Dataplane Specialization for High-performance OpenFlow Software Switching

László Molnár*, Gergely Pongrácz*, Gábor Enyedi*, Zoltán Lajos Kis*,
Levente Csikor†, Ferenc Juhász*,†, Attila Kőrösi‡, Gábor Rétvári†,‡
*TrafficLab, Ericsson Research
†Department of Telecommunications and Media Informatics, BME
‡MTA-BME Information Systems Research Group

ABSTRACT

OpenFlow is an amazingly expressive dataplane programming language, but this expressiveness comes at a severe performance price as switches must do excessive packet classification in the fast path. The prevalent OpenFlow software switch architecture is therefore built on flow caching, but this imposes intricate limitations on the workloads that can be supported efficiently and may even open the door to malicious cache overflow attacks. In this paper we argue that instead of enforcing the same universal flow cache semantics to all OpenFlow applications and optimize for the common case, a switch should rather automatically specialize its dataplane piecemeal with respect to the configured workload. We introduce ESWITCH, a novel switch architecture that uses on-the-fly template-based code generation to compile any OpenFlow pipeline into efficient machine code, which can then be readily used as fast path. We present a proof-of-concept prototype and we demonstrate on illustrative use cases that ESWITCH yields a simpler architecture, superior packet processing speed, improved latency and CPU scalability, and predictable performance. Our prototype can easily scale beyond 100 Gbps on a single Intel blade even with complex OpenFlow pipelines.

CCS Concepts

•**Networks** → **Bridges and switches;** *Network performance modeling;*

Keywords

OpenFlow software switching, packet classification, template-based code generation

SIGCOMM '16, August 22-26, 2016, Florianopolis, Brazil

DOI: http://dx.doi.org/10.1145/2934872.2934887

1. INTRODUCTION

The OpenFlow switch is perhaps the most generic packet processing device ever conceived [1, 2]. Depending on the configuration from the control plane, an OpenFlow switch can identify flows based on a broad combination of layer-2, layer-3, and layer-4 (L2–L4) protocol attributes and apply essentially any meaningful packet processing action on the packets of these flows. This facilitates the rapid provisioning and central administration of highly reconfigurable network services on top of a potentially heterogeneous inventory of switches and thereby providing Infrastructure-as-a-Service capabilities [3, 4]; it unlocks networks for radical innovation, allowing to mix circuit and packet switched infrastructures [5], heterogeneous control paradigms [6], and legacy with clean-slate protocols [1, 7]; and it still retains the familiar network operations mental model in expressive high-level declarative policies [8–10]. Accordingly, OpenFlow has found its use in a wide range of application areas, from enterprise networks [11], data centers [12] and multi-tenant clouds [13], optical transport networks [14], software-defined Internet exchanges [15], WANs [6] and WAN gateways [16], all the way to telco load-balancers, stateless firewalls, and mobile access gateways [17, 18]. Crucially, the OpenFlow dataplane must support all these diverse applications with reasonable efficiency.

Unfortunately, this genericity conflicts with performance; to quote [19], "OpenFlow is expressive but troublesome to make fast on x86". Embedded intrinsically into the OpenFlow fast path is a series of costly packet classification steps, needed to associate packets with flows for applying corresponding actions, and, in spite of decades of continuous research and development [20–24], *OpenFlow software packet classification still remains too expensive* for today's line rates [19]. Reasons for this all trace back to the rich packet processing capabilities [25], the need to perform basically arbitrary wildcard matches on a broad selection of packet header fields (40+ as of OpenFlow 1.4, [2]), flow priorities imposing a firm ordering on classification that may break elementary networking conventions like IP longest prefix matching (LPM), or the fact that a pipeline may include dozens of stages and the corresponding classifications must be performed successively, each stage using the results from the

previous stages. This rich semantics is already hard to support in hardware firewalls and intrusion-detection middleboxes, let alone in OpenFlow software switches [25]. With the rise of server virtualization in data center networking and network function virtualization, however, OpenFlow appliances increasingly run on a stock x86 platform that lacks the necessary hardware-based packet classification components [5, 6, 11–13, 15–18].

Consequently, most OpenFlow softswitch implementations recur to excessive flow caching at the fast path, in order to amortize the computational costs of packet classification over the packets of flows [26–28]. It is well-known, however, that flow caching performs poorly for many important applications that require forwarding decisions depending on diverse "high-entropy" packet fields, like transport-layer firewalls, or produce short-lived flows, e.g., peer-to-peer protocols, MapReduce, or network monitoring [19, 29]. Consequently, OpenFlow switches often exhibit abrupt performance regressions in various hard-to-predict combinations of flow tables and traffic patterns [29–34], opening the door to malicious denial-of-service-like attacks even on as innocently looking traffic patterns as port scans [19, 29, 35].

In this paper, we argue that these adverse phenomena stem from the fact that *flow caching over-generalizes*: by enforcing the same universal flow cache semantics to fundamentally diverse use cases it optimizes for the lowest common denominator. For instance, the prevalent OpenFlow software switch implementation, Open vSwitch (OVS), uses a hash-based wildcard match store as (one of the hierarchy levels of) its flow cache, which works fairly well for simple OpenFlow pipelines but inherently breaks down for large-scale IP routing (that would rather require LPM, [30]) or flow tables that heavily match across layer boundaries [19, 29].

Instead of relying on an overly general-purpose fast path, we argue, *an OpenFlow switch should rather automagically specialize itself for the actual workload*, into an optimal exact matching switch when the flow tables specify pure L2 MAC forwarding [36], an LPM engine for L3 routing [37, 38], or a fast, optimized packet classifier for L4 ACLs [20–24], and a reasonable combination of these building blocks whenever the OpenFlow pipeline matches heterogeneous protocol header fields.

We present ESWITCH, a new OpenFlow switch framework that radically breaks with general-purpose datapaths and embraces a fully customized dataplane. We view OpenFlow as a declarative language to program the dataplane [8, 9] and we cast ESWITCH as *a compiler that transforms a declarative pipeline specification into efficient machine code*. Underlying ESWITCH is the observation that, similarly to many natural programming languages [39], most real-world OpenFlow applications compose the same small set of simple forwarding behaviors, or patterns, and therefore OpenFlow pipelines lend themselves readily to be rewritten in terms of a small set of predefined static templates. ESWITCH, accordingly, uses dynamic template-based code generation to emit optimized OpenFlow fast paths that sidestep flow caching altogether. Thanks to the template abstraction we can even construct meaningful performance models for the generated code and reason about the fast path in simple quantitative terms.

We implemented ESWITCH on top of the Intel DataPlane Development Kit (DPDK, [40]). The dataplane is written entirely in assembly, hand-optimized to the x86 and ARM platforms. We present extensive measurements to show that, compared to OVS, ESWITCH *features predictable and superior performance, latency and multi-core scalability* with up to two orders of magnitude improvement on complex OpenFlow pipelines, *while supporting similar update intensities*. Our rudimentary prototype easily scales beyond 100 Gbps transmission speeds, downright beating many contemporary hardware OpenFlow switches by a large margin [41].

The rest of the paper is structured as follows. In Section 2 we discuss OpenFlow switch architectures and present our critiques for flow caching. In Section 3 we introduce ESWITCH and in Section 4 we demonstrate pipeline compilation on some common use cases, we analyze performance related aspects, and we sketch simple analytic switch models. Finally, we review related work in Section 5 and we conclude the paper in Section 6.

2. THE OPENFLOW PIPELINE

The crux of the OpenFlow dataplane is the *pipeline*, an abstract description of the forwarding functionality programmed into a switch. The pipeline is a linked hierarchy of *flow tables*, each flow table specifying a logically distinct stage of packet forwarding in the form of a list of *flow entries*. A flow entry in turn consists of a *rule* to be matched on packet header fields, *counters* for maintaining statistics, and *actions* to be applied to a packet whenever a match is found. Rules designate flows and actions establish pipeline processing for these flows, by triggering forwarding on a particular port, updating packet contents, sending to a next stage flow table for further processing, etc. Flow entries are managed by the controller via a dedicated OpenFlow channel, either *reactively* (online, in response to received packets) or *proactively* (offline, e.g., after a topology change).

Packet processing starts at the first flow table ("Table 0"), trying to match the header field tuple against the first flow entry and then, should this fail, against successive flow entries in decreasing order of priority[1]. Processing terminates when the matching flow entry does not specify a next table to be visited (using a `goto_table` instruction), at which point the actions associated with the packet are executed. Unmatched packets cause a table miss and, depending on switch configuration, can be dropped or sent to the controller for further consideration.

Fig. 1a gives the pipeline for a simple firewall, arbitrating packets between an Internet-facing `external` port and an `internal` port connected to a web server hosted at the IP address `192.0.2.1`. The pipeline contains a single flow table; the first flow entry requires that packets received at the `internal` port are forwarded to the `external` port unconditionally, while in the reverse direction only HTTP

[1]When not stated otherwise, we use the convention that flow entries are listed in decreasing order of priority.

Table:0

in_port	ip_src	ip_dst	tcp_src	tcp_dst	action
internal	*	*	*	*	output:external
external	*	192.0.2.1	*	80	output:internal
external	*	*	*	*	drop

(a) sample flow table

Table:0

in_port	action
internal	output:external
external	goto_table:1

Table:1

ip_dst	tcp_dst	action
192.0.2.1	80	output:internal
*	*	drop

(b) equivalent pipeline

Figure 1: A simple firewall: (a) single-stage pipeline and (b) an equivalent multi-stage pipeline. Flow entries are listed in decreasing order of priority and priorities are not marked explicitly. Note that in (b) we omitted irrelevant match fields (`ip_src` and `tcp_src`).

packets (`tcp_dst=80`) are admitted and the rest of the traffic is dropped.

Industry best practices recommend to split the pipeline into multiple consecutive stages, to modularize pipeline design by decoupling packet processing in different protocol layers and to obtain simpler representations for complex forwarding semantics and sidestep cross-product flow-state explosion effects [2, 17, 18]. For example, the VMware NVP network virtualization controller sets up more than a dozen stages in its packet processing pipeline [13]. For our sample firewall, an equivalent multi-stage OpenFlow pipeline is specified in Fig. 1b. Here the first stage flow table forwards between switch ports, directing external packets to a second stage flow table that filters web traffic.

2.1 OpenFlow Software Datapaths

The art and science of OpenFlow switch architectures revolve around the organization of functionality inside the software pipeline implementation (the *datapath*) in order to permit processing flow entries and applying actions as fast as possible, without sacrificing OpenFlow's inherent expressiveness. Existing implementations fall into the below two coarsely defined categories.

Direct datapath. A direct datapath performs packet classification right on the flow tables. A simple implementation strategy would be to organize flow entries into a linked list according to the order imposed by priorities and iterating over this list priority-wise, possibly jumping to another table whenever a match is found and starting linear iteration anew. Thusly, a direct datapath in the worst case loops through all flow entries in all flow tables until it finally finds a matching flow or can signal a table miss. Correspondingly, direct datapaths are generally considered an inferior implementation strategy but, thanks to their simplicity, they still find important use in reference designs and experimental implementations (OpenFlow Reference Switch [42], CPqD [43], xDPd [44], LINC [45]), or as a last resort for complex pipelines to which no fast specialized classification strategies apply [19].

Indirect datapath. Indirect datapath designs adopt the venerable fast-path/slow-path separation principle [46]. Here, the switch maintains multiple "views" of the pipeline: the fast path is constituted by one or more increasingly compact and incomplete pipeline representations, while a complete representation serves as the slow path, used as a fallback when the fast path cannot decide on the fate of a packet. This way, only the first packet of a flow is subjected to full-scale slow path pipeline processing, the resulting flow-specific rules and actions are registered in the fast path classifier and the rest of the flow's packets take the fast path without having to recurse to the slow path again. In fact, the fast path functions as a *flow cache*, storing the forwarding decisions for recent flows. Aggressive flow caching, consequently, allows to amortize the cost of packet classification over subsequent packets of a flow [26–28], contributing to increased performance without loss of expressiveness and genericity (see OVS [19], 6WINDGate [47]).

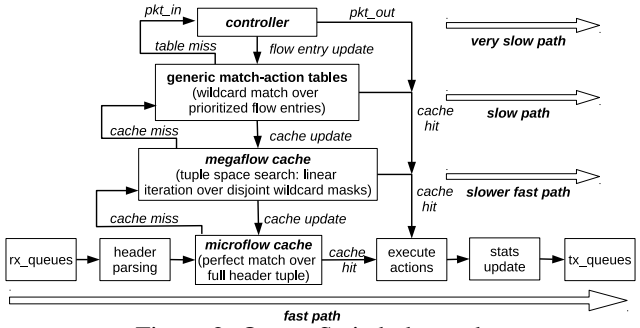


Figure 2: Open vSwitch datapaths.

2.2 A Sophisticated Indirect Pipeline: OVS

The most popular OpenFlow software switch implementation today is undoubtedly Open vSwitch (OVS, [19]). In this paper, we use OVS to showcase the pros and cons of the indirect datapath approach. Indeed, OVS brings this design to the extreme by adapting a datapath hierarchy of as many as four levels (see Fig. 2).

Microflow cache: per-transport-connection exact match store. The microflow cache stores the forwarding decisions for the least recently seen transport connections in a very fast collision-free hash. Since exact matching occurs over all relevant tuple fields, essentially *any* change in the packet header inside an established flow (e.g., the IP TTL field) results in a cache miss. The microflow cache is managed by the second-level cache, the megaflow cache (see below), in that the microflow cache indexes into the megaflow cache and megaflow cache hits trigger a microflow cache update.

Megaflow cache: wildcard match store for traffic aggregates. The second-level megaflow cache allows to bundle multiple microflows into a single megaflow aggregate and impose common forwarding behavior to the entire bundle, saving on cache entries (e.g., by applying the same action to all incoming HTTP connections, regardless of the source TCP port) at the cost of increased processing time. The megaflow cache uses a tuple space search strategy [23]: flow entries are divided into groups based on the combination of header fields they match on and every group matches on only the fields relevant for the group. In practice, this entails linearly iterating over a list of key/mask pairs for each

packet. The megaflow cache is managed by the third datapath level (vswitchd, see next) *reactively*: packets missing the cache are encapsulated and sent to vswitchd that returns the packet with the actions to be applied and updates the cache accordingly. Since the megaflow cache does not "know" about flow priorities, matches can never overlap and so megaflows must be disjoint. Consequently, *all header fields from all flow entries a packet traverses*, those that caused a match as well as those higher priority ones that did not, need to be taken into consideration in tuple space search. Note also that OVS uses a single megaflow cache for the entire pipeline, hence cache entries collapse together the behavior from all relevant flow tables.

vswitchd: complete OpenFlow pipeline. The penultimate level is a fully blown realization of the OpenFlow pipeline. Besides the use of standard techniques like multithreading, read-copy-update (RCU), and extensive batching, OVS adopts a number of clever tricks to improve tuple space search and megaflow cache management, like tuple priority sorting (to cut down on pipeline stage iterations), staged lookup (per-protocol-layer caches for opportunistic early exit), prefix tracking (optimal prefix masks for IP address range tracking), and classifier partitioning (by metadata fields).

OpenFlow controller. Somewhat unconventionally, we consider the controller as the highest level of the OVS datapath hierarchy as it fulfills precisely the same role for vswitchd as vswitchd does for the megaflow cache: it manages entries at the next lower level of the datapath hierarchy plus serves as a last resort for packets missing that level.

2.3 The Case Against Flow Caching

Indirect pipelines proved a remarkably successful implementation strategy to break down the complexity of OpenFlow software packet processing. Yet, we argue that inherent to general purpose flow caches are a number of under-the-hood limitations, which not only significantly curtail achievable switch performance but also raise a number of deep architectural concerns.

Unpredictable packet processing. Flow caches work best when traffic exhibits sufficient spatial locality (header fields vary in a small range and thus only a few megaflows are enough to cover all traffic) and temporal locality (flows' packets are finely spaced in time to keep cache entries warm) and/or when the pipeline omits high-entropy header fields [26–28]. But only a single fine-grained rule is enough to "punch a hole" in all aggregates, leading to heavy megaflow fragmentation [13, 19]. We found that even the same packets and the same pipeline can yield vastly different flow caches depending on the packet arrival sequence (see Fig. 3). Correspondingly, in a flow-cache-oriented architecture it is very difficult to reliably predict when and how a particular packet will traverse the switch dataplane.

Performance artifacts. As long as flow caching assumptions hold, indirect datapaths work reliably and efficiently. Failing these, however, brings undue cache thrashing and packets recurring to the slow path. This is then perceived by the user as perplexing throughput drops, latency spikes, and downright service interruptions, on various hard-to-predict combinations of flow tables and input traffic [29–34].

dl_type	tcp_dst	action
0x0800	255	output:port
*	*	drop

(a) flow table

#	decimal	binary
1	190	10111110
2	189	10111101
3	187	10111011
4	183	10110111
5	175	10101111
6	159	10011111
7	191	10111111

(b) pkt port seq 1

#	decimal	binary
1	191	10111111
2	190	10111110
3	189	10111101
4	187	10111011
5	183	10110111
6	175	10101111
7	159	10011111

(c) pkt port seq 2

Figure 3: The flow table (a) yields 7 megaflow cache entries when the TCP destination port arrivals are as of seq 1 (b) (for each zero bit in positions 2, . . ., 8), while if destination port 191 arrives first as of seq 2 (c) then only a single entry arises (matching at position 2, covering all subsequent packets).

Vexing cache management complexity. The flip side of flow caching is the complexity of managing the cache. First, computing "good" megaflows is already a very hard problem [29, 33] (cf. Fig. 3). But updating cache contents when changes to the switch configuration are made is also rather onerous, due to the difficulty of tracking exactly which cache entries are affected by a change and need invalidation[2]. Ensuring cache coherence across switch threads, furthermore, necessitates fine-grained locking, impeding multi-core scalability. This complexity can also make dataplane debugging cumbersome and conceal software bugs and security flaws.

Opens the door to malicious attacks. It is well-known that caches are inherently vulnerable to a wide spectrum of security threats, like cache poisoning or cache overflow [35], and may leak information through side-channel attacks [48]. An attacker may easily spawn a denial-of-service attack, by executing timing attacks on an OpenFlow switch to infer flow table contents and crafting a malicious packet trace to overflow flow caches. This is especially troublesome for cloud infrastructure switches because only a single misbehaving user can produce a widespread service disruption, by exploiting that shared flow caches break tenant isolation and create a coupling between logical pipelines (see Section 4.3).

Brittle architectural constraints. Flow caches bring along some insidious but inevitable architectural choices. For instance, OVS must recreate essentially the entire functionality of the OpenFlow protocol at the vswitchd–megaflow-cache interface, complete with flow entry management and packet in/out operations, and it does this in a rather inefficient *reactive* way, forcing packets to the slow path in order to populate the caches. But flow caching may also hinder dataplane innovation, since cache semantics must be fit piecemeal to new proposals like NOSIX [34] or P4 [49].

3. DATAPLANE SPECIALIZATION

ESWITCH is a new OpenFlow switch architecture we created with the aim to discover the design space beyond general purpose switch fast paths. ