a) each NF chain receives at least $t_{\min}$; (b) the NFs allocated to the PISA switch collectively *fit* into the switch; (c) the placement respects the server core counts of every server; (d) aggregate traffic resulting from the placement does not exceed the capacity of any network link.

Placer aims to generate a feasible placement with maximal aggregate marginal throughput: the difference between a chain's *estimated* throughput and its $t_{\min}$. To check whether an NF chain meets its SLO, Placer estimates the throughput of each chain [16]. As discussed in §2, this objective is natural in our setting because it maximizes revenue for the ISP.[2]

Challenges. Several aspects make this placement problem hard. First, some NFs can be placed on servers, switches, or smart NICs, while others have limited placement options (*e.g.,* PISA switches cannot currently perform payload encryption). Second, different hardware resources have different constraints. PISA switches can process NFs at line rate, but have limited pipeline stages and memory. Servers are less constrained and more general, but are slower. Smart NICs occupy a midpoint. Beyond placement, Placer has to meet SLOs. To do this, it needs to estimate the throughput achieved by an NF chain in a given placement. This throughput is primarily constrained by the processing on servers and smart NICs (since PISA switches process at line rate).

When it places NFs on servers, Placer must also *minimize overhead* in NF execution, which will allow efficient packing of NFs into servers. Consider two successive NFs A and B in a chain: Placer must decide, while meeting SLOs, whether to place these on the same core (to avoid copying costs), or on different cores (to permit parallelism) [46]. To meet the SLO, it may be necessary to *replicate* A across several cores.

Alternative Approaches. For concreteness, consider two strawman approaches. The first places an NF on the PISA switch whenever a switch implementation exists. This may be infeasible depending on the chain, since it may exceed the number of switch stages. The second, at the other end of the spectrum, places an NF on a server if a software version exists, which may be infeasible because there may be too few cores to satisfy $t_{\min}$ for one or more chains.

Prior work has considered placing *VM-based* NFs on server cores while satisfying SLOs, a mixed integer programming problem [22, 28] solved either using heuristics [22] or with an MILP solver [28]. Other work has explored minimum bounce placements [32]. However, in this context, such bounces between nodes may be unavoidable. Consider a chain with five NFs A-E where B and D only have software implementations, A and E only switch implementations, but C can be executed on either. A minimum-bounce placement would force server placement. This may be sub-optimal and fail to meet $t_{\min}$: another NF chain could use the core(s) allocated to C to achieve a feasible solution, or one with higher marginal throughput.

[2]More fine-grained objectives may also make sense in our setting; an ISP may wish to allocate higher marginal rates to certain customers, or ensure proportional fairness in rate allocation. We leave this to future work.

3.2 The Placement Algorithm

Lemur's placement algorithm overcomes these challenges.

Profiling and Estimated Throughput. To estimate the throughput of an NF chain, Placer precomputes *profiles* for each NF on a server and/or smart NIC. NF B's profile is the CPU cycle count c to execute it.[3] Given the CPU clock rate f, the estimated rate for B is $\frac{f}{c}$. Placer might allocate k cores to B, in which case its rate is $k\frac{f}{c}$. If an NF chain placement has multiple server (or smart NIC) placed NFs, the estimated rate of the NF chain is the minimum of all the per-NF (or, per NF sub-group, as discussed below) estimated rates.[4]

The cycle count of an NF may be a function of NF state or traffic. For example, `ACL` processing may depend on table sizes; we profile cycle counts for different sizes and use a linear model to predict the processing costs. In other cases, such as `NAT`, we may not know the size of the state a priori, in which case we aim to compute a *worst-case* cycle count. Finally, for some NFs such as `Dedup`, the cycle count might depend on the degree of redundancy in the packet; in this case we compute a worst-case cycle count, and plan to explore better profiling techniques in the future. Placer decouples profiling from placement, so can directly leverage improvements in profiling (such as based on operator-specific knowledge).

Brute-force Placement. Placement lends itself to an optimization formulation. We cast the placement problem as an MILP, but for one key component: *it is hard to estimate a priori the number of PISA switch stages* used by a placement because the PISA compiler (for Barefoot's Tofino [29]) performs stage packing. We could have modeled the PISA switch placement conservatively [14], but this would have resulted in stranded resources. An alternative is *brute-force* placement, which: (a) enumerates placement patterns, (b) searches through core allocations for each pattern, and (c) finds the max marginal throughput for a pattern and core allocation.

Enumerating Placement Patterns. Brute-force placement first enumerates **patterns** of all possible NF placements across available hardware for the given DAG. For example, for a chain `A->B->C->D`, one possible placement is A on the PISA switch, B and C on a server, and D back on the PISA switch. Another placement might place D on a smart NIC. The space of patterns is large but constrained by the fact that not all NFs can run on all platforms: *e.g.,* a `Dedup` NF that de-duplicates packet payloads can only run on a server.

Dealing with branches in chains. In enumerating placement patterns for NF chains with branches, we decompose such chains into linear chains. Thus, if a chain branches from NF X to two NFs Y and Z, and then merges back into an NF W, we decompose these into two chains `X->Y->W` and `X->Z->W`. Here we assume knowledge of traffic splits across the two chains (in our discussions, operators

[3]We consider eBPF [42]-capable NICs that can be profiled this way.

[4]As ResQ [41] shows, there are subtle NF performance interactions in software that must be accounted for, such as cache effects; ResQ is complementary to Lemur and could be used to improve our estimation [41].

estimate these using historical measurements). Later we merge the throughput estimates for X and W.

Searching through Core Allocations. For each pattern, brute-force placement must search all possible *core allocations* for server NFs because, to meet the SLO or to increase the aggregate marginal throughput, we may need to allocate multiple cores to an NF and split traffic across the instances. Currently we (a) coalesce successive server NFs using *run-to-completion* (discussed below), and (b) do not replicate stateful NFs or those where a branch or a merge occurs.

In our example above, if B and C are assigned to a server, we run them to completion on one core (*i.e.,* a packet batch is fully processed by both NFs before B starts processing the next batch). In this case, we say B and C are part of a single *subgroup*, and, in making core allocation decisions, we treat a subgroup as a single entity. Subgrouping has two advantages. First, run-to-completion is fast because it permits zero-copy packet transfers between NFs, has no scheduling overhead, and has no cross-core communication. Second, subgrouping involves a search of fewer patterns and fewer core allocations.

Subgrouping and run-to-completion are not always optimal. Consider two NFs B and C (where C comes after B in an NF chain) each with a cycle cost of x. The throughput of the subgroup BC is $\frac{f}{2x}$. Instead of sub-grouping, one can run B and C on separate cores; the throughput for the two NFs would be $\frac{2f}{2x+\delta}$ where δ is the cross-core or cross-socket cost. Depending on the relative values of x and δ, this throughput can be higher than run-to-completion. However, modeling cross-core and cross-socket costs is complex, especially considering cache effects [41]; we profile conservatively (§5.3), leaving more sophisticated profiling to future work.

Replicating stateful and branch/merge NFs. Brute-force placement does not replicate any subgroup containing one or more stateful NFs even though it may be possible to do so. For example, `NAT` can be replicated by partitioning the port space to minimize cross-core communication. Our current implementation does not do this yet in part because automatically generating this replication in a meta-compiler (§4) is difficult, and we have left this to future work. We plan to leverage work on stateful NF scaling, such as S6 [44]. Meta-compilation complexity also motivates us to avoid replicating NFs where branching or merging occurs.

Finding Maximum Marginal Throughput. For a given placement pattern and a given core allocation, we can find the maximum throughput achievable for each NF chain from our cycle cost profiles. The throughput is constrained either by a server subgroup or a smart NIC NF. We compute the NF chain's estimated throughput, as discussed above, as the minimum of the throughputs of all NF subgroups or smart NIC NFs in the chain.

