Domain Specific Run Time Optimization for Software Data Planes

Sebastiano Miano
Queen Mary University London
London, United Kingdom
s.miano@qmul.ac.uk

Alireza Sanaee
Queen Mary University London
London, United Kingdom
a.sanaee@qmul.ac.uk

Fulvio Risso
Politecnico di Torino
Turin, Italy
fulvio.risso@polito.it

Gábor Rétvári
Budapest University of Technology
and Economics & Ericsson Research
Budapest, Hungary
retvari@tmit.bme.hu

Gianni Antichi
Queen Mary University London
London, United Kingdom
g.antichi@qmul.ac.uk

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 eBPF and DPDK programs including Katran, Facebook's production-grade load balancer. We compare Morpheus against state-of-the-art optimization frameworks and show that it can bring up to 2x throughput improvement, while halving the 99th percentile latency.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; **Data center networks**; *End nodes*; • **Software and its engineering** → **Dynamic compilers**; *Just-in-time compilers*.

KEYWORDS

Data Plane Compilation, LLVM, eBPF, XDP, DPDK

1 INTRODUCTION

Software Data Planes, packet processing programs implemented on commodity servers, are widely adopted in real deployments [10, 39, 46, 55, 78, 89, 96, 97]. 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 [7, 61]) during the compilation process [41, 59]. These are mainly *static* transformations, independent of the actual input the code will process in operation, as this is unknown until then [11, 30]. 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 [8, 30, 40]. 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 [30], JavaScript [40], and C/C++ [8]) and often produces orders of magnitude more efficient code as shown for data-caching services [77], data mining [23] and databases [57, 100]. Unfortunately, this is not the case for packet-processing programs [7, 31, 71], since their performance critically depends on *highly dynamic domain-specific knowledge*, such as traffic patterns, match-action table content, and network configuration (§2). 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 [56]. Therefore, existing solutions tailored for the networking domain (Table 2) 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.

Table 1: A comparison of some popular dynamic optimization frameworks and Morpheus.

Name	Domain specific	Unsupervised adaptation to control plane actions	Unsupervised adaptation to data plane traffic	Data plane agnostic	Description
Bolt [77]	✗	-	-	✓	Offline profile-guided optimizer for generic software code.
AutoFDO [23]	✗	-	-	✓	Offline profile-guided optimizer for generic software code.
eSwitch [71]	✓	✓	✗	✗	Policy-driven optimizer for DPDK-based OpenFlow software switches.
P5 [6]	✓	✗	✗	✗	Policy-driven optimizer for P4/RMT packet-processing pipelines.
P2GO [94]	✓	✗	✗	✗	Offline profile-guided optimizer for P4/RMT packet-processing pipelines.
PacketMill [36]	✓	✗	✗	✗	Packet metadata management optimizer for DPDK-based software data planes.
NFReducer [31]	✓	✗	✗	✓	Policy-driven optimizer for network function virtualization.
Morpheus	✓	✓	✓	✓	Run time compiler and optimizer framework for arbitrary networking code.

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 (§3), 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 protect the consistency of the specialized code against changes to input that is considered invariant by injecting *guards* (§4).

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 (§5). 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, using synthetic and real-world traffic traces (§6). 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%. Finally, we measured Morpheus against state-of-the-art network code optimization frameworks such as ESwitch [71] and PacketMill [36]: we show that Morpheus boosts the throughput by up to 80% and 294%, respectively, compared to existing work.

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;
- share the code in open source to foster reproducibility [65][1].

2 THE CASE FOR DOMAIN-SPECIFIC OPTIMIZATIONS

State-of-the-art profile guided optimization tools (PGO), such as Google's AutoFDO [23, 48] and Facebook's Bolt [77], 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 [33], 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 §6 for the details of the configuration). In line with the expectations [23, 56, 77], 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 [7, 31, 36, 71, 82].

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 [46], 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 [50] or adopt a single virtualization technology (VLAN/VxLAN/ GRE/Geneve/GTP) and so switches would never see other encapsulations in operation [55, 76]. This implies that, depending on dynamic input that is unknown at compile time, a huge body of

---

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

unused code gets assembled into the program, boosting code size and causing excess branch prediction misses, negatively impacting the overall performance [7, 31, 62, 69].

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 Class-Bench [93]. This opens up an 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 [47, 74] or whitelist-based access control, most firewall rules are fully-specified; for instance, in the official Stanford ruleset [54] 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 with this simple modification.

A similar effect is visible with the *load-balancer* (Fig. 1c): configuring Katran as an HTTP load balancer [15, 73] 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 [71], P4 software [82, 94] and hardware targets [6], network functions [31], and programmable switches [36] (see Table 1). 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 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.
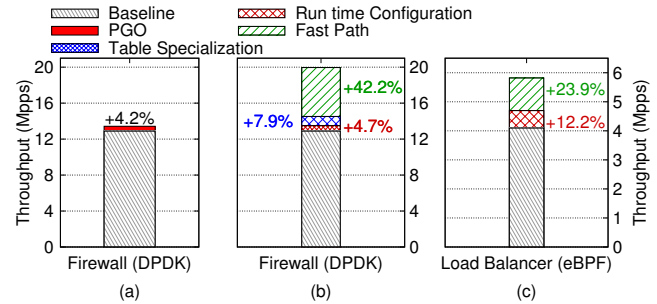

Figure 1: (a) Impact of AutoFDO+Bolt (PGO) and performance breakdown when applying a set of domain specific optimizations to both (b) the DPDK *firewall* and (c) the Facebook's Katran eBPF *load balancer*. We were unable to run PGO on the latter, since existing tools do not support eBPF targets.

