Morpheus: A Run Time Compiler and Optimizer for Software Data Planes

Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, Gianni Antichi

***Abstract*—State-of-the-art approaches to design, develop and optimize software packet-processing programs are based on static compilation: the compiler's input is a description of the forwarding plane semantics and the output is a binary that can accommodate any control plane configuration or input traffic.**

In this paper, we demonstrate that tracking control plane actions and packet-level traffic dynamics at run time opens up new opportunities for code specialization. We present Morpheus, a system working alongside static compilers that continuously optimizes the targeted networking code. We introduce a number of new techniques, from static code analysis to adaptive code instrumentation, and we implement a toolbox of domain specific optimizations that are not restricted to a specific data plane framework or programming language. We apply Morpheus to several systems, from eBPF and DPDK programs including Katran, Meta's production-grade load balancer to container orchestration solutions such a Kubernets. We compare Morpheus to state-of-the-art optimization frameworks and show that it can bring up to 2x throughput improvement, while halving the 99th percentile latency.

***Index Terms*—Data Plane Compilation, LLVM, eBPF, XDP, DPDK**

I. INTRODUCTION

SOFTWARE Data Planes, packet processing programs implemented on commodity servers, are widely adopted in real deployments [2]–[9]. Since data plane programs tend to be performance-critical, the code is usually transformed through a sequence of offline optimization steps (e.g., inlining, loop unrolling, branch elimination, or vectorization [10], [11]) during the compilation process [12], [13]. These are mainly *static* transformations, independent of the actual input the code will process in operation, as this is unknown until then [14], [15]. Thus, the resulting code is *generic*, as it contains logic for protocols and features that may never be triggered in a deployment, performs costly memory loads to access values that are only known at run time, and takes difficult-to-predict branches conditioned on variable data.

Dynamic compilation, in contrast, enables program optimization based on invariant data computed at run time and produces code that is *specialized* to the input the program is processing [15], [23], [24]. The idea is to continuously collect run time data about program execution and then re-compile it to improve performance. This is a well-known practice adopted by generic programming languages (e.g., Java [15], JavaScript [24], and C/C++ [23]) and often produces orders of magnitude more efficient code as shown for data-caching services [16], data mining [17] and databases [25], [26]. Unfortunately, this is not the case for packet-processing programs [11], [18], [22], since their performance critically depends on *highly dynamic domain-specific knowledge*, such as traffic patterns, match-action table content, and network configuration (§II). Obtaining and tracking this knowledge efficiently is extremely challenging: lightweight *online* tracing tools (e.g., Linux `perf` [27]) are restricted to CPU performance counters, whereas capturing *all* domain-specific information requires tracking packet-level and instruction-level logs which is prohibitively costly. As an example, GCC FDO instrumentation, when applied in this context, may easily incur ∼900% mean overhead [28]. Therefore, existing solutions tailored for the networking domain (Table II) resort to *offline* profiling, which requires operators to collect representative samples of data-plane configuration and match-action tables from production deployments and still completely miss out on dynamic traffic-level insights. The main challenge we tackle in this paper is *unsupervised dynamic compilation* for network code, which captures just enough domain-specific knowledge to enable efficient dynamic performance optimization, but inexpensive enough to be run online, inside the data plane pipeline.

We present Morpheus, a system to optimize network code at run time using *domain-specific dynamic optimization techniques*. Morpheus operates in unsupervised mode: it does not require any *a priori* knowledge about control plane configuration or data plane traffic patterns. We discuss its design challenges (§III), such as automatically tracking highly variable input (e.g., inbound traffic) that may change tens, or hundreds of millions times *per second*. We show that the required profiling and tracing facilities, if implemented carelessly, can easily nullify the performance benefit of code specialization. We introduce several novel techniques; we leverage *static code analysis* to build an understanding of the program *offline* and propose a low-overhead *adaptive instrumentation* mechanism to minimize the amount of data collected *online*. Then, we invoke several dynamic *optimization passes* (e.g., dead code elimination, data-structure specialization, just-in-time compilation, and branch injection) to specialize the code against control plane actions and data plane traffic patterns. Finally, we

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART") and by the NKFIH/OTKA Project #135606, the MTA-BME Information Systems Research Group and the MTA-BME Network Softwarization Research Group. Preliminary results were presented at ACM ASPLOS '22 [1]. *(Corresponding author: Sebastiano Miano).*

Sebastiano Miano is with the Politecnico di Milano, 20158, Italy (email: sebastiano.miano@polimi.it).

Alireza Sanaee is with the Queen Mary University of London, E1 4NS, United Kingdom (email: a.sanaee@qmul.ac.uk).

Fulvio Risso is with Politecnico di Torino, 10129, Italy (email: fulvio.risso@polito.it).

Gábor Rétvári is with Budapest University of Technology and Economics & Ericsson Research, 1111, Hungary (email: retvari@tmit.bme.hu).

Gianni Antichi is with Politecnico di Milano, 20158, Italy (email: gianni.antichi@polimi.it).

Name	Domain specific	Unsupervised adaptation to control plane actions	Unsupervised adaptation to data plane traffic	Data plane agnostic	Description
Bolt [16]	✗	-	-	✓	Offline profile-guided optimizer for generic software code.
AutoFDO [17]	✗	-	-	✓	Offline profile-guided optimizer for generic software code.
eSwitch [18]	✓	✓	✗	✗	Policy-driven optimizer for DPDK-based OpenFlow software switches.
P5 [19]	✓	✗	✗	✗	Policy-driven optimizer for P4/RMT packet-processing pipelines.
P2GO [20]	✓	✗	✗	✗	Offline profile-guided optimizer for P4/RMT packet-processing pipelines.
PacketMill [21]	✓	✗	✗	✗	Packet metadata management optimizer for DPDK software data planes.
NFReducer [22]	✓	✗	✗	✓	Policy-driven optimizer for network function virtualization.
Morpheus	✓	✓	✓	✓	Run time compiler and optimizer framework for arbitrary networking code.

TABLE I
A COMPARISON OF SOME POPULAR DYNAMIC OPTIMIZATION FRAMEWORKS AND MORPHEUS.

protect the consistency of the specialized code against changes to input that is considered invariant by injecting *guards* (§IV).

Our implementation exploits the LLVM JIT compiler toolchain to apply the above ideas at the LLVM Intermediate Representation (IR) level in a generic fashion and separates data plane specific code to several backend plugins to minimize the effort in porting Morpheus to a new architecture (§V). The code currently contains an eBPF and a DPDK/C plugin. We apply Morpheus to a number of packet processing programs, including the production-grade L4 load balancer Katran from Facebook and Kubernetes, the state-of-the-art container orchestration system, using synthetic and real-world traffic traces (§VI). Our results show that Morpheus can improve the performance of the unoptimized (statically compiled) eBPF application up to 94%, while reducing packet-processing latency by up to 123% at the 99th percentile. Applying Morpheus to a DPDK program, we increase performance by up to 469%. We measured Morpheus against state-of-the-art network code optimization frameworks such as ESwitch [18] and PacketMill [21]: we show that Morpheus boosts the throughput by up to 80% and 294%, respectively, compared to existing work. Finally, we demonstrate that Morpheus can successfully optimize the performance of the Kubernetes Container Networking Interface (CNI) plugin used by cloud-native applications.

Contributions In this paper, we:

- demonstrate that tracking packet-level dynamics opens up new opportunities for network code specialization;
- design and implement Morpheus, a system working with standard compilers to optimize network code at run time;
- extensively evaluated Morpheus by applying it to two different I/O technologies (i.e., DPDK and eBPF), and a number of programs including production-grade software;
- show the applicability of Morpheus to real-world environment such as Kubernetes container networking [29];
- share the code in open source to foster reproducibility [30][1].

II. THE CASE FOR DOMAIN-SPECIFIC OPTIMIZATIONS

State-of-the-art profile guided optimization tools (PGO), such as Google's AutoFDO [17], [31] and Facebook's Bolt [16], can dynamically rewrite the targeted code using execution profiles recorded offline; e.g., by simplifying load instructions or reordering basic blocks to speed up the most frequently executed code paths. Fig.1a shows the single-core throughout obtained when applying AutoFDO and Bolt combined (PGO) to a sample DPDK *firewall* application [32], which performs basic L2/L3/L4 packet processing followed by a lookup into an access control list (ACL), over a stream of 64-byte packets at 40Gb line rate (see §VI for the details of the configuration). In line with the expectations [16], [17], [28], we managed to improve the performance of the targeted code by a mere 4.2%.

The behavior of packet-processing code can, however, be deeply influenced by specific metrics (e.g., match-action table access patterns, table sizes and content) that cannot be tracked with generic profiling mechanisms (i.e., Linux perf) used by standard PGO tools. Lacking such *domain-specific* insight, meaningful only in the packet processing context, generic purpose PGO tools cannot be fully exploited for dynamically optimizing network code [11], [18], [21], [22], [33].

To understand the potential of *domain-specific* optimizations, we present a series of preliminary benchmarks using real network code. We consider two applications: the DPDK sample *firewall* discussed above and Katran [6], Facebook's open-source L4 eBPF/XDP *load balancer*.

The promise of policy-driven optimizations. Most data-plane programs are developed as a single monolithic block containing various features that might be activated depending on the specific network configuration in use at any instance of time. For example, many large-scale cloud deployments still run on pure IPv4 and so the hypervisor switches would never have to process IPv6 packets [34] or adopt a single virtualization technology (VLAN/VxLAN/GRE/Geneve/GTP) and so switches would never see other encapsulations in operation [5], [35]. This implies that, depending on dynamic input that is unknown at compile time, a huge body of unused code gets assembled into the program, boosting code size and causing excess branch prediction misses, negatively impacting the overall performance [11], [22], [36], [37].

Removing unused code based on *run time configuration* can have a profound effect on software performance. To show this, we configured our *firewall* as a TCP signature-based Intrusion Detection System (IDS), with only TCP *wildcard* rules generated with ClassBench [38]. This opens up an

[1] A working version with latest updates is also available here: https://github.com/Morpheus-compiler/Morpheus

opportunity for optimization: all non-TCP packets can bypass the ACL table, avoiding a wasteful lookup. Fig. 1b shows the run time benefit of this optimization (under the *Run time configuration* bar) for a synthetic input traffic trace where only about 10% of the packets are UDP. Although around 90% of the traffic still has to undergo an ACL lookup, just avoiding this costly operation for a small percentage of traffic increases performance with about 4.7%, without changing the semantic in any way. In many practical scenarios, like DDoS blocking, security groups [29], [39] or whitelist-based access control, most firewall rules are fully-specified; for instance, in the official Stanford ruleset [40] on average ∼45% of the rules are purely exact-matching. This opens up another dynamic optimization opportunity: add in front of the ACL an exact-matching lookup table to sidestep the costly wildcard lookup. The result in Fig. 1b (under the *Table specialization* bar) shows a further ∼8% performance improvement.

A similar effect is visible with the *load-balancer* (Fig. 1c): configuring Katran as an HTTP load balancer [41], [42] allows to dynamically remove all the branches and code unrelated to IPv4/TCP processing, which reduces the number of instructions by ∼58% (as reported by the Linux `perf` tool), yielding ∼17,1% decrease in the number of L1 instruction cache-load misses. Better cache locality then translates into ∼12% performance improvement (from 4.09 Mpps to 4.69 Mpps).

Takeaway #1: Specializing networking code for slowly changing input, like flow-rules, ACLs and control plane policies, substantially improves the performance of software data planes.

The need for tracking packet-level dynamics. The potential to optimize code for specific network configurations has been explored in prior work, for OpenFlow [18], P4 software [20], [33] and hardware targets [19], network functions [22], and programmable switches [21] (see Table I). In order to maximize performance, however, we need to go beyond specializing the code for relatively stable run time configuration and apply optimizations at the packet level.

Consider the DPDK *firewall* application. We installed 1000 wildcard rules and generated highly skewed traffic, so that from the thousand active unique 5-tuple flows only 5% accounts for 95% of the traffic. This opens up the opportunity to inline the match-action logic for the recurring rules. As the results show (Fig. 1b, under the *Fast Path* bar), we obtain ∼42% performance improvement with this simple traffic-dependent optimization. With the eBPF *load balancer* the effect is also quite visible: configuring 10 Virtual IPs (VIP) (both TCP and UDP), each with hundred different back-end servers, a similarly skewed input traffic trace presents the same opportunity to inline code, yielding ∼24% performance edge (Fig.1c).

Takeaway #2: For maximum performance, networking code must be specialized with respect to inbound traffic patterns, despite the potentially daunting packet-level dynamics.

III. Challenges

Static compilation performs optimizations that depend only on compile-time constants: it does not optimize variables whose value is invariant during the execution of the program but remain unknown until then. Dynamic compilation, in contrast, enables specializing the code with respect to *invariant* run time data [23]. This opens up a broad toolbox of optimization opportunities, to propagate, fold and inline constants, remove branches and eliminate code never triggered in operation, or even to completely sidestep costly match-action table processing. The *unsupervised optimization of networking code*, however, presents a number of unique challenges:

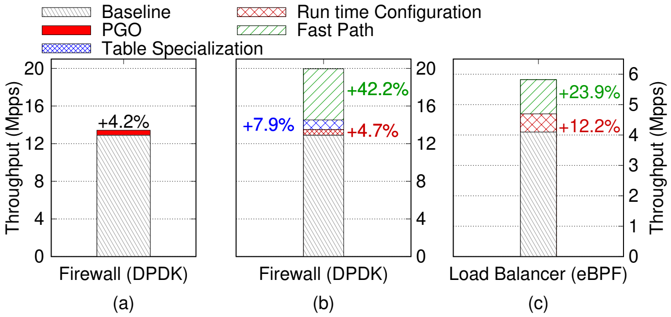


Fig. 1. (a) Impact of AutoFDO+Bolt (PGO) and performance breakdown when applying a set of domain specific optimizations to both (b) the DPDK *firewall* and (c) the Facebook's Katran eBPF *load balancer*. We were unable to run PGO on the latter, since existing tools do not support eBPF targets.

Challenge #1: Low-overhead run time instrumentation. Unsupervised dynamic optimization rests on the assumption that program variables remaining invariant for an extended period of time are promptly detected. This is the job of profiling tools. Although low-overhead solutions exist [27], [43]–[45], they track high-level code behavior information such as cache events, branch misses or memory accesses which is not enough for packet-processing code (§II). Less lightweight and more accurate tools [46]–[48], instead, are not practical to be used at data-plane time scales: recording at run time instruction-level logs for code that processes potentially tens of millions of packets per second can introduce an overhead that makes the subsequent optimization pointless. For example, GCC FDO instrumentation can easily incur ∼900% mean overhead [28]. We tackle this challenge in Morpheus by using *static code analysis* to understand the structure of the program offline (§IV-A) and leveraging an *adaptive instrumentation* mechanism to minimize the amount of data that is collected online (§IV-B).

Challenge #2: Dynamic code generation. Once run time profiling information is available, the dynamic compiler applies domain-specific optimizations to specialize the code for that profile. Here, *code generation* must integrate seamlessly into the compiler toolchain, to avoid interference with the built-in optimizations. Furthermore, a toolbox of *domain-specific optimization passes* must be identified, which, when applied to networking code, promise significant speedup (§IV-C).

Challenge #3: Consistency. The dynamically optimized data plane is contingent on the assumption that the data considered invariant during the compilation indeed remains so: any update to such data would immediately invalidate the specialized code. Here, the challenge is to guarantee data plane consistency under any modification to the invariants on which the specialized code relies. We tackle this challenge by injecting *guards* at critical points in the code that allow the execution to fall back to the generic unoptimized path whenever an invariant changes. Since the performance burden on each packet, possibly taking several guards during its journey, can be taxing, we introduce a *guard*

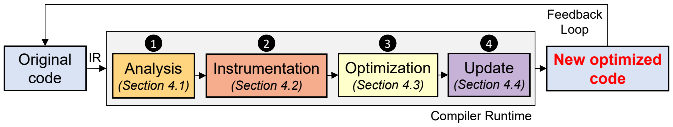


Fig. 2. The Morpheus compiler pipeline.

elision heuristic to sidestep useless guards (§IV-C). To do so, our static code analysis tool must have enough understanding of the program to separate *stateless from stateful code* (§IV-C). Finally, mechanisms are needed to *atomically update* the data plane once the code is re-optimized for the new invariants (§IV-D).

IV. Morpheus Compilation Pipeline

We designed Morpheus with an ambitious goal: to build a portable dynamic software data plane compilation and optimization toolbox. The system architecture is shown in Fig. 2. Morpheus accepts the input code at the Intermediate Representation (IR) level. The pipeline is triggered periodically at given time slots to readjust the code for possibly changed traffic patterns and control plane updates. At each invocation, the compiler performs an extensive offline code analysis to understand the program control/data flow (see §IV-A) and then reads a comprehensive set of instrumentation tables to extract run time match-action table access patterns (see §IV-B). Finally, Morpheus invokes a set of dynamic compilation passes to specialize the code (see §IV-C) and then replaces the running data plane with the new, optimized code on the fly (see §IV-D).

Below, we review the above steps in more detail. We use the simplified main loop of the *Katran* XDP/eBPF load balancer [6] as a running example (see Listing 1). The main loop is invoked by the Linux XDP datapath for each packet. It starts by parsing the L3 (line 4) and the L4 (line 5) header fields, using a special case for QUIC traffic as this is not trivial to identify [49]. In particular, QUIC flows are marked by a flag stored in the VIP record (line 12); if the flag is set, then a special function is called to deal with the QUIC protocol. Otherwise, a lookup in the connection table (line 17) is done: in case of a match, the ID of the backend assigned to the flow is returned; if no connection tracking information is found, a new backend is allocated and written back to the connection table (line 20). Finally, the IP address of the backend associated with the packet is read from the backend pool (line 24), the packet is encapsulated (line 25) and sent out (line 26).

A. Code Analysis

To be able to specialize code, we need to have a good understanding of the possible inputs it may receive during run time. Networking code tends to be fairly simplistic in this regard: commonly, the input consists of the *context*, which in eBPF/XDP corresponds to the raw packet buffers, and the content of match-action tables named *maps* in the eBPF world (Listing 1). Since input traffic may be highly variable and provides limited visibility into program operation, Morpheus does not monitor this input *directly* [11]. Rather, it relies on tracking the map access patterns and uses this information to

```
1  int process_packet (packet pkt) {
2    u32 backend_idx;
3
4    parse_l3_headers(pkt);
5    parse_l4_headers(pkt);
6
7    vip.vip = pkt.dstIP;
8    vip.port = pkt.dstPort;
9    vip.proto = pkt.proto;
10   vip_info = vip_map.lookup(vip);
11
12   if (vip_info->flags & F_QUIC_VIP){
13      backend_idx = handle_quic();
14      goto send;
15   }
16
17   backend_idx = conn_table.lookup(pkt);
18   if(!backend_idx) {
19      backend_idx = assign_to_backend(pkt)
20      conn_table.update(pkt, backend_idx);
21   }
22
23 send:
24    backend = backend_pool.lookup(backend_idx);
25    encapsulate_pkt(backend->ip);
26    return XDP_TX;
27 }
```

Listing 1. Simplified Katran main loop

indirectly reconstruct aggregate traffic dynamics and identify invariants along frequently taken control flow branches.

In the first pass, Morpheus uses comprehensive statement-level *static code analysis* to identify all map access sites in the code, understand whether a particular access is a read or a write operation, and reason about the way the result is used later in the code. In particular, *signature-based call site analysis* is used to track map lookup and update calls, and then a combination of *memory dependency analysis* [50] and *alias analysis* [51] is performed to match map lookups to map updates. Maps that are never modified from within the data plane are marked as read-only (RO) and the rest as read-write (RW). Note that RO maps may still be modified from user space, but such control-plane actions tend to occur at a coarser timescale compared to RW maps, which may be updated with each packet. This observation will then allow to apply more aggressive optimizations to *stateless* code, which interacts only with relatively stable RO maps, and resort to conservative optimization strategies when specializing stateful code, which depend on potentially highly variable RW maps.

Running example. Consider the *Katran* main loop (Listing 1). Morpheus leverages the domain-specific knowledge, provided by the eBPF data-plane plugin (§V-A), to identify map reads by the `map.lookup` eBPF helper signature and map writes either via `map.update` calls or a direct pointer dereference. Thus, map `backend_pool` is marked as RO and `conn_table` as RW. For `vip_map`, memory dependency analysis finds an access via a pointer (line 12), but since this conditional statement does not modify the entry and no other alias is found, `vip_map` is marked as RO as well.

B. Instrumentation

In the second pass, Morpheus profiles the dynamics of the input traffic by generating *heatmaps* of the maps' access

Optimization	Description	Small RO maps	Large RO maps	RW maps	Traffic-dependent
JIT (§IV-C1)	inline frequently hit table entries into the code	✓	✓	✓	✓
Table Elimination (§IV-C1)	remove empty tables	✓	✓	✗	✗
Constant Propagation (§IV-C2)	substitute run time constants into expressions	✓	✓	✗	✓
Dead Code Elimination (§IV-C3)	remove branches that are not being used	✓	✓	✗	✓
Data Structure Specialization (§IV-C4)	adapt map implementation to the entries stored	✓	✓	✓	✗
Branch Injection (§IV-C5)	prevent table lookup for select inputs	✓	✓	✗	✗
Guard Elision (§IV-C6)	eliminate useless guards	✓	✓	✗	✗

TABLE II

Dynamic optimizations in Morpheus. Applicability of each optimization depends on the map size, access profile (RO/RW), and availability of instrumentation information. Note that optimizations marked as ”traffic-dependent” can also be applied, at least partially, without packet-level information (e.g., small RO maps can always just-in-time compiled). For full efficiency, these passes rely on timely instrumentation information (e.g., to JIT heavy hitters from a large map as a fast-path).

patterns, so that the collected statistics can then be used to drive the subsequent optimization passes. Specifically, Morpheus uses a *sketch* to keep track of map accesses[2], by storing instrumentation data in a LRU (least-recently-used) cache alongside each map and adapting the sampling rate along several dimensions to control the run time cost of profiling. The dimensions of adaptation are as follows. (1) *Size:* small maps are unconditionally inlined into the code and hence instrumentation is disabled for these maps. (2) *Dynamics:* Morpheus does not record each map access, but rather it samples just enough information to reliably detect heavy hitters [53]. (3) *Locality:* instrumentation caches are per-CPU and hence track the local traffic conditions at each execution thread separately, i.e., specific to the RSS context. This improves per-core heavy hitter detection in presence of highly asymmetric traffic. (4) *Scope:* after identifying heavy hitters in the CPU context, local instrumentation caches are run together to identify global heavy hitters. (5) *Context:* if a map is accessed from multiple call sites then each one is instrumented separately, so that profiling information is specific to the calling context. (6) *Application-specific insight:* the operator can manually disable instrumentation for a map if it is clear from operational context that access patterns prohibit any traffic-dependent optimization (see Table II). Traffic-independent optimizations are still applied by Morpheus in such cases.

Running example. Consider the `vip_map` in our sample program, identified as an RO map in the first pass. In addition, suppose that there are hundreds of VIPs associated with TCP services stored in the `vip_map` and only a single one is running QUIC, but the QUIC service receives the vast majority of run time hits. Then, instrumentation will identify the QUIC VIP as a heavy hitter and Morpheus will seize the opportunity to specialize the subsequent QUIC call-path explicitly. Note that this comes without *direct* traffic monitoring, only using *indirect* traffic-specific instrumentation information.

C. Optimization Passes

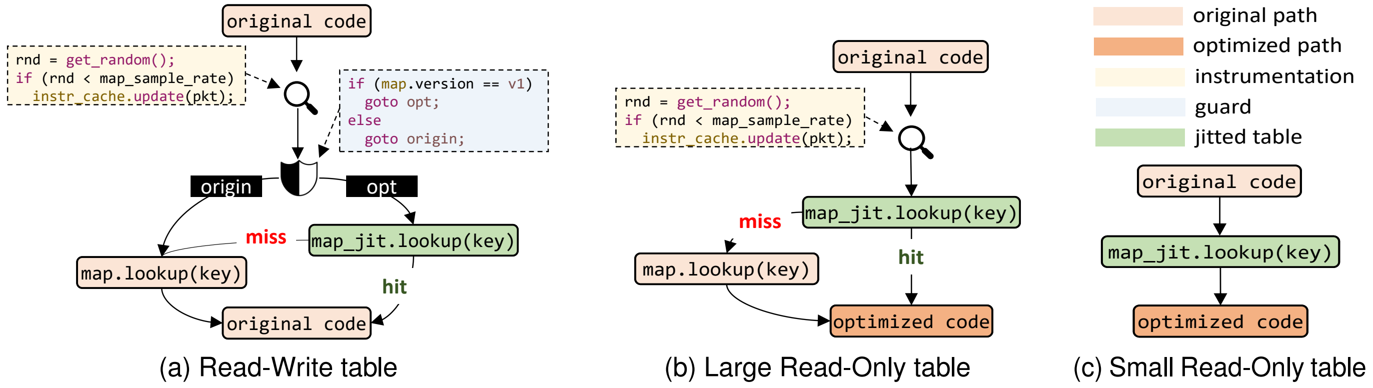
The third step of the compilation pipeline is where all online code transformations are applied. Before deploying any code transformation, Morpheus has to protect the consistency of the new specialized code against changes to the invariants the optimizations depend on. To do this, Morpheus uses *guards*, a standard mechanism used by dynamic compilers to guarantee code consistency by injecting simple run time version checks at specific points in the code [54]. When the control flow reaches a guard, it atomically checks if the version of the guard is the same as the version of the optimized code; if yes, execution jumps to the optimized version, otherwise it falls back to the original code (“deoptimization”). Below, we describe all the various run time optimizations currently applied by Morpheus; see Table II for a summary.

1) Just-in-time compilation (JIT): Empirical evidence (see §II) suggests that table lookup is a particularly taxing operation for software data planes. This is because certain match-action table types (e.g., LPM or wildcard), that are relatively simple in hardware, are notoriously expensive to implement in software [55]. Therefore, Morpheus specializes tables at run time with respect to their content and dynamic access patterns, as learned in the instrumentation pass. Specifically, empty maps are completely removed, small maps are unconditionally just-in-time (JIT) compiled into equivalent code, and larger maps are preceded by a similar JIT compiled fast-path cache, which is in charge of handling the heavy hitters. Note that the consistency of the the fast-path cache must be carefully protected against potential changes made to the specialized map entries; Morpheus places guards into the code to ensure this (see later).

Running example. Consider again Listing 1 and suppose that there are only two VIPs configured in the `vip_map`. Being an exact-matching hash it is trivial to compile the `vip_map` into an “if-then-else” statement, representing each distinct map key as a separate branch. To do so, Morpheus uses the insights from the code analysis phase to discover that relevant fields in the lookup are the destination address (`pkt.dstIP`), port (`pkt.dstPort`) and the IP protocol (`pkt.proto`). Then, for each entry in the map, it builds a separate “if” conditional to compare the entry’s fields against the relevant packet header fields and chains these with “else” blocks. Since the instrumentation and the just-in-time compiled map are specific to unique combinations of destination address/ port and protocol, the lookup semantics is correctly preserved even for longest prefix matching (LPM) caches and wildcard lookup.

2) Constant propagation: Specializing a table does not only benefit the performance of the lookup process but has far reaching consequences for the rest of the code. This is because a specialized table contains all the constants (keys and values)

[2] In implementing the sketch, we applied the same insights as those uncovered in [52]. Specifically, for the eBPF implementation, instead of invoking a dedicated helper function to obtain a random number for controlling the instrumentation rate, we employ an additional ARRAY map containing a collection of pre-generated random numbers.



```
rnd = get_random();
if (rnd < map_sample_rate)
  instr_cache.update(pkt);
```

```
if (map.version == v1)
  goto opt;
else
  goto origin;
```

```
rnd = get_random();
if (rnd < map_sample_rate)
  instr_cache.update(pkt);
```

(a) Read-Write table (b) Large Read-Only table (c) Small Read-Only table

Fig. 3. Morpheus handles the optimizations and provide code consistency mechanisms that are table-dependent.

inlined, which makes it possible to propagate these constants to the surrounding code so to inline memory accesses. In Morpheus, constant propagation opportunistically extends to larger maps that cannot be wholly just-in-time compiled: if a certain table field is found to be constant across *all* entries, then it is also inlined into the surrounding code. This optimization is thereby two-faceted: it can be used to specialize the code with respect to the inbound traffic (traffic-dependent, former case) but can also be applied without packet-level information (traffic-independent, the latter case). Morpheus does not implement constant propagation itself; rather, it relies on the underlying compiler toolchain to perform this pass.

Running example. Suppose there are only two backends in the `backend_pool`. Here, the map lookup (line 24) is rewritten into an “if-then-else” statement, with two branches for each backend. Correspondingly, in each branch the value of the `backend` variable is constant, which allows to save the costly memory dereference `backend->ip` (line 25) by inlining the backend IP address right into the specialized code.

3) Dead code elimination: Depending on the specific configuration, a large portion of code may sit unused in memory at any point in time. Such “dead code” can be found using a combination of static code analysis and the instrumentation information obtained from the previous pass. Upon detection, Morpheus removes all dead code on the optimized code path. As previously, this operation is outsourced to the compiler.

Running example. Consider the `vip_map` lookup site (line 10) and suppose that there are no QUIC services configured. As a consequence, the `vip_info->flags` is identical across all the entries in the `vip_map` and the constant propagation pass inlines this constant into the subsequent conditional (line 10). Thus, the condition `vip_info->flags & F_QUIC_VIP` always evaluates to `false` and the subsequent branch can be safely removed.

4) Data Structure Specialization: Morpheus adapts the layout, size and lookup algorithm of a table against its content at run time. For example, if all entries share the same prefix length in an LPM map, then a much faster exact-matching cache [18] can be used. This is done by first associating a backend-specific cost function with each applicable representation (this can be automatically inferred using static analysis and symbolic execution [16], [56]), generate the expected cost of each candidate, and finally implement the table that minimizes the cost.

5) Branch Injection: This pass applies to the cases when certain fields take only few possible values in a table, which makes it possible to eliminate subsequent code that handles the *rest* of the values. This optimization was used in §II to sidestep the ACL lookup for UDP packets in the *firewall* use case: if we observe that the “IP protocol” field can have only a single value in the ACL (e.g., TCP), then we can inject a conditional statement *before* the ACL lookup to check if the IP protocol field in a packet is TCP, use symbolic execution to track the use of this value throughout the resultant branch, and invoke dead code elimination to remove the useless ACL lookup on the non-TCP “else” branch.

6) Guard elision: As discussed before, Morpheus uses *guards* to protect the consistency of the optimized code. Since each packet may need to pass multiple checks while traversing the datapath, guards may introduce non trivial run time overhead [57]. To mitigate this, Morpheus heuristically eliminates as many guards as possible; this is achieved by using different schemes depending if changes to the code are made from the control plane or from the data-plane itself, as in the case when a program implements a stateful network function.

Handling control plane updates. Theoretically, each table should be protected by a guard when the contents are modified from the control plane. This would require packets to perform one costly guard check for each table. To reduce this overhead, Morpheus collapses all table-specific guards protecting against control plane updates into a single program-level guard, injected at the program entry point. Once an RO map gets updated by the control plane, the program-level guard directs all incoming packets to the *original* (unoptimized) datapath until the next compilation cycle kicks in to re-optimize the code with respect to the new table content.

Handling updates within the data plane. The optimized datapath must be protected from data-plane updates as well, which requires an explicit guard at all access sites for RW maps. If the guard tests valid then a query is made into the just-in-time compiled fast-path map cache and, on cache hit, the result is used in the subsequent code. Once a modification is made to the map from *inside* the data plane, the guard is invalidated and map lookup falls back to the original map.

Fig. 3 presents a breakdown of the strategies Morpheus uses to protect the consistency of optimized code. For RW maps (Fig. 3a), first an instrumentation cache is inlined at the access sites, followed by a guard that protects the just-in-time compiled fast-path against data-plane updates. Note that the constant propagation and dead code elimination passes are suppressed, since these passes may modify the code *after* the map lookup and the guard does not protect these optimizations. In contrast, RO map lookups (Fig. 3b and Fig. 3c) elide the guard, because only control-plane updates could invalidate the optimizations in this case but these are covered by the program-level guard. RO maps are specialized more aggressively than RW maps, by enabling all optimization passes. Finally, additional overhead can be shaved off for small RO tables by removing the fall-back map all together (Fig. 4c).

Running example. Once static code analysis confirms that the `vip_map` and `backend_pool` maps are RO, Morpheus opportunistically eliminates the corresponding guards at the call site. This then implies that, as long as the VIPs and the backend pool are invariant, the optimized code elides the guard. Since the `conn_table` map is RW, it is protected with a specific guard at the call site (line 17). Thus, the specialized map is used only as long as the connection tracking module's state remains constant; once a new flow is introduced into `conn_table` (line 20) the specialized code is immediately invalidated by bumping the data-plane version. This does not invalidate all optimizations: as long as the rest of the (RO) maps are not updated by the control plane, the program-level guard remains valid and the corresponding RO map specializations still apply.

D. Update

Upon invocation, Morpheus executes the above passes to create the optimized datapath and uses the native compiler toolchain to transform the optimized code to target native code. Meanwhile, control plane updates are temporarily queued without being processed. This allows the "old" code to process packets without any disruption while the optimization takes place. Once compilation is finished, the optimized code is injected into the data path, the program-level guard is updated [58] and the outstanding table updates are executed.

V. Implementation

Morpheus is implemented in about 6000 lines of C++ code and it is openly available at https://github.com/Morpheus-compiler/Morpheus, with the artifacts archived on Zenodo [30]. As shown in Figure 4, the code is separated into a data plane independent portable *core*, containing the compiler passes, and technology-specific *plugins* to interact with the underlying technology (i.e., eBPF, DPDK).

The Morpheus core extends the LLVM [13] compiler toolchain (v10.0.1) and the *ORCv2/MCJIT* [59], [60] toolset for code manipulation and run time code generation. We opted to implement Morpheus at the *intermediate representation* (IR) level as it allows to reason about the running code using a relatively high-level language framework without compromising on code generation time. Moreover, this also makes the Morpheus core portable across different data plane frameworks and programming languages [61].

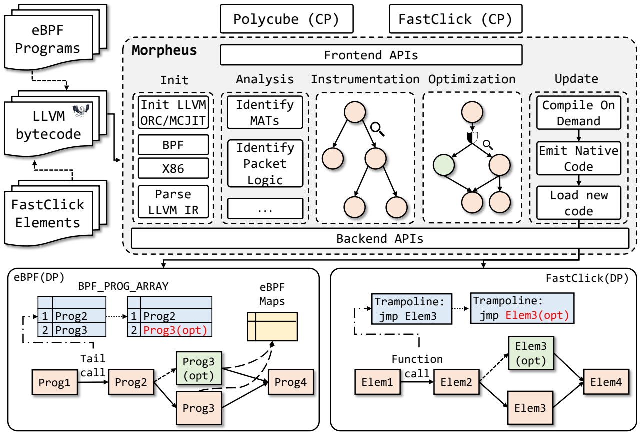


Fig. 4. Implementation details of Morpheus compilation pipeline. In the bottom part of the figure we show the details of the pipeline swap for the eBPF and FastClick plugins.

As shown in the upper part of Figure 4, Morpheus requires to implement a set of *frontend APIs* for each supported framework. These APIs provide control plane related information to Morpheus, such as when a new program is instantiated, or to provide a mechanism for the Morpheus core to intercept, inspect, and queue any update made by the control plane. This allows Morpheus to internally register the program to be optimized at runtime, or to trigger the compilation pipeline when Morpheus intercepts a control plane event, e.g., an update to a table. The *frontend APIs* also offer an abstraction layer that allows the CP program to communicate with the optimized data plane in the same way as with the original program. This makes all the automatic and runtime optimization that Morpheus performs entirely transparent to users.

On the other hand, the data plane plugins are abstracted via a *backend API*. This API exports a set of functions for the core to identify match-action table access sites based on data-plane specific call signatures; compute cost functions for data structure specialization; rewrite data plane dependent code using templates; and provide an interface to inject guards. Additionally, the backend can channel instrumentation data from the data plane to the compiler core and implement the data plane dependent parts of the pipeline update mechanism. Currently, only eBPF (fully) and DPDK (partially) are supported, but the architecture is generic enough to be extended to essentially any I/O framework, like netmap [62] or AF_XDP [63].

Initialization: Once the framework starts, the LLVM JIT framework is initialized with the corresponding *native* target (i.e., eBPF or x86) and, depending on the instantiated program it loads the associated LLVM bytecode. Following the usual analysis, instrumentation, and optimization, it activates the *compile-on-demand* layer, which compiles the program on-the-fly, emitting the final code. Later, the backend-specific APIs are used to load the new code into the pipeline, as shown in the subsequent sections.

A. The eBPF Plugin

Morpheus leverages the Polycube [64] framework as an eBPF backend to manage chains of in-kernel packet processing programs. Polycube readily delivers almost all the needed components for an eBPF backend. We added a mechanism for updating the data plane program on-the-fly and defined templates to inject guards. We discuss these components next.

Pipeline update. Once the optimized program is built, Morpheus calls the eBPF LLVM backend to generate the final eBPF native code, loads the new program into the kernel using the `bpf()` system call, and directs execution to the new code. In Polycube, a generic data plane program is usually realized as a chain of small eBPF programs connected via the eBPF *tail-call* mechanism, using a `BPF_PROG_ARRAY` map to get the address of the entry point of the next eBPF program to execute. Thus, injecting a new version of an eBPF program boils down to atomically update the `BPF_PROG_ARRAY` map entry pointing to it with the address of the new code, as shown in Figure 4. When loading the optimized code, it is possible that the verifier will reject it. Currently, Morpheus doesn't incorporate additional mechanisms to ensure that the regenerated code can pass the verifier. Ideed, in Morpheus each optimization pass is carefully designed to ensure that the modification made to the program are deterministic and that the generated code is verifiable. While it's possible that a bug in a specific pass could lead to an unsafe program rejected by the verifier, Morpheus addresses this by preserving the latest stable version of the program and falling back to the original program if the latest version does not implement the new runtime configuration. During subsequent optimization cycles, one optimization pass is disabled at a time, and the newly generated code is reloaded to check verifier acceptance. This iterative process helps identify the combination of passes that resulted in the problematic code.

Guards. Morpheus relies on guards to protect the specialized code against map updates. The program-level guard is implemented as a simple run time version check [58]. For stateful processing, Morpheus installs a guard at each map lookup site and injects a *guard update pre-handler* at the instruction address corresponding to the map update eBPF function (`map_update_elem`). This handler will then safely invalidate the guard before executing the map update.

B. The DPDK Plugin

Morpheus leverages FastClick [65], a framework to manage packet-processing applications based on DPDK. FastClick makes implementing most components of the backend API trivial; below we report only on pipeline updates and guards.

Pipeline update. A FastClick program is assembled from primitive network functions, called *elements*, connected into a dataflow graph. As shown in Figure 4, every FastClick element holds a pointer to the next element along the processing chain. To switch between different element implementations at run time, Morpheus adds a level of *indirection* to the FastClick pipeline: every time an element would pass execution to the next one, the corresponding function call is conveyed through a trampoline, which stores the *real* address of the next element to be called. Then, atomic pipeline update simplifies into rewriting the corresponding trampoline to the address of the new code. In contrast to eBPF, which explicitly externalizes into separate maps all program data intended to survive a single packet's context, a FastClick element can hold non-trivial internal *state*, which would need to be tediously copied into the new element. As a workaround, our DPDK plugin disables dynamic optimizations for stateful FastClick elements.

Guards. Since stateful FastClick elements are never optimized in Morpheus and RO elements always elide the guard, our DPDK plugin currently does not implement guards, except a program-level version check at the entry point.

VI. Evaluation

Our testbed includes two servers connected back-to-back with a dual-port Intel XL710 40Gbps NIC. The first, a 2x10-core Intel Xeon Silver 4210R CPU @2.40GHz with support for Intel's Data Direct I/O (DDIO) [66] and 27.5 MB of L3 cache, runs the various applications under consideration. The second, a 2x10 Intel Xeon Silver 4114 CPU @2.20GHz with 13.75MB of L3 cache, is used as packet generator. Both servers are installed with Ubuntu 20.04.2, with the former running kernel 5.10.9 and the latter kernel 4.15.0-112. We also configured the NIC Receive-Side Scaling (RSS) to redirect all flows to a single receive queue, forcing the applications to be executed on a single CPU core, while Morpheus was pinned to another CPU core on the device-under-test (DUT).

In our tests, we used `pktgen` [67] with DPDK v20.11.0 to generate traffic and report the throughput results, and the DPDK burst replay tool [68] to replay the different packet traces. Unless otherwise stated, we report the average single-core throughput across five different runs of each benchmark, measured at the minimum packet size (64-bytes). For latency tests, we used *Moongen* [69] to estimate the round-trip-time of a packet from the generator to the DUT and back. Finally, we used *perf* v5.10 to characterize the micro-architectural metrics of the DUT (e.g., cache misses, cycles, number of instructions).

In order to benchmark Morpheus on real applications, we chose four eBPF/XDP-based packet processing programs from the open-source eBPF/XDP reference network function virtualization framework Polycube [70], plus Facebook's Katran load-balancer used earlier as a running example [6].

The *L2 switch*, the *Router* and the *NAT* applications were taken from Polycube [70]. The *L2 switch* use case is a functional Ethernet switch supporting 802.1Q VLAN and STP, with STP and flooding delegated to the control plane while learning and forwarding implemented entirely in eBPF, using an exact-matching MAC table supporting up to 4K entries. The *Router* use case implements a standard IP router, with RFC-1812 header checks, next-hop processing and checksum rewriting, configured with an LPM table taken from the Stanford routing tables [40]. The *NAT* is an eBPF re-implementation of the corresponding Linux Netfilter application, configured with a single two-way SNAT/masquerading rule: the source IP of every packet is replaced with the IP of the outgoing NAT port and a separate L4 source port is allocated for each new flow. *BPF-iptables* is an eBPF/XDP clone [71] of the well-known Linux *iptables* framework, configured with 5-tuple rules generated by Classbench [38]. We used the Classbench trace generator [72] to generate packets matching the created rule

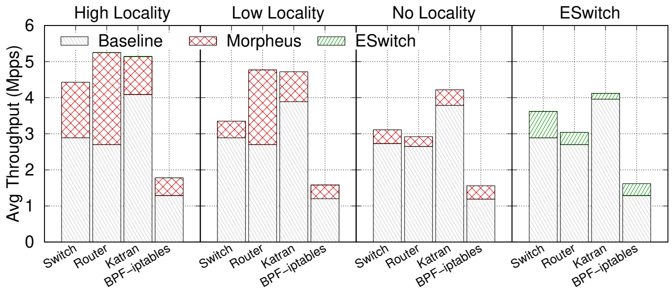


Fig. 5. Single core throughput (64B packets) varying input traffic locality. The optimizations adopted by Morpheus are traffic-dependent, while the ones from ESwitch [18] are not. For this reason, the ESwitch throughput (shown in the right box) is the same across the different traffic localities.

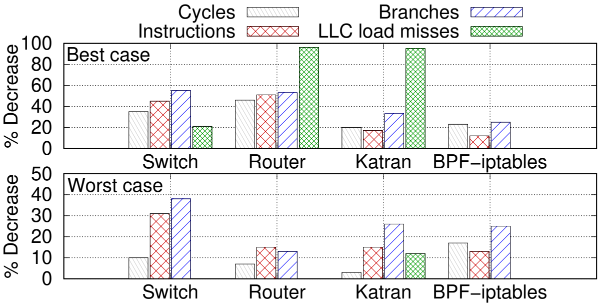


Fig. 6. Effect of Morpheus optimizations on PMU counters, obtained with *perf* at the default frequency (40KHz). The top panel shows the percentage of *decrease*, per packet, of different metrics for *high locality* traffic (best-case for Morpheus), and the bottom panel for *no locality* traffic (worst-case).

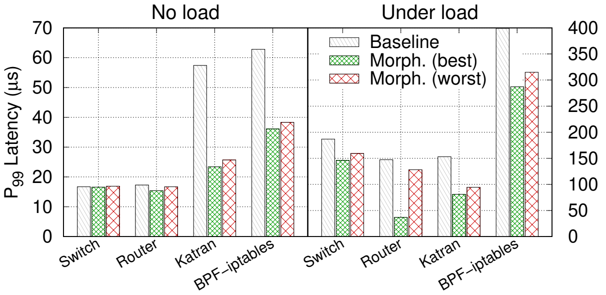


Fig. 7. 99th percentile (P_{99}) latency with Morpheus. The graph shows both the latency for the *optimized* and *non-optimized* code paths, under small load (10pps) and heavy load (highest rate without packet drop).

set using a *Pareto* cumulative density function to control the locality of reference. We used the same default parameters suggested by the ClassBench paper [38] to generate traces of varying locality, in particular the *no-locality* trace uses $\alpha = 1, \beta = 0$ as Pareto parameters, the low locality uses $\alpha = 1, \beta = 0.0001$, and the high locality uses $\alpha = 1, \beta = 1$. Finally, *Katran* [6] was configured as a web-frontend, with 10 TCP services/VIPs and 100 backend servers for each VIP.

For each benchmark, we generated 3 traffic traces with varying locality, to demonstrate the ability of Morpheus to track packet-level dynamics and optimize the programs accordingly. In particular, we created a *high-locality* traffic trace, where few flows account for most of the traffic, a *no-locality* trace with flows generated at random by a uniform distribution and a *low-locality* trace that sits in the middle between the two previous cases.