However, the sum of NF chain rates can overwhelm a NIC, so we must find assignments for NF chains that respect NIC capacities while maximizing aggregate marginal throughput. This problem is complex when two subgroups in an NF chain may be placed on different servers, or different NIC interfaces on a server with multiple NICs. Brute-force placement uses a linear program to determine the max marginal throughput.

Putting it all together. Brute-force placement lists possible *placements* (where a placement includes a pattern, a core allocation for each subgroup, and the rates assigned to NF chains), ordered by decreasing maximum marginal throughput. We then iteratively call a PISA compiler to find the highest-ranked placement within the switch's stage constraints.

A Fast, Scalable Heuristic. Brute-force placement has two expensive pieces: enumerating placements and core allocations, and compiling placements on a PISA switch's compiler. Next we describe a low-complexity heuristic that reduces the cost of both of these and is several orders of magnitude faster than brute-force placement. Unless otherwise indicated, Placer uses this heuristic, which has three steps.

1. Check stage constraints. Placer greedily places as many NFs on the PISA switch as possible. If this placement exceeds the switch's stages, it iteratively moves the lowest cycle cost NF away from the switch until it finds a placement that respects the stage's constraints. The rationale behind removing the lowest cycle cost NF first is: since the PISA switch guarantees line-rate for any chain that fits the switch resources, if a high cost NF and a low cost NF use the same number of stages, it is always better to remove the low-cost NF (since it is more likely that we can pack this on the server and satisfy SLOs). Thus, unlike brute-force placement, Placer checks the switch stage constraint *first*, which more effectively prunes the search space. The output of this step is a *baseline placement*. The next step may remove NFs assigned to the PISA switch in the baseline placement to explore alternative placements (as described below); however, it never adds an NF to a PISA switch, guaranteeing that the final placement always respects the switch constraint.

2. Coalesce sub-groups. Even with the baseline placement, the search space is still large: we can *offload* each PISA switch NF (or combinations thereof) to the server to see if these result in higher marginal throughputs. Each such offload presents an opportunity to *coalesce* sub-groups. To see why, consider a chain `{A->B}->C->{D->E}` where the `{}` denote server-placed subgroups. In this example, `C` is a PISA switch NF. Moving `C` to the server enables coalescing the two sub-groups into a single sub-group, freeing up a core that can help make another NF chain feasible, or increase overall marginal throughput.

To make an optimal coalescing decision, Placer needs to consider core allocation, but this can involve an expensive search since other factors (such as NIC link capacity) constrain core allocation. Thus it decouples coalescing from core allocation and uses three simple rules to coalesce sub-groups.

Consider two subgroups `{A->B}` and `{D->E}`. Placer coalesces these only if the resulting marginal throughput from allocating two cores to the coalesced sub-group is higher than allocating one core to each sub-group. We call this *strict coalescing*. However, there are other situations in which coalescing might be beneficial (with appropriate core allocation) because they can free up cores for use by other NF chains, and we consider two. In *aggressive coalescing*, Placer coalesces two subgroups as long as the SLO is not violated; this is aggressive because it can potentially backfire and result in lower overall marginal throughput. In *conservative coalescing*, Placer coalesces two sub-groups only if the chain's throughput does not decrease.

The output of this step is three different placements: the baseline placement, an aggressive placement which applies strict and aggressive coalescing, and a conservative placement which applies strict and conservative coalescing.

3. Maximize marginal throughputs. For each of the three placements, Placer generates core allocations, runs the LP to compute marginal throughput under link constraints, and picks the configuration with the highest marginal throughput.

Dynamics. The placement algorithm runs when an NF chain config changes: *e.g.,* when an operator adds or removes an NF from a chain, or changes an SLO, or updates the traffic aggregate associated with a chain. As we show in §5, Placer is fast enough to handle these dynamics. However, these kinds of changes need additional run-time support to dynamically reconfigure NF chains without impacting traffic; such support is usually found in NFV orchestration frameworks, into which we expect Lemur to be integrated.

4 THE META-COMPILER

Lemur's *meta-compiler* integrates many different execution platforms, each with its own execution model, language(s), and toolchains. It takes as input the NF chain specifications (§2), and automates the entire process of generating and running code for all NF chains. To do this, it parses the NF chain specifications, and develops an intermediate graph representation of all the NFs. In this *NF-graph*, nodes are NFs, links represent data-flows, and each node is associated with attributes that govern placement and other information. The meta-compiler then feeds the NF-graph to Placer to find the highest marginal throughput placement. Using this placement, the meta-compiler synthesizes (a) NF chain routing and (b) NF code generation. These synthesis tasks are aided by the meta-compiler's *library* of NF implementations.

4.1 Synthesizing NF Chain Routing

Given a placement, traffic that matches an NF chain must traverse each of its NFs, across different platforms in the correct order. Lemur must synthesize routing configurations to deliver packets from each NF to the next NF in the NF chain, which may be on a different platform. To do this, we use the Network Service Header (NSH) [35], which tags packets with a service path index (SPI) and service ID (SI); a service path is equivalent to a linear NF chain, and a service ID helps sequence execution of NFs within a single chain.

The meta-compiler's first step, after placement, is to assign SPI and SI values nodes in the NF-graph. Then it needs to synthesize code for routing between NFs in each chain. For this, the meta-compiler pre-defines implementations of two modules for each platform–encap and decap–which respectively add and remove NSH or its equivalent. Having determined the SPI and SI values, the meta-compiler must generate code for each platform to affect the routing between NFs. To minimize encap and decap overhead, Lemur concatenates NFs in a single service path and only generates encap and decap modules at the head and tail of that service path.

For example, consider an NF chain `A->B->C->D`, where *B* and *C* are on a server and *A* and *D* are on the switch. The meta-compiler inserts code to set the initial SPI/SI values in the switch, then inserts code after NF *A* to forward the packet to the server. On the server, *B* and *C* run to completion, but the meta-compiler must insert code to increment the SI value, and, after *C*'s completion, code to route the packet back to the switch. Finally, it must add code to strip NSH after *D*. This example shows a key part of Lemur's design: here, the PISA switch *coordinates* execution of the NF chain via routing. This is natural as all traffic enters/exits the PISA switch ToR.

4.2 Code Generation

The meta-compiler generates code from the built-in NF implementation library. For example, the library might have an `ACL` implementation for the PISA switch and an x86 server, and a `Dedup` implementation only for an x86 server (because PISA switches cannot implement `Dedup`). Using these, the meta-compiler can generate code for the NF chain `ACL->Dedup` as follows: if `ACL` is placed on the PISA switch, it generates the appropriate routing PISA code as above, prepends it to `ACL`'s PISA implementation, and performs similar steps for the x86 `Dedup` code. This is conceptually easy but complicated by the platforms: a Barefoot Tofino-based programmable switch and x86 commodity servers running BESS [2].[5]

Synthesizing P4 NF chains. PISA switches are programmed using P4 [3] with *monolithic programs* for packet processing. However Lemur requires composability of NF chains in P4. To enable this, programmers must be able to write *standalone* P4 NFs that can then be composed into NF chains. Rather than invent a new language, we minimally extended P4's syntax to allow users to specify standalone NFs. We also developed an associated pre-processor to the meta-compiler that parses these extensions. The following paragraphs describe our extensions to the P4 syntax, and the pre-processor.

Defining standalone P4 NFs. In Lemur, NF-developers can write a *standalone P4 NF* in much the same way as they write a regular P4 program: by defining headers, per-packet metadata, header parser specification, match/action tables, and control flow of the packet processing pipeline. Lemur makes small changes in the way programmers specify headers, metadata and parsers.

For each P4 NF to be standalone, the NF developer cannot know the actions a packet will be subject to after the NF is processed. Lemur assumes that the NF will pass all packets to the next NF in the chain. However, the programmer can set the `drop_flag` in metadata to ensure that a packet is *not* passed to the next NF. This is useful in implementing firewalls.

