PacketMill: Toward Per-Core 100-Gbps Networking

Alireza Farshin*
KTH Royal Institute of Technology
Stockholm, Sweden

Tom Barbette*
KTH Royal Institute of Technology
Stockholm, Sweden

Amir Roozbeh
KTH Royal Institute of Technology
Stockholm, Sweden
Ericsson Research
Stockholm, Sweden

Gerald Q. Maguire Jr.
KTH Royal Institute of Technology
Stockholm, Sweden

Dejan Kostić
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

We present PacketMill, a system for optimizing software packet processing, which (i) introduces a new model to efficiently manage packet metadata *and* (ii) employs code-optimization techniques to better utilize commodity hardware. PacketMill grinds the whole packet processing stack, from the high-level network function configuration file to the low-level userspace network (specifically DPDK) drivers, to mitigate inefficiencies and produce a customized binary for a given network function. Our evaluation results show that PacketMill increases throughput (up to 36.4 Gbps – 70%) & reduces latency (up to 101 µs – 28%) and enables nontrivial packet processing (e.g., router) at ≈100 Gbps, when new packets arrive > 10× faster than main memory access times, while using only *one* processing core.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; **Network servers**; *Network adapters*; **Programming interfaces**; • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → **Compilers**; **Source code generation**.

KEYWORDS

PacketMill, X-Change, Packet Processing, Metadata Management, 100-Gbps Networking, Middleboxes, Commodity Hardware, LLVM, Compiler Optimizations, Full-Stack Optimization, FastClick, DPDK.

ACM Reference Format:
Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward Per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3445814.3446724

*Both authors contributed equally to the paper.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446724

1 INTRODUCTION

Networking has shifted from inflexible, proprietary, and specialized hardware toward Software-defined Networking (SDN) and Network Functions Virtualization (NFV). Today many network appliances are realized using commodity hardware and the network functions are increasingly software-driven. The flexibility and programmability of such platforms has led to the emergence of many software networking solutions (e.g., Open vSwitch (OVS) [79], Click-based frameworks [6, 29, 65], BESS [36, 37], and Vector Packet Processing (VPP) [2, 27]). Unfortunately, the introduction of multi-hundred-gigabit network equipment and dramatic increases in the telecommunication bandwidth strain the performance of commodity hardware [63], due to the demise of Moore's law and Dennard scaling putting a cap on commodity hardware's performance [22]. While many try to introduce in-network processing via modern hardware (e.g., P4 architecture [11] and modern/programmable Network Interface Card (NIC)) to address the performance limitations [92], many network functions are deployed on commodity hardware, via *unspecialized* modular software, as software-based packet processing is being promoted by Ericsson, Cisco, and Intel [21, 49, 91]. Unfortunately, software-driven networking solutions have come at the price of lower performance. Two critical factors imposing performance limitations on software-driven networking to process packets at multi-hundred-gigabit rates are: (i) code inefficiency mainly coming from generality and modularity of networking frameworks; and (ii) suboptimal metadata management.

Our objective is to produce an optimized binary executable while maintaining high-level modularity and flexibility, as opposed to relying on handwritten assembly code [64]. This paper shows that performing efficient metadata management (to specialize Data Plane Development Kit (DPDK) buffers) and employing code optimizations (to minimize unnecessary memory accesses, improve cache locality, etc.) facilitates realizing our goal of software-based packet processing at 100-Gbps *and beyond* on commodity hardware.

We design, build, and evaluate a system, called PacketMill, to optimize the performance of a popular modular framework used for composing complex network functions on top of commodity hardware. PacketMill proposes a new metadata management model that realizes customized buffers when using DPDK, rather than relying on the generic `rte_mbuf` structure. Additionally, our proposed system performs a set of common & uncommon code optimizations to (i) the source code and (ii) the

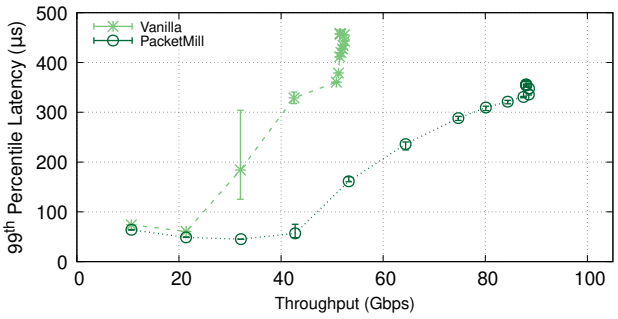


Figure 1: PacketMill improves per-core packet processing. Overlapped markers show that the performance can be capped despite the increasing offered load.

intermediate representation (IR) code while also employing link-time optimization (LTO) techniques. More specifically, PacketMill exploits the already-known information defining a Network Function (NF), i.e., the input processing graph, to mitigate virtual calls, improve constant propagation & constant folding, and reorder commonly used data structures in modular software processing frameworks.

Our evaluation results demonstrate that PacketMill improves not only microarchitectural metrics (i.e., it reduces cache misses) but also application-level metrics (i.e., it decreases latency and increases throughput) of network functions running at 100 Gbps. Figure 1 demonstrates that PacketMill improves the packet processing at 100 Gbps when a router forwards packets using a *single core* running at 2.3 GHz. More specifically, our proposed model & techniques shift the knee of the tail latency vs. throughput curve, i.e., achieving lower latency *even* when the load is higher. PacketMill's improvements are not limited to single-core NFs, see §4.5.

Our main insight is that efficient packet processing at 100-Gbps calls for holistic system optimization, specifically *milling* the entire software stack to *squeeze* every bit of performance from the hardware. We believe we are the first to (i) empirically examine/optimize metadata management models for packet processing and (ii) advocate the importance of low-level optimizations to process packets at near-100-Gbps rates with only *one* core. Although we focus on optimizing one specific framework, our results and techniques should be useful in other performance-sensitive contexts interested in nanosecond- and microsecond-level improvements. We think that our tool could be a starting point for further research on optimizing software packet processing frameworks and, more generally, on networking applications. Moreover, our optimization techniques can be used in combination with modern NICs.

Why now? Our work is motivated by two main recent trends:

First, the introduction of 100-Gbps network interfaces dramatically decreases the time budget for processing small packets, i.e., 6.72 ns to process a 64-B packet before receiving the next one, making *nanosecond level savings* count. Some works [24, 25, 89, 90] have advocated better cache management and avoiding any memory access to realize packet processing at multi-hundred-Gbps link rates. Additionally, CacheDirector [24] showed that *nanosecond* improvements in Last Level Cache (LLC) access latency can result in microsecond-scale reductions in latency.

Second, data center companies such as Facebook and Google have recently taken an active interest in profile-guided and post-link binary optimizations [15, 33, 51, 68, 71, 75], where they often achieve *sub-ten-percent* speedups. These works suggest that utilizing the full potential of the underlying hardware, even a little bit more, is essential for cost-effectively providing Internet services.

Observing these trends motivated us to dive a level deeper to find the underlying inefficiencies in modular packet processing to improve the performance of network functions running on top of commodity hardware. We rely on the fact that modular packet processing frameworks are similar to general-purpose software, i.e., they contain lots of code inefficiencies and perform lots of unnecessary memory accesses & indirect branches, which could lead to many opportunities for optimization. Additionally, for a given network function and workload, there are a subset of all of the execution paths that are very frequently used, hence improvements to these execution paths can have a large impact on performance.

Contributions. In this paper, we:

- Highlight the importance of metadata management in packet processing (§2.2) and propose a new model, called X-Change*, to mitigate its inefficiencies (§3.1),
- Design & implement PacketMill* to optimize the performance of packet processing frameworks via low-level optimizations (§3.2),
- Operate at >100-Gbps rates by employing code optimizations & efficient metadata management (§4).

2 SOFTWARE PACKET PROCESSING

Many network operators/providers have shifted toward pure software solutions that can be run on commodity off-the-shelf (COTS) servers, aka commodity hardware, to (i) avoid proprietary inflexible hardware middleboxes and to (ii) reduce capital expense (CAPEX) & operating expense (OPEX). These efforts can be classified into three main categories [14, 54]:

(1) Low-level building blocks for realizing I/O frameworks (e.g., DPDK [18], PF_RING ZC [70], netmap [83], and XDP [39]).
(2) Specialized virtual NF as a unified piece of software (e.g., OVS [79], ESwitch [64], PacketShader [38], and DPDKStat [95]).
(3) Modular frameworks for composing network functions (e.g., Click [65], FastClick [6], VPP [27], the Snabb NFV project [77], and BESS [36, 37]).

This paper focuses on the third category, i.e., modular network function composition frameworks. We briefly discuss some of the popular frameworks for packet processing.

Click introduced one of the first modular architectures for building software routers [65]. The building blocks of Click are called elements, which can be connected together to compose a graph defining a complex network function. Each element implements a simple function (e.g., packet classification, queuing, and decrementing TTL). During the initialization phase, Click parses the input processing graph, provided by the user, and *virtually* builds the control flow graph. Later, Click executes the

*Our source code is publicly available [23], see packetmill.io and Appendix B.

elements while traversing the graph for every packet. FastClick [6] is a high-speed variant of Click that leverages different acceleration techniques (e.g., linked-list batching) and integrates kernel-bypass networking frameworks (i.e., DPDK and netmap) into Click.

VPP or Vector Packet Processing framework is a software router, developed by Cisco, which focuses on L2–L4 packet processing based on vector processing. VPP is part of Fast Data Project (FD.io), i.e., a collaborative open-source project aimed at establishing a high-performance I/O services framework for dynamic compute environments, and it has teamed up with Intel to take advantage of SIMD instructions (e.g., SSE & AVX) as much as possible.

BESS or Berkeley Extensible Software Switch (aka SoftNIC [37]) is another modular framework that was inspired by Click but simplifies & extends Click's design choices. BESS was designed with an eye toward utilizing new hardware NIC features and kernel-bypass technologies (e.g., DPDK), thereby achieving better performance compared to Click [65] by leaving out the unused & old implementation.

Next, we discuss two important factors imposing performance limitations to process packets at multi-hundred-gigabit rates, i.e., (i) code inefficiency and (ii) patched* metadata management.

2.1 Code Inefficiency

Packet processing frameworks are built based on a modular design to bring a higher degree of flexibility and to simplify the composition of complex network services, by customizing and connecting simple monolithic elements. These frameworks usually adapt a general-purpose binary based on an input configuration file, which results in many inefficiencies, such as virtual calls, dead code, and unordered basic blocks. More specifically, the binary dynamically creates the control flow graph based on the input file and then executes it. All of the above frameworks utilize different acceleration techniques, such as kernel-bypass techniques and batch processing to achieve performance comparable to custom hardware appliances. However, as general-purpose processors were not optimized for packet processing, they do not provide the same performance as specialized hardware. Moreover, the modular & flexible design of these frameworks prevents them from achieving the performance of the underlying hardware.