A. Benefits of Optimizations

We first show the impact of Morpheus on the mentioned programs, when attached to the XDP hook of the ingress interface.
Morpheus improves packet-processing throughput. In Fig. 5, we show the impact of Morpheus under different traffic conditions. Throughput is defined as the maximum packet-rate sustained by a program without experiencing packet loss. When a small subset of flows sends the majority of traffic (high-locality), Morpheus consistently delivers over 50% throughput improvement compared to the baseline, achieving a 2× speed-up for the *Router*. This is because it can track heavy flows and optimize the code accordingly. Notably, traffic-dependent optimization contributes to ∼65% of the improvement for high-locality traces, with JIT optimization being the most significant contributor. Data-structure optimization accounts for ∼26%, while DCE/Constant Propagation contributes to the remainder. In low-locality traces, the improvement from traffic-dependent optimization decreases to around 40% of the overall throughput enhancement. For no-locality traces, policy-driven optimization primarily drives improvements, accounting for ∼25%. However, this improvement is offset by the instrumentation overhead, as detailed in Section VI-B. To confirm the benefit of packet-level optimizations in Morpheus, we compared it to a faithful eBPF/XDP re-implementation of ESwitch, a dynamic compiler that does not consider traffic dynamics [18]. The results (Fig 5) show that Morpheus delivers 5–10× the improvement compared to ESwitch for high-locality traces, while it falls back to ESwitch for uniform traffic.
Morpheus benefits at the micro-architectural scale. In Fig. 6, we show that Morpheus reduces the last-level CPU cache misses by up to 96% and halves the instructions and branches executed per packet. At low or no traffic locality, the effects of packet-level optimizations diminish, but Morpheus can still bring considerable performance improvement: we see ∼ 30% margin for *BPF-iptables* even for the *no-locality* trace. This is because the optimization passes in Morpheus are carefully selected to be applicable independently from packet-level dynamics (see Table II).
Morpheus reduces packet-processing latency. In Fig. 7, we compared the 99th percentile baseline latency for each application against the one obtained with Morpheus, both in a *best-case scenario* when all packets travel through the optimized code path (e.g., the right branch in Fig. 3a), and a *worst-case scenario* with all packets falling back to the default branch instead of taking the fast-patch cache for each map (the left branch in Fig. 3a). The left panel in Fig. 7 shows the latency measured at low packet rate (10pps) so to avoid queuing effects [73], whereas the right panel shows latency under the maximum sustained load without packet drops [74]. First, we observe that Morpheus never *increases* latency, despite the considerable additional logic it injects dynamically into the code (guards, instrumentation; see below); in fact, it generally reduces it even in the worst case scenario. Notably, it reduces Katran's packet-processing latency by about 123%.

B. What is the cost of code instrumentation?

Clearly, the price for performance improvements is the additional logic, most prominently, instrumentation, injected by Morpheus into the fast packet-processing path. To understand

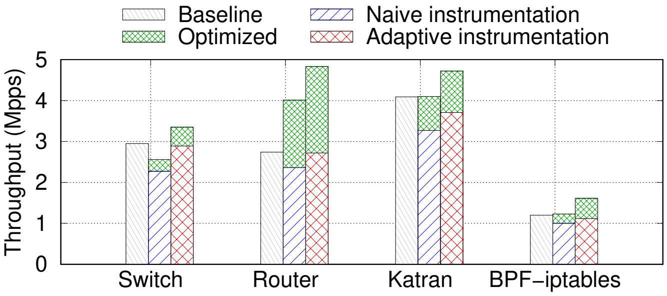


Fig. 8. Naive vs adaptive instrumentation (low locality traffic). In the naive case all map lookups are recorded, while adaptive instrumentation adjusts data sampling selectively for the access patterns at each lookup call site.

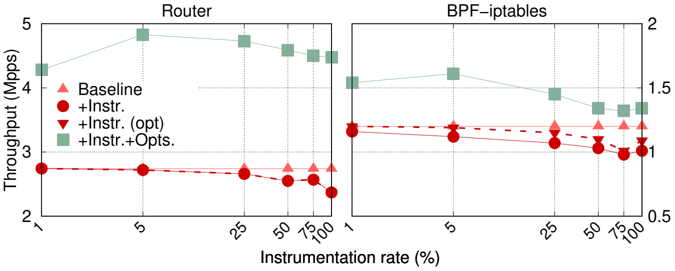


Fig. 9. Effectiveness of instrumentation at varying sampling rates (Router and BPF-iptables, low-locality traffic).

this price, we compared our *adaptive* instrumentation scheme (§IV-B) against a *naive* approach where all map lookups are explicitly recorded. Fig. 8 shows that instrumentation involves visible overhead: the instrumented code performs worse than the baseline. The naive approach imposes a hefty 14–23% overhead, but adaptive instrumentation reduces this to just 0.9%–9%. Most importantly, this reduction does not come at a prohibitive cost: adaptive instrumentation provides enough insight to Morpheus to make up for the performance penalty imposed by it and still attain a considerable throughput improvement on top (see the green stacked barplots). In contrast, the performance tax of naive instrumentation may very well nullify optimization benefits, even despite full visibility into run time dynamics (e.g., for the L2 switch or Katran).

We also studied the impact of packet sampling rate on instrumentation. Indeed, Morpheus collects information on packet-level dynamics only on a subset of input traffic in order to minimize the overhead. Fig. 9 highlights that Morpheus can strike a balance between overhead and efficiency by adapting the sampling rate. At a low sampling rate (e.g., recording every 100th packet) Morpheus does not have enough visibility into dynamics, which renders traffic-dependent optimizations less effective (but the traffic-invariant optimizations still apply). Higher sampling rates provide better visibility but also impose higher overhead. At the extreme (*BPF-iptables*, 100% instrumentation rate), optimization is just enough to offset the price of instrumentation. In conclusion, we found that setting the sampling rate at 5%–25% represents the best compromise.

Finally, we tested a different implementation of the Morpheus instrumentation, which uses an additional ARRAY map that contains a set of pre-generated random numbers generated in accordance with the Morpheus sampling rate. This is used to reduce the cost associated with the BPF `get_prandom_u32` helper function, as also suggested in [52]. Results shown in Figure 9 revealed that when the number of instrumented maps is small (fewer than 2) such as in the *router* case, the performance improvement with this approach is almost negligible, since the overhead of the helper function is replaced by the overhead of the new lookup call needed to retrieve the pre-generated random numbers. However, when the number of instrumented maps is larger such as in the *BPF-iptables* case, the new approach brings a notable performance improvement by reducing the overall instrumentation overhead of Morpheus.

TABLE III
TIME (IN MS) TO EXECUTE THE ENTIRE MORPHEUS COMPILATION PIPELINE AND INSTALL THE OPTIMIZED DATAPATH. LOC IS CALCULATED USING *cloc* (V1.82) EXCLUDING COMMENTS AND BLANK LINES WHILE INSTRUCTION COUNT IS MEASURED WITH *bpftool* V5.9.

Application	C LOC	BPF Insn	Compilation (ms)				Injection (ms)	
			Best		Worst		Best	Worst
			t_1	t_2	t_1	t_2		
L2 Switch	243	464	81	62	140	78	0.5	0.9
Router	331	458	87	65	196	91	1.1	1.3
BPF-iptables*	220	358	95	62	105	87	0.6	0.5
Katran	494	905	287	115	569	151	3.4	6.1

* Uses a chain of eBPF programs; since Morpheus optimizes every eBPF program separately, values shown refer to the most complex program in the chain.
t_1 Time to analyze the program, instrument it and read the maps.
t_2 Time to generate the final eBPF code.

C. How fast is the compilation?

In Table III, we indicate with t_1 the time to analyze, instrument and optimize the LLVM IR code, and with t_2 the time to generate the final eBPF code, starting from the LLVM IR. Note that t_1 is highly dependent on table size: the bigger the table, the more time needed to read and analyze it. We show the results for high-locality and no-locality traffic. The former is the *best case* since Morpheus needs to track fewer flows, thus requiring lighter instrumentation tables that are faster to analyze. The latter is the worst case. Generally, table read time (i.e., t_1) dominates over compilation time, consistently staying below 100ms and reaching only for Katran in the worst-case scenario almost 600ms. This is because Katran uses huge static maps with tens of thousands of entries to implement consistent hashing. Recent advances in the Linux kernel allow to read maps in batches, which would cut down this time by as much as 80% [75], reducing recompilation time for Katran below 100ms. Furthermore, the time needed to inject the optimized datapath into the kernel depends on the complexity of the program, since all eBPF code must pass the in-kernel verifier for a safety check before being activated. This also ensures that a mistaken Morpheus optimization pass will never break the data plane. In our tests, injection time varies between 0.5 to 3.4ms in the best case and at most 6.1ms in the worst case. Finally, in all the tests we run the Morpheus compilation pipeline on a separate core with respect to the data plane application, and we noticed that in most of the case it consumes, on average, ~5% of CPU, with a peak of 15% for applications that consist of a pipeline of multiple programs chained together (e.g., BPF-iptables) and with lot of table entries to be analyzed (e.g., Katran).

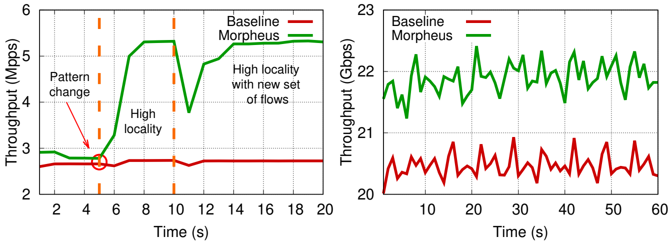


Fig. 10. (a) Single core throughput over time with Morpheus on the *Router* use case, with dynamically changing traffic patterns, and (b) with a CAIDA [76] trace.

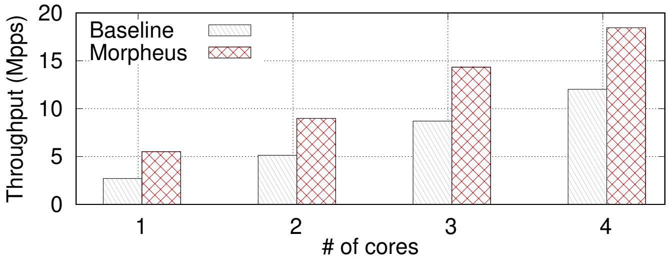


Fig. 11. Multicore application (router) with Morpheus.

D. Morpheus in action

To test the ability of Morpheus to track highly dynamic inputs, we fed the *Router* application with time-varying traffic and observed the throughput over time (Fig. 10a). Recompilation period was conservatively set to 1 second. In the first 5 seconds we generate uniform traffic; here, the traffic-independent optimizations applied by Morpheus yield roughly 15% performance improvement over the baseline. At the 5th second, the traffic changes to a high-locality profile: after a quick learning period Morpheus specializes the code, essentially doubling the throughput. We see the same effect from the 10th second, when we switch to another high-locality trace with a new set of heavy-hitters: after a brief training period Morpheus dynamically adapts the optimized datapath to the new profile and attains 60–100% performance improvement. We also repeated the same test using a real-world traffic trace (CAIDA 2019 dataset, *equinix-nyc* [76]), counting 30M packets with an average size of 910B. The trace experiences also a low degree of traffic locality, with the most hit entry matched around 0.4% overall. In Figure 10b, we show how Morpheus consistently improves the throughput of the router by factor of ∼10%. Increasing the most hit entry matched rate from 0.4% to 5% in the CAIDA trace increased the average throughput with the Morpheus optimizations from 22Gbps to ∼22.5Gbps, with peaks reaching ∼23Gbps.

Finally, in Figure 11, we report the multi-core scaling of Morpheus. Here, we still used the router when processing input traffic characterized by low-locality. The constant performance increase is enabled mainly by our adaptive instrumentation mechanism, which is able to track the flow states across the different cores, i.e., specific to the RSS context and, depending on their distribution optimize the code accordingly.

