# Decoupling Algorithms and Optimizations in Network Functions

Omid Alipourfard
Yale University
omid.alipourfard@yale.edu

Minlan Yu
Harvard University
minlanyu@seas.harvard.edu

## ABSTRACT

Network function virtualization promises a path to rapid innovation in networks. However, due to the complexity of developing these functions, innovations have been slow. Designing a network function is a daunting task that requires combining packet processing optimizations with the network function logic. It is not possible to ignore packet processing optimizations either: an optimized pipeline can have three times better performance than an unoptimized pipeline in our experiments. In this paper, we introduce NFMorph, a framework wherein the algorithms (i.e., the network function logic) are decoupled from the packet processing optimizations. Developers would specify the packet processing algorithms in a high-level language. The runtime then identifies the best set of optimizations on the algorithms. This is done based on the domain knowledge that operators provide as input to NFMorph as well as optimization templates we have developed for common NF primitives. NFMorph can also just-in-time reoptimize based on workloads and environment constraints.

## 1 INTRODUCTION

Middleboxes are network appliances that run packet processing functions such as firewalling, caching, and load balancing. In the past, dedicated hardware boxes performed most of these functions, but hardware boxes lack the agility to keep up with the rapid change and growth of the network.

To get around this limitation, industry proposed the use of commodity servers to run the network functions (NFs) [3, 4]. The spearheads of this effort promised rapid innovation (since changing the software on commodity servers is easy) and reduced CAPEX cost (as commodity servers are cheap and repurposing them for new functionalities comes at the cost of changing the software).

The benefits of network functions only happen when we fully optimize their performance (e.g., throughput, latency) and achieve a better performance-cost tradeoff than hardware boxes. However, fully optimizing the performance is not easy, because there is a large set of optimizations depending on the environment that runs the NF pipeline and the input workloads.

**Optimizations based on environment constraints**: NF pipelines often have different environment constraints such as cache size, memory size, and the types of cores in the platform. The environmental constraints also change when NFs share resources with other applications. These environment constraints impact the choice of optimizations. For example, when the memory size for an NF is constrained, we may use Cuckoo hash tables that achieve high utilization as opposed to a large linear hash table [8]. When an NF shares cache with another cache aggressive application, we should prefetch instructions to improve its performance. However, when the NF runs on multiple cores, it may be better not to prefetch because software prefetchers may get overloaded which adversely impacts performance

**Optimizations based on workloads**: The choice of optimizations also depends on workloads (§2). For example, for NFs that deal with skewed traffic patterns, we can build a fast path by partially computing values for a set of high-volume flows, which drastically improves performance. However, under heavy-tailed traffic patterns, the same fast path lowers throughput because it increases the code size and reduces cache locality (§2.2). As another example, Trumpet [19] exploits the fact that applications often generate a burst of packets on the same flow and thus decides to cache and reuse hash function computations.

Whenever environments or workloads change, we need a major code refactoring to adjust optimizations. For example, we may need to unroll loops, identify the right data structures, or change data layouts (§2.2). Furthermore, one optimization may change the effect of other optimizations. For example, an optimization that increases cache locality but uses more instructions may reduce the instruction cache hit rate and negatively impact other optimizations.

Today, developers often take an iterative process when performing optimizations: they have to run the NF pipeline, find the bottlenecks, change the code based on their experiences, and repeat. This process is largely manual because commodity servers and compilers were not designed for packet processing pipelines in NFs and thus cannot automatically identify the right set of optimizations. Most optimizations require the domain knowledge of the specific network functions such as their performance requirements, environment constraints, and workloads [10, 11, 19].



*HotNets-XVII, November 15–16, 2018, Redmond, WA, USA*



ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286073

Thus it often takes many expert engineers a large amount of effort to develop highly optimized monolithic NF software (e.g., VPP [7] and OpenVSwitch [22]). It is hard for an average engineer to add a new feature to these NFs and optimize the performance because it requires a full understanding of the code. Such a tedious process significantly hinders innovations in network functions and increases the CAPEX and OPEX costs to develop and maintain these functions.

We observe that the fundamental obstacle for optimization here is the tight coupling between algorithms and the optimizations in network functions. Developers must understand network function algorithms and the large space of packet processing optimizations in detail to efficiently optimize network functions for a new environment or workload.

To fundamentally address the complexity of optimizations and facilitate innovation, we propose to *decouple* packet processing algorithms from optimizations in NFs. Developers should simply focus on developing the packet processing algorithms in NFs. We will then design a runtime that automatically makes optimization choices based on the environment and workloads.

Although automatic optimization of a general program is challenging, it is possible to optimize NFs automatically because of three reasons: First, rather than rather than focusing on low-level metrics such as cache locality used in general code optimization, we only need to optimize for the performance metrics for NFs such as throughput and latency. Second, code optimizations are often challenging because there are complex dependencies between instructions. Fortunately, most NFs have only packet-based dependencies (i.e., packets are the only units that flow across NFs and instruction modules within an NF). Third, NFs often have a common set of primitives such as hash tables, longest-prefix matching. If we can optimize these primitives, we can improve the performance for a large part of NF code.

Using the above domain knowledge of NFs, we introduce NFMorph that automatically optimizes NF pipelines based on environments and workloads. NFMorph includes an NF language that allows developers to expresses the logic of NFs (i.e., the algorithm) together with the packet dependencies and common primitives that helps the optimization. NFMorph also provides a set of optimization templates that can enable different tradeoffs of instruction locality, cache locality, and pipelining. NFMorph's runtime periodically selects the best set of optimizations from the templates based on code profiling on the current workloads and automatically swaps in the selected optimizations using just-in-time compilations.

Our prototype of NFMorph on an example pipeline shows that we have the potential of improving the throughput by 322% compared to statically optimized counterparts (§2.2). NFMorph also makes it easy to migrate codes to new environments while still ensuring high performance.

## 2 BACKGROUND AND MOTIVATION

To illustrate the impact of optimizations for packet processing, consider a simple NF pipeline with three modules: (1) a *routing module* that forwards packets using a radix trie (similar to that of click [17]) and contains 1614 rules from one of the Stanford routing tables [16], (2) a *measurement module* that uses a count-array sketch to count traffic of each flow [8], and (3) a *packet checksum module* that calculates the TCP checksum of every packet. We synthetically create a heavy-tailed traffic pattern with a Zipfian packet distribution ($\alpha = 0.5$) across 10 million unique flows where the top 1% of the flows have 10% of the total traffic. We chose a synthetic traffic because we can control the parameters of the traffic distribution and show the impact of each optimization technique.

To implement the NF pipeline, we develop our own framework on top of DPDK [5], a data plane library for fast packet processing. We opted not to use well-known frameworks that already use DPDK, e.g., VPP [7]. These frameworks typically have a rigid structure and enforce specific ways of processing packets, making it difficult to show the effect of each optimization. For example, VPP is optimized to work with batches of 256 packets. Our framework is modular and we can specify any pipeline as a graph of interconnected modules.

For the baseline, we implement each module's algorithm in the simplest form. We do not include packet specific optimizations outside those provided by DPDK. Our measurement module looks as follows[1].

```
count(pkt, table, table_size):
    value = hash(pkt.src_ip, pkt.dst_ip,
                 pkt.src_port, pkt.dst_port)
    table[value & (array_size - 1)]++
```

Here, `table` is an array of 32 bit integers and `hash` refers to Murmur32 hash. Although this code is easy to understand and maintain, it only achieves a throughput of 4.4 Mpps.