One key feature of P4 is that it is protocol-independent. When Lemur aims to unify standalone P4 NFs, it must ensure agreement on how to parse headers. Hence for each P4 NF, programmers must specify headers and header layouts, and then specify how these headers are parsed. An NF-developer may not know a priori all the headers and their associated layouts; this is only known after placement is finalized. So an NF-developer must specify headers and parsers in a manner amenable to composability: the meta-compiler must be able to combine header parsers of P4 NFs when generating code for a set of NF chains. To achieve this, Lemur provides an interface for NF-developers to describe headers and parsers. It provides a library of predefined headers (along with their layouts). NF-developers may extend this library. When writing a

[5]The meta-compiler supports eBPF on Netronome's Agilio CX 1x40 Gbps SmartNIC; the code generation technique described above suffices.

P4 NF, they simply list the headers they wish to use, and describe an *NF-local parser* via a simple graph definition language.

Composing P4 NFs into chains. After Placer runs, Lemur reads in the P4 NF modules and merges them together as a single unified P4 program. In addition to name mangling P4 NFs to ensure uniqueness, and eliminating redundant headers, the meta-compiler implements two key algorithms.

The first algorithm auto-generates the unified parser from the NF-local parsers, specified by the NF developer, by *merging* the NF-local parsers: it takes the union of the next header choices for each unique header in a parse tree. The meta-compiler then auto-generates the headers and the parser definitions, using layouts from the header library (see §A.2.1).

The meta-compiler must also assemble the per-NF tables and actions into a global sequence of tables and actions consistent with dependencies between NFs in the NF chain definitions. One naive solution is to generate code for NFs in a topological-sort order, and place a check at the beginning of each NF. However, P4 programs generated in this manner can waste many switch stages. Our experience with resource mappings for many compiled P4 programs points to two important facts that must be considered when generating a unified P4 pipeline: (1) (no loop) a match/action table cannot be revisited in the pipeline; consider a merge at the end of a branch, where NFs *A* and *B* merge into *C*. The tables from *C* must be applied in the unified pipeline exactly once, and after all tables from *A* and *B*. (2) two match/action tables cannot be packed on the same stage if they have dependencies between them. Therefore, the challenge is to convert a DAG of NFs (with many branching and merging points) into a tree struct that must respect all dependencies in the original NF DAG and must not introduce unnecessary dependencies between NFs. The meta-compiler statically analyzes the NF-graph for these dependencies and generates tables with this property (details in §A.2.2).

Resource-Aware Code Generation. The generated NF code must conserve constrained resources on our platforms: PISA pipeline stages, and server cores.

Minimizing PISA switch stage usage. Of the various constraints (DRAM, TCAM, matching bits, stages), the number of stages is most constraining (it is the constraint that is easiest to violate). As previous studies [17] have shown, in a P4 switch table dependencies can rapidly consume available stages. Therefore, we optimized switch stage usage by eliminating table dependencies ([17] uses similar techniques for standard P4 programs). These optimizations use a static analysis of the NF chain graph, similar to the one described above, and execute the following assertions: (a) Do not generate code to insert an NSH header if a chain is placed by Placer entirely on the PISA switch; (b) Instead of updating the SI values after each P4 NF, update it once at the end of a chain of sequential NFs; (c) To steer packets returning from the server to the correct next NF in the chain, incorporate the steering into the first switch stage which also steers previously unseen packets; and (d) Allow the P4 compiler to pack parallel branches into the same set of switch stages by expressing the exclusivity among these branches explicitly in the generated P4 code.

Chain	Specification
Chain 1	`BPF->`*Subchain 7*`->BPF->UrlFilter->`*Subchain 8* ↘ *Subchain 8* ↘ *Subchain 8*
Chain 2	`Encrypt->LB->`*3x*`NAT(branched)->IPv4Fwd`
Chain 3	`Dedup->ACL->Limiter->LB->IPv4Fwd`
Chain 4	`Dedup->ACL->Monitor->Tunnel->BPF->` *3xSubchain 6 (branched)* `->IPv4Fwd`
Chain 5	`ACL->UrlFilter->Fast Encrypt->IPv4Fwd`
Subchain 6	`LB->Limiter->ACL`
Subchain 7	`ACL->Limiter`
Subchain 8	`Detunnel->Encrypt->IPv4Fwd`

Table 2: Five canonical NF chains used for evaluation.

Codegen for BESS packet steering and NF scheduling. BESS is a DPDK-based software switch that supports standalone NFs and NF-chaining, so we did not need to extend the BESS programming model (see also §A.1.1).

Placer determines NF run-to-completion subgroups and how many cores are allocated per subgroup. The meta-compiler must generate code to demultiplex packets to the right subgroup and further the right subgroup instance. This demultiplexer module also decapsulates NSH headers because BESS NF implementations aren't aware of this header; an auto-generated multiplexer module at the end re-inserts this header. In Lemur, the demultiplexer runs on a single core, pulls packets from the NIC, and steers packets to the subgroup (§A.1.2). This incurs cross-core communication costs; in future work we intend to generate PISA switch code to tag and steer packets to specific cores as in Metron [18].

Finally, the meta-compiler uses BESS's scheduler, which supports hierarchical scheduling policies via a per-core tree with NF leaves and policy interior nodes. Given subgroups and core allocations, the meta-compiler specifies NF scheduling. Placer might choose to allocate multiple subgroups to the same core, and the meta-compiler generates code to schedule these subgroups round-robin. We also use the scheduler to enforce $t_{\max}$. (More discussions in §A.1.3)

5 EVALUATION

We compare Lemur against several other alternatives, and illustrate features of Lemur's design.

5.1 Methodology

Implementation. Our Lemur implementation has three key pieces: 1) NF implementations in C, C++, and P4, 2) the Placer, and 3) the meta-compiler. Our NFs require 1396 lines of C++ (new BESS modules), 412 lines of C (eBPF code to run on the SmartNIC), and 1273 lines of P4 (P4 libraries). The Placer consists of 841 lines of Python. The meta-compiler consists of 6450 lines of Python composed of 2564 lines for the parser core, 312 lines for the BESS code generator, 3142 lines for the P4 code generator, 434 lines for OpenFlow, and 120 lines of ANTLR to parse NF chain specifications.

Experiment setup. Most of our experiments use two servers connected to a PISA switch functioning as a ToR. Both servers run BESS, one as a traffic generator and the other for NFs. Our PISA hardware

NF	Spec	C++	P4	eBPF	OF
Encrypt	128-bit AES-CBC	•			
Decrypt	128-bit AES-CBC	•			
Fast Enc.	128-bit Chacha	•		•	
Dedup	Network RE [1]	•			
Tunnel	Push VLAN tag	•	•	•	•
Detunnel	Pop VLAN tag	•	•	•	•
IPv4Fwd	IP Address match	•	•	•	•
Limiter	Token bucket	•			
Url Filter	HTML Filter	•			
Monitor	Per-flow statistics	•			•
NAT	Carrier-grade NAT	•	•		
LB	Layer-4 load balance	•	•	•	
Match	Flexible BPF Match	•	•	•	
ACL	ACL on src/dst fields	•	•	•	•

Table 3: NFs and available placement choices. We artificially limit IPv4Fwd as P4-only for the sake of evaluation.

is an Edgecore 100BF-32X with a Barefoot Tofino switching chip with 32x100G ports. The traffic generator is a dual-CPU 40-core 2.2 GHz Xeon E5-2630 with one Mellanox 100Gbps MCX515A-CCAT NIC. The BESS server for Lemur is a dual-CPU 8-core 1.7 GHz Xeon Bronze 3106 with one 40Gbps single-port XL710 Intel NIC. In some experiments, we use a Netronome Agilio CX 1x40 Gbps NIC or Edgecore AS5712-54X OpenFlow switch.

NFs and NF chains. Our experiments use five different canonical chains, shown in Table 2. These represent a range of use cases selected from [21] and from our discussions with ISPs. These canonical chains are composed of numerous NF implementations across the three platforms for which we have developed Lemur thus far. We include a summary of each network function, its corresponding implementation, and the placement choices available in Table 3. Two NFs, in bold, cannot be replicated across multiple cores.

Comparison. We compare Lemur, which runs the heuristic placement described in §3.2, against alternative strategies. Each of these alternatives corresponds to approaches described in prior work. *Optimal* runs the brute-force placement algorithm. *HW Preferred* places as many modules as possible on the PISA switch, which models the preferential use of accelerated hardware [27]. *SW Preferred* places all NFs with software implementations in software (BESS), which models the preferential deployment of NFs on commodity servers with kernel-bypassing techniques [33]. *Minimum Bounce* minimizes the bounces between the switch and servers, emulating prior work (*e.g.,* using Kernighan-Lin in E2 [32]). *Greedy* selects HW-preferred chain placement and allocates cores to first meet the minimum rate requirement of each chain; once the minimum requirement is satisfied, it greedily allocates spare cores to chains sequentially by index. Once a chain's maximum rate is reached, it moves on to the next chain to allocate spare cores, possibly causing the first chain to take resources that another chain would need to achieve a more globally-ideal allocation. Our evaluation goal is to show that an approach which holistically considers both accelerators and commodity servers, and trades off traffic bounces when necessary to accommodate more chains, can do much better than approaches that focus on a single dimension.

Experiment Design. The input to our experiments is a collection of chains, together with an SLO for each chain. The space of possible SLOs is large. We systematically explore part of this space as follows. For each chain, we first define its *base rate* as the rate it would achieve if only one core were allocated to the slowest software NF in the chain. Then, we perform experiments in which each chain's $t_{\min}$ is set to δ times the base rate. We vary δ from 0.5 to 4.0, in steps of 0.5. As δ increases, it becomes harder for schemes to satisfy SLOs, since they need to either allocate more switch resources or cores. In all experiments, we set $t_{\max}$ to be 100 Gbps.

Metrics. For each experiment, we first compute the placement generated by Lemur and the other schemes and then use the meta-compiler to generate code. Thereafter, we execute the NF chain configuration on the testbed, but only when the placement is *feasible* (*i.e.,* meets SLOs). We measure and report the *aggregate throughput* achieved, from which we can derive the *aggregate marginal throughput* of each scheme.

5.2 Comparison Results

We test Lemur against alternatives with (subsets of) Chains 1-4 in Table 2 (we use the fifth for Smart NIC experiments).

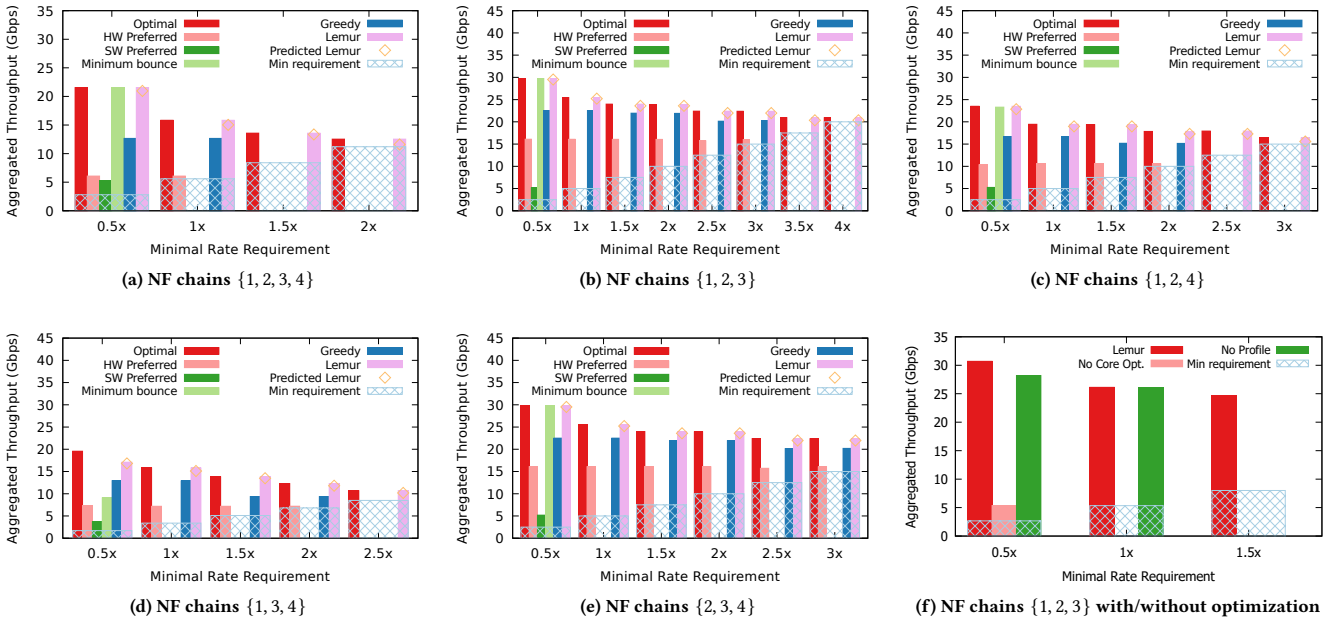
Overall results. Figure 2 compares Lemur performance with the alternatives. These graphs show δ on the x-axis and aggregate throughput in Gbps on the y-axis. Each scheme is shown by a vertical bar, and the *aggregate* $t_{\min}$ is shown by a hashed blue rectangle for each value of δ. The difference between each vertical bar and the top of the hashed rectangle is its aggregate marginal throughput. The absence of a vertical bar for a scheme for a given δ indicates that the scheme *could not generate a feasible solution* at that δ.

Figure 2(a-e) show experiments for different chain combinations: all four of chains 1-4, and all 3-chain combinations of these 4 chains. In all experiments, as δ increases, Lemur is the *only one that produces a feasible solution.*

Comparison with Optimal. Moreover, across all experiments, Lemur's heuristic is able to find an SLO-satisfied solution for all 29 sets, matching the brute-force placement. In addition, Lemur achieves the same marginal throughput as Optimal in all but one of the experiments; in that one case, Lemur still outperforms all other alternatives.

Moreover, as δ increases, the total aggregate throughput of the chains decreases. This is because all chains place greater demands to meet their minimum rates and some chains are significantly more expensive than others; as a result, increasing δ forces the reallocation of resources towards expensive chains (in order to meet their SLOs) and away from faster chains that could have delivered aggregate throughput gains.

Four chain experiment. In Figure 2a Lemur performs better than the alternatives as it frees up and then uses spare cores to meet chain SLOs; there are either insufficient cores or insufficient switch pipeline stages for other schemes that waste cores on chains that will overshoot their SLOs while failing to meet the SLOs for others. At a δ of 0.5, all approaches find feasible solutions, but Lemur has the



(a) NF chains $\{1, 2, 3, 4\}$ **(b) NF chains** $\{1, 2, 3\}$ **(c) NF chains** $\{1, 2, 4\}$

(d) NF chains $\{1, 3, 4\}$ **(e) NF chains** $\{2, 3, 4\}$ **(f) NF chains** $\{1, 2, 3\}$ **with/without optimization**

Figure 2: Performance comparison of alternative schemes and of Lemur with toggled optimizations.

highest marginal throughput. By δ 1.0, only the Greedy approach and HW Preferred approach are able to compete with Lemur. At δ of 1.5, Lemur is the only approach that provides a feasible solution. The reasons for these are varied, and somewhat nuanced, and are better illustrated by our 3-chain experiments.

Three chain experiments. In 3-chain configurations (Figures 2b–2e) we find that Lemur consistently provides higher marginal throughputs than the alternatives, and finds feasible solutions at higher δ even when other alternatives cannot.

Minimum Bounce. Minimum bounce provides comparable marginal throughput to Lemur for low values of δ, but beyond a δ of 1.0, it fails to find a solution. This is because it is unwilling to move an intermediate NF to P4 as it attempts to avoid bounces. Adding the bounce might allow an NF to use P4, freeing up server resources to satisfy SLO.