E. What can go wrong?

The primary limitation of Morpheus lies in the potential for misdirected runtime code transformations that could adversely impact performance. This drawback is analogous to situations in generic languages where a dynamic compiler can steal CPU cycles from the running code [15], [77], necessitating manual compiler parameter tuning and in-depth application-specific knowledge to mitigate performance losses [78]. In the context of dynamically optimizing *network code*, we can illustrate this limitation through the example of a NAT use case [64]. The NAT involves a large connection tracking table that undergoes frequent updates within the data plane for each new flow. This represents a worst-case scenario for Morpheus: fully stateful code, so that guards cannot be opportunistically elided, coupled with potentially high traffic dynamics. Yet, since traffic-independent optimizations can still be applied (Table II) Morpheus can improve throughput by around 5% (from 4.36 to 4.58 Mpps) in the presence of high-locality traffic. However, for low-locality traffic we see about 6% performance degradation compared to the baseline. Intuitively, Morpheus just keeps on recompiling the conntrack fast-path with another set of potential heavy hitters, just to immediately remove this optimization as a new flow arrives. Our tests mark micro-architectural reasons behind this: the number of branch misses and instruction cache loads increases by 90% and 75%, respectively, both clear symptoms of frequent code changes. Similar patterns are observed in other stateful applications, such as the *L2 switch* and *Katran*. However, the speed-up achieved through dead code elimination, constant propagation, and branch injection can offset the performance impact in these cases. As with Java, such cases require human intervention; manually disabling optimization for the connection tracking module's table safely eliminates the performance degradation on the NAT use case. In general, Morpheus tends to excel in scenarios where both the runtime configuration and the data structures used within the code remain relatively stable, i.e., they do not undergo frequent changes (where "frequent" is defined within the timescale of Morpheus' compilation periodicity). When focusing solely on policy-driven optimizations, Morpheus consistently proves effective. These optimizations don't rely on instrumentation or guards for data plane consistency, thus avoiding additional overhead in these scenarios.

F. Morpheus with DPDK programs

We applied Morpheus to a DPDK program, the FastClick [65] version of the eBPF *Router* application, the same one used in PacketMill's paper [21]. We configured it with either 20 or 500 rules from the Stanford routing tables [40] and generated traffic with different levels of locality. We compared the throughput and the latency of the baseline code, the Morpheus optimized and its version transformed with PacketMill, state-of-the-art DPDK packet-processing optimizer. In our tests PacketMill uses the following optimizations: removing virtual function calls, inlining variables, and allocating/defining the elements' objects in the source code.

Fig. 12a reports the average throughput results. For only 20 prefix rules and with low locality traffic, PacketMill

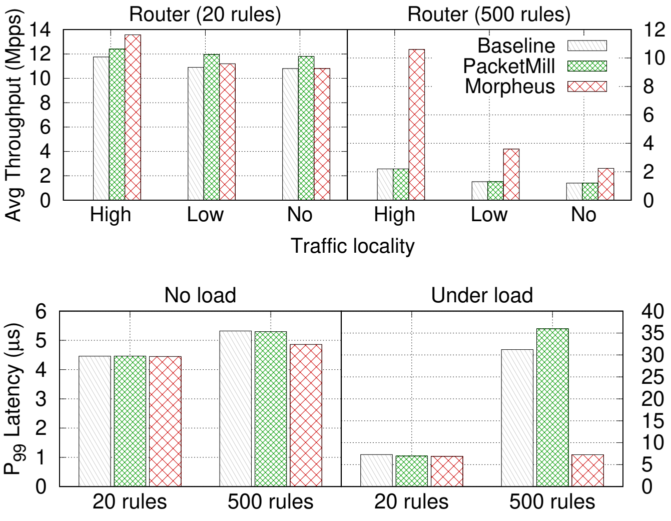


Fig. 12. Comparison between vanilla FastClick, PacketMill and Morpheus for the Router FastClick (DPDK) application with 20 and 500 rules.

outperforms Morpheus by about 9%, whereas for high-locality traffic and larger forwarding tables Morpheus produces a whopping 469% improvement over PacketMill. The reason for the large performance drop from 20 rules to 500 rules is that LPM lookup is particularly expensive in FastClick (linear search), but Morpheus can largely avoid this costly lookup by inlining heavy hitters. The 99th percentile latency results (Fig. 12b) confirm this finding, with Morpheus decreasing latency 5-fold compared to PacketMill with high-locality traffic.

On the other hand, the reason for the lower performance of Morpheus in the low/no locality case (20 rules) can be found in the main difference between the two systems. First, Morpheus requires instrumentation to track table access patters, which produces some run time overhead, while PacketMill does not apply online optimizations and so it does not need instrumentation at all. Second, PacketMill implements some optimizations that Morpheus does not (although nothing prevents us from implementing them), but in most cases the effect of these additional optimizations is masked by the speedup brought by the Morpheus traffic-level optimizations.

VII. Morpheus with Kubernetes

In the previous section, we demonstrated the ability of Morpheus to enhance the performance of various eBPF and DPDK network functions, including real-world applications like Katran, Facebook's load balancer. In this section, our focus is on showcasing the impact of Morpheus on real-world cloud-native scenarios, particularly within the context of Kubernetes [29]. We aim to evaluate how Morpheus can improve the performance of a CNI plugin for Kubernetes and compare the results with existing production-grade solutions.

For our evaluation, we considered three different CNI plugins. Cilium [79] and Calico [80] are both eBPF-based production-ready networking plugins for Kubernetes that provide security and observability features for container workloads. The Polycube [81] CNI is built on the Polycube [70] framework, offering standard CNI functionality by chaining various Polycube network functions, such as switches, load balancers, and NAT, to achieve the desired outcome for the default Kubernetes networking interface. While Polycube has limited functionality compared to the other two solutions, especially in terms of security and observability, our tests focused solely on the common networking functionality shared among these providers.

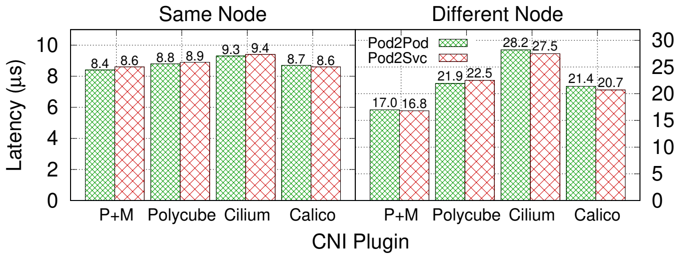


Fig. 13. TCP latency comparison between different CNI plugins when the communication is performed between Pods on the same node or different nodes (Pod-2-Pod), and between Pods using the Service IP (Pod-2-Svc). Morpheus is applied on top of the Polycube CNI plugin (P+M).

We deployed Morpheus on top of the Polycube CNI and compared its performance with the other CNI providers. The experiments were conducted on the NSF Cloudlab [82] using two XL170 nodes. Each node was equipped with a 10-core Intel E5-2640v4 CPU@2.4 GHz, 64GB memory, and a 25Gb Mellanox ConnectX-4 NIC. The setup employed Ubuntu 20.04 with kernel version 5.16.

To perform the experiments, we utilized the standard benchmark suite available in the Kubernetes repository [83]. The suite employed a custom docker container with a go binary, *iperf v3.1.3*, and *qperf* integrated within it. We deployed four pods[3] - two worker pods running iperf server and client on the same node, and another worker pod on a different node. An orchestrator pod coordinated the workers to execute the tests. We tested the communication between pods co-located on the same node and remotely located pods. For both cases, we tested communication using the pod's direct IP or its Service IP[4]. We conducted these tests to examine the scenarios involving additional components of the CNIs, such as NATTing and load-balancing, when communicating between Pods and Service IPs.

Figure 13 presents the latency results obtained using the *qperf* tools. The optimizations implemented by Morpheus resulted in a significant reduction in overall latency. When comparing with the original Polycube CNI, Morpheus achieved a 25% latency reduction in Pod-to-Pod communication between different nodes, and similar results were observed in Pod-to-Service communication. Moreover, Morpheus achieved a 40% latency reduction compared to Cilium. Regarding throughput, our optimized Morpheus on top of the Polycube CNI achieved approximately 61Gbps for Pod-2-Pod communication and around 60.2Gbps for Pod-2-Svc on a single node. In contrast, Cilium and Calico achieved 56Gbps and 57Gbps, respectively.

[3] A pod is the smallest execution unit in Kubernetes, representing a group of containers deployed on the same host.

[4] In Kubernetes, *Services* are an abstract way to expose an application running on a set of Pods. Services can have a cluster-scoped virtual IP address that clients can connect to, and Kubernetes load-balances traffic to that service across the different backing pods.

The significant performance improvement observed in inter-Pod communication with Morpheus can be attributed to the specific characteristics of the Polycube CNI [81]. When Pods communicate within the same node, they primarily interact with an internal load balancer that forwards packets directly to the destination Pod. However, cross-node communication involves traversing an additional set of NFs, including a firewall, an external load balancer, and a router. Morpheus provides benefits even for same-node communication, but its impact is most notable when packets traverse all three NFs during cross-node communication. Specifically, Morpheus tailors the packet processing pipeline based on the CNI's configuration, eliminating unnecessary processing to support a variety of services. This reduction in latency is primarily achieved by eliminating additional BPF map lookup calls made by the load balancer and firewall services for packet redirection. Additionally, Morpheus' traffic-level optimization creates fast paths for packets received on a particular node, while still maintaining mappings for all Services and Pods in the cluster within the CNI on every node.

VIII. Discussions

Add other optimizations to Morpheus compilation pipeline. Morpheus is orthogonal to most optimizations proposed in recent literature and can be extended to support them. For example, PacketMill optimizations [21], such as the reordering of metadata fields, could be easily integrated into Morpheus with the added benefit that, having access to the number of accesses to a given variable thanks to Morpheus's instrumentation, we could obtain a more accurate reordering compared to PacketMill, which only estimates access patterns. Finally, traditional PGO optimizations can be used with Morpheus too, allowing the compiler to optimize the code on-the-fly, as opposed to traditional PGO approaches where the profile is collected offline.

Extend Morpheus to other data plane technologies. Morpheus comprises a data plane-independent core implementing the bulk of the optimization passes, and separate data plane-specific plugins. This design allows Morpheus to potentially support various network functions with different logic, provided they are part of the existing frameworks and data plane-specific plugins currently supported by Morpheus, such as Polycube for eBPF-based network functions and FastClick for DPDK-based ones. Furthermore, it simplifies the process of porting Morpheus to new data planes with well-defined APIs. In such cases, developers would need to specify function signatures for relevant API calls and providing simple operators for Morpheus to interact with match-action table content and modify the underlying code (e.g., inject a guard).

Extend Morpheus to generic network programs. As previously mentioned, Morpheus is designed to work seamlessly with networking code that follows a structured approach and maintains a distinct separation between its operations and interactions with the external environment. Adapting Morpheus to generic code presents more challenges. Generic code lacks the specific structural constraints and clear differentiation between data structures and packet processing logic making the automatic inference of such information and its subsequent recompilation at runtime complex. We are exploring this idea further, as outlined in [84].

The choice of working at the IR level. In Morpheus, all the optimizations are directly applied at the LLVM IR level. A major drawback of this approach is that by doing so we lose direct access to the low-level machine code, making certain optimizations impossible: peephole, vectorization/SIMD, or other micro-architectural optimizations [85]–[89]. Nevertheless, this choice provides also a series of benefits: *(i)* IR code is in the Static Single Assignment (SSA) form and SSA simplifies the use of different compiler optimization algorithms; *(ii)* Morpheus optimization passes can exploit flow information performed in the compiler itself to gather information about the code under consideration: for instance, we use LLVM MemorySSA analysis to retrieve information about variable and load/store dependencies; *(iii)* working at the IR level allows Morpheus to re-use part of the other optimizations already available in the compiler suite; *(iv)* finally, it allows to keep the optimization passes as generic as possible with respect to the language in which the data plane is written.

It is not all about table lookups. Morpheus heavily optimizes also the code surrounding the table lookups using the insights it obtains during code analysis. It separates table lookup code into a fast-path, with the lookup results specialized for the heavy hitters, and a generic slow-path. It uses the table lookup code to also gather information about the table's content, and how it is used in the rest of the code. This allows for optimizing the entire fast-path code after the table lookup without affecting the slow path in any way. For example, constants are folded from the JITted lookup code (effectively a set of nested if-then-else statements) into the surrounding code (i.e., each branch of the if-then-else is specific to a certain value of the conditional), unreachable code is removed or tables are specialized depending on their run time entries.