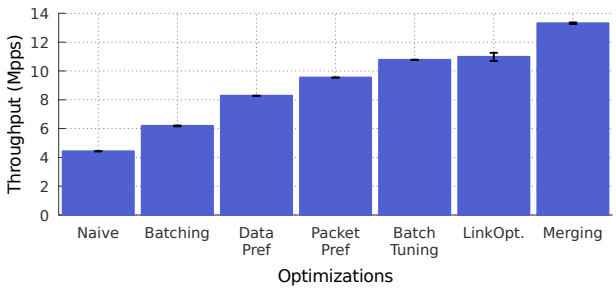


**Figure 1: Impact of different optimizations on the NF throughput. Each bar contains all the optimizations described in the bars to its left and under it. Error bars show the standard deviation of throughput across 5 runs.**

[1]Due to lack of space, we do not show the code for checksum and the routing module, but their code follows a similar trend.

## 2.1 Optimization knobs

We take commodity x86 servers as an example to show a variety of optimization knobs we can choose and their impact on throughput.

**Batching packets**: A common optimization in packet processing pipelines is batching incoming packets and processing them together. Batching helps by amortizing the static computation costs over the batch size. For example, instead of calling a function many times with one packet, we can call the function once with many packets. By passing packets in a batch of 32 (a commonly used batch size), we can improve the performance of the previous pipeline to 6.2 Mpps.

Since NFs operate on packets, there are many opportunities to perform batching—we can pass a batch of packets to every NF or break down an NF into multiple stages and process each stage using a batch of packets. The key question is how to find the right set of stages for batching while ensuring that there is no data-dependency between two consecutive stages. Another challenge is to decide the optimal batch size for trading-off throughput and latency.

**Prefetching data**: In packet processing pipelines most memory accesses are triggered by an incoming packet. There is a latency gap of up to 100 ns between CPU and memory. It is possible to hide this access latency by prefetching packets, that is, we can break down the computation into stages and process other packets while we are waiting for the prefetchers to pull the data from memory.

A developer could identify memory bottlenecks by using profiling tools. For example, using Precise Event-Based Sampling (PEBS) counters [6], we find that the table cell increment operation in the measurement module uses more than 68% of the sampled cycles and incurs 43% of the cache misses. By prefetching the table cell's data, we can improve the throughput to 8.2 Mpps (1.36 IPC). We can further improve the throughput to 9.5 Mpps (1.62 IPC) by prefetching the packet headers. The code at this stage looks as follows:

```
count(pkts, table, table_size):
    for pkt in pkts:
        prefetch pkt.tcpip_header
    index tmp[len(pkts)];
    index idx = 0;
    for pkt in pkts:
        ...
        prefetch tmp[idx]
    idx = 0;
    for _ in pkts:
        table[tmp[idx++]]++
```

**Variations on loop iteration pattern**: Different loop iteration patterns, when processing batches of packets, result in different localities of instruction and data cache. The current iteration pattern processes all the packets in the same batch together. Instead, we can improve the data cache locality and reduce the number of jumps by unrolling loops and grouping packets in smaller batches to ensure that the packet data does not get evicted from the lower level caches. Following this, we can reimplement the measurement module as follows:

```
count(pkts, table, table_size):
    for pkt in pkts:
        prefetch pkt.tcpip_header
    for pkt_grp_of_4 in pkts[4..]:
        for pkt in pkt_grp_of_4:
            /* Hash and prefetch cur grp */
        for pkt in prev_grp:
            /* Set the value for prev grp*/
        prev_grp = pkt_grp_of_4
```

This implementation has a throughput of 10.2 Mpps and has lower latency than when the batch size is smaller than 32 (e.g., because the latency of packet processing cannot exceed a certain threshold)[2].

Other variations of loop patterns are also possible. For example, we can tune the group size, change iteration patterns within each group, or use constructs such as Duff's device to reduce the number of branches.

**Amalgamation of the modules**: Rather than optimizing individual NFs, we can optimize a pipeline of NFs. Since NFs pass packets around, we merge the NFs' code and perform whole-program optimizations that improve the performance.

However, we cannot simply rely on generic compiler optimizations (e.g., link time optimizations such as -flto for GCC and Clang/LLVM). In our experiments, such optimizations can only improve the throughput from 9.5 Mpps to 9.8 Mpps. The reason is that compilers cannot fully utilize the packet processing context. For example, they do not merge or split packet batches to increase the instruction or data cache locality and improve throughput.