3 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 [8]. 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:

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, 28, 42, 60], they track high-level code behavior information such as cache events, branch misses or memory accesses which is not enough for packet-processing code (§2). Less lightweight and more accurate tools [9, 20, 29], 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 [56]. We tackle this challenge in Morpheus by using *static code analysis* to understand the structure of the program offline (§4.1) and leveraging an *adaptive instrumentation* mechanism to minimize the amount of data that is collected online (§4.2).

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 (§4.3).

Challenge #3: Consistency. The dynamically optimized data plane is contingent on the assumption that the data considered

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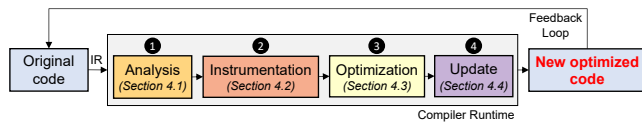

Figure 2: The Morpheus compiler pipeline.

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 elision* heuristic to sidestep useless guards (§4.3). To do so, our static code analysis tool must have enough understanding of the program to separate *stateless from stateful code* (§4.3). Finally, mechanisms are needed to *atomically update* the data plane once the code is re-optimized for the new invariants (§4.4).

4 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 §4.1) and then reads a comprehensive set of instrumentation tables to extract run time match-action table access patterns (see §4.2). Finally, Morpheus invokes a set of dynamic compilation passes to specialize the code (see §4.3) and then replaces the running data plane with the new, optimized code on the fly (see §4.4).

Below, we review the above steps in more detail. We use the simplified main loop of the *Katran* XDP/eBPF load balancer [46] 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 [58]. 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).

4.1 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

Listing 1: Simplified Katran main loop

```
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   }
```

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* [7]. Rather, it relies on tracking the map access patterns and uses this information to *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* [4] and *alias analysis* [3] 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 (§5.1), 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.

Table 2: 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).

Optimization	Description	Small RO maps	Large RO maps	RW maps	Traffic-depend.
JIT (§4.3.1)	inline frequently hit table entries into the code	✓	✓	✓	✓
Table Elimination (§4.3.1)	remove empty tables	✓	✓	✗	✗
Constant Propagation (§4.3.2)	substitute run time constants into expressions	✓	✓	✗	✓
Dead Code Elimination (§4.3.3)	remove branches that are not being used	✓	✓	✗	✓
Data Structure Specialization (§4.3.4)	adapt map implementation to the entries stored	✓	✓	✓	✗
Branch Injection (§4.3.5)	prevent table lookup for select inputs	✓	✓	✗	✗
Guard Elision (§4.3.6)	eliminate useless guards	✓	✓	✗	✗

4.2 Instrumentation

In the second pass, Morpheus profiles the dynamics of the input traffic by generating *heatmaps* of the maps' access 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, 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 [35]. (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 2). 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.

4.3 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 [45]. 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 2 for a summary.

4.3.1 Just-in-Time Compilation (JIT). Empirical evidence (see §2) 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 [37]. 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.

4.3.2 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) *inlined*, which
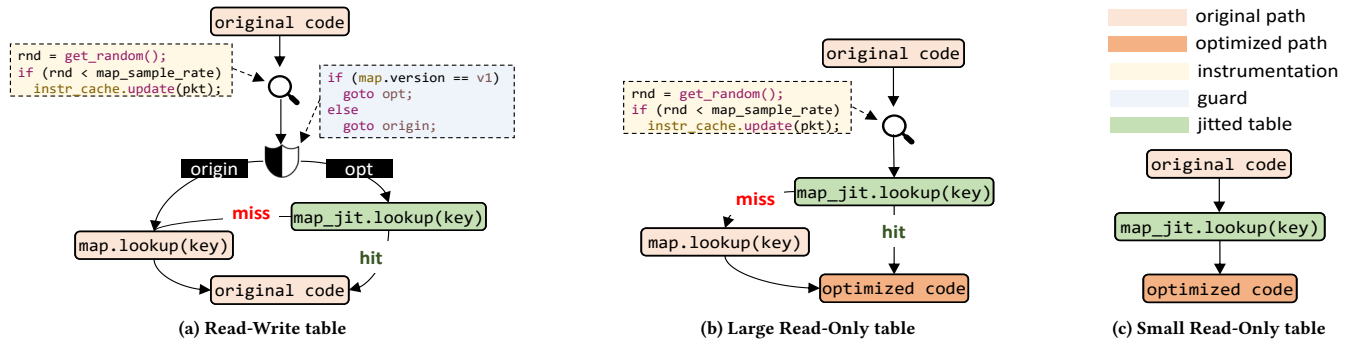
(a) Read-Write table   (b) Large Read-Only table   (c) Small Read-Only table

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

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.

4.3.3 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.

4.3.4 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 [71] 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 [77, 79]), generate the

expected cost of each candidate, and finally implement the table that minimizes the cost.

4.3.5 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 §2 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.

4.3.6 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 [95]. 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.

4.4 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 [43] and the outstanding table updates are executed.

5 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 [65]. 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 [59] compiler toolchain (v10.0.1) 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 [88].

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, implement the data plane dependent parts of the pipeline update mechanism, and provide a mechanism for the Morpheus core to intercept, inspect, and queue any update made by the control plane. The latter allows the compilation pipeline to be triggered when Morpheus intercepts a control plane event, e.g., an update to a table. 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 [84] or AF_XDP [2].

5.1 The eBPF Plugin

Morpheus leverages the Polycube [68] 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.

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 [43]. 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.

5.2 The DPDK Plugin

Morpheus leverages FastClick [14], 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. 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.

6 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) [1] 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` [32] with DPDK v20.11.0 to generate traffic and report the throughput results, and the DPDK burst replay tool [83] 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* [34] 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 [46].

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 [54]. 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 [67] of the well-known Linux *iptables* framework, configured with 5-tuple rules generated by Classbench [93]. We used the Classbench trace generator [92]
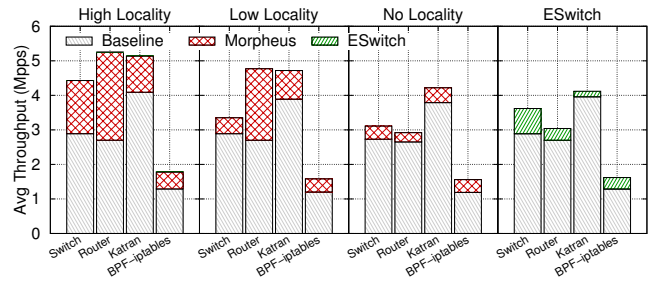

Figure 4: Single core throughput (64B packets) varying input traffic locality. The optimizations adopted by Morpheus are traffic-dependent, while the ones from ESwitch [71] are not. For this reason, the ESwitch throughput (shown in the right box) is the same across the different traffic localities.
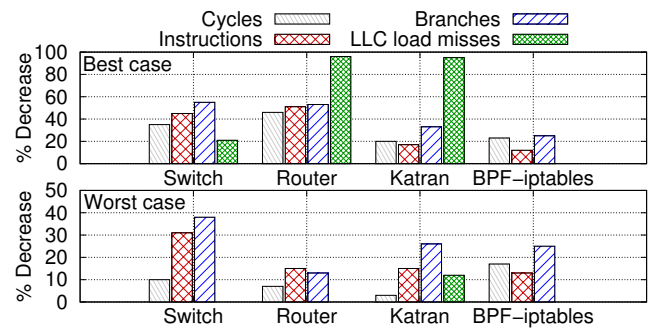

Figure 5: Effect of Morpheus optimizations on PMU counters, obtained with *perf* at the default frequency (40KHz). The top panel shows the percentage of *decrease*, per packet, of different metrics for *high locality* traffic (best-case for Morpheus), and the bottom panel for *no locality* traffic (worst-case).

to generate packets matching the created rule set using a *Pareto* cumulative density function to control the locality of reference. We used the same default parameters suggested by the ClassBench paper [93] 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* [46] 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.

6.1 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. 4, we show the impact of Morpheus under different traffic conditions. Throughput is defined as the maximum packet-rate sustained by
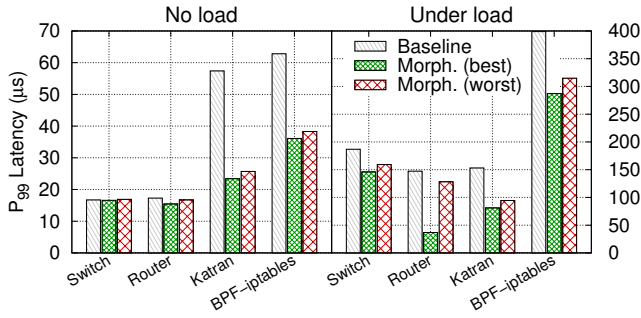
Figure 6: 99th percentile (P$_{99}$) latency with Morpheus. The graph shows both the latency for the *optimized* and *non-optimized* code paths, under small load (10pps) and heavy load (highest rate without packet drop).

a program without experiencing packet loss. When a small subset of flows sends the majority of traffic (high-locality), Morpheus consistently provides more than 50% throughput improvement over the baseline, with a 2× speed-up for the *Router*. This is because it can track heavy flows and optimize the code accordingly. 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 [71]. The results (Fig 4) 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. 5, 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 2).

Morpheus reduces packet-processing latency. In Fig. 6, 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. 6 shows the latency measured at low packet rate (10pps) so to avoid queuing effects [17], whereas the right panel shows latency under the maximum sustained load without packet drops [18]. 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%.

6.2 What Is the Cost of Code Instrumentation?

Clearly, the price for performance improvements is the additional logic, most prominently, instrumentation, injected by Morpheus
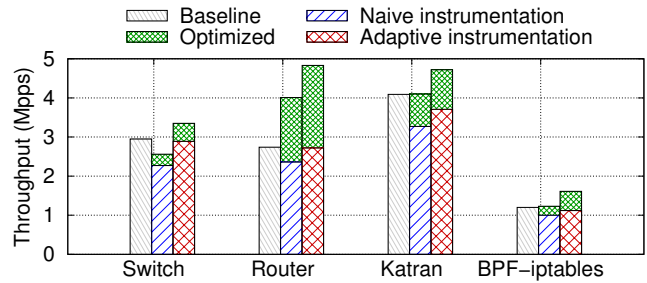

Figure 7: Naive vs adaptive instrumentation (low locality traffic). In the naive case all map lookups are recorded, while adaptive instrumentation adjusts data sampling selectively for the access patterns at each lookup call site.
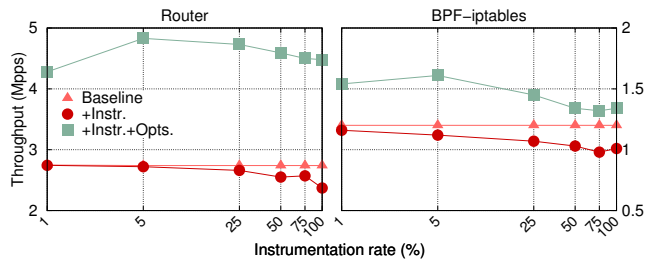

Figure 8: Effectiveness of instrumentation at varying sampling rates (Router and BPF-iptables, low-locality traffic).

into the fast packet-processing path. To understand this price, we compared our *adaptive* instrumentation scheme (§4.2) against a *naive* approach where all map lookups are explicitly recorded. Fig. 7 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. 8 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.