Because of this, caching/JITting table lookups is just one, albeit very important, optimization that Morpheus performs. As shown in Figure 5, the results with the "no locality" trace demonstrate the combined benefits of all the optimizations that are independent from the input traffic, such as dead code elimination, constant propagation and data structure specialization, while the rest of the cases ("low/high-locality") show the additional effect of traffic-dependent optimizations. Note that some optimizations cannot be directly measured since they are the results of a combination of other passes; e.g., the contribution of dead code elimination is dependent on constant propagation.

Morpheus dependence on compilation periodicity. The performance of Morpheus depend on how fast Morpheus can recompile the targeted code (see Table III). In the presence of traffic changes that are faster than that, then traffic-dependent optimizations become less effective. Nevertheless, Morpheus can still speedup the original function, since other *traffic-independent* optimizations are still valid (Figure 4). Potentially, we could disable traffic-level optimizations when Morpheus discovers highly variable traffic that goes under the recompilation period (Section VI-E); this would reduce the impact of guards and instrumentation and increase the benefits of traffic-independent optimizations. Examining these techniques remains as future work.

Use Morpheus with the other Kubernetes CNI plugins. As discussed in Section V-A, Morpheus relies on the Polycube framework to manage chains of eBPF packet processing programs. To simplify the writing and manipulation of eBPF programs, Polycube uses the BPF Compiler Collection (BCC), which integrates Clang+LLVM to statically compile programs written in a higher level syntax into the necessary eBPF object files. By seamlessly integrating with Polycube and BCC, Morpheus adds the necessary compiler runtime, on-the-fly instrumentation, and optimization functionality.

Unlike Polycube, CNI plugins like Cilium and Calico do not use BCC to build their eBPF dataplanes. Instead, they use the *libbpf* library to compile eBPF objects offline and inject the program during the startup phase of the CNI plugin. While Morpheus supports libbpf-based eBPF dataplanes, it requires additional engineering effort to integrate Morpheus APIs into the CNI plugin control plane, which is out-of-scope. Despite this, the results demonstrate that Morpheus can still provide benefits in a real-world scenario, and we leave the process of integrating Morpheus with the other CNIs as future work.

IX. Related work

Generic code optimization has a long-standing stream of research and prototypes [16], [17], [31], [88]–[92]. In the context of networking, domain-specific data-plane optimization has also gained substantial interest lately.

Static optimization of data-plane programs. Several packet I/O frameworks present specific APIs for developers to optimize network code [9], [93]–[96], or implement different paradigms to efficiently execute packet-processing programs sequentially or in parallel [36], [85], [86], [97]–[101]. Other proposals aim to remove redundant logic or merge different elements together [102]–[104]. These works provide *static* optimizations; Morpheus, on top of these, also considers run time insight to specialize generic network code.

Dynamic optimization of packet-processing programs. ESwitch [18], [33] was the first functional framework for the unsupervised *dynamic* optimization of software data planes with respect to the packet-processing program, specified in OpenFlow, being executed. PacketMill [21] and NFReducer [22] leverage the LLVM toolchain [13] instead of OpenFlow: PacketMill targets the FastClick datapath by exploiting the DPDK packet I/O framework and NFReducer aims to eliminate redundant logic from generic packet processing programs using symbolic execution. Morpheus is strictly complementary to these works: (1) it applies some of the same optimizations but it also introduces a toolbox of new ones (e.g., branch injection or constant propagation for stable table entries); (2) Morpheus can detect packet-level dynamics and apply more aggressive optimizations depending on the specific traffic patterns; and (3) Morpheus is data-plane agnostic, in that it performs the optimizations at the IR-level using a portable compiler core and relies on the built-in compiler toolchain to generate machine code and a data-plane plugin to inject it into the datapath.

Profile-guided optimization for packet-processing hardware. P2GO [20] and P5 [19] apply several profile-driven optimizations to improve the resource utilization of programmable P4 hardware targets. Some of the ideas presented in this work can also be used with programmable P4 hardware, provided it is possible to re-synthesize the packet processing pipeline without traffic disruption, with a notable difference: P2GO and P5 require *a priori* knowledge (i.e., the profiles) while Morpheus aims at *unsupervised* dynamic optimization.

X. Conclusions & Future Work

We presented Morpheus, a run time compiler and optimizer framework for arbitrary networking code. We demonstrated the importance of tracking packet-level dynamics and how they open up opportunities for a number of domain-specific optimizations. We proposed a solution, Morpheus, capable of applying them without any *a priori* information on the running program and implemented on top of the LLVM JIT compiler toolchain at the IR level. This allows to decouple our system from the specific framework used by the underlying data plane as much as possible. Finally, we demonstrated the effectiveness of Morpheus on a number of programs written in eBPF and DPDK, on Kubernetes, and released the code in open-source to foster reproducibility of our results.

We consider Morpheus only as a first step towards more intelligent systems that can adapt to network conditions. As future work, we intend to integrate a run time performance prediction model [56], [105]–[108] into Morpheus, enabling the compiler to reason about the effect of each different dynamic optimization pass. This would allow for selecting the most efficient subset of optimizations and adapt the recompilation timescales to the current network conditions.

References

[1] S. Miano *et al.*, “Domain Specific Run Time Optimization for Software Data Planes,” ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1148–1164. [Online]. Available: https://doi.org/10.1145/3503222.3507769

[2] Sourcefire, “Snort - Network Intrusion Detection & Prevention System,” nov 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://www.snort.org/

[3] I. Authors, “Istio - Connect, secure, control, and observe services,” nov 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://istio.io/

[4] O. I. S. Foundation, “Suricata - intrusion detection system,” nov 2020, [Online; accessed 07-August-2021]. [Online]. Available: https://suricata-ids.org/

[5] J. Kempf, B. Johansson, S. Pettersson, H. Luning, and T. Nilsson, “Moving the Mobile Evolved Packet Core to the Cloud,” ser. WIMOB ’12. USA: IEEE Computer Society, 2012, p. 784–791. [Online]. Available: https://doi.org/10.1109/WiMOB.2012.6379165

[6] C. Hopps, “Katran: A high performance layer 4 load balancer,” September 2019, https://github.com/facebookincubator/katran.

[7] D. Wragg, “Unimog - Cloudflare’s edge load balancer,” sep 2020. [Online]. Available: https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/

[8] M. Xhonneux, F. Duchene, and O. Bonaventure, “Leveraging EBPF for Programmable Network Functions with IPv6 Segment Routing,” ser. CoNEXT ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–72. [Online]. Available: https://doi.org/10.1145/3281411.3281426

[9] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “NetBricks: Taking the V out of NFV,” ser. OSDI’16. USA: USENIX Association, 2016, p. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[10] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of Performance Acceleration Techniques for Network Function Virtualization," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019.

[11] O. Alipourfard and M. Yu, "Decoupling Algorithms and Optimizations in Network Functions," ser. HotNets '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 71–77. [Online]. Available: https://doi.org/10.1145/3286062.3286073

[12] GNU Project, "GNU Compiler Collection," [Online; accessed 13-November-2023]. [Online]. Available: https://gcc.gnu.org/

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[14] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 1–12. [Online]. Available: https://doi.org/10.1145/349299.349303

[15] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling java just in time," *IEEE Micro*, vol. 17, no. 3, p. 36–43, may 1997. [Online]. Available: https://doi.org/10.1109/40.591653

[16] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A Practical Binary Optimizer for Data Centers and Beyond," ser. CGO 2019. IEEE Press, 2019, p. 2–14. [Online]. Available: https://dl.acm.org/doi/10.5555/3314872.3314876

[17] D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications," ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–23. [Online]. Available: https://doi.org/10.1145/2854038.2854044

[18] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane Specialization for High-Performance OpenFlow Software Switching," ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 539–552. [Online]. Available: https://doi.org/10.1145/2934872.2934887

[19] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-Driven Optimization of P4 Pipeline," ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 136–142. [Online]. Available: https://doi.org/10.1145/3050220.3050235

[20] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2GO: P4 Profile-Guided Optimizations." ACM, 2020.

[21] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "PacketMill: Toward per-Core 100-Gbps Networking," ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–17. [Online]. Available: https://doi.org/10.1145/3445814.3446724

[22] B. Deng, W. Wu, and L. Song, "Redundant Logic Elimination in Network Functions," ser. SOSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 34–40. [Online]. Available: https://doi.org/10.1145/3373360.3380832

[23] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, "Fast, Effective Dynamic Compilation," ser. PLDI '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 149–159.

[24] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based Just-in-Time type specialization for dynamic languages," ser. PLDI '09, 2009, p. 465–478.

[25] R. Zhang, S. Debray, and R. T. Snodgrass, "Micro-specialization: Dynamic code specialization of database management systems," ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 63–73. [Online]. Available: https://doi.org/10.1145/2259016.2259025

[26] A. Kohn, V. Leis, and T. Neumann, "Adaptive Execution of Compiled Queries," 2018, pp. 197–208.

[27] W. contributors, "Perf (linux)," 2018, [Online; accessed 13-November-2023]. [Online]. Available: https://en.wikipedia.org/wiki/Perf_(Linux)

[28] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling." USENIX Association, July 2021, pp. 163–181. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/khan

[29] G. Inc., "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io, July 2019, [Online; accessed 07-August-2021].

[30] S. Miano, "Morpheus: Domain Specific Run Time Optimization for Software Data Planes - Artifact for ASPLOS'22," Zenodo, December 2021, version 1.3. [Online]. Available: https://doi.org/10.5281/zenodo.5830832

[31] G. Inc., "Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker," Oct 2019, [Online; accessed 13-November-2023]. [Online]. Available: https://github.com/google/llvm-propeller

[32] DPDK, "L3 forwarding with access control sample application," 2021, [Online; accessed 07-August-2021]. [Online]. Available: https://doc.dpdk.org/guides/sample_app_ug/l3_forward_access_ctrl.html

[33] G. Rétvári, L. Molnár, G. Enyedi, and G. Pongrácz, "Dynamic Compilation and Optimization of Packet Processing Programs," *ACM SIGCOMM NetPL*, 2017.

[34] J. Jackson, "Kubernetes long road to dual IPv4/IPv6 support," 2019, [Online; accessed 13-November-2023]. [Online]. Available: https://thenewstack.io/it-takes-a-community-kubernetes-long-road-to-dual-ipv4-ipv6-support

[35] "The Open Virtual Network architecture: Tunnel encapsulations," 2018, [Online; accessed 07-August-2021]. [Online]. Available: http://www.openvswitch.org/support/dist-docs/ovn-architecture.7.html

[36] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions," ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 504–517. [Online]. Available: https://doi.org/10.1145/3230543.3230563

[37] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating Complex Network Services with eBPF: Experience and Lessons Learned," 2018, pp. 1–8.

[38] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM transactions on networking*, vol. 15, no. 3, pp. 499–511, 2007.