By re-optimizing the whole-pipeline, we can increase the throughput to 13.2 Mpps. This final representation of the code has a throughput that is 3 times faster than our original strawman implementation. However, it is much harder to modify the code as we have mingled the algorithm with the optimizations. What used to be 19 lines of code across three modules is now 102 lines of code in a single module due to the optimizations—an overhead of 430%.

The interface of each network function is well defined for network functions: each module accepts a packet and can output many packets. The interface makes it easy to combine the functions and redo the optimizations for the whole pipeline. Even then, optimizing NF pipelines is difficult as there are many such pipelines. Each operator uses a unique combination of functions. Here is the source code after applying optimizations across our three modules (checksum, routing, and measurement):

```
pipeline(pkts, table, table_size):
    for pkt_grp_of_4 in pkts[4..]:
        for pkt in pkt_grp_of_4:
            prefetch pkt.tcpip_header
        for pkt in prev_grp:
```

[2]Since all the packets in the same batch are processed together, larger batch sizes translate to higher packet processing latency.

```
        /* Hash and prefetch prev grp and
           perform routing and checksum
           calculations */

    for pkt in prev_prev_grp:
        /* update the measurement tbl.*/
    prev_prev_grp = prev_grp
    prev_grp = pkt_grp_of_4
```

## 2.2 Workload based optimizations

The workload affects the choice of optimizations. Continuing with the above NF pipeline example, we consider a new workload of skewed traffic pattern with 100k unique flows where the top 1% of the flows represent 99% of the traffic. Under this traffic pattern, we can apply two additional optimizations to improve the code's throughput.

**A small number of flows**: Knowing that the number of unique flows is low, we can reduce the data-structure footprint, remove prefetching instructions (as most of the data stays in the cache), increase the batch size (because the number of unique flows in each batch is low), and use a different loop iteration pattern.

To illustrate the impact of these optimizations, consider the NF pipeline before the amalgamation. The throughput of the pipeline for the skewed traffic is 11.3 Mpps (10.8 Mpps for the heavy-tailed traffic). By reducing the table size and removing the prefetching instructions in the measurement module, we can improve the throughput to 11.7 Mpps, but this lowers the throughput for the heavy-tailed traffic pattern to 10.4 Mpps. Further, we can tune the batch size and modify the iteration pattern and reach 12.7 Mpps, but this lowers the throughput of heavy-tailed traffic to 10.3 Mpps.

**A small number of heavy-hitters**: Because we have a small number of heavy-hitter flows, we can improve the pipeline throughput by breaking the pipeline into a common case for the high volume flows and a rare case for all other flows. More concretely, we can cache the routing computations for the most popular flows and keep separate counters for them. Through this optimization, we can improve the throughput to 13.2Mpps; but this further lowers the throughput of the heavy-tailed traffic to 9.8Mpps. The lower throughput is because popular flows' share of traffic in the heavy-tailed case does not justify the cost of branching and additional tables whereas for the skewed traffic case the *common case* happens often enough to make it worthwhile.

With these optimizations, we improve the throughput of the pipeline for the skewed traffic by 17% but lower the throughput of the heavy-tailed traffic by 10%. The result touches two types of problems: (1) Optimizations do not work well across all workloads. When we optimize the code for one workload, the code may have a worse performance under other types of workloads). (2) When building systems without assumptions on the workloads, developers have to consider many types of workloads during the optimization process [19], making the optimization a time-consuming and challenging task.

Finally, by amalgamating the code, we can easily achieve line rate packet processing on a single core for the skewed traffic pattern for our pipeline of routing, measurement, and checksum modules.

## 2.3 Environment based optimizations

All the above optimizations and the performance improvement was based on the x86 server we use. If environments such as cache sizes, memory sizes, and processor types change, we may make different optimization tradeoffs. If we have SmartNICs [13], we have another set of optimizations such as offloading the checksum verification modules to the SmartNIC.