Optimization Efforts. Two relevant attempts to overcome code inefficiencies in packet processing frameworks are:

① Kohler et al. [48] introduced a Click optimization toolkit to eliminate modular inefficiencies in Click and improve its performance. This toolkit is mainly a source-to-source tool that scans a Click-language file and employs optimization techniques, resembling general compiler optimizations, to transform the code into a more efficient version. More specifically, the Click optimization toolkit reads a Click configuration file, builds a graph of elements, analyzes & transforms the graph, and finally produces a more optimized configuration file and/or source code. The Click optimization toolkit includes a number of tools. The most relevant tool to our work is `click-devirtualize`, which is a static class analysis tool that devirtualizes function calls, i.e., replacing virtual function calls during graph traversal with direct calls extracted from the graph analysis.

② Protocol space mismatch can occur between development and deployment phases, leading to redundant logic. Thus, Bangwen et al. [17] proposed a tool, called NFReducer, which employs classic compiler optimization techniques to eliminate redundant logic from a configured NFV instance. NFReducer has been developed using LLVM and it has utilized a symbolic execution engine (i.e., KLEE [13]) to filter out infeasible paths of NFs. This tool performs three main optimizations based on the NF configuration: (i) excluding unrelated logic from NFV code; (ii) applying constant propagation, constant folding, and dead code elimination; and (iii) eliminating cross-NF redundancy when multiple NFs are chained.

PacketMill also employs code optimizations and is complementary to these efforts (see §3.2).

2.2 Patched Metadata Management

Packet processing usually requires additional information beyond the raw packets (i.e., bits received from the wire). The extra information is divided into two categories: (i) metadata and (ii) packet annotations (aka user/application metadata). The former contains additional details on the raw packet/buffer itself, such as its length, timestamp, checksum, and pointers to different protocol stacks in the packet, which is required to operate on the raw packets. The latter is the information used during the packet processing–i.e., the information that has to be calculated/extracted from the packet at one place and used in *another* place [65], such as VLAN ID, MPLS label, source & destination IP addresses & ports, statistics, and Wi-Fi association. The metadata is usually defined by the driver and the NIC, whereas the (packet) annotations are derived and used by the application. Next, we briefly explain the evolution of metadata management, starting from the Linux kernel.

The Linux kernel uses `sk_buff` data structures, aka socket buffers, to manage/handle network packets†. An `sk_buff` contains many metadata fields (the size/number of which depends on the protocol standards) to facilitate manipulation of packets, see linux/skbuff.h. Each `sk_buff` also provides a *fixed* 48-B free space, aka control buffer (cb), which can be used for application-specific annotations [81, 97].

Since Click started as a kernel-space packet processing framework, it developed a metadata class, called `Packet`, for handling packets, inspired by `sk_buff`. `Packet` had pointers to different protocol headers (e.g., network layer and transport layer). Additionally, it likewise reserved 48 B to be used by Click elements for storing packet annotations. As 48-B space may not be enough, developers had to carefully prevent collisions.

Modern packet processing frameworks (e.g., FastClick, BESS, and VPP) utilize kernel-bypass libraries (e.g., DPDK and netmap) to achieve zero-copy packet transfers and eliminate Linux kernel stack costs. In the rest of this paper, we focus on DPDK-based packet processing.

DPDK uses mbufs to carry network packets/buffers. Each mbuf has three sections: (i) a `rte_mbuf` data structure containing the metadata, (ii) a fixed-size headroom reserved for prepending/appending data, and (iii) a data segment used for storing the raw

*We use *patch* since packet processing frameworks have tried to adapt themselves to work with DPDK.

†BSD buffers are called `mbuf`s.

packets [19]. Each `rte_mbuf` struct is only two cache lines* (i.e., 128 B) to keep it as small as possible, leaving the management of packet annotations to the application (i.e., the packet processing framework). DPDK provides userspace NIC drivers, aka Poll Mode Driver (PMD), which enables the *direct* interaction of an application and the NIC. DPDK allocates mbufs (metadata + headroom + data) in the initialization phase. Subsequently, PMD uses these pre-allocated mbufs to receive/transmit packets at run-time. To receive packets, PMD passes the mbuf data address & its driver-specific descriptors to the NIC so that the NIC can DMA the received packets and their metadata to these addresses. Later, when the PMD detects a DMA completion via polling, PMD copies the relevant information from the driver descriptors to the mbuf metadata (i.e., `rte_mbuf` struct). To transmit packets, PMD performs a similar operation (i.e., updating driver-specific descriptors) before passing the mbuf data address & driver descriptors to the NIC. Unfortunately, integrating DPDK with packet processing frameworks causes metadata management to become a bottleneck, thereby achieving *suboptimal* performance at multi-hundred-gigabit rates. The performance degradation happens for three reasons:

First, modern packet processing frameworks typically employ batch processing to improve cache locality, i.e., an application receives a batch of packets, processes them, and then asks for another batch. Therefore, the number of packets' *metadata* required at any given time is equal to the batch size (i.e., the number of packets received from the PMD). However, DPDK uses a *distinct* `rte_mbuf` for every packet, which reduces the probability of the metadata data structures remaining in the cache. More specifically, warm cache lines containing recently processed packets' metadata may be evicted to make room for the newly arrived packets' metadata. An optimal solution would use a limited number of metadata data structures (e.g., `rte_mbuf`) and keep them in cache.

Second, since the `rte_mbuf` struct does not provide enough space for storing/keeping (packet) annotations, packet processing frameworks have to allocate larger data structures to enable/facilitate packet processing. Figure 2 compares the two common ways to extend the `rte_mbuf` metadata, which we refer to as "Copying" vs. "Overlaying".

Copying. This method is mainly used by Click & FastClick. They *handpick*, via copying or converting, the information useful for packet processing from the `rte_mbuf` struct into their own data structure (i.e., `Packet` class [26]) that contains a 48-B space for annotations, see 1 in Figure 2. Unfortunately, this method is inefficient because it involves *two* copy/conversion operations: (i) driver descriptors to `rte_mbuf` struct and (ii) `rte_mbuf` struct to `Packet` object.

Overlaying. Some packet processing frameworks (e.g., BESS) *overlay* the beginning of their data structures on the `rte_mbuf` and cast it to avoid copying & conversion, see 2 in Figure 2. They insert their dynamic metadata or annotations *after* the `rte_mbuf` struct and before the headroom. More specifically, BESS uses the `sn_buff` struct (recently renamed to `Packet` [9]), where they provide a 384-B space for storing the `rte_mbuf` struct, 64-B for the immutable fields (e.g., packet address and socket ID), 128-B

*The most frequently used fields are defined to be in the first cache line.

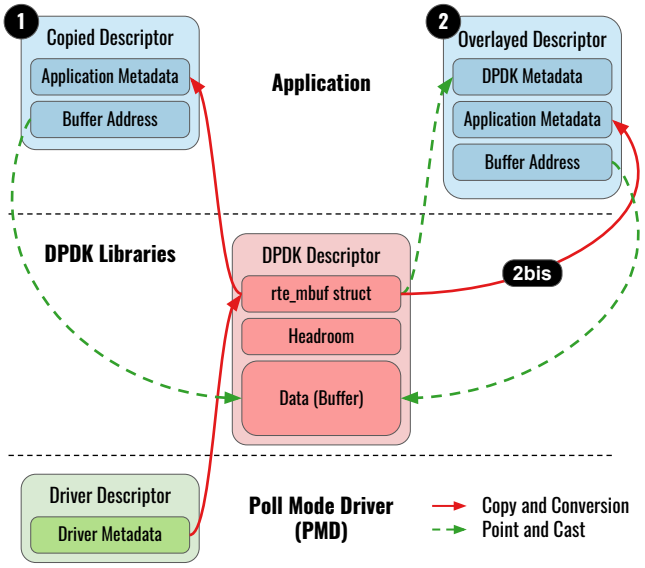


Figure 2: Common metadata management methods in packet processing frameworks (Copying vs. Overlaying).

static/dynamic metadata fields, and 64-B space for a module's & driver's internal use [8].

VPP follows a similar approach, where they use `vlib_buffer_t` to store the buffers' metadata. `vlib_buffer_t` is also known as primary buffer metadata used by the vector library (vlib). VPP overlays the beginning of its data structure with the `rte_mbuf` struct, but it does not use it. Instead, it copies/converts some fields from the DPDK data structure into the `vlib_buffer_t`, as it needs to make the metadata format fit for SSE instructions, see 2bis in Figure 2.

FastClick also supports Overlaying, which should be enabled at compile time. It casts every `rte_mbuf` into a `Packet` object and then (similar to BESS) inserts its annotations after it [26].

Although overlaying mitigates the cost of copying, it is still inefficient since packet processing frameworks have to adapt their format to the DPDK format (e.g., BESS) and/or do a transformation (e.g., VPP), which often results in carrying unnecessary fields while processing packets, thereby reducing cache locality.

Third, since different NFs require different information for processing a packet, using *one* standard data structure to keep the metadata & packet annotations is *non*-optimal, as it could spread the required information over multiple cache lines, thereby increasing cache occupancy and increasing the number of memory accesses needed to process a packet. A performant design should change the type and/or order of variables used to keep the metadata & packet annotations based on the functionally of a given NF.

3 PACKETMILL

This section explains PacketMill, our proposed system for optimizing the performance of modular software packet processing frameworks. Our goal is to mitigate the inefficiencies discussed in §2.1 & §2.2. To do so, PacketMill introduces a new metadata

management model, called X-Change, to enable metadata customization while improving cache locality. Additionally, it modifies the code based on the input configuration file – as this contains information that assists in the compilation process. Figure 3 shows our proposed pipeline to produce a specialized binary for a given NF configuration, where numbered/green shapes are proposed by us. We start by explaining the metadata management model and then continue with our efforts to mitigate code inefficiencies.

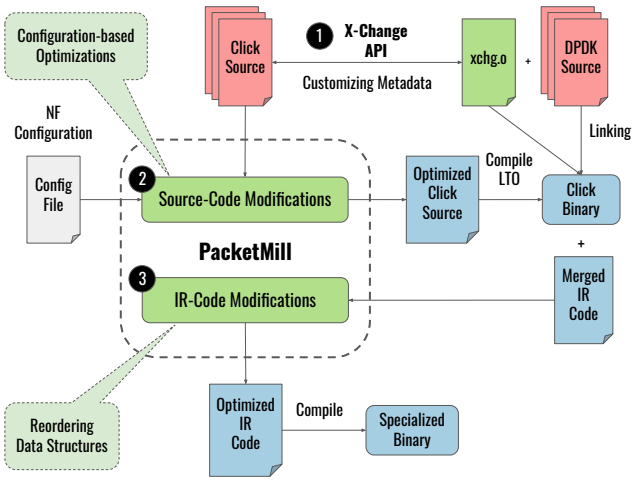


Figure 3: PacketMill Overview.

3.1 Efficient Metadata Management

§2.2 discussed the problems associated with the current ways of managing metadata in DPDK-enabled packet processing frameworks. Current packet processing frameworks rely on the generic `rte_mbuf` to store metadata, which requires adaptation & extra overhead/effort to process packets efficiently. PacketMill introduces a new metadata management model (X-Change) that enables frameworks developed on top of DPDK to *exchange* their customized/specialized metadata buffers with the userspace DPDK drivers (i.e., PMD) and to bypass `rte_mbuf`, thus addressing the second problem in §2.2. Additionally, PacketMill's metadata management model makes it possible to use a limited number of metadata buffers (e.g., 32) to improve cache locality and prevent unnecessary cache evictions, thus solving the first problem in §2.2. It aims to provide efficiency while ensuring backward compatibility for previously developed DPDK-based applications. To achieve our goals, we develop an Application Programming Interface (API) within DPDK, which requires some changes to DPDK's PMDs, see 1 in Figure 3.

Implementation. To showcase X-Change, we modified MLX5* PMD (used by Mellanox NICs). We add a header file (`.h`) and a source file (`.c`) to the MLX5 source code. The header file defines conversion functions for both receive (RX) & transmit (TX) paths to assign/copy different metadata fields, as opposed to the current typical DPDK implementation where PMD *directly* assigns the metadata to a `rte_mbuf` struct. Note that these functions will eventually get inlined, as we use LTO. Listing 1 compares our proposed approach and the default DPDK behavior with a simple example.

```
/* Default DPDK */
pkt->vlan_tci = rte_be_to_cpu_16(cqe->vlan_info);
```

```
/* X-Change */
xchg_set_vlan_tci(pkt, rte_be_to_cpu_16(cqe->vlan_info));
```

Listing 1: X-Change introduces conversion functions instead of directly writing to the `rte_mbuf` struct. The code shows an example for setting the VLAN TCI field in the driver.

The source file implements the standard behavior of DPDK (i.e., using `rte_mbuf`). Consequently, since DPDK compiles our pre-implemented source file by default, X-Change enables full backward compatibility to metedata-agnostic applications. However, it enables developers to re-implement the conversion functions and customize DPDK's metadata to their needs, thereby enabling the PMD to write the metadata directly to their application's data structures. Moreover, X-Change makes it possible to *easily* try out (or switch between) different metadata management models with low overhead (i.e., simply by linking to a different object file implementing the conversion functions). Listing 2 shows an example re-implementation of a conversion function.

```
/* X-Change Implementation of Default DPDK */
void
xchg_set_vlan_tci(struct xchg* pkt, uint16_t vlan_tci) {
    ((struct rte_mbuf*)pkt)->vlan_tci = vlan_tci;
}
```

```
/* X-Change Implementation for Custom Buffers */
void
xchg_set_vlan_tci(struct xchg* pkt, uint16_t vlan_tci) {
    SET_VLAN_ANNO((Packet*)pkt, vlan_tci);
}
```

Listing 2: X-Change simplifies the metadata management. The code compares the default DPDK and a custom implementation to set the VLAN TCI field.

PacketMill's model also proposes a new way to interact with PMD. The default DPDK implementation asks applications to provide an empty array of pointers in order to receive packets. Later, PMD fills this array with the address of received packet buffers (i.e., the metadata and the raw packet), and DPDK provides new *untouched* packet buffers to the PMD to replace the *cached* buffers used for the received packets. However, X-Change enables applications to provide their own packet buffers to the PMD. By doing so, the PMD can directly write into the application's metadata and "exchange" the used buffers (containing received packets) with the newly received ones from the application. X-Change uses a similar workflow for the TX path. After processing the received packets, the application passes the processed buffers to the PMD. Subsequently, the PMD copies/converts the application's metadata to the NIC descriptors and does a *swap* of the previously sent buffers, sitting in the transmit ring, with to-be-sent buffers; thus,

*Our current prototype does not support vectorized PMD, so we have disabled it in all of our experiments, except in §4.1.

the application has as many empty buffers as it has sent (which can be exchanged again in the RX path).

Summary. X-Change is an optimization to DPDK that provides custom buffers to drivers; thus, metadata can be directly written into the applications' buffers rather than using an intermediate DPDK metadata (i.e., `rte_mbuf`). X-Change uses conversion functions instead of direct assignment to set the metadata fields. The X-Change implementation (i.e., the definition of different conversion functions) is dependent on the driver's features and descriptors. A recent work called TinyNF, done by Pirelli et al. [80], proposes a simple and *formally verifiable* driver model (for Intel 82599 NIC) that removes the need for dynamic packet metadata. However, it prevents buffering of packets, such as switching packets between cores, reordering packets, and stream processing, which introduces lots of drawbacks for some network functions. In contrast, X-Change also reduces the number of metadata buffers, but *without* imposing those restrictions. Moreover, X-Change is more generic (as opposed to TinyNF) since it pushes programmability into the driver, making it possible to implement buffer exchanging, or even TinyNF, without even re-compiling DPDK. X-Change results in the following improvements:

- Enables applications to use their tailored metadata and to bypass the generic `rte_mbuf`, thus avoiding unnecessary copy/transform operations and cache evictions;
- Pushes down part of the application's RX/TX loops to initialize packet annotations into the PMD, thereby simplifying the application's processing path;
- Limits the amount of metadata used to the application's requirement (i.e., proportional to the RX burst size + the number of packets enqueued in software), keeping metadata cache lines warmer and making the most out of DDIO [25];
- Skips buffer allocation/release operations through DPDK buffer pools, which are inefficient due to supporting/maintaining many (unnecessary) features; and
- Makes it possible for the application to easily use different packet chaining models (e.g., vector, linked list, or a combination of both) to better fit their needs.

3.2 Optimized Code

PacketMill performs two main types of code optimizations: (i) source-code modifications and (ii) IR code modifications. The former embeds & modifies the source code, as *early* as possible, based on the information provided by the NF configuration file. Informing the compiler of this known information should enable many optimizations, as compilers have become much smarter [32]. The latter exploits the LLVM toolchain to modify the final IR bitcode produced by LTO, making it possible to perform optimizations, as late as possible, i.e., when the *whole* program's IR bitcode is available.

3.2.1 Source Code Modifications. Our first step to produce a more specialized binary is to perform configuration-based optimization, which gets rid of unnecessary pointers to already-known data structure/variables, while removing unused code. To do so, we *embed* some of the already-known information about the NF into the source code, see ❷ in Figure 3. More specifically, we use (i) the packet processing graph and (ii) constant element parameters defined in the NF configuration file, see Listing 3.

```
// Elements Definition
input :: FromDPDKDevice(PORT 0, N_QUEUES 1, BURST 32);
output :: ToDPDKDevice(PORT 0, BURST 32);
// Processing Graph
input -> EtherMirror -> output
```

Listing 3: A Click's NF configuration file defining a simple forwarder that receives packets from a DPDK-enabled NIC, swaps the Ethernet MAC addresses, and transmits the packets. The graph contains three elements: (i) `FromDPDKDevice` *alias* input, (ii) `EtherMirror`, and (iii) `ToDPDKDevice` *alias* output, chained together sequentially.

Embedding the packet processing graph, i.e., the processing elements and their connections, informs the compiler about the NF's control flow graph (CFG), resulting in better code layout. It also makes it possible to declare the processing elements statically in the source code, i.e., allocating them in a *static* .data or .bss segment (or stack) rather than the heap*, *potentially* resulting in a less fragmented access pattern and fewer translation lookaside buffer (TLB) misses. Moreover, using statically defined elements *and* defining the CFG enables us to perform full devirtualization, i.e., inlining the virtual calls, as opposed to `click-devirtualize` that only defines the type of the function pointer *rather than* the actual object reference.

Constant element parameters define the characteristics of processing elements. For example, an element receiving packets (e.g., `FromDPDKDevice` in Listing 3) should define the maximum number of packets fetched from the I/O device (i.e., a BURST size). Embedding these values in the source code enables the compiler to perform constant propagation & constant folding while removing/eliminating dead code & unrolling loops, thereby improving cache locality.

Implementation. To benefit from the `click-devirtualize` toolchain [48] and reduce the implementation overhead, we resurrected & adopted `click-devirtualize` to work with FastClick [6] and then implemented additional optimization on top of it. These optimizations are similar to NFVReducer [17], as both share the same goal, i.e., removing redundant/unnecessary code. However, NFVReducer focuses on optimizing the performance of popular Intrusion Detection Systems (IDSs), whereas our techniques are applicable to *any* kind of NFs composed by modular packet processing frameworks. Despite these differences, PacketMill can potentially be combined with NFVReducer to filter out infeasible paths via symbolic execution. Finally, it is important to highlight that our proposed optimizations are *not* limited to Click-based frameworks; they could be useful for other software packet processing frameworks.

3.2.2 Intermediate Code Modifications. Modern compilers (e.g., gcc and clang/LLVM) support LTO [30, 31, 57], making it possible to perform inter-procedural optimizations during the linking phase where the *whole* program is visible to the linker/optimizer. When

*We define the element objects in the source code and re-initialize them properly after executing the binary.

LTO is enabled, compilers typically produce IR code rather than regular object files (containing machine code), so that whole-program analysis and optimization can be done during linking. LTO can realize better code layout and smaller binaries, as it is easier for the compiler to collect/use the information about symbols, variables, functions, and the callgraph to eliminate dead code and reorder functions. Additionally, since the whole program is available, developers can potentially implement customized optimization passes to optimize the executable even further.

PacketMill exploits LLVM's LTO to address the third problem in §2.2, i.e., lack of per-NF data structure speciality. Our goal is to specialize/customize the *one* standard metadata used in a packet processing framework for a given NF. To do so, we develop an optimization pass (via LLVM) that reorders the variables/fields of a metadata structure based on the access pattern of a given NF, see ❸ in Figure 3. By doing so, the more frequently accessed fields will be placed at the beginning of a data structure (i.e., the first cache line(s)), prevent extra accesses to multiple cache lines. More specifically, our pass finds the references (done by the NF) to different variables/fields of a metadata structure within a *module* and then sorts these variables based on the *estimated** number of accesses to the variables. Later, the pass fixes these references so that LLVM's GetElementPtrInst (GEPI)† instructions perform the correct accesses. Listing 4 shows an example LLVM IR bitcode for accessing a variable of a C++ object. The current version of our pass only sorts the variables, but one could also *remove* unused variables/fields. Additionally, it is possible to extend our pass to consider other sorting criteria (e.g., order of access), which remains as our future work. To examine the full potential of LTO, we extend DPDK's build system to work with clang and produce LLVM IR bitcode‡. It is worth mentioning that using LTO increases the compilation time, which could be reduced by using a scalable variant of LTO (e.g., ThinLTO [58]).

```
1  /* A Simple Metadata Class */
2  class Packet {
3  public:
4    long unusedlong;
5    void *unusedptr;
6    void *data;
7    char unusedchar;
8    int length; // <-- accessed
9  };
10 /* Access Example */
11 Packet p;
12 p.length = 100;
```

```
; Class Declaration (line 2-9)
%class.Packet = type { i64, i8*, i8*, i8, i32 }
; Object Definition + Initialization (line 11-12)
%1 = alloca %class.Packet, align 8 ; Allocate p
%2 = getelementptr inbounds %class.Packet,
    %class.Packet* %1, i32 0, i32 4 ; Get addr. of length
store i32 100, i32* %2, align 4 ; Store 100
```

Listing 4: Accessing a metadata field in LLVM IR bitcode (bottom). The top shows the C++ version of the code.

Challenges. While some compilers (e.g., Rust) support structure reordering [82], C & C++ compilers are forbidden to reorder data structures (e.g., struct or class) [74], which makes reordering variables/fields of a data structure at IR level challenging. When compilers make some assumptions about a data structure's order [72–74], reordering cannot be done for *any* data structures *without* careful consideration. Particularly, it requires deep knowledge of the workflow of the code and the relationship between different data structures. For instance, reordering the data structures that exchange data with hardware could break the program's correctness. Two common scenarios where this problem can occur are: ① when a piece of code relies on the order of the variables in a data structure (e.g., (i) using vector instruction to initialize or process a data structure and (ii) interacting with hardware) and ② when dynamically linked libraries access the data structure. Note that the references in statically linked libraries can be fixed (repaired) if we apply the reordering when the whole program's IR bitcode is available. Moreover, in case of aggregation/composition, it is important to fix the references to the container class/struct. Our pass currently does not perform any verification, but it is possible to verify the correctness of the NF when reordering the metadata, see §5.

Implementation. To mitigate these challenges, our pass reorders the metadata structure (i.e., `Packet` in FastClick) *only* when the metadata management model is set to use the Copying model. However, it is possible to apply a similar approach for the Overlaying model by extending our pass to reorder `rte_mbuf` & PMD's descriptors. To fix the references, our pass takes into account references done by `class.WritablePacket` and `class.PacketBatch`, which are dependent classes of `class.Packet`. We apply our pass via LLVM's `opt` on the pre code generation output of LTO (i.e., `click.0.5.precodegen.bc`) [56]. However, it would be possible to develop a custom LTO pass and then extend clang's C++ frontend to define a keyword for our proposed optimization.

To the best of our knowledge, PINstruct [46] is the only *published* work that has considered reordering data structures for a C program. They traced memory accesses via MemPin (a tool that uses the Intel Pin tool [60]) and reordered the OpenMPI data structures manually. However, they neither automated the data structure reordering nor evaluated its benefits.

4 EVALUATION

To better understand each optimization done by PacketMill, this section discusses them individually and then evaluates their impact on microarchitectural & application-level metrics. Later, we apply all of the optimizations together and demonstrate their combined impact.

NF configurations. We focus on five network functions: (i) a simple forwarder, (ii) a router, (iii) an IDS followed by the router, (iv) a Network Address Translation (NAT), and (v) a synthetic memory- & compute-intensive NF, check Appendix A for details. The simple forwarder & the router represents those scenarios where a network function is relatively *I/O bound* rather than *CPU-bound*, whereas the IDS+router, the NAT, and the synthetic NF demonstrate more sophisticated network functions that require more processing.

Testbed. We use a testbed with two (Skylake) servers equipped with Mellanox ConnectX-5 VPI and interconnected via a 100-Gbps link. One server acts as a packet generator, and generates/sinks

*The *real* number of accesses depends on the received workload.

†It calculates the address of a sub-element of an aggregate data structure.

‡Note that DPDK currently only supports LTO for gcc and icc that produce `fat-lto-objects` containing both machine and IR codes.

packets and measures the end-to-end latency & throughput, while the other server acts as a Device Under Test (DUT) and processes packets based on the given input NF configuration file. The packet generator and the DUT are equipped with 2×8-core Xeon Gold 6134 @ 3.2 GHz and 2×18-core Xeon Gold 6140 @ 2.3 GHz (nominal frequency), respectively. Both servers run Ubuntu 18.04.4 (Linux kernel 4.15.0-112). We use `perf` to measure microarchitectural metrics. Additionally, we isolate the DUT's CPU socket on which we run the experiment to increase our measurement accuracy. We use LLVM 10.0.0 (trunk 375507) to compile and optimize FastClick [6] and DPDK (v20.02). To prevent Intel Data Direct I/O (DDIO) from becoming a bottleneck in our measurements, we change `IIO LLC WAYS`' register's value to `0x7F8` (i.e., 8 set bits) [25]. Additionally, we set the uncore frequency to 2.4 GHz (i.e., the maximum frequency in our testbed) to minimize DRAM and LLC latency [35, 88]. We use the Copying model (i.e., the default metadata management model in FastClick) unless stated otherwise. We use the Network Performance Framework (NPF) tool [93] to facilitate reproducibility of our tests.

Generated traffic. We use two types of traffic in our evaluation: (i) a 28-min campus trace and (ii) synthetically generated traces with fixed-size packets (see §4.3 and §4.6). The campus trace which has 799 M packets with an average size of 981 B. In each run, we replay the first two million packets of the trace 25 times. We repeat each test five times and report the median values when there are no error bars. Note that the achieved throughput is proportional to processed packets packets per second (pps) × packet size. Therefore, replaying a trace with smaller average packet size could result in lower throughput, but the same improvements (i.e., more pps).

4.1 Do PacketMill's Code Optimizations Improve Packet Processing at 100 Gbps?

We evaluate the router's performance when processing the repeated campus trace at different clock frequencies. We change the processor's clock frequency of the DUT to assess the impact of PacketMill on different classes of processors, i.e., more cores with lower frequency vs. fewer cores with higher frequency. Additionally, reducing the clock frequency somewhat emulates the situation where the processor receives traffic at a higher rate (than the injected rate, e.g., >100 Gbps) or when the NF is more CPU-bound.

Configuration-based optimizations. Figure 4 & Table 1 show the results of our experiments when applying different source-code optimization techniques: (i) devirtualization (done by `click-devirtualize`), (ii) constant embedding, and (iii) static graph (i.e., defining the elements statically and their connections in the source code). These results demonstrate that all techniques & their combination have positive impact on the number of cache misses, throughput, and median latency. More specifically, using a static graph rather than a dynamic one improves throughput by up to 20% (or 14.8 Gbps) and dramatically reduces the *LLC misses* (up to ~300×), see the second row of Table 1. Note that 10-Gbps-throughput improvements are significant, as they would translate to supporting more 10-Gbps links, thus reducing the number of NFs in the network.

LTO & structure reordering. Applying LTO and reordering `Packet` class of FastClick for the router configuration (running at 3 GHz) increases throughput and decreases median latency, at no additional cost, by up to 5.4 Gbps (6.8%) and 13 µs (~3.8%), respectively. Reordering contributes to one third of the improvements. As mentioned earlier, these sub-ten-percent improvements should not be ignored, as these are essential for cost effectively deploying Internet services, facilitating service providers meeting their Service Level Objectives (SLOs). Note that other frequencies also result in similar improvements.

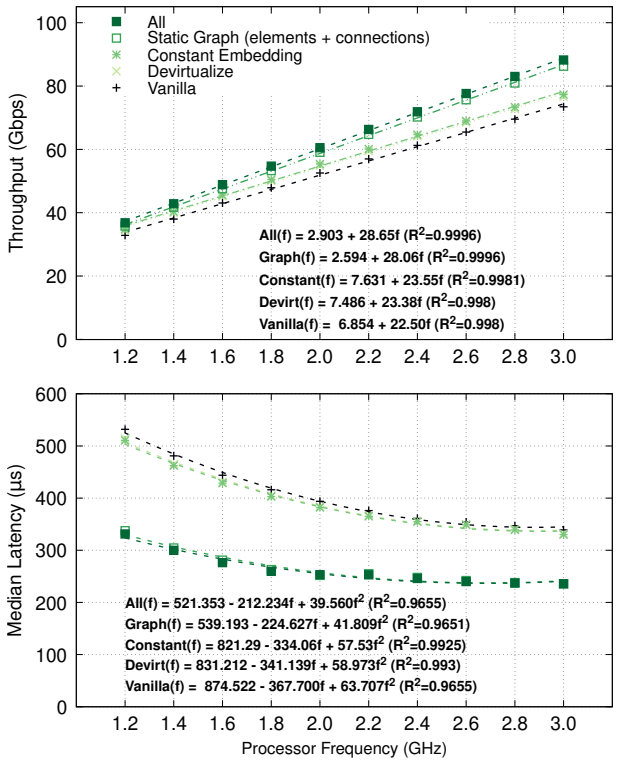


Figure 4: Exploiting the information in the router configuration improves throughput and median latency. The server is processing packets with *one* core running at different frequencies (f in GHz).

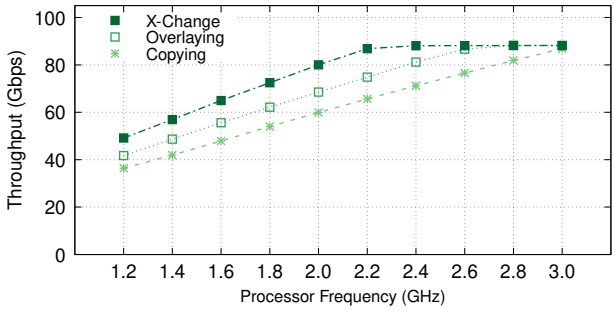
Table 1: PacketMill's code optimizations improve microarchitectural metrics by up to 300× (i.e., reducing the number of LLC load misses). We measure cache misses & IPC with `perf` every 100 ms and report average measured during the experiment, performed at 3 GHz.

Metric \ Scenario	Vanilla	Devirtualization	Constant Embedding	Static Graph	All
LLC kilo loads	1097	1159	1176	24	26
LLC kilo load-misses	803	841	845	0.98	2.58
instructions per cycle (IPC)	2.24	2.30	2.28	2.58	2.59
Million packets per second (Mpps)	8.66	9.05	9.12	10.16	10.41

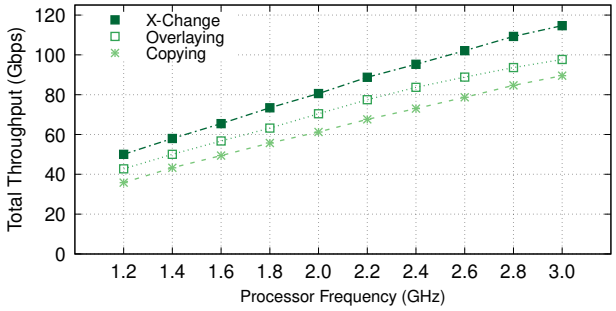
4.2 How Effective is PacketMill's Model (X-Change) Compared to the Existing Metadata Management Models?

We use FastClick to compare all three methods (i.e., Copying, Overlaying, and X-Change) for a simple forwarding configuration. We disable PacketMill's code optimizations to examine the impact of metadata management alone. We enable LTO in all scenarios to have the best achievable performance of each model. Note that disabling LTO could underestimate the benefits of X-Change, due to not inlined conversion calls. However, DPDK could be recompiled with X-Change's source file included as a header file to achieve a similar result *without* LTO.

Figure 5a demonstrates that PacketMill's metadata management model (X-Change) improves throughput significantly by mitigating inefficiencies of the other two models. These results show that increasing the processor's frequency does not improve the throughput of X-Change & Overlaying models after a certain frequency (i.e., 2.2 GHz & 2.6 GHz, respectively), as there may be other bottlenecks in the system (e.g., using one RX/TX queue or other NIC-related issues [42]). To investigate the full potential of X-Change, we set up a 200-Gbps testbed, where two servers generate traffic toward the DUT equipped with two NICs (connected to the *same* CPU socket). DUT forwards the received packets from the two generators via only *one* core.



(a) One NIC via one core.



(b) Two NICs via one core.

Figure 5: X-Change makes it possible to forward packets at >100-Gbps rates. The experiments with two NICs reports the sum of throughput achieved by one core.

Figure 5b shows that X-Change is the only metadata management model which enables a *single* core to forward packets at >100 Gbps. Additionally, both Figure 5a and 5b show that Overlaying model performs better than Copying, as Copying involves double copy/transform operations. Moreover, our measurements show that X-Change significantly reduces the number of LLC load misses; for instance, the X-Change-enabled forwarder running at 3 GHz *only* results in ~200 misses per 100 ms, whereas Copying and Overlaying cause ~3000 and ~6000 misses per 100 ms, respectively . We also observed that the performance of Copying+Overlaying method (used by VPP) is similar to Copying model. In summary, an inefficient metadata management model prevents the system from processing packets at higher rate, i.e., degrades throughput by >10 Gbps (i.e., *typical* data center link speed).

4.3 How does the Workload/Trace Affect PacketMill?

We have performed most of our experiments with the campus trace, as we thought using fixed-size packets could increase the effect of measurement/layout bias. Additionally, we reported throughput in bytes per second since we were targeting 100-Gbps networking. In this section, we use FastClick to generate fixed-size packets to show that PacketMill's improvements are not trace-dependent. Figure 6 reports the throughput in both bytes per second and packets per second (PPS) for a router running at 2.3 GHz while receiving fixed-size packets. It shows that PacketMill's improvements are consistent for different packet sizes, as long as there are no other bottlenecks in the system. Note that increasing the packet size after a certain point (e.g., ~800 B) would reduce the number of processed packets per second, due to the PCIe bandwidth [25, 67].

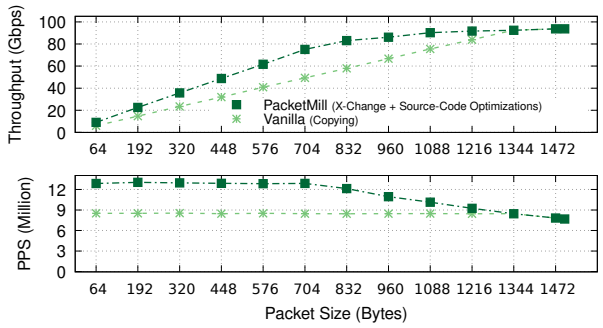
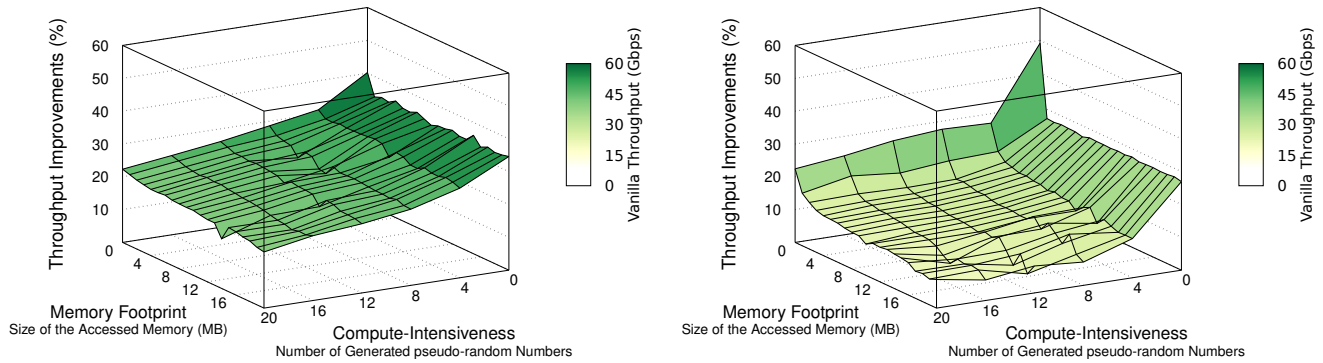


Figure 6: For a router running at 2.3 GHz, PacketMill improves the number of processed packets per second for different packet sizes.

4.4 How about more Sophisticated Network Functions?

So far, we have shown the individual (§4.2, §4.2) and combined* benefits (see Figure 1 and §4.3) of using different optimizations proposed by PacketMill on the router and the simple forwarder configuration. We showed that using PacketMill improves the

*Combined impact does not take into account data structure reordering.



(a) 1 access per packet (N = 1).

(b) 5 accesses per packet (N = 5).

Figure 7: PacketMill is effective for sophisticated network functions. Left and right figures show synthetic NFs that perform one and five memory accesses per packet, respectively. The improvements reduces when memory footprint, compute-intensiveness, and/or number of accesses per packet of an NF increases. A WorkPackage+router is running at 2.3 GHz**.**

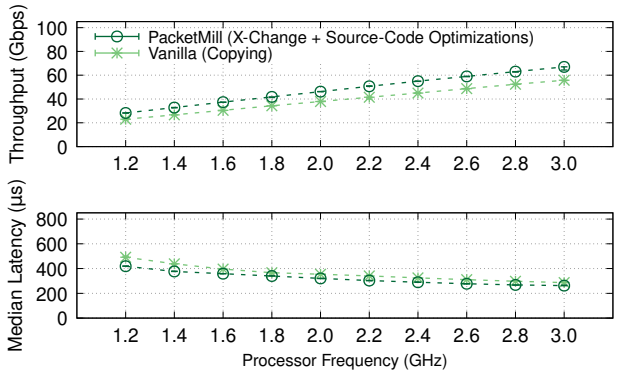


Figure 8: PacketMill improves the performance of a more compute-intensive NF (i.e., an IDS+router running at 2.3 GHz**), by up to 20% throughput and 17% latency.**

system's performance when performing relatively lightweight processing. This section investigates the impact of PacketMill on more sophisticated NFs. We start by applying PacketMill to an IDS followed by a router, which requires more processing to check the correctness of TCP, UDP, and ICMP headers and encapsulating the packet in a VLAN header. Figure 8 shows the throughput & median latency vs. frequency curve for the IDS+router. These results demonstrate that PacketMill is also beneficial for more CPU-demanding NFs.

To generalize this notion to more sophisticated NFs, we use FastClick's `WorkPackage` [3] element to emulate the behavior of more memory- and compute-intensive functions. This element generates N (1 and 5 in our configuration) random accesses (per-packet) to a static memory of S MB. Additionally, it generates W pseudo-random numbers to simulate more CPU-bound workloads. Figures 7a and 7b show the 3D colormaps of improvements for different values of W and S, when a core is performing different number of random accesses per packet (i.e., N). In these figures, X & Y axes represent W & S while the Z axis & colormap show the PacketMill's improvements & Vanilla's throughput, respectively. While it is difficult to come up with a unified benchmark that succinctly captures a wide variety of applications, these results demonstrates that PacketMill is beneficial for a wide range of CPU- and memory-bound NFs. PacketMill's gain reduces when the application becomes less I/O intensive; in other words when its throughput decreases, i.e., the lighter the color, the lower the Z. Additionally, comparing Figures 7a and 7b (i.e., $N = 1$ vs. $N = 5$) shows that increasing the number of accesses per packet amplifies the impact of increasing memory intensiveness, reducing Vanilla's throughput and PacketMill's improvements. The key behind these gains is PacketMill's highly efficient use of the underlying hardware. It is worth mentioning that the results of this section could underestimate PacketMill's improvements for real NFs, as the emulated NFs presented in this section do not contain a complicated processing graph (as opposed to real NF chains).

Impact of memory intensiveness. To have a more detailed understanding of memory intensiveness, we zoom into a slice of Figure 7a where an NF performs a single memory access per packet ($N = 1$) and generates four random numbers ($W = 4$), i.e., doing light-weight processing*. Figure 9 shows the impact of changing memory footprint on throughput, LLC load misses, and LLC kilo loads. Comparing the three sub-figures, we can make the following observations:

(1) We notice that Vanilla's throughput is inversely proportional to Vanilla's LLC loads–PacketMill also shows a similar behavior.
(2) The number of LLC loads gets saturated when the size of the accessed memory is increased to ~3 MB, which suggests the threshold where almost all of the memory accesses are being done through LLC, see the bottom sub-figure in Figure 9.

*This specific slice can represent an emulated simple Key-Value Store (KVS) with variable memory footprints.

(3) The number of LLC loads is never zero, even for accessed memory sizes of smaller than 1 MB. This observation implies that the application is still not L1/L2 bound, as there is always a considerable amount of LLC accesses, most probably due to the application footprint and DDIO [25].
(4) The percentage of LLC load misses increases after ~14 MB, highlighting the point where the application starts accessing the main memory (i.e., DRAM), see the middle sub-figure in Figure 9. However, the performance does not get substantially affected, as significant number of LLC loads are still hitting LLC (i.e., ~90% hits).
(5) PacketMill results in more LLC loads and LLC load misses per 100 ms, as PacketMill is processing more packets.
(6) PacketMill's improvements are consistent for this specific NF that performs one memory access per packet.

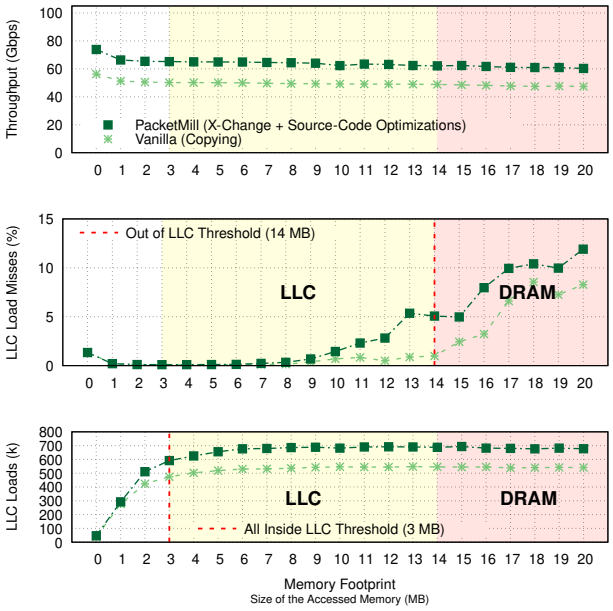


Figure 9: Increasing memory intensiveness results in larger number of LLC loads that is inversely proportional to the performance of the synthetic NF. From top to bottom, the figures show throughput, LLC load misses, and LLC loads, respectively.

4.5 Is PacketMill Useful for Multicore Network Functions?

The evaluation mainly focused on showing the single-core performance to highlight the gains achieved by our approach. Figure 10 shows that applying PacketMill to multicore NFs is also beneficial; in this case, for a NAT with different numbers of cores*. These results demonstrate that the benefits of applying PacketMill to multicore NFs is comparable to the single-core improvements shown in Figure 7.

*We use RSS to distribute packet among different cores.

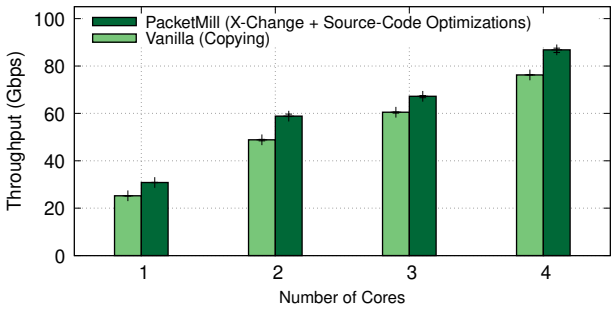


Figure 10: PacketMill also improves the performance of multicore NFs. A NAT is running at 2.3 GHz**.**

4.6 How about state-of-the-art Packet Processing Frameworks?

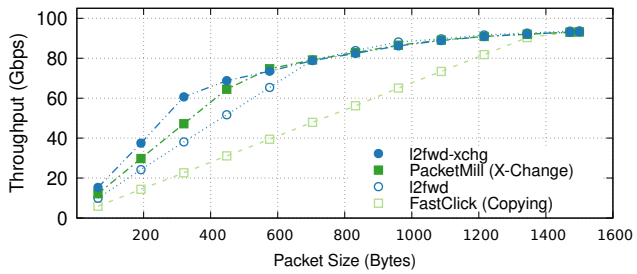
A fair comparison between PacketMill and state-of-the-art packet processing frameworks (e.g., BESS [36, 37] & VPP [2, 27]) requires (i) developing a high-performance NF in pure DPDK, (ii) modifying those other frameworks to enable the case for source-to-source optimizations & LLVM pass that reorders metadata, and (ii) devising scenarios/experiments to avoid any incorrect conclusions, which is beyond the scope of this paper (as done by [102] previously in a separate research publication). However, we have performed a simple comparison to, specifically, show the full potential of X-Change. This section compares the performance of a simple forwarding application running via FastClick, PacketMill, DPDK, DPDK+X-Change, BESS, and VPP.

For the DPDK+X-Change case, we developed a sample application, called `l2fwd-xchg`, for DPDK to support X-Change, which is a modified version of the L2 forwarding sample application (`l2fwd`). In this example, the metadata is reduced to *two* simple fields (i.e., the buffer address and packet length) instead of the 128-B `rte_mbuf`. This application† can also serve as a template for developers to write their own applications, benefiting from X-Change. Note that since X-Change currently does not support vectorized PMD, we disabled it for all experiments. However, this should not affect the improvement trend, as a full vectorized implementation of X-Change would still result in the same benefits, addressing the inefficiencies of current metadata management models. Extending X-Change to support vectorized PMD remains as our future work.

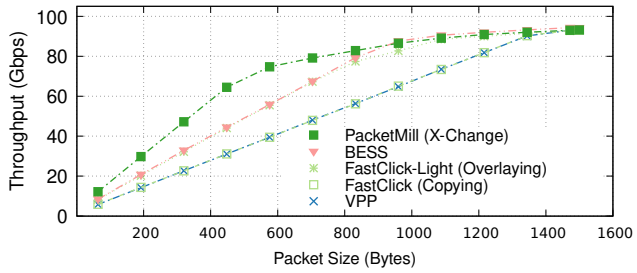
Figure 11 shows the results of our experiments when different frameworks/applications are forwarding fixed-size packets while a single core is running at 1.2 GHz. Increasing the frequency would eventually result in the same behavior as Figure 5a, as these applications perform simple forwarding operations, hiding the full potential of X-Change due to other bottlenecks (e.g., using one RX/TX queues).

FastClick vs. DPDK. Figure 11a compares the performance of DPDK-based forwarding applications with default FastClick (i.e., it uses Copying model) and PacketMill (i.e., it uses X-Change). These results shows that PacketMill enables FastClick to process packets

†It is available at tbarbette/xchange/examples/l2fwd-xchg



(a) X-Change improves the performance of two DPDK-based applications (i.e., FastClick and `l2fwd` application). Moreover, PacketMill enables FastClick to process packets faster than a simple forwarding application (i.e.,`l2fwd`) with limited metadata.



(b) PacketMill performs better than other packet processing frameworks.

Figure 11: Comparison of packet processing frameworks forwarding fixed-size packets with a single core.

faster than `l2fwd` that is a simple forwarding application with minimal features & footprint and limited metadata. Additionally, it shows that applying X-Change to even simple DPDK-based applications could result in significant improvements, as `l2fwd-xchg` is forwarding packets up to ~59% faster than `l2fwd`.

FastClick vs. BESS vs. VPP. Figure 11b compares the achieved throughput of different modular packet processing frameworks when they *only* perform simple forwarding. These results show that PacketMill achieves the best overall performance. Additionally, both VPP and default FastClick achieve similar performance, as both of them use the Copying model that performs an extra copying. Moreover, these results show that FastClick can achieve similar throughput as BESS after disabling extra features and using the Overlaying model .

5 FREQUENTLY ASKED QUESTIONS

Why should I use PacketMill instead of PGO? Specializing a binary for a specific workload via profiling + re-compiling, aka profile-guided optimization (PGO), has recently gained a lot of interest, due to modular programming language (PL) tools (e.g., LLVM toolchain) and convenient/accessible profiling tools (e.g., perf). For example, Bolt [75] and Propeller [33] have shown that exploiting dynamic program control flow information, extracted via profiling, could improve large-scale applications' performance. However, these techniques work best when there exists *a* defined workload, rather than different and varying workloads.

Software packet processing frameworks typically have to process various kinds of packets. For instance, a router should handle ARP requests while forwarding different versions/types of IP packets, which requires different execution paths, thereby reducing cache locality. To improve locality, Metron [43] exploited modern networking equipment to distribute traffic classes to different cores–i.e., each core operates on a specific workload independently. Although Metron defines a per-core workload, it cannot substantially benefit from PGO* since it relies on one codebase for all execution paths (or traffic classes). PacketMill could be extended to duplicate the necessary code per-traffic class and benefit from PGO, which remains as our future work.

Does PacketMill affect the correctness? Premature optimizations could be *the root of all evil* [47], as it may result in unexpected bugs. While our goal was to emphasize the impact of low-level optimizations for 100-Gbps packet processing, deploying optimized network functions should be accompanied by a verification stage to *formally* prove the correctness of the optimized NF and avoid bugs. Since PacketMill relies on optimizing the whole IR code, it is possible to integrate our system with a IR-based symbolic execution engine (e.g., KLEE [13]), as done by VigNAT [100] and Vigor [99]. Furthermore, using a symbolic execution engine could facilitate further optimizations as demonstrated in NFReducer [17] and Castan [78].

Why should I trust your measurements? Mytkowicz et al. [66] showed that measurement bias is a common phenomenon in evaluating computer systems. We have taken the following measures to *avoid* drawing incorrect conclusions. ① We use NPF to perform our experiments to ensure testbed consistency– as it clones/compiles repositories, sets up/configures the testbed, and collects measurements. Additionally, it randomizes the environment variables in each run to mitigate the measurement bias due to stack alignment. ② We randomize the location where the binary is loaded for each run of the experiment using Address Space Layout Randomization (ASLR). ③ We use different NF configurations and various traces to broaden our evaluation space. The best approach to mitigate measurement bias is to use/develop tools, such as STABILIZER [16].

Is PacketMill applicable to other frameworks? Although we have designed PacketMill with an eye toward optimizing click-based packet processing, our optimizations are not limited to Click, as the code inefficiencies (e.g., unembedded graph and parameters) are also present in other packet processing frameworks such as BESS and VPP. Moreover, PacketMill's X-Change can be easily adapted to improve the performance of other frameworks since it modifies DPDK userspace drivers, making it a common optimization for DPDK-based frameworks. As our target was to achieve 100-Gbps per core, we modified the MLX5 driver used by Mellanox NICs. However, X-Change is applicable to other drivers, as other (e.g., Intel) drivers are implemented similarly and have the same inefficiencies. Moreover, our techniques/optimizations could be combined with other existing middlebox optimizations such as Metron [43], see Appendix A.

Would PacketMill still be relevant given current technology trends? While many try to exploit recent accelerators to improve

*See PacketMill's public repository for more information, see Appendix B.

packet processing, we think our approach would still be relevant for two reasons: (i) current accelerators (e.g., programmable switches & NICs) have many limitations, which force applications to perform all stateful processing in the commodity hardware, and (ii) our research shows that performing holistic optimizations makes it possible to achieve performance similar to accelerators while still benefiting from the flexibility of commodity hardware. Moreover, we believe PacketMill optimizations become even more critical with increasing link speeds (i.e., 200 and 400 Gbps).

What are future directions for PacketMill? PacketMill's use of LTO and access to the whole program's IR facilitates application of additional techniques to improve/evaluate the performance of software packet processing frameworks, such as: llvm-mca [10] for performance estimation, IR-based *superoptimizers* [62] (e.g., Souper [34, 84]), and/or customized code generation/instruction scheduling [59]. Additionally, it is possible to extend our code optimizations with new passes, e.g., performing liveness analysis to overlay the metadata that is not alive at the same processing stage. Examining these techniques remains as future work.

6 RELATED WORKS

In addition to the works discussed throughout the paper, the work on NFV performance acceleration can be classified into three categories: ① relies on hardware accelerators to improve processing speed by offloading (part of) packet processing into an FPGA, GPU, or modern NIC [20, 28, 45, 52, 53, 69, 87, 96, 98, 101, 104, 105]; ② focuses on NFV execution models and tries to improve the performance of either the pipeline/parallelism model [43, 55, 61, 86, 103] or run-to-completion (RTC) model [37, 76]; and ③ improves the performance of NFV by reducing/eliminating redundant operations and/or merging similar packet processing elements into (one) consolidated optimized equivalent [1, 12, 40, 44, 55, 85]. The second category also includes efforts toward better scheduling & load balancing [4, 5, 7, 41, 50, 94] or more efficient I/O [24, 25]. Our work is orthogonal and complementary to these.

7 CONCLUSION

Despite the availability of 100- & 200-Gbps interfaces, networked systems are generally unable to operate at these rates, due to both software and hardware limitations. PacketMill addresses some of these software limitations in packet processing by mitigating code inefficiencies & improving the metadata management. Our goal was to highlight the importance of *low-level* optimizations in order to utilize the full potential of commodity hardware. Our main takeaway is that efficient packet processing at multi-hundred-gigabit rates calls for holistic system optimizations, i.e., *milling* the whole solution/software stack to *squeeze* every bit of performance from the available hardware. We hope our work motivates the system community to pay extra attention to multi-hundred-gigabit networking.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Jacob Nelson, for his insightful suggestions; the anonymous reviewers for their valuable comments; the anonymous artifact evaluators for their effort to review our artifact; and Voravit Tanyingyong for providing feedback on the paper. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work was also funded by the Swedish Foundation for Strategic Research (SSF). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

A NETWORK FUNCTION CONFIGURATIONS

This section explains the details of NF configurations used in the evaluation section (§4).

A.1 Simpler Forwarder

The simpler forwarder receives packets from DPDK, rewrites the Ethernet MAC addresses via `EtherRewrite` element, and transmits them.

A.2 Router

The router configuration is the standard Click router, compliant with IP routing standards[65]. The processing graph produced by this configuration exhibits multiple, more complex paths. The whole IP header is loaded in memory, as the routing element (with only *one* rule per port) does a lookup for each destination IP address. We do not consider the impact of more rules, as they could be potentially offloaded to the NIC [43].

A.3 NAT, IDS, and VLAN

All these configurations are optional supplements for the two previous configurations (i.e., the simple forwarder and the router). The NAT is a standard Network Address and Port Translation (NAPT) that rewrites source IP addresses of outgoing packets. The NAT configuration is stateful and it uses the DPDK Cuckoo hash table, resulting in more lookups and higher memory usage. The IDS checks the correctness of TCP, UDP, and ICMP headers, except for the checksum that can be verified in hardware. The VLAN supplement eventually encapsulates the packet in a VLAN header.

A.4 WorkPackage

This configuration uses `WorkPackage` element that is a purely synthetic element built for microbenchmarking [3]. It generates N random accesses to a static memory of S MB. Additionally, it generates W pseudo-random numbers to artificially emulate more CPU-bound workloads. The WorkPackage element enables us to study the impact of our optimizations on more memory- & compute-intensive configurations. The WorkPackage is also supplementary to the base configurations. In our tests, we use it along with the forwarding configuration.

B ARTIFACT APPENDIX

B.1 Abstract

PacketMill is a system that optimizes the performance of network functions via holistic inter-stack optimizations. More specifically, PacketMill provides a new metadata management model, called X-Change, enabling the packet processing frameworks to provide their custom buffer to DPDK and fully bypass rte_mbuf.

Additionally, PacketMill performs a set of source-code & IR-code optimizations.

Our paper's artifact contains the source code, the experimental workflow, and additional information to (i) set up PacketMill & its testbed, (ii) perform some of the experiments presented in the paper, and (iii) validates the reusability & effectiveness of PacketMill.

B.2 Artifact check-list (meta-information)

- **Program:** FastClick (PacketMill branch), DPDK (with X-Change modifications), and Network Performance Framework (NPF).
- **Compilation:** LLVM Toolchain and Clang (10.0).
- **Hardware:** Mellanox ConnectX-5 (Ethernet or VPI in Ethernet mode) and Intel Xeon processors.
- **Metrics:** Throughput and Latency.
- **Output:** An optimized `click` binary and its measured performance while receiving traffic.
- **How much disk space required (approximately)?:** ≈20 GB.
- **How much time is needed to prepare workflow (approximately)?:** ≈1 hour.
- **How much time is needed to complete experiments (approximately)?:** ≈1 day.
- **Publicly available?:** Code, measured results, and experiments are publicly available. The only unpublished material is our campus trace (due to GDPR).
- **Code licenses (if publicly available)?:** We have three repositories: (i) PacketMill repo has MIT License, (ii) X-Change uses the same license as DPDK (i.e., BSD), and (iii) FastClick (PacketMill branch) uses the same license as Click/FastClick (i.e., MIT license).
- **Workflow framework used?:** We use Network Performance Framework (NPF) [93] to clone/compile repositories, set up/configure the testbed, and collect measured results.
- **Archived (provide DOI)?:** 10.5281/zenodo.4435970

B.3 Description

B.3.1 How to access. Our artifact and guidelines for using/evaluating PacketMill is publicly available at the following GitHub repositories:

- **PacketMill source:** https://github.com/tbarbette/fastclick/tree/packetmill
- **X-Change:** https://github.com/tbarbette/xchange
- **Experimental workflow and LLVM Optimization Passes:** https://github.com/aliireza/packetmill

Please refer to PacketMill main **`README.md`** for more information: `https://github.com/aliireza/packetmill/blob/master/README.md`

B.3.2 Hardware dependencies. PacketMill's metadata management model (X-Change) only supports MLX5 driver in DPDK. Although MLX5 driver is used by several Mellanox NICs, we have only tested Mellanox Connect-X 5 NICs. To perform PacketMill's experiments, you need two servers (preferably with Xeon processors) equipped with Mellanox Connect-X 5 NICs and interconnected via a 100-Gbps link.

B.3.3 Software dependencies. Our source-code & IR-code optimizations currently only works on FastClick (PacketMill branch). Additionally, IR-code optimizations currently only supports Copying model (i.e., the default model in FastClick).

B.4 Installation & Experiment Workflow

PacketMill's `README.md` (https://github.com/aliireza/packetmill/blob/master/README.md) describes the testbed preparation, installation process, and the experimental workflow to use PacketMill and perform different experiments.

B.5 Evaluation and Expected Results

This section explains the available experiments in our repository and their expected/sample results.

Most of the experiments in our paper have been performed using a captured trace from a campus network. Unfortunately, we are unable to make the campus trace available to the public due to GDPR. While it is possible to use our experiments with other traces, our artifact provides some scripts to perform some experiments with synthetic traces (i.e., using fixed-size packets) to validate the reusability and effectiveness of PacketMill. Our artifact also includes (i) the experimental data captured using our campus trace, and (ii) the scripts to generate the graphs presented in the paper.

B.5.1 Source-code Modifications (Router). This experiment shows the benefits of using our proposed source-code optimizations when a router is receiving fixed-size packets (e.g., 64-B and 1024-B packets). The results should follow a similar trend as Figure 4. This experiment uses Copying metadata management model.

B.5.2 X-Change (Forwarder). This experiment compares the performance of different metadata management model: (i) Copying, (ii) Overlaying, and (iii) X-Change, where a single core is forwarding fixed-size packets. The output results should be similar to Figure 5a. Note that this experiment uses LTO in all configurations.

B.5.3 PacketMill (Router). This experiment applies PacketMill to a router that receives fixed-size packets. The results should result in similar improvements as Figure 1. As opposed to Figure 1, this experiment uses a fixed-size packets and a fixed rate, which only considers the "saturated" condition in that figure.

B.5.4 Sophisticated Network Functions (Router+IDS+VLAN). This experiment demonstrates the benefits of PacketMill for a router+IDS configuration, where a single core is receiving fixed-size packets. The expected results should be similar to Figure 8.

B.5.5 Multicore Network Functions (Router+NAT). This experiment shows the benefits of PacketMill for a router+NAT configuration, where different number of cores are receiving 1024-B packets. The expected results should be similar to Figure 10.

B.5.6 Others. We believe the five mentioned experiments are sufficient to demonstrate the reusability & effectiveness of PacketMill. However, we have included more experiments in our publicly available repository, which can benefit the community. Please check PacketMill's experiments `README.md` for more information.

REFERENCES

[1] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. 2015. Programming Slick Network Functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/2774993.2774998

[2] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine* 56, 12 (2018), 97–103. https://doi.org/10.1109/MCOM.2018.1800069

[3] Tom Barbette. 2018. *Architecture for programmable network infrastructure.* Ph.D. Dissertation. University of Liege. http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1249035, accessed 2020-12-23.

[4] Tom Barbette, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. *Stateless CPU-Aware Datacenter Load-Balancing.* Association for Computing Machinery, New York, NY, USA, 548–549. https://doi.org/10.1145/3386367.3431672

[5] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3359989.3365412

[6] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Oakland, California, USA) *(ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16. https://doi.org/10.1109/ANCS.2015.7110116

[7] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[8] BESS. 2017. sn_buff Layout. https://github.com/NetSys/bess/blob/master/core/snbuf_layout.h.

[9] BESS. 2019. Packet. https://github.com/NetSys/bess/blob/master/core/packet.h.

[10] Andrea Di Biagio and Matt Davis. 2020. llvm-mca - LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html, accessed 2020-06-15.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

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

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[14] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle. 2018. Fast Packet Processing: A Survey. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 3645–3676. https://doi.org/10.1109/COMST.2018.2851072

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*. New York, NY, USA, 12–23.

[16] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/2451116.2451141

[17] Bangwen Deng, Wenfei Wu, and Linhai Song. 2020. Redundant Logic Elimination in Network Functions. In *Proceedings of the Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '20)*. Association for Computing Machinery, New York, NY, USA, 34–40. https://doi.org/10.1145/3373360.3380832

[18] DPDK. 2020. Data Plane Development Kit (DPDK). https://dpdk.org.

[19] DPDK. 2020. Mbuf Library. https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html.

[20] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 345–362. https://www.usenix.org/conference/atc19/presentation/eran

[21] Ericsson. 2017. *Supercharging the Evolved Packet Gateway.* Technical Report. Ericsson. https://www.ericsson.com/assets/local/digital-services/doc/Supercharging-the-Evolved-Packet-Gateway.pdf https://www.ericsson.com/assets/local/digital-services/doc/Supercharging-the-Evolved-Packet-Gateway.pdf, accessed 2020-07-24.

[22] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[23] Alireza Farshin and Tom Barbette. 2021. *PacketMill: Toward per-core 100-Gbps Networking - Artifact for ASPLOS'21.* https://doi.org/10.5281/zenodo.4435970 Note that this is just an archive for ASPLOS'21 artifact evaluation; you can access the latest version at https://github.com/aliireza/packetmill.

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

[25] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 673–689. https://www.usenix.org/conference/atc20/presentation/farshin

[26] FastClick. 2019. Packet Class. https://github.com/tbarbette/fastclick/blob/master/include/click/packet.hh.

[27] FD.io. 2017. *Vector Packet Processing - One Terabit Software Router on Intel Xeon Scalable Processor Family Server.* Technical Report. Cisco, Intel Corporation, FD.io. https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf, accessed 2020-07-24.

[28] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

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

[30] GCC. 2009. Link Time Optimization. https://gcc.gnu.org/wiki/LinkTimeOptimization, accessed 2020-06-15.

[31] Taras Glek and Jan HubiČka. 2010. Optimizing real world applications with GCC Link Time Optimization. arXiv:1010.2196 [cs.PL] http://sciencewise.info/media/pdf/1010.2196v2.pdf, accessed 2020-06-15.

[32] Matt Godbolt. 2020. Optimizations in C++ Compilers. *Commun. ACM* 63, 2 (Jan. 2020), 41–49. https://doi.org/10.1145/3369754

[33] Google. 2020. GitHub - Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller, accessed 2020-06-15.

[34] Google. 2020. GitHub - Souper: A superoptimizer for LLVM IR. https://github.com/google/souper, accessed 2020-06-15.

[35] Corey Gough, Ian Steiner, and Winston A. Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization* (1st ed.). Apress, USA.

[36] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html, accessed 2020-07-22.

[37] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware.* Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 195–206. https://doi.org/10.1145/1851275.1851207

[39] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[40] Y. Jiang, Y. Cui, W. Wu, Z. Xu, J. Gu, K. K. Ramakrishnan, Y. He, and X. Qian. 2019. SpeedyBox: Low-Latency NFV Service Chains with Cross-NF Runtime Consolidation. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 68–79. https://doi.org/10.1109/ICDCS.2019.00016

[41] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[42] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q. Maguire Jr. 2021. What you need to know about (Smart) Network Interface Cards. In *Proceedings of the Passive and Active Measurement (PAM) Conference*. Springer International Publishing.

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

[44] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr., and Dejan Kostić. 2016. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science* 2, e98. https://doi.org/10.7717/peerj-cs.98

[45] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[46] Rainer Keller and Shiqing Fan. 2013. *PINstruct – Efficient Memory Access to Data Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 127–128. https://doi.org/10.1007/978-3-642-35893-7_14

[47] Donald E. Knuth. 1974. Structured Programming with Go to Statements. *ACM Comput. Surv.* 6, 4 (Dec. 1974), 261–301. https://doi.org/10.1145/356635.356640

[48] Eddie Kohler, Robert Morris, and Benjie Chen. 2002. Programming Language Optimizations for Modular Router Configurations. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 251–263. https://doi.org/10.1145/605397.605424

[49] Maciek Konstantynowicz, Patrick Lu, and Shrikant M. Shah. 2017. *Benchmarking and Analysis of Software Data Planes*. Technical Report. Cisco, Intel Corporation, FD.io. https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf, accessed 2019-07-24.

[50] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. 2020. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. *IEEE/ACM Transactions on Networking* 28, 2 (2020), 639–652. https://doi.org/10.1109/TNET.2020.2969971

[51] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: Inter-Procedural Basic Block Layout Optimization. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 65–75. https://doi.org/10.1145/3302516.3307358

[52] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2934872.2934897

[53] X. Li, X. Wang, F. Liu, and H. Xu. 2018. DHL: Enabling Flexible Software Network Functions with FPGA Acceleration. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 1–11. https://doi.org/10.1109/ICDCS.2018.00011

[54] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. 2019. Survey of Performance Acceleration Techniques for Network Function Virtualization. *Proc. IEEE* 107, 4 (2019), 746–764. https://doi.org/10.1109/JPROC.2019.2896848

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

[56] LLVM. 2018. Four bitcode generated with plugin-opt=save-temps. http://lists.llvm.org/pipermail/llvm-dev/2018-May/123341.html, accessed 2020-06-15.

[57] LLVM. 2020. LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html, accessed 2020-06-15.

[58] LLVM. 2020. ThinLTO. https://clang.llvm.org/docs/ThinLTO.html, accessed 2020-06-15.

[59] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 17 (July 2019), 53 pages. https://doi.org/10.1145/3332373

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

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

[62] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 122–126. https://doi.org/10.1145/36206.36194

[63] Niall McDonnell and Gage Eads. 2020. Queue Management and Load Balancing on Intel Architecture. https://tinyurl.com/yxv9cgpj, accessed 2020-08-08.

[64] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. 2016. Dataplane Specialization for High-Performance OpenFlow Software Switching. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 539–552. https://doi.org/10.1145/2934872.2934887

[65] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. 1999. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 217–231. https://doi.org/10.1145/319151.319166

[66] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[67] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. ACM, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[68] Andy Newell and Sergey Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/tc.2020.2982888

[69] G. S. Niemiec, L. M. S. Batista, A. E. Schaeffer-Filho, and G. L. Nazar. 2020. A Survey on FPGA Support for the Feasible Execution of Virtualized Network Functions. *IEEE Communications Surveys Tutorials* 22, 1 (2020), 504–525. https://doi.org/10.1109/COMST.2019.2943690

[70] ntop. 2020. PF_RING ZC (Zero Copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/, accessed 2020-08-02.

[71] G. Ottoni and B. Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[72] Stack Overflow. 2008. Why doesn't GCC optimize structs? https://stackoverflow.com/questions/118068/why-doesnt-gcc-optimize-structs, accessed 2020-06-15.

[73] Stack Overflow. 2012. Why can't C compilers rearrange struct members to eliminate alignment padding? https://tinyurl.com/yxncnqk8, accessed 2020-08-07.

[74] Stack Overflow. 2016. Struct Reordering by compiler. https://stackoverflow.com/questions/38244689/struct-reordering-by-compiler, accessed 2020-06-15.

[75] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14.

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

[77] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho. 2015. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 86–92. https://doi.org/10.1109/NFV-SDN.2015.7387411

[78] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 372–385. https://doi.org/10.1145/3230543.3230573

[79] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch.

In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[80] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 225–241. https://www.usenix.org/conference/osdi20/presentation/pirelli

[81] Sekhar Reddy. 2014. What is SKB in Linux kernel? What are SKB operations? Memory Representation of SKB? How to send packet out using skb operations? http://amsekharkernel.blogspot.com/2014/08/what-is-skb-in-linux-kernel-what-are.html.

[82] The Rust Language Reference. 2008. Struct Types. https://github.com/rust-lang/reference/blob/master/src/types/struct.md, accessed 2020-06-15.

[83] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[84] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

[85] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 323–336. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar

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

[87] W. Sun and R. Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and click. In *Architectures for Networking and Communications Systems*. 25–35. https://doi.org/10.1109/ANCS.2013.6665173

[88] Vaibhav Sundriyal, Masha Sosonkina, Bryce M. Westheimer, and Mark Gordon. 2018. Comparisons of Core and Uncore Frequency Scaling Modes in Quantum Chemistry Application GAMESS. In *Proceedings of the High Performance Computing Symposium* (Baltimore, Maryland) *(HPC '18)*. Society for Computer Simulation International, San Diego, CA, USA, Article 13, 11 pages.

[89] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. 2018. Dark Packets and the End of Network Scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems* (Ithaca, New York) *(ANCS '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3230718.3230727

[90] Shelby Thomas, Geoffrey M. Voelker, and George Porter. 2018. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud18/presentation/thomas

[91] Georgii Tkachuk, Maciek Konstantynowicz, and Shrikant M. Shah. 2019. *Benchmarking Software Data Planes - Intel Xeon Skylake vs. Broadwell.* Technical Report. Cisco, Intel Corporation, FD.io. https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking_sw_data_planes_skx_bdx_mar07_2019.pdf https://www.lfnetworking.org/wp-content/uploads/sites/55/2019/03/benchmarking_sw_data_planes_skx_bdx_mar07_2019.pdf, accessed 2020-07-24.

[92] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/3302424.3303979

[93] Tom Barbette. 2020. Network Performance Framework (NPF). https://github.com/tbarbette/npf, accessed 2020-07-24.

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

[95] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, and D. Rossi. 2017. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Communications Magazine* 55, 3 (2017), 163–169. https://doi.org/10.1109/MCOM.2017.1600756CM

[96] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 321–332. https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[97] James M. Westall. 2011. Management of *sk_buffs*. https://people.cs.clemson.edu/~westall/853/notes/skbuff.pdf.

[98] Xiaodong Yi, Jingpu Duan, and Chuan Wu. 2017. GPUNFV: A GPU-Accelerated NFV System. In *Proceedings of the First Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/3106989.3106990

[99] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3341301.3359647

[100] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/3098822.3098833

[101] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU Sharing in NFV Systems. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 187–200. https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

[102] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. 2019. Comparing the Performance of State-of-the-Art Software Switches for NFV. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3359989.3365415

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

[104] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[105] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (2014), 32–41. https://doi.org/10.1109/MM.2014.61