HW Preferred. HW Preferred delivers the same rate regardless of δ because it maximizes P4 processing, and otherwise allocates spare cores evenly among chains. While effective at lower δ values, it fails once the SLO for a slower chain cannot be satisfied because of insufficient cores. In both the 4-chain and the 3-chain experiments, the HW Preferred solution fits in the switch; below we discuss an example where alternatives exceed switch stage limits, but Lemur does not.

SW Preferred. SW Preferred fails to scale because all NFs are in one subgroup, and we do not replicate stateful NFs or branch/merge NFs. So, SLOs cannot be satisfied even at low δ. Lemur, *though it is subject to the same replication constraints* is able to find a solution with high marginal throughput.

Greedy. Greedy performs quite well in all our experiments as it uses hardware when possible and attempts to meet the minimum SLO using differential core allocation across chains. Greedy differs from HW Preferred as it does not evenly distribute cores to chains but instead does so preferentially to meet SLOs using Lemur's profiling (§3.2). Even so, it fails to find a feasible placement at higher values of δ when Lemur can. The reason is subtle: while Greedy is the only one of our alternatives that targets SLOs, it starts with a HW Preferred placement instead of a full exploration. Thus Greedy may fail to satisfy SLOs because of a lack of cores. Consider a chain `A->B->C->D->E`, where B and D are on the server, the rest on the switch. Greedy is forced to allocate one core each to B and D, while Lemur can potentially place C on the server and allocate a core to the subgroup `B->C->D`. (Our heuristic implements such optimizations.)

Specific chains. In Figure 2b all schemes deliver higher rates and meet higher δ requirements since this experiment omits Chain 4, which, as seen in Table 2, is complex.

In NF chains $\{1, 2, 3\}$ and NF chains $\{1, 2, 4\}$, when $\delta = 0.5$, we note that Minimum bounce is competitive to Lemur, but in NF chains $\{1, 3, 4\}$ it underperforms. This is due to Chain 1, for which Minimum Bounce tries to minimize the number of bounces although it is possible to move some modules to P4. Under the same constraint as discussed in §3.2, Lemur finds that one more bounce of Chain 1 is beneficial and enables an expensive subgroup to be replicated, which allows spare cores to be allocated to faster chains. Instead, Minimum Bounce is unwilling to trade the bounce and it allocates the cores to slow chains, with lower total throughput.

Comparison Summary. Across all experiments, Lemur can **always** find a feasible solution while other approaches only do 17-76%

of the time. Moreover, overall, Lemur obtains a marginal throughput lead ranging from 500 Mbps to nearly 24 Gbps (at the latter end, more than 50% of link capacity).

Profiling and Performance prediction. In all figures in Figure 2, we show the predicted aggregate throughput as a ⋄ above the Lemur bar. This prediction is the sum of estimated rates (§3.2) of all chains. In general, the *predicted throughput closely matches the measured throughput*; in this section, we explore why the match is close, and when it is not perfect.

Predictions are conservative. This prediction depends on profiling, and is not always perfect. For example, Greedy, which uses profiles, exhibits non-monotonic aggregate throughput for NF chains $\{1, 2, 4\}$ for $\delta = 1$ and $\delta = 0.5$. When we profile an NF, we pick the worst-case cycle count reported by BESS. In some runs, NFs see lower cycle counts and therefore higher rates. In this case, the predicted aggregate rate for both values of δ were the same, but $\delta = 1$ saw lower cycle counts and a higher rate in our experiments.

Cross-socket costs. Our profiles assume worst-case cross-socket costs. Our server has 2 sockets, and the NIC is connected to one of them. If a subgroup is replicated on cores on the same socket as the NIC, our measured rates will be higher than predicted; this occurs in most experiment sets.

Data-dependent NFs. Performance prediction may be inaccurate for an NF like `dedup`. This does not occur in our experiments, but this NF is interesting in two ways: (a) the number of cycles to process a packet can vary due to packet contents; and (b) the NF's packet egress rate is less than its ingress rate. We leave exploration of this to future work.

The stability of profiled cycle costs. Table 4 shows the statistics of cycle costs for several NFs, across 500 profiling runs.[6] In general, these profile costs are extremely stable, with the worst-case cycle cost being within 6.5% of the average cycle cost. This is surprising, but explains why our predicted throughput matches the measured throughput so well. To understand this better, we tried to understand the effect of under-estimating the cycle costs. We conducted an experiment in which we reduced the profiled costs by a fraction, ranging from 1% to 10%, mimicking errors in profiling. *We found that, even with these errors, Lemur produces a configuration with the same aggregate marginal throughput as the baseline, up to 8% errors.* The stability of the cycle costs, and the relative insensitivity of Lemur to errors, is encouraging and explains why our predicted throughputs match measured throughputs.

Cache effects. ResQ [41] examined NF cache effects and showed NF profiling can be cache sensitive, especially if packet queues before NFs are large and are shared across NFs. In Lemur, we have experimentally verified that our queues are short; in this setting

[6]We generate traffic in two ways that exercise worst-case NF behavior. For NFs that perform poorly with long-lived traffic, we generated 30-50 uniformly distributed long-lived flows. For NFs that perform poorly with short-lived flows, we generated 3.2 mpps of traffic with 10,000 new flows/sec each lasting 1 second.

NF	NUMA	Mean	Min	Max
Encrypt	Same	8593	8405	8777
Encrypt	Diff	8950	8755	9123
Dedup	Same	30182	29202	30867
Dedup	Diff	31188	29969	33185
ACL (1024 rules)	Same	3841	3801	4008
ACL (1024 rules)	Diff	4020	3943	4091
NAT (12000 entries)	Same	463	459	477
NAT (12000 entries)	Diff	496	491	507

Table 4: Example profiled NF costs (CPU cycles/packet)

[41] shows that NF profiling variability can be bounded to within 3%, consistent with our findings.

An extreme configuration: P4 stage constraints. So far switch stage constraints are implicit since link and core constraints dominate. Next we consider an extreme NF chain configuration that causes the switch to run off stages. This is a variant of Chain 2 without encryption: `BPF->11 NAT(branched)->IPv4Fwd`, and we choose $\delta = 0.5$ for which we expect the chain minimum rate requirement will be about 44.9 Gbps. Here SW Preferred fails to satisfy SLOs while all other alternatives exceed the number of switch stages. By contrast, Lemur finds a feasible solution, placing 10 of the `NAT`s in the switch, and one on the server.

This illustrates the importance of using the P4 compiler to compute stage usage. Initially, we attempted to estimate stage usage by analyzing placement results, using a recent work technique [14]. But, such estimates were very conservative. For the 10 `NAT` placement, it estimated 14 stages, while the compiler could fit these into 12 stages using internal black-box optimizations. This shows the importance of our dependency elimination algorithms for stage compaction (§4.2); without it, the 10 `NAT` placement would have required *27* stages.

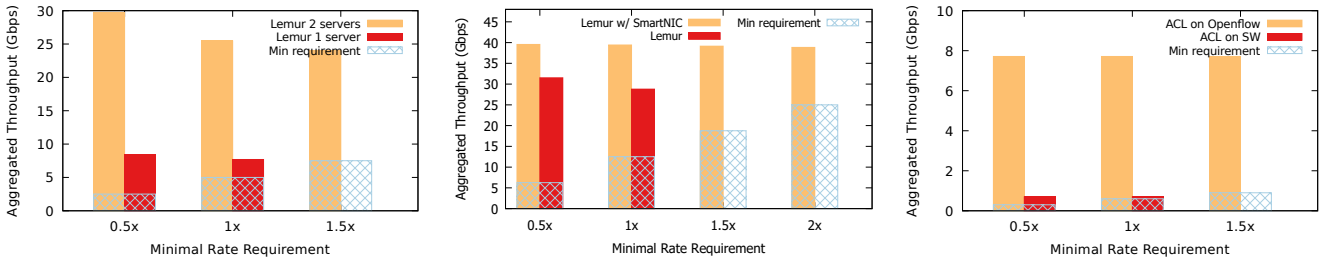
5.3 Other Experiments

Importance of Lemur Components. Lemur uses both NF profiling and subgroup scaling with core allocation to meet SLOs. Figure 2f considers removal of each feature in turn.