Decoupling NF algorithms from their optimizations makes it easier to identify the best optimizations for the current environment and allow NFs to run across environments.

# 3 NFMORPH DESIGN

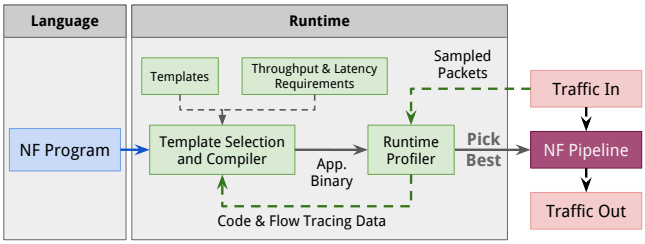


**Figure 2: NFMorph Design.**

We now outline the design of NFMorph. NFMorph consists of a language and a runtime (Figure 2). Developers write an NF program in NFMorph's domain specific language (DSL). The goal of the DSL is to make it easy to capture packet dependencies. It also comes with primitives matching packet processing pipelines.

The runtime is responsible for optimizing the NF program to match the operator specified throughput and latency requirements. The runtime's approach to optimization is similar to that of profile-guided optimizers: the runtime iteratively applies a set of program transformations (in the form of templates) on the NF program, profiles the performance of the NF using representative traffic, and optimizes the program based on the profile. Once it finds a good realization of the NF program, it swaps the current NF pipeline with the new realization.

## 3.1 Domain specific language

The goal of NFMorph's language is to allow developers to express transformations on individual packets, e.g., how to modify the headers or the payload of each packet. Packets are first-class types in NFMorph, which makes it easier to perform dependency analysis on NF programs. This dependency is crucial for selecting optimizations. For example, the compiler can use such dependency information to change the loop iteration pattern or to reorder NFs in the pipeline. NFMorph's language also allows developers to specify compiler-hints

that allow the runtime to make aggressive optimization decisions. For example, a hint specifying that a function is pure allows the runtime to precompute and cache the value of that function for some of the heavy-hitter flows.

NFMorph's language contains packet processing primitives that map to current hardware. For example, the language contains a hashing primitive. Such primitives allow us to offload the computation to a programmable device if it has matching offloading capabilities. We leave the detailed specifications of NFMorph domain specific language for future work.

Finally, NFMorph's NF language provides constructs for a modular block-based NF pipeline design, similar to Click [17]. Such a design allows NFMorph's optimizer to perform whole program dependency extraction, which paves the way for interleaving or dividing modules.

## 3.2 NF Runtime

NFMorph's runtime is responsible for finding the realization of the NF pipeline that achieves near-optimal performance based on the NF code, the workloads, and the constraints of the execution environment. It can also leverage different platforms, e.g., SmartNICs, if available. Our runtime starts with a preliminary pipeline (a pipeline compilation with no assumption about the workload or the architecture) and overtime applies optimizations by profiling the NF's performance.

**Traffic sampling**: NFMorph uses a traffic trace to periodically optimize the pipeline, e.g., by modifying the data structures or building fast-paths. This traffic trace should be representative of the incoming traffic for the optimizations to be useful. To build this traffic trace, we could redirect all the traffic to a secondary core/machine, but that is expensive and unnecessary. Instead, we use lightweight traffic sampling and preserve the traffic properties that affect the pipeline's throughput: first, we need to ensure that the relative portion of traffic to each branch remains intact. Even though the sampling may miss some rarely used branches, it does not affect the choice of optimizations because throughput bottlenecks often come from commonly used branches. Second, we need to mimic the access pattern on both memory and the CPU cache. To do so, we can upsample the sampled traffic such that the resulting trace has the same number of unique flows as that of the original traffic. The goal of such upsampling would be to ensure that the size of the data pulled into the cache and the eviction pattern of data from the cache is not drastically altered.