[39] OpenStack Authors, "OpenStack," oct 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://www.openstack.org/

[40] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks." San Jose, CA: USENIX Association, April 2012, pp. 113–126. [Online]. Available: https://dl.acm.org/doi/10.5555/2228298.2228311

[41] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer." Renton, WA: USENIX Association, April 2018, pp. 125–139. [Online]. Available: https://dl.acm.org/doi/10.5555/3307441.3307453

[42] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. M. Jr., P. Papadimitratos, and M. Chiesa, "A High-Speed Load-Balancer design with guaranteed Per-Connection-Consistency." Santa Clara, CA: USENIX Association, February 2020, pp. 667–683. [Online]. Available: https://dl.acm.org/doi/10.5555/3388242.3388291

[43] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," ser. SIGPLAN '82. New York, NY, USA: Association for Computing Machinery, 1982, p. 120–126. [Online]. Available: https://doi.org/10.1145/800230.806987

[44] J. Levon and P. Elie, "Oprofile: A system profiler for linux," 2004, [Online; accessed 13-November-2023]. [Online]. Available: https://oprofile.sourceforge.io/news/

[45] W. contributors, "Dtrace," 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://en.wikipedia.org/wiki/DTrace

[46] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," ser. CGO '03. USA: IEEE Computer Society, 2003, p. 265–275. [Online]. Available: https://dl.acm.org/doi/10.5555/776261.776290

[47] G. Authors, "AutoFDO tutorial," 2016, [Online; accessed 13-November-2023]. [Online]. Available: https://gcc.gnu.org/wiki/AutoFDO/Tutorial

[48] I. Corporation, "Pin - A Dynamic Binary Instrumentation Tool," dec 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html

[49] M. Kuehlewind and B. Trammell, "Manageability of the QUIC transport protocol," Working Draft, Internet-Draft draft-ietf-quic-manageability-09, January 2021.

[50] "LLVM MemorySSA," Feb 2021, [Online; accessed 13-November-2023]. [Online]. Available: https://llvm.org/docs/MemorySSA.html

[51] "LLVM Alias Analysis," Feb 2021, [Online; accessed 13-November-2023]. [Online]. Available: https://llvm.org/docs/AliasAnalysis.html

[52] S. Miano, X. Chen, R. B. Basat, and G. Antichi, "Fast In-Kernel Traffic Sketching in EBPF," *SIGCOMM Comput. Commun. Rev.*, vol. 53, no. 1, pp. 3–13, apr 2023. [Online]. Available: https://doi.org/10.1145/3594255.3594256

[53] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," ser. SIGCOMM '02. New York, NY, USA: Association

for Computing Machinery, 2002, p. 323–336. [Online]. Available: https://doi.org/10.1145/633025.633056

[54] U. Hölzle and D. Ungar, "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback," ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 326–336. [Online]. Available: https://doi.org/10.1145/178243.178478

[55] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," vol. 3, 2000, pp. 1193–1202 vol.3.

[56] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated Synthesis of Adversarial Workloads for Network Functions," ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 372–385. [Online]. Available: https://doi.org/10.1145/3230543.3230573

[57] J. Worthington, "Eliminating unrequired guards," 2018, [Online; accessed 13-November-2023]. [Online]. Available: https://6guts.wordpress.com/2018/09/29/eliminating-unrequired-guards/

[58] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. W. Moore, and P. G. Neumann, "Blueswitch: enabling provably consistent configuration of network switches," 2015, pp. 17–27.

[59] L. Authors, "ORC Design and Implementation," https://llvm.org/docs/ORCv2.html, February 2023, [Online; accessed 18-February-2023].

[60] ——, "MCJIT Design and Implementation," https://llvm.org/docs/MCJITDesignAndImplementation.html, February 2023, [Online; accessed 18-February-2023].

[61] M. Shahbaz and N. Feamster, "The Case for an Intermediate Representation for Programmable Data Planes," ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2774993.2775000

[62] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 9. [Online]. Available: https://dl.acm.org/doi/10.5555/2342821.2342830

[63] "Linux AF_XDP," Feb 2021, [Online; accessed 13-November-2023]. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/af_xdp.html

[64] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, J. Pi, and A. Shaikh, "A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services," 2019, pp. 1–9. [Online]. Available: https://doi.org/10.1109/ANCS.2019.8901880

[65] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," ser. ANCS '15. USA: IEEE Computer Society, 2015, p. 5–16.

[66] "Intel Data Direct I/O Technology," Feb 2021, [Online; accessed 13-November-2023]. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/io/data-direct-i-o-technology.html

[67] DPDK, "Pktgen traffic generator using dpdk," aug 2018, [Online; accessed 07-August-2021]. [Online]. Available: http://dpdk.org/git/apps/pktgen-dpdk

[68] J. Ribas, "Dpdk burst replay tool," Jun 2019, [Online; accessed 13-November-2023]. [Online]. Available: https://github.com/FraudBuster/dpdk-burst-replay

[69] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[70] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A Framework for eBPF-based Network Functions in an Era of Microservices," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[71] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, "Securing Linux with a Faster and Scalable Iptables," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, p. 2–17, November 2019. [Online]. Available: https://doi.org/10.1145/3371927.3371929

[72] D. E. Taylor and J. S. Turner, "Classbench filter set & trace generator," accessed: 2023-11-11. [Online]. Available: https://www.arl.wustl.edu/classbench/

[73] S. Bradner, "Benchmarking Terminology for Network Interconnection Devices," Internet Requests for Comments, RFC Editor, RFC 1242, July 1991. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1242.txt

[74] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," Internet Requests for Comments, RFC Editor, RFC 2544, March 1999, http://www.rfc-editor.org/rfc/rfc2544.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2544.txt

[75] Yonghong Song, "bpf: adding map batch processing support," Aug 2019, [Online; accessed 12-August-2021]. [Online]. Available: https://lwn.net/Articles/797808/

[76] CAIDA, "The CAIDA UCSD Anonymized Internet Traces," 2019, [Online; accessed 13-November-2023]. [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml

[77] StackOverflow, What can cause my code to run slower when the server JIT is activated?, 2011, https://stackoverflow.com/questions/2923989/what-can-cause-my-code-to-run-slower-when-the-server-jit-is-activated.

[78] Oracle, "Java HotSpot VM Options," 2021, [Online; accessed 13-November-2023]. [Online]. Available: https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

[79] C. Authors, "Cilium: Api-aware networking and security using ebpf and xdp," 2020, https://github.com/cilium/cilium.

[80] ——, "Tigera adds ebpf support to calico," 2019, https://www.projectcalico.org/tigera-adds-ebpf-support-to-calico/.

[81] F. Parola, L. D. Giovanna, G. Ognibene, and F. Risso, "Creating disaggregated network services with ebpf: the kubernetes network provider use case," 2022, pp. 254–258. [Online]. Available: https://doi.org/10.1109/NetSoft54395.2022.9844062

[82] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 1–14.

[83] Kubernetes, "Benchmarking kubernetes networking performance," 2023, [Online; accessed 13-November-2023]. [Online]. Available: https://github.com/kubernetes/perf-tests/tree/master/network/benchmarks/netperf

[84] F. Shahinfar, S. Miano, G. Siracusano, R. Bifulco, A. Panda, and G. Antichi, "Automatic kernel offload using bpf," ser. HOTOS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 143–149. [Online]. Available: https://doi.org/10.1145/3593856.3595888

[85] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling Network Function Parallelism in NFV," ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 43–56. [Online]. Available: https://doi.org/10.1145/3098822.3098826

[86] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't forget the i/o when allocating your llc," ser. ISCA '21. IEEE Press, 2021, p. 112–125. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00018

[87] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, *VeGen: A Vectorizer Generator for SIMD and Beyond*, ser. ASPLOS'21. New York, NY, USA: Association for Computing Machinery, 2021, p. 902–914. [Online]. Available: https://doi.org/10.1145/3445814.3446692

[88] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 394–403. [Online]. Available: https://doi.org/10.1145/1168857.1168906

[89] R. Joshi, G. Nelson, and K. Randall, "Denali: A Goal-Directed Superoptimizer," ser. PLDI '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 304–314. [Online]. Available: https://doi.org/10.1145/512529.512566

[90] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," 2017. [Online]. Available: https://arxiv.org/abs/1711.04422

[91] M. Mukherjee, P. Kant, Z. Liu, and J. Regehr, "Dataflow-Based Pruning for Speeding up Superoptimization," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, November 2020. [Online]. Available: https://doi.org/10.1145/3428245

[92] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up Superoptimization," ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 297–310. [Online]. Available: https://doi.org/10.1145/2872362.2872387

[93] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," 2015.

[94] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, and C. Kim, "PVPP: A Programmable Vector Packet Processor," ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 197–198. [Online]. Available: https://doi.org/10.1145/3050220.3060609

[95] ——, "The Case for a Flexible Low-Level Backend for Software Data Planes," ser. APNet'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 71–77. [Online]. Available: https://doi.org/10.1145/3106989.3107000

[96] L. Foundation, "Vector Packet Processing (VPP) platform," Oct 2020, [Online; accessed 13-November-2023]. [Online]. Available: https://wiki.fd.io/view/VPP

[97] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV Service Chains at the True Speed of the Underlying

Hardware." Renton, WA: USENIX Association, April 2018, pp. 171–186. [Online]. Available: https://dl.acm.org/doi/10.5555/3307441.3307457

[98] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," ser. NSDI'14. USA: USENIX Association, 2014, p. 459–473. [Online]. Available: https://dl.acm.org/doi/10.5555/2616448.2616491

[99] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling End-Host Network Functions," ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 493–507. [Online]. Available: https://doi.org/10.1145/2785956.2787493

[100] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and Implementation of a Consolidated Middlebox Architecture." San Jose, CA: USENIX, 2012, pp. 323–336. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar

[101] F. Parola, F. Risso, and S. Miano, "Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway," 2021, pp. 221–225. [Online]. Available: https://doi.org/10.1109/NetSoft51509.2021.9492571

[102] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 511–524. [Online]. Available: https://doi.org/10.1145/2934872.2934875

[103] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015, https://doi.org/10.1145/2774993.2775000.

[104] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "SNF: synthesizing high performance NFV service chains," *PeerJ Computer Science*, vol. 2, p. e98, November 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.98

[105] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 270–282. [Online]. Available: https://doi.org/10.1145/3387514.3405868

[106] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions." USENIX Association, February 2019, pp. 517–530.

[107] A. Bhardwaj, A. Shree, V. B. Reddy, and S. Bansal, "A preliminary performance model for optimizing software packet processing pipelines," ser. APSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3124680.3124747

[108] F. Rath, J. Krude, J. Rüth, D. Schemmel, O. Hohlfeld, J. A. Bitsch, and K. Wehrle, "Symperf: Predicting network function performance," ser. SIGCOMM Posters and Demos '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 34–36. [Online]. Available: https://doi.org/10.1145/3123878.3131977

Sebastiano Miano received his M.S. and Ph.D degree in Computer Engineering from Politecnico di Torino. He is now Assistant Professor at the Politecnico di Milano at the Department of Electronics, Information and Bioengineering. Previously, he was PostDoc researcher at the School of Electronic Engineering and Computer Science of Queen Mary Univerisity of London (QMUL). He is particularly interested in programmable data planes and high-speed network function virtualization with a focus on eBPF and XDP.

Alireza Sanaee received his BSc and MSc degrees in computer science from the School of Computer Engineering, Iran University of Science and Technology (IUST) in 2016 and 2018, respectively, supervised by Prof Mohsen Sharifi. Currently, he is a graduate student at Queen Mary University of London, advised by Dr. Gianni Antichi. His research interests focus on designing operating and networked systems at different levels of abstraction for data centers to achieve higher performance.

Fulvio Risso received the M.Sc. (1995) and Ph.D. (2000) in computer engineering from Politecnico di Torino, Italy. He is currently Associate Professor at the same University. His research interests focus on high-speed and flexible network processing, edge/fog computing, software-defined networks, network functions virtualization. He has co-authored more than 100 scientific papers.

Gábor Rétvári received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology and Economics in 1999 and 2007. He is now the Ericsson Systems Professor at the Department of Telecommunications and Media Informatics and a co-founder and CTO of L7mp.io, an expert group focusing on cloud-native real-time communications. His research interests include all aspects of network routing and switching, the programmable data plane and the cloud-native network stack.

Gianni Antichi received the M.Sc. and Ph.D. degrees in telecomunication engineering from the University of Pisa in 2007 and 2011. He is now Associate Professor at Politecnico di Milano and Senior Lecturer at Queen Mary University of London. His research interests are at the intersection of systems and networking with a special focus on end-host networking and dataplane offloading.