No Profiling. Here we assume all NFs have the same cycle cost. Because this variant is unable to distinguish between expensive and cheap NFs, it generally has lower marginal throughput, and becomes infeasible for higher values of δ because it needlessly gives cores to cheap NFs.

No Core Allocation. Here we assign no extra cores to scale subgroups; this variant can only satisfy SLOs at $\delta = 0.5$.

Placement across multiple servers. Above we evaluate with a 16-core server with a single NIC, but this is not a limitation for Lemur. Lemur can reason about multi-server placements. To show this, we run an experiment using NF chains $\{1, 2, 3\}$ where Lemur is used to place the 3-chain set on (a) a single 8-core server, and (b) two 8-core servers. Figure 3a shows that, when $\delta = 0.5$, the single server gets less than half the aggregate throughput of the 2-server experiment. However, for a very subtle reason, when $\delta = 1.5$, Lemur cannot find a feasible solution for the single serve case. Chain 3 contains the following sub-chain `Dedup->ACL->Limiter`. The base rate of this chain is bottlenecked by `Dedup`. When $\delta = 0.5$, Lemur



(a) Two servers (b) SmartNIC (c) OpenFlow switch

Figure 3: Performance comparison of Lemur running with different hardware and on multiple servers.

is able to allocate one core to the subgroup `Dedup->ACL->Limiter`, because the additional modules `ACL` and `LimiterS` are relatively lightweight, so one core can satisfy their SLO. However, when $\delta = 1.5$, this subgroup assignment can no longer satisfy the SLO, and Lemur, to satisfy the chain's SLO (a) offloads `ACL` to the switch, (b) replicates `Dedup` on two cores, and (c) allocates a 3rd core for `Limiter` which is non-replicable. It thus runs out of cores in the 1-server case.

Placement on a SmartNIC. Lemur can accelerate NFs across multiple types of hardware. To demonstrate this, Figure 3b shows an experiment on Chain 5, which includes the ChaCha NF [30, 38]. ChaCha cannot be implemented in the P4 switch but can be in the SmartNIC and on x86 servers. Our SmartNIC implementation (which uses eBPF on a 40G Netronome NIC) is more than 10× faster than on the server. In Figure 3b, Lemur is able to achieve higher aggregate throughput at lower δ by offloading ChaCha to the SmartNIC. At $\delta = 1.5$, Lemur cannot produce a server-only solution since the $t_{\min}$ is too high for a server (even with multiple cores). This shows that Lemur can achieve close to the line rate of 40 Gbps, lower only due to NSH header overhead.

Placement on an Openflow switch. OpenFlow switches are ubiquitous today, unlike PISA switches. We show how Lemur can use an OpenFlow switch in place of a PISA switch. Unlike a PISA switch, an OpenFlow switch has fixed table order, so the Placer must check whether a configuration violates the switch table order to execute NFs. Also, Openflow switches do not support NSH; Lemur uses VLAN in its place (specifically, the 12-bit vid field as SPI-SI to demultiplex packets for different subgroups). This somewhat limits how many chains and how many NFs can be configured, but using Lemur with an OpenFlow switch still provides overall benefits. To demonstrate how an Openflow switch can accelerate NFs, we compare offloading `ACL`, or not, to an OpenFlow switch on chain 3, as shown in Figure 3c; this can accommodate up to 7710 Mbps traffic for that chain, while stitching `ACL` via a commodity server can only achieve 693 Mbps.

Lemur decouples the performance optimization and the code generation from the actual deployment of NF chains. This makes it extensible to other hardware platforms as well.

Adding latency constraints. Lemur can reason about latency constraints, which are encoded into our LP formulation (§3.2). The switch vendor's EULA disallows reporting latency details, but we show a single experiment in which we model chain latency using (a) propagation and transmission latency between switch and server, and (b) NF execution delay. Here we used Chain 1 and Chain 4, and assigned each a latency constraint of 45 μs. This allows Lemur to increase marginal throughput at the expense of additional bounces between the server and switch, and for this case we get over 21 Gbps. When we constrain the latency to 25 μs, Lemur is forced to reduce the number of bounces and can only achieve 9 Gbps.[7]

Meta-compiler Benefits and Overhead. The meta-compiler automates coding tasks that would otherwise have to be performed by a system administrator. We quantify this benefit by counting the lines of code auto-generated by Lemur. The most significant code generation component is for P4, and for NF chains {1, 2, 3, 4} more than a third of the total code (about 820 out of 1700 lines) is auto-generated, with most of the auto-generated code (600 lines) providing packet steering.

Flexible NF-chain composition comes at a cost which takes two forms: additional stage usage in P4, and additional cycle costs in BESS. We have to burn two P4 stages, one each to encapsulate and decapsulate packets. Our BESS cycle cost overheads for these are modest at about 220 cycles. The server also incurs about 180 cycles to load-balance packets when a subgroup is allocated to multiple cores. These overheads are a small fraction of NF cycle costs and of the coordination overheads imposed by any framework or virtual switch.

Scaling Placer Computation. Brute-force placement is slow; for the 4-chain case (34 NF instances in total) it takes 14901 seconds (~4 hours). Our heuristic is far faster, taking 3.5 s for the 4-chain case, motivating our careful design.

6 RELATED WORK

Lemur builds upon prior work but is unique in meeting SLOs while targeting diverse hardware platforms.

NFV frameworks. NFV frameworks [5, 12, 31] help develop, chain, execute, and deploy NFs. NetBricks [33] provides abstractions to write NFs, and a fast, safe runtime implemented in Rust. NFP [40] enables parallelism for NFV by avoiding NF-dependencies. E2 [32]

[7] Sources of latency include DPDK and switch queueing, and encap/decap overheads. The 9 Gbps drop occurs because the higher throughput placement violates latency SLO (due to multiple bounces), so Lemur picks an SLO-compliant alternative with lower throughput.

demonstrates the potential of orchestration of NF chains using commodity servers. We see Lemur as a key future component of such frameworks.

vNF placement and orchestration. Most relevant to our work is SmartChain [43], which explores the placement of vNFs between a smartNIC and CPU cores on a server. It focuses on optimizing one NF-chain's latency on one machine. While this is important, Lemur tackles a more practical deployment scenario, *i.e.,* many chains deployed in a cluster where a switch interconnects many servers with or without a smartNIC. Moreover, Lemur's Placer can find a placement with bounces that satisfy both throughput and latency SLO requirements. SmartChain, however, ignores the possibility of utilizing the full capability of hardware, and allows only one NIC-CPU transmission. Other work on vNF placement [7, 13, 23, 34] solves an optimization problem targeting a general deployment scheme. Lemur's heterogeneous-hardware architecture and run-to-completion execution are novel in this context, and our model reflects practical deployment constraints, optimizing for throughput while meeting latency SLOs. Of these, Cziva *et al.* [7] and Laghrissi *et al.* [23] consider a case where user mobility affects the end-to-end NF-chain latency. The former proposes a predictive placement algorithm. The latter proposes a dynamic-migration algorithm that minimizes the latency violations and the number of vNF migrations. Lemur targets a different setting where mobility is less of a concern: packet processing at the ingress to a cellular provider's backhaul network. In [13], Gouareb *et al.* model the end-to-end latency by considering both the link delay and the inter-cloud delay. They assume NFs are hosted on separate clusters. Lemur focuses on latency SLOs within the cluster, since that is what a service provider can control. Finally, in [34], Pham *et al.* model and optimize for the system operational cost of running an NFV infrastructure. While the objective is different, Lemur addresses this problem from a different perspective: it utilizes hardware accelerators to reduce the number of commodity servers.

Hardware acceleration. UNO [24] offloads NFs to a Smart NIC on a single host to save CPU time and energy. Metron [18] shows how switch hardware can steer traffic to specific CPU cores on servers running NFs, thereby avoiding cross-core costs; Lemur can easily be augmented to include this as discussed in §3.2. Lemur is distinguished by its focus on meeting SLOs across chains and hardware platforms.

Several accelerate NFV platforms using high-end networking hardware including GPUs [20, 45], FPGAs [25], and programmable switching chips [11, 27] for specific types of NFs. Lemur generalizes this line of work by considering placement across heterogeneous hardware, but is also unique in considering SLOs. Sonata [14] explores hardware acceleration for accurate network measurement; by contrast, Lemur is a generic framework for accelerating NFs. Several other papers have also demonstrated the power of P4 [11, 19, 27] in accelerating specific network functions, and motivate our focus on offloading NF execution to programmable hardware.

Only two prior papers meet SLOs for NF chains with new hardware. ResQ uses Intel's CAT to help meet SLOs [41]. Lemur targets macro-level SLOs across NF chains on many platforms, and can use ResQ's techniques. Grus [47] lowers latency for NFs on GPUs given a bound; in future work we can extend Lemur to GPUs using techniques from Grus.

7 DISCUSSION

Dynamics. Lemur must deal with two different types of dynamics. One is customers' traffic dynamics. ISPs sign service level agreements with their customers to offer a certain bandwidth and if permitted, they are willing to allow their customers to add more bandwidth to burst above the minimal SLO requirement. If customers do not pay an additional fee to increase their bursty bandwidth, the ISP would apply a rate limiter to control the traffic volume into the service chain, and hence Lemur assumes that there is a rate limiter before the traffic for each service chain. Because Lemur provisions for rate limit, it is immune to this type of dynamics. A second form of dynamics is customers requesting higher bandwidth for their service chains for which our heuristics can easily compute a new placement, and we assume we can adopt other well-known techniques (*e.g.,* Metron [18], OFM [39]) for NF migration.

Lemur's SLO model matches today's ISP pricing models which provide either fixed pipes, or pipes with guaranteed minimum rates and a variable pay-as-you-go burst. So, today customers have no way of optimizing their costs based on diurnal variations. If, in the future, ISPs provide time-varying SLOs (*e.g.,* minimum rate of x between 10am and 4pm), Lemur can precompute chain placements for those SLOs and install them accordingly.

Failures. Lemur leverages on-path hardware (PISA switches, NICs), so if those fail, it will have to re-route traffic to another rack or server. If the new path does not have enough hardware offload resources, Lemur can always fall back to using server-based NFs. Its Placer can make these decisions either reactively (after failure), or proactively (perhaps by reserving some spare capacity to ensure fast failover).

8 CONCLUSION

Lemur tackles a key problem in NFV deployments: meeting SLOs while leveraging available hardware. Its Placer considers several competing objectives to increase marginal throughput, while its meta-compiler coordinates NF chain execution seamlessly across several platforms. Lemur outperforms several other alternatives, and its placement computation scales well. Its structure enables it to both integrate recent work on NF scaling, and to integrate into existing frameworks.

ACKNOWLEDGEMENTS

This work was supported in part by US National Science Foundation grant CNS-1901523, the CONIX Research Center (one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA), the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) grant 205154/2018-2, USC's Annenberg Fellowship, Intel, and Cisco.

REFERENCES

[1] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. EndRE: An End-system Redundancy Elimination Service for Enterprises. In *Proceedings of USENIX/ACM NSDI*, 2010.

[2] Berkeley Extensible Software Switch. https://github.com/NetSys/bess, 2019.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[5] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of ACM SIGCOMM*, 2016.

[6] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[7] Richard Cziva, Christos Anagnostopoulos, and Dimitrios P Pezaros. Dynamic, latency-optimal vnf placement at the network edge. In *Ieee infocom 2018-ieee conference on computer communications*, pages 693–701. IEEE, 2018.

[8] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation*, pages 224–231, June 2010.

[9] Abdessalam Elhabbash, Assylbek Jumagaliyev, Gordon S Blair, and Yehia Elkhatib. Slo-ml: A language for service level objective modelling in multi-cloud applications. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 241–250, 2019.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[11] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2015.

[12] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of ACM SIGCOMM*, 2014.

[13] Racha Gouareb, Vasilis Friderikos, and Abdol-Hamid Aghvami. Virtual network functions routing and placement for edge cloud latency minimization. *IEEE Journal on Selected Areas in Communications*, 36(10):2346–2357, 2018.

[14] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In *Proceedings of ACM SIGCOMM*, 2018.

[15] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of ACM SIGCOMM*, 2017.

[16] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 517–530, Boston, MA, February 2019. USENIX Association.

[17] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *Proceedings of USENIX/ACM NSDI*, 2015.

[18] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *Proceedings of USENIX/ACM NSDI*, 2018.

[19] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, 2016.

[20] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[21] Surendra Kumar, Mudassir Tufail, Sumandra Majee, Claudiu Captari, and Shunsuke Homma. Service Function Chaining Use Cases In Data Centers. Internet-Draft draft-ietf-sfc-dc-use-cases-06, Internet Engineering Task Force, February 2017. Work in Progress.

[22] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. Deploying chains of virtual network functions: On the relation between link and server usage. In *Proceedings of IEEE INFOCOM*, 2016.

[23] Abdelquoddouss Laghrissi, Tarik Taleb, Miloud Bagaa, and Hannu Flinck. Towards edge slicing: Vnf placement algorithms for a dynamic & realistic edge cloud environment. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[24] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: unifying host and smart NIC offload for flexible packet processing. In *Proceedings of ACM SoCC*, 2017.

[25] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of ACM SIGCOMM*, 2016.

[26] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 160–161. IEEE, 2007.

[27] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of ACM SIGCOMM*, 2017.

[28] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, KK Ramakrishnan, and Timothy Wood. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, 2015.

[29] Barefoot Networks. The world's fastest & most programmable networks. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[30] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

[31] Network Operators. Network functions virtualization, an introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow SDN and OpenFlow World Congress*, 2012.

[32] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of ACM SOSP*, 2015.

[33] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *Proceedings of USENIX/ACM OSDI*, 2016.

[34] Chuan Pham, Nguyen H Tran, Shaolei Ren, Walid Saad, and Choong Seon Hong. Traffic-aware and energy-efficient vnf placement for service chaining: Joint sampling and matching approach. *IEEE Transactions on Services Computing*, 2017.

[35] Paul Quinn, Uri Elzur, and Carlos Pignataro. Network Service Header (NSH). RFC 8300, January 2018.

[36] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[37] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 329–343, 2020.

[38] Speeding up and strengthening HTTPS connections for Chrome on Android. https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html, 2014.

[39] Chen Sun, Jun Bi, Zili Meng, Xiao Zhang, and Hongxin Hu. Ofm: Optimized flow migration for nfv elasticity control. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[40] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of ACM SIGCOMM*, 2017.

[41] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *Proceedings of USENIX/ACM NSDI*, 2018.

[42] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), February 2020.

[43] Shuhe Wang, Zili Meng, Chen Sun, Minhu Wang, Mingwei Xu, Jun Bi, Tong Yang, Qun Huang, and Hongxin Hu. Smartchain: Enabling high-performance service chain partition between smartnic and cpu. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[44] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *Proceedings of USENIX/ACM NSDI*, 2018.

[45] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU Sharing in NFV Systems. In *Proceedings of USENIX/ACM NSDI*, 2018.

[46] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A closer look at nfv execution models. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 85–91, 2019.

[47] Zhilong Zheng, Jun Bi, Haiping Wang, Chen Sun, Heng Yu, Hongxin Hu, Kai Gao, and Jianping Wu. Grus: Enabling latency slos for gpu-accelerated nfv systems. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 154–164. IEEE, 2018.

A META-COMPILER

In this section, we include hardware and software switch implementation details to reproduce Lemur's experiments.

A.1 x86-based commodity server

A.1.1 BESS script generation. Lemur's chain specification is inspired by BESS' script language. When specifying a pipeline, the user of BESS writes simple languages to concatenate NFs with arrows, and our Lemur user-level configuration adopts BESS script language style with small revisions. As such, Lemur can use BESS's parser with two small modifications, described below.

Instance Name parsing BESS allows users to define several module instances that belong to the same module class, and our configuration file language also supports instance naming convention. Lemur user can declare several instance names to represent multiple BESS module instances. For example, there is an access control (ACL) module class, and users can define an 'ACL0' instance that uses ACL module class. Analogously, Lemur users are allowed to create instance name 'ACL0' for ACL NF. Lemur also supports macro definitions for arguments for module creation. To support both of these, we added functionality to the parser.

A.1.2 Shared Modules. In a BESS pipeline, there are some modules that are shared by all contiguous subgroups. Specifically, 'PortInc' and 'PortOut' modules are used to pull and push packets from the NIC in poll mode. Similarly, all packets are required to decapsulate the NSH header and be distributed to a corresponding contiguous subgroup for further processing, and for this we introduce a custom 'NSHdecap' module into each BESS pipeline, to be shared by all contiguous subgroups. Before pushing packets to the NIC, packets are required to be encapsulated with the NSH header again to indicate the downstream module in another platform what next NF processing should be applied. Hence, the final step to wrap up a subgroup processing in BESS is to use a custom 'NSHencap' module to tag the next service path index and service index pair.

A.1.3 Core Assignment. Given the placement solution returned from Lemur Placer, BESS code generator automatically translates the optimization result to manage the pipeline scheduler. BESS's scheduler is responsible for managing the execution of modules to process traffic in a whole pipeline; BESS separates the module graph from the scheduler tree, which is a per-core tree of logical (interior nodes) or physical (leaf nodes) schedule-able entities akin to Linux `tc`, enabling the implementation of complex hierarchical scheduling policies. By default, a single pipeline is assigned to the first system core under a round-robin root node and would be assigned with one core to handle corresponding traffic.

When BESS's code generator receives a placement solution from the optimizer, it pre-computes the optimal core placement to maximize throughput. According to this core allocation, we allocate cores to contiguous NF subgroups via the BESS scheduler. This allocation of cores to subgroups is done carefully to avoid violation of mutual exclusion in the NF DAG and to avoid fragmentation of stateful NFs. Ultimately, the overall chain throughput is limited by some contiguous subgroup, and since NF and/or subgroups can have dramatically different costs, we find that despite multi-chain allocation it is ultimately most meaningful to analyze each chain independently.

A.2 PISA switch

Deploying NFs in a switch hardware is different from deploying them in a server. Hardware switches process packets with a pipeline of switch stages. P4 switches require a platform-specific compiler that compiles a P4 program into a binary configuration for switch ASICs. The binary configuration maps a switch abstraction into the underlying hardware resources, such as per-stage register bits, TCAM, SRAM, ALUs and so on.

To generate a P4 program and get it compiled into the switch hardware, Lemur's meta compiler unifies many P4 NFs and generate a single P4 program that (1) has a unified packet header parser that can recognize all possible packet headers from each individual NFs, and (2) has a complete set of match-action tables and ensures packets traverse through them in the correct order. With Lemur, NF developers can build new P4 NFs as standalone NFs. They use Lemur's extended P4 syntax to design new P4 NFs, or modify the P4 implementations of NFs slightly to make them recognizable by Lemur's meta-compiler. Then, Lemur's meta-compiler is responsible for unifying P4 NFs into a final P4 program by unifying header parser trees, and composing them into a switch pipeline.

A.2.1 Algorithm of unifying P4 parsers. In an abstract P4 switch model, a header parser is a parser tree that has each tree node representing a unique packet header and is usually rooted at Ethernet header. It is an ordered tree and contains a number of transitions from one header to next possible headers. To unify P4 NFs, Lemur's meta-compiler must provide an unified header parser that parses all necessary packet headers. To do so, the meta-compiler starts from an empty parse tree and merges each P4 NF's parse tree into that unified tree. To merge a new parse tree, it traverses the new tree and visits all parsing states (i.e. headers). At each parsing state, it compares all state transitions between the new tree and the unified tree, and integrates any non-existing transitions and new headers into the unified tree. If the meta-compiler encounters a conflicting header transitions, then it rejects this placement because at least two NFs conflict with each other and cannot be placed at the P4 switch together.

A.2.2 Algorithm of generating the global P4 pipeline. Another important aspect of Lemur's meta-compiler is to unify match-action tables from all P4 NFs into one final P4 switch pipeline.

At the pre-processing stage, Lemur's meta-compiler concatenates NFs into a subgroup if they are in a sequential order and have no branches or merges in between. This saves switch's resources because packets' NSH headers do not get unnecessarily updated and matched when they traverse NFs in the subgroup. Concatenating P4 NFs into subgroups also simplifies the control flow of the final P4 pipeline. The output is a P4 subgroup DAG. In this DAG, nodes are categorized as normal (leaf) nodes, branching nodes and merging nodes. To convert a P4 subgroup DAG to a P4 subgroup tree, the meta-compiler handles branching nodes and merging nodes differently.

A branching node is a subgroup node that has multiple downstream subgroup nodes. Traffic is split into downstream subgroups

with a set of BPF rules according to the NF chain specification. To handle a branching node, Lemur's code generator generates and inserts a customized traffic-splitting table that is pre-populated with BPF rules to split traffic. When packets arrive at the branching point, the table matches on packets' traffic classes and decides which branch the packet should be forwarded to. Decisions are stored in a per-packet metadata field. Then, the meta-compiler steps into all branches one at a time. It generates code for each branch individually and places a condition checking before that branch. This is to make sure that only destined packets should be processed by a branch. This design only introduces necessary dependencies between upstream subgroups and downstream subgroups, and does not introduce unnecessary dependencies among different downstream subgroups. This allows a platform-specific P4 compiler to pack parallel branches into the same set of switch stages whenever possible.

A merging node is a subgroup node with multiple upstream subgroup nodes. It is the merging point of multiple branches. In a P4 switch pipeline, a table cannot be revisited twice as the pipeline must be a tree structure. Therefore, Lemur must choose the right place to generate code for a merging node, and ensure that all its upstream branches can finally reach the merging subgroup node. The code generator implements this by detaching a merging node from the P4 subgroup DAG and re-attaching it to its all direct-predecessors' common ancestor node. That ancestor node has just the right scope to ensure that all branches can reach the merging node. The merging node is placed at the same level as the ancestor's children. When traversing the P4 subgroup tree, the code generator must visit all non-merging nodes first. The code generator also places a condition check on packets' metadata to select packets that are necessarily processed by NFs in merging nodes.

After dealing with branching and merging nodes, Lemur's code generator takes the P4 subgroup tree. It traverses the tree recursively in Preorder, and generates P4 code for each subgroup node.

A.3 Execution: smartNIC

We use the Netronome Agilio CX 1x40 Gbps SmartNIC. This smartNIC is capable of executing eBPF (extended Berkeley Packet Filter) programs. The NFs are programmed in C language and then compiled to the eBPF target, which is an intermediate assembly representation [42].

We then load the eBPF program in the SmartNIC, offloading the computation to the NIC. XDP (eXpress Data Path) is used to hook the ingress traffic to the SmartNIC, which is running the eBPF program.

Programming the SmartNIC with eBPF technology presents some challenges. It has only 512 bytes of memory stack. It can only load 4196 instructions. There can be no function call. Moreover, to load the program in the SmartNIC, the code has to pass a verifier. The verifier does not allow back-edge jump (for, while). We solved these challenges by optimizing the code for 64-bit implementation, using loop unrolling to avoid for (back-edge), and inlining all function calls.