Table 3: 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.

6.3 How Fast Is the Compilation?

In Table 3, 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% [98], 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).

6.4 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 the time (Fig. 9a). 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
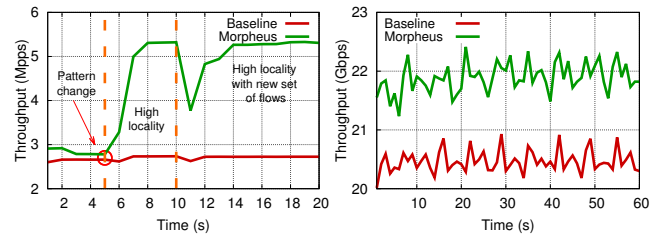

Figure 9: (a) Single core throughput over time with Morpheus on the *Router* use case, with dynamically changing traffic patterns, and (b) with a CAIDA [22] trace.
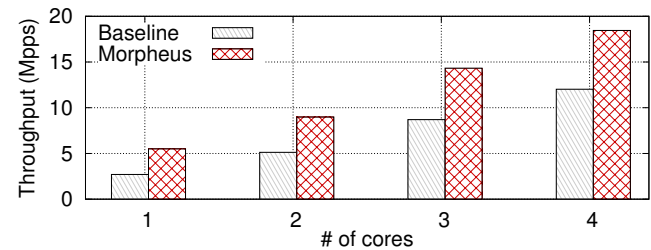

Figure 10: Multicore application (router) with Morpheus.

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* [22]), 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 9b, we show how Morpheus consistently improves the throughput of the router by factor of ~10%.

Finally, in Figure 10, 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.

6.5 What Can Go Wrong?

The flip side of dynamic optimization is the potential for a misguided run time code transformation to harm performance. With generic languages this can happen when the dynamic compiler steals CPU cycles from the running code [30, 90]; in such cases careful manual compiler parameter tuning and deep application-specific knowledge is needed to make up for the lost performance [75]. Similar issues may arise with dynamically optimizing *network code*, as we show below on the *NAT* use case [70]. The NAT is organized as a single large connection tracking table, updated from within the data plane on 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 2) Morpheus can improve throughput by around 5% (from 4.36 to 4.58
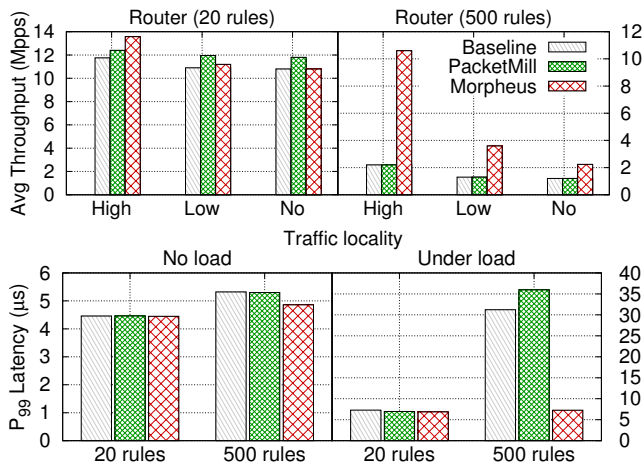
Figure 11: Comparison between vanilla FastClick, PacketMill and Morpheus for the Router FastClick (DPDK) application with 20 and 500 rules.

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. The rest of the stateful applications (*L2 switch* and *Katran*) exhibit a similar pattern, but the speed-up enabled by dead code elimination, constant propagation and branch-injection can make up for this. 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.

6.6 Morpheus with DPDK Programs

We applied Morpheus to a DPDK program, the FastClick [14] version of the eBPF *Router* application, the same one used in PacketMill's paper [36]. We configured it with either 20 or 500 rules from the Stanford routing tables [54] 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. 11a reports the average throughput results. For only 20 prefix rules and with low locality traffic, PacketMill 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. 11b) 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.

7 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 [36], 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 makes it relatively easy to port Morpheus to new dataplanes that use well defined APIs (e.g, for table lookups or packet reception). This is the case of eBPF/XDP and DPDK FastClick, but any data plane with a clear API can be potentially used with Morpheus. To do this, the developer would need to write a new data plane plugin, providing the function signatures for the relevant API calls as well as some simple operators for Morpheus to read match-action table content and manipulate the underlying code (e.g., inject a guard).

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 [13, 24, 51, 91, 99]. 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 4, 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 3). 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 6.5); this would reduce the impact of guards and instrumentation and increase the benefits of traffic-independent optimizations. Examining these techniques remains as future work.

8 RELATED WORK

Generic code optimization has a long-standing stream of research and prototypes [13, 23, 48, 51, 72, 77, 80, 85]. 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 [25, 26, 38, 44, 78], or implement different paradigms to efficiently execute packet-processing programs sequentially or in parallel [12, 52, 62, 64, 86, 91, 99]. Other proposals aim to remove redundant logic or merge different elements together [19, 53, 87]. 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 [71, 82] 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 [36] and NFReducer [31] leverage the LLVM toolchain [59] 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 [94] and P5 [6] 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.

9 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 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 [16, 49, 63, 79, 81] 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.

A ARTIFACT APPENDIX

A.1 Abstract

Morpheus is a system working alongside static compilers that continuously optimizes the targeted networking code. It introduces a number of new techniques to manipulate the code on-the-fly depending on runtime traffic patterns and control plane configurations. The Morpheus core exploits the LLVM compiler toolchain for code manipulation and run-time code generation. It works 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.

Our paper's artifact contains the source code, the experimental workflow, and additional information to (i) compile and build Morpheus, (ii) install the software dependencies and setup the testbed to run all the experiments, (iii) the scripts that can be used to perform some of the experiments presented in the paper, and (iv) the scripts to generate the plots based on the obtained results.

A.2 Artifact Check-list (Meta-information)

- **Program:** Morpheus, Polycube (Morpheus branch), BCC (Morpheus branch), libbpf (21f90f6).
- **Compilation:** GCC v9.3.0 to compile Morpheus, LLVM Toolchain and Clang (v10.0.1) for run-time code generation.
- **Data set:** High/low/no locality traces generated with Classbench [93] trace generator with different Pareto parameters; CAIDA 2019 dataset, *equinix-nyc* [22] to test real-world scenarios.
- **Hardware:** Two servers connected back-to-back with a dual-port Intel XL710 40Gbps NIC and Intel Xeon processors.
- **Execution:** The artifacts can be executed in a local testbed, or on a Cloudlab [5] setup.
- **Metrics:** Throughput, Latency and Compilation Time.
- **Output:** A modified Polycube binary with Morpheus support [66] that optimizes the target program at runtime. This artifact produces also raw performance measurements and plots of the different NFs while receiving traffic.
- **Experiments:** *(1)* Baseline vs Morpheus *throughput* experiments (Figure 4), *(2)* Effect of Morpheus optimizations on PMU counters (Figure 5), *(3)* Baseline vs Morpheus *latency* experiments (Figure 6), *(4)* Naive vs Adaptive instrumentation (Figure 7), *(5)* Instrumentation overhead (Figure 8), *(6)* Effect of Morpheus optimizations with dynamic changing traffic (Figure 9a).
- **How much disk space required (approximately)?:** ~30GB.
- **How much time is needed to prepare workflow (approximately)?:** ~1.5 hour with a local testbed, ~30 mins with Cloudlab setup.
- **How much time is needed to complete experiments (approximately)?:** ~1 day.
- **Publicly available?:** Code, setup scripts, measured results, and experiments are publicly available. The CAIDA 2019 dataset, *equinix-nyc* should be requested separately on the CAIDA website [21].
- **Code licenses (if publicly available)?:** Morpheus, Polycube and BCC use the Apache License 2.0, libbpf uses the 2-Clause BSD License.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5830832.

A.3 Description

A.3.1 How to Access. Our artifacts are archived under Zenodo [65]. We also published the software and artifact, with the corresponding setup instructions at the following Github repository (https://github.com/Morpheus-compiler/Morpheus).

Note: The GitHub repository is the preferred method to access our artifacts, since it provides the latest updates and fixes. The repository is linked with Zenodo so that every release is automatically archived.

A.3.2 Hardware Dependencies. In order to fully reproduce the results presented in the paper and run the majority of the experiments, this artifact requires two servers connected back-to-back with 2 different interfaces. The first server is used as *packet generator*, while the second is used as *Device Under Test (DUT)* running the data plane application together with the Morpheus compiler/optimizer. The driver used by the two interface in the DUT server should be compatible with *XDP Native* mode[2]; we have tested our artifact only with Intel cards, in particular a dual-port Intel XL710 40Gbps NIC. In addition, the NIC should should have the possibility to set hardware rules to redirect packets to a single core (e.g., Intel Flow Director technology).

A.3.3 Software Dependencies. We provide here a tentative list of software dependency required to run our artifact, for both packet generator server (pktgen) and Device Under Test server (DUT). The setup script available in the repository will take care of installing all the libraries and software needed for the artifact, just make sure you meet the following requirements:

- Ubuntu 20.04 LTS (pktgen and DUT).
- DPDK v20.11.3 (pktgen).
- At least 10G of large hugepages (1G each) required to send large PCAPs (pktgen).
- Linux kernel v5.10+ (the paper's test have been performed on kernel v5.12) (DUT).
- The `bpf_jit_enable` flag enabled on the kernel[3] (DUT).

A.4 Installation

If you have a local testbed that meets the hardware (Section A.3.2) and software (Section A.3.3) requirements, you can directly follow the instructions in Section A.5. If you do not have a local testbed, we also prepared a Cloudlab [5] profile to run the artifacts.

A.4.1 CloudLab Setup. To run our artifact on Cloudlab, you can follow these steps:

(1) Login to Cloudlab (https://cloudlab.us/) using your username and password.
(2) Go to `Experiments > Start Experiment` on the left tab.
(3) Click on `Change Profile` button on the right side of the page, and select the *Morpheus-ASPLOS* profile; then click `Next`. You can now choose a hardware node type. We already provide a default type (i.e., *c220g1*) but you can use another (we only tested it on *c220g1*).
    Note: Although c220g1 hardware nodes are similar to our paper's testbed, it is not the same; the absolute results may be different, but the Morpheus behavior should be evident.

---

[2]A list of drivers that support XDP Native is available here: https://github.com/xdp-project/xdp-project/blob/master/areas/drivers/README.org.
[3]This flag is enabled by default on kernel v4.18+.

(4) After the experiment is instantiated, you just need to wait for the setup to complete; then you can click on the `List View` tab to read the *ssh* command to log into the servers.