**Flow tracing and code profiling**: The flow tracer collects statistics about the percentage of flows (and their volumes) that go through different code paths in the program. Using this information, we can find hot code paths and optimize them. For example, the profiler can identify a hash function as the main bottleneck, and the tracer can attribute 90% of the computations of this hash function to a particular flow. Thus, we can then cache and reuse hash computations for this flow and save cycles.

We can control the overhead of tracing and profiling by dynamically changing the sampling rate at the expense of optimization accuracy. Our results show that the throughput overhead of using a PEBS counter [6] and a sampling frequency of 100Hz is only 0.05% at 14Mpps and when running on a single core. The 100Hz sampling rate was sufficient for identifying the code bottlenecks.

To reduce the overhead of tracing flows, we can leverage static code analysis to remember branching points for different traffic classes only. For example, if HTTP flows always take the true branch of an if-condition, we would be able to maintain all necessary information by just tracking the flow ID, meaning that we do not need to remember that branching point.

**Template selection**: The optimizer contains a repository of transformations and rewriting rules, i.e., templates, that change the abstract syntax tree (AST) of the NF's packet processing algorithm. As discussed in Section 2, the transformation includes four classes: (1) Batching templates, (2) Prefetching templates, (3) Partial computation caching, and (4) Data structure tuning.

We can associate a set of transformations to each type of the bottleneck so that the template selection engine knows when each optimization is useful. For example, if the code profiler detects a memory access bottleneck, we can use any one of the transformations that inject prefetching instructions to hide the memory latency.

The optimizer's job is to search for a close-to-optimal set of transformations to build the final NF pipeline. Different optimizations impact each other. Therefore, greedy approaches such as gradient descent may get stuck in local optima. We can instead use evolutionary algorithms that do not make any assumption about the optimization space.

Evolutionary algorithms run in generations. In each generation, we first select a set of candidates based on a fitness criterion—throughput, latency, or a combination of both. We next build the next generation using mutation and crossover operations. Mutations choose a set of (semi) randomly selected transformations to apply on a selected group of candidates from the previous generation. Crossovers combine transformations from two candidates from the previous generation in order to create new ones. Such algorithms have been successfully used to optimize the performance of image processing pipelines [9, 23, 24].

One advantage of evolution algorithms is that the tracing and profiling components need not be accurate and can tolerate a degree of noise. This is because the fitness criteria already incorporate some randomness (as a source of noise). In fact, this noise is crucial to avoid local optima when using evolutionary algorithms.

**Code swapping**: Once we find a good candidate pipeline, we have to swap the old pipeline with the new candidate. The code-swapper is responsible for changing the pipeline code while ensuring per-packet consistency and preserving processing states between the two pipelines.

Since we read packets in batches from the NIC, we can seamlessly change the code by swapping the pipeline on the

arrival of a new batch. To do so, we have to ensure that no packets are queued in the previous pipeline. We also have to transfer states from one pipeline to the next. For example, to remove a fast path from an old pipeline, we need to have all its data-structures merged before swapping. In order to ensure that no packets are queued, we can perform bookkeeping on every packet to detect packets that are not released by the pipeline (even dropped packets need to release their memory eventually or the NIC ring gets full). In order to ensure that we properly transfer the state from one pipeline to the next, we need to make sure that the memory pointers to the stateful components remain consistent. We can achieve this easily by reusing pointers from previous generations of the pipeline. For example, if a pipeline uses a hash table, we can reuse the hash table pointer in the next generation of the pipeline, safely, assuming that the hash table is untouched between the two pipelines. In addition, we need to ensure that if we break a stateful element into multiple elements, these elements are merged back before swapping the pipelines. This breakdown is hard, and we leave it to future work.

# 4 RELATED WORK

**Network function frameworks**: There is a large body of frameworks [2, 7, 14, 21] for building network functions. These frameworks expect the developers to use optimizations when writing NF code. The runtime of these frameworks are static, rely on general purpose compilers for optimizations, and do not optimize the user code based on the workload nor the environment that runs the code.

**Software switches:** OpenVSwitch [22] and VFP [12] are software-based switches that use table-based programming with a controller pushing rules into the tables that specify the policy of the network. It is possible to implement simple NFs on top of these tables, e.g., routing, NAT, or Firewalls, but more sophisticated NFs such as IDS, require dedicated binaries. It is also possible to use these switches to pass the packets between more sophisticated NFs, but there is usually a large overhead associated with passing packets around in this fashion. NFMorph, on the other hand, aims to provide a holistic NF framework.

**Berkeley packet filters:** Berkeley Packet Filters (BPF) [18] can inject packet processing algorithm into the networking stack in most Unix-like systems. The extended BPF, which comes with Linux, has an additional set of predefined data-structure, e.g., histograms, to collect statistics. Most implementations of BPF come with a JIT-optimizer that translates the BPF instructions into machine code. Even then, the goal of BPF and eBPF is to run within the kernel and safety of BPF programs played a big factor during their design process-they run on a virtual CPU with limited capabilities to ensure that they finish in finite time and do not block other kernel tasks. NFMorph, on the other hand, is a generic framework for writing packet processing algorithm. NFMorph uses trace-based JIT optimization and allows the developers to use arbitrary libraries for packet processing.

**Network function isolation**: It is important to ensure that in a pipeline running network functions from multiple vendors, the network functions remain isolated from each other [20, 21, 25]. Even then, for memory isolation, we can reuse previously used isolation techniques such as using containers [15] or relying on programming languages with memory safety guarantees [21]. For performance isolation, we can either rely on NF schedulers [20] or in case the NFs are on the same machine use CPU features such as CAT. [25].

**JIT optimizations**: Prior work has successfully applied JIT optimization in other domains [1, 23]. JVM [1] optimizes hot-code during the execution of the program. Halide [23] builds an optimized version of user code for image processing pipelines considering the executing platform. NFMorph leverages both types of optimizations and works on a different domain.

# 5 CONCLUSION

Given the complexity of writing optimized NF code and the stagnation in innovation caused by this complexity, we propose that we should decouple the NF logic from packet processing optimization. We discuss how an optimizer combines NF logic with packet processing optimizations through evolution algorithms and transformations on the NF program's abstract syntax tree. We further discuss the benefits of online optimizations and show how to build realizations of the pipeline based on the workload and environment at runtime.

# 6 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. We thank our shepherd, Manya Ghobadi, for her thoughtful interaction. This work is supported in part by the grants CNS-1829349, CNS-1834263, and CNS-1413978, and Facebook Graduate Fellowship award.

# REFERENCES

[1] Java Virtual Machine. URL https://java.com.
[2] Network Function Framework for GO. URL https://github.com/intel-go/nff-go.
[3] Network Function Virtualization, 2012. URL https://portal.etsi.org/NFV/NFV_White_Paper.pdf.
[4] AT&T Domain 2.0 Vision White Paper, 2013.
[5] Dpdk: Data plane development kit, 2018. URL https://www.dpdk.org/.
[6] Hardware Event-based Sampling Collection, 2018. URL https://software.intel.com/en-us/vtune-amplifier-help-hardware-event-based-sampling-collection.
[7] Vector Packet Processing. https://fd.io/technology/, 2018.
[8] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. SOSR, 2018.
[9] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. PLDI, 2009.
[10] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. Technical Conference on Linux Networking, 2017.
[11] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. NSDI, 2016.
[12] Daniel Firestone. VFP: A Virtual Switch Platform for Host Sdn in the Public Cloud. NSDI, 2017.

[13] 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. NSDI, 2018.

[14] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. UCB/EECS-2015-155, 2015.

[15] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. NetVM: high performance and flexible networking using virtualization on commodity platforms. IEEE TNSM, 2015.

[16] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. 2012, NSDI.

[17] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click modular router. TOCS, 2000.

[18] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. USENIX, 1993.

[19] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. SIGCOMM, 2016.

[20] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. SOSP, 2015.

[21] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. OSDI, 2016.

[22] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. NSDI, 2015.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. PLDI, 2013.

[24] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[25] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. NSDI, 2018.