(5) Follow the instructions in Section A.5 to install Morpheus.

A.5 Morpheus Installation

We provide a set of script to automatically install all the software dependencies, compile the code and setup the testbed (e.g., setup the hugepages or bound the interfaces to DPDK on the pktgen server). Morpheus's `README.md` describes all the steps needed to prepare the testbed and install the software. Here we provide a summary of the steps:

(1) **(Pktgen & DUT)** Clone or download Morpheus repository.

(2) **(Pktgen)** Install dependencies.

```
$ ./setup_pktgen.sh
```

(3) **(DUT)** Install Morpheus and dependencies.

```
$ ./setup_dut.sh
```

(4) **(DUT)** Upgrade the kernel to v5.12 (if needed). We provide a `upgrade-kernel.sh` script to simplify the process.

(5) **(Pktgen & DUT)** Fill the `experiment/config.sh` file with all the information specific to the testbed (e.g., IP, MAC, username). This file is used by all the experiments scripts to automatically configure the experiments to the testbed.

(6) **(Pktgen & DUT)** SSH Configuration. The experiments' scripts automatically connect to the remote DUT and configure Morpheus. This requires the DUT to be accessible through SSH key.

(7) **(Pktgen & DUT)** SUDO Configuration. All the scripts require `sudo` without password to be enabled for the user specified in the `experiment/config.sh` file.

A.6 Experiment Workflow

Every experiment in this artifact is meant to be executed on the *pktgen* server. Once executed, the script reads the configuration parameters set in the `experiment/config.sh` file, starts the `poly-cubed` daemon with Morpheus on the *DUT*, instantiates the NF under test and configures it. If those steps are executed successfully, it start the DPDK packet generator to replay the high/low/no locality trace and collects the results in the `results` folder of every single service. Once the experiments are completed, the user can parse the results with the appropriate script under the `experiments/plot` folder and generate the final figures.

A.7 Evaluation and Expected Results

In this Section we present the main experiments included in this artifact. For more information please refer to the experiments' `README.md` of the Morpheus repository.

A.7.1 Baseline vs Morpheus Throughput Experiment. This experiment measures the single core throughput (with 64B packets) with varying input traffic locality. This is the main experiment, which evaluates the performance improvements provided by Morpheus with 4 different applications (Router, Switch, BPF-iptables and Katran). The results should follow a similar trend as Figure 4. The experiment can be executed with the following command:

```
$ ./run_all_throughput_tests.sh -r 5 #5 runs
$ cd plot && python3 parse_throughput_loc_results.py
$ make throughput_loc # generate figure
```

A.7.2 Effect of Morpheus Optimizations on PMU Counters. This experiment measures the effect of Morpheus optimizations on PMU counters, obtained with the Linux `perf` tool, for 4 different applications (Router, Switch, BPF-iptables and Katran). The results should follow a similar trend as Figure 5. The experiment can be executed with the following command:

```
$ ./run_all_perf_tests.sh -r 5 #5 runs
$ cd plot && python3 parse_perf_loc_results.py
$ make perf_loc # generate figure
```

A.7.3 Baseline vs Morpheus Latency Experiment. This experiment measures the effect of Morpheus optimizations on the latency for the different services. In particular, it calculates the latency in the best and worst case scenario depending on the Morpheus optimizations applied. The results should follow a similar trend as Figure 6. The experiment can be executed with the following command:

```
$ ./run_all_latency_tests.sh -r 5 #5 runs
$ cd plot && python3 parse_latency_results.py
$ make latency # generate figure
```

A.7.4 Naive vs Adaptive Instrumentation. This experiment measures the effect of Morpheus instrumentation mechanism. In particular, it measures the overhead that would be introduced by the *naive* instrumentation, compared to the overhead of the *adaptive* instrumentation, which is the one applied by Morpheus. The results should follow a similar trend as Figure 7. The experiment can be executed with the following command:

```
$ ./run_all_adaptive_naive_tests.sh -r 5 #5 runs
$ python3 parse_adaptive_naive_instr_results.py
$ make adaptive_vs_naive # generate figure
```

A.7.5 Instrumentation Overhead. This experiment measures the effect of Morpheus instrumentation rate on the optimizations applied. It measures the throughput of the *Router* and *BPF-iptables* service with varying instrumentation rate. The results should follow a similar trend as Figure 8.
Note: To run this experiment, you first need to run the Experiment A.7.1, since to generate the final figure, we rely on the results of that test. The experiment can be executed with the following command:

```
$ ./run_all_instr_overhead_tests.sh -r 5 #5 runs
$ cd plot && python3 parse_instr_overhead_results.py
$ make instr_overhead # generate figure
```

A.7.6 Effect of Morpheus Optimizations with Dynamic Changing Traffic. This experiment measures the effect of Morpheus optimizations when the input traffic suddenly changes locality distribution. It measures the throughput of the *Router* service. The results should follow a similar trend as Figure 9a. The experiment can be executed with the following command:

```
$ cd router
$ ./start_dynamic_traffic_test.sh
$ cd plot && make dynamic_traffic # generate figure
```

REFERENCES

[1] 2021. Intel Data Direct I/O Technology. https://www.intel.co.uk/content/www/uk/en/io/data-direct-i-o-technology.html. [Online; accessed 12-August-2021].

[2] 2021. Linux AF_XDP. https://www.kernel.org/doc/html/latest/networking/af_xdp.html. [Online; accessed 12-August-2021].

[3] 2021. LLVM Alias Analysis. https://llvm.org/docs/AliasAnalysis.html. [Online; accessed 12-August-2021].

[4] 2021. LLVM MemorySSA. https://llvm.org/docs/MemorySSA.html. [Online; accessed 12-August-2021].

[5] 2022. CloudLab. https://cloudlab.us/. [Online; accessed 26-January-2022].

[6] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. 2017. P5: Policy-Driven Optimization of P4 Pipeline. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 136–142. https://doi.org/10.1145/3050220.3050235

[7] Omid Alipourfard and Minlan Yu. 2018. Decoupling Algorithms and Optimizations in Network Functions. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 71–77. https://doi.org/10.1145/3286062.3286073

[8] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 149–159. https://doi.org/10.1145/231379.231409

[9] GCC Authors. 2016. AutoFDO tutorial. https://gcc.gnu.org/wiki/AutoFDO/Tutorial. [Online; accessed 07-August-2021].

[10] Istio Authors. 2020. Istio - Connect, secure, control, and observe services. https://istio.io/. [Online; accessed 07-August-2021].

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/349299.349303

[12] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 493–507. https://doi.org/10.1145/2785956.2787493

[13] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

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

[15] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://doi.org/10.5555/3388242.3388291

[16] Ankit Bhardwaj, Atul Shree, V. Bhargav Reddy, and Sorav Bansal. 2017. A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 26, 7 pages. https://doi.org/10.1145/3124680.3124747

[17] Scott Bradner. 1991. *Benchmarking Terminology for Network Interconnection Devices.* RFC 1242. RFC Editor. 1–12 pages. https://www.rfc-editor.org/rfc/rfc1242.txt

[18] Scott Bradner and Jim McQuaid. 1999. *Benchmarking Methodology for Network Interconnect Devices.* RFC 2544. RFC Editor. http://www.rfc-editor.org/rfc/rfc2544.txt http://www.rfc-editor.org/rfc/rfc2544.txt.

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

[20] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 265–275. https://dl.acm.org/doi/10.5555/776261.776290

[21] CAIDA. 2019. The CAIDA Anonymized Internet Traces Data Access. https://www.caida.org/catalog/datasets/passive_dataset_download/. [Online; accessed 26-January-2022].

[22] CAIDA. 2019. The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/data/passive/passive_dataset.xml. [Online; accessed 07-August-2021].

[23] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[24] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. *VeGen: A Vectorizer Generator for SIMD and Beyond.* Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[25] Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. 2017. PVPP: A Programmable Vector Packet Processor. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 197–198. https://doi.org/10.1145/3050220.3060609

[26] Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. 2017. The Case for a Flexible Low-Level Backend for Software Data Planes. In *Proceedings of the First Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 71–77. https://doi.org/10.1145/3106989.3107000

[27] Wikipedia contributors. 2018. Perf (linux). https://en.wikipedia.org/wiki/Perf_(Linux). [Online; accessed 07-August-2021].

[28] Wikipedia contributors. 2020. Dtrace. https://en.wikipedia.org/wiki/DTrace. [Online; accessed 07-August-2021].

[29] Intel Corporation. 2020. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html. [Online; accessed 07-August-2021].

[30] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. 1997. Compiling Java Just in Time. *IEEE Micro* 17, 3 (may 1997), 36–43. https://doi.org/10.1109/40.591653

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

[32] DPDK. 2018. Pktgen Traffic Generator Using DPDK. http://dpdk.org/git/apps/pktgen-dpdk. [Online; accessed 07-August-2021].

[33] DPDK. 2021. L3 Forwarding with Access Control Sample Application. https://doc.dpdk.org/guides/sample_app_ug/l3_forward_access_ctrl.html. [Online; accessed 07-August-2021].

[34] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator *(IMC '15)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/2815675.2815692

[35] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/633025.633056

[36] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3445814.3446724

[37] A. Feldman and S. Muthukrishnan. 2000. Tradeoffs for packet classification. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, Vol. 3. 1193–1202 vol.3. https://doi.org/10.1109/INFCOM.2000.832493

[38] Linux Foundation. 2020. Vector Packet Processing (VPP) platform. https://wiki.fd.io/view/VPP. [Online; accessed 07-August-2021].

[39] Open Information Security Foundation. 2020. Suricata - Intrusion Detection System. https://suricata-ids.org/ [Online; accessed 07-August-2021].

[40] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 465–478. https://doi.org/10.1145/1542476.1542528

[41] GNU Project. [n.d.]. GNU Compiler Collection. https://gcc.gnu.org/. [Online; accessed 07-August-2021].

[42] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

[43] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav Dave, Andrew W. Moore, and Peter G. Neumann. 2015. Blueswitch: enabling provably consistent configuration of network switches. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 17–27. https://doi.org/10.1109/ANCS.2015.7110117

[44] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. (2015).

[45] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[46] Christian Hopps. 2019. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[47] Google Inc. 2019. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io. [Online; accessed 07-August-2021].

[48] Google Inc. 2019. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller. [Online; accessed 07-August-2021].

[49] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. USENIX Association, 517–530.

[50] Joab Jackson. 2019. Kubernetes Long Road to Dual IPv4/IPv6 Support. https://thenewstack.io/it-takes-a-community-kubernetes-long-road-to-dual-ipv4-ipv6-support. [Online; accessed 07-August-2021].

[51] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-Directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

[52] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://dl.acm.org/doi/10.5555/3307441.3307457

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

[54] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://dl.acm.org/doi/10.5555/2228298.2228311

[55] James Kempf, Bengt Johansson, Sten Pettersson, Harald Luning, and Tord Nilsson. 2012. Moving the Mobile Evolved Packet Core to the Cloud. In *Proceedings of the 2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob) (WIMOB '12)*. IEEE Computer Society, USA, 784–791. https://doi.org/10.1109/WiMOB.2012.6379165

[56] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 163–181. https://www.usenix.org/conference/osdi21/presentation/khan

[57] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208. https://doi.org/10.1109/ICDE.2018.00027

[58] Mirja Kuehlewind and Brian Trammell. 2021. *Manageability of the QUIC Transport Protocol.* Internet-Draft draft-ietf-quic-manageability-09.

[59] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

[60] John Levon and Philippe Elie. 2004. Oprofile: A system profiler for linux. https://oprofile.sourceforge.io/news/. [Online; accessed 07-August-2021].

[61] Leonardo Linguaglossa, Stanislav Lange, Salvatore Pontarelli, Gábor Rétvári, Dario Rossi, Thomas Zinner, Roberto Bifulco, Michael Jarschel, and Giuseppe Bianchi. 2019. Survey of Performance Acceleration Techniques for Network Function Virtualization. *Proc. IEEE* 107, 4 (2019), 746–764. https://doi.org/10.1109/JPROC.2019.2896848

[62] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. 2018. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 504–517. https://doi.org/10.1145/3230543.3230563

[63] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/3387514.3405868

[64] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 459–473. https://dl.acm.org/doi/10.5555/2616448.2616491

[65] Sebastiano Miano. 2021. *Morpheus: Domain Specific Run Time Optimization for Software Data Planes - Artifact for ASPLOS'22.* https://doi.org/10.5281/zenodo.5830832

[66] Sebastiano Miano. 2021. Polycube (Morpheus branch). https://github.com/Morpheus-compiler/polycube/tree/morpheus.

[67] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a Faster and Scalable Iptables. *SIGCOMM Comput. Commun. Rev.* 49, 3 (Nov. 2019), 2–17. https://doi.org/10.1145/3371927.3371929

[68] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, J. Pi, and A. Shaikh. 2019. A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 1–9. https://doi.org/10.1109/ANCS.2019.8901880

[69] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 1–8. https://doi.org/10.1109/HPSR.2018.8850758

[70] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu. 2021. A Framework for eBPF-based Network Functions in an Era of Microservices. *IEEE Transactions on Network and Service Management* (2021), 1–1. https://doi.org/10.1109/TNSM.2021.3055676

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

[72] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-Based Pruning for Speeding up Superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 177 (Nov. 2020), 24 pages. https://doi.org/10.1145/3428245

[73] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 125–139. https://dl.acm.org/doi/10.5555/3307441.3307453

[74] OpenStack Authors. 2020. OpenStack. https://www.openstack.org/. [Online; accessed 07-August-2021].

[75] Oracle. 2021. Java HotSpot VM Options. https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html. [Online; accessed 12-August-2021].

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

[77] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14. https://dl.acm.org/doi/10.5555/3314872.3314876

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

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

[80] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for

Computing Machinery, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

[81] Felix Rath, Johannes Krude, Jan Rüth, Daniel Schemmel, Oliver Hohlfeld, Jó Á. Bitsch, and Klaus Wehrle. 2017. SymPerf: Predicting Network Function Performance. In *Proceedings of the SIGCOMM Posters and Demos* (Los Angeles, CA, USA) *(SIGCOMM Posters and Demos '17)*. Association for Computing Machinery, New York, NY, USA, 34–36. https://doi.org/10.1145/3123878.3131977

[82] Gábor Rétvári, László Molnár, Gábor Enyedi, and Gergely Pongrácz. 2017. Dynamic Compilation and Optimization of Packet Processing Programs. *ACM SIGCOMM NetPL* (2017).

[83] Jonathan Ribas. 2019. DPDK burst replay tool. https://github.com/FraudBuster/dpdk-burst-replay. [Online; accessed 07-August-2021].

[84] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 9. https://dl.acm.org/doi/10.5555/2342821.2342830

[85] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. https://arxiv.org/abs/1711.04422

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

[87] Muhammad Shahbaz and Nick Feamster. 2015. The Case for an Intermediate Representation for Programmable Data Planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2774993.2775000 https://doi.org/10.1145/2774993.2775000.

[88] Muhammad Shahbaz and Nick Feamster. 2015. The Case for an Intermediate Representation for Programmable Data Planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (Santa Clara, California) *(SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2774993.2775000

[89] Sourcefire. 2020. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/. [Online; accessed 07-August-2021].

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

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

[92] David E Taylor and Jonathan S Turner. [n.d.]. *Classbench Filter Set & Trace Generator*. https://www.arl.wustl.edu/classbench/

[93] David E Taylor and Jonathan S Turner. 2007. Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking* 15, 3 (2007), 499–511. https://doi.org/10.1109/TNET.2007.893156

[94] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P2GO: P4 Profile-Guided Optimizations. In *Hot Topics in Networks (HotNets)*. ACM.

[95] Jonathan Worthington. 2018. Eliminating unrequired guards. https://6guts.wordpress.com/2018/09/29/eliminating-unrequired-guards/. [Online; accessed 12-August-2021].

[96] David Wragg. 2020. Unimog - Cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/

[97] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. 2018. Leveraging EBPF for Programmable Network Functions with IPv6 Segment Routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/3281411.3281426

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

[99] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 112–125. https://doi.org/10.1109/ISCA52012.2021.00018

[100] Rui Zhang, Saumya Debray, and Richard T. Snodgrass. 2012. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 63–73. https://doi.org/10.1145/2259016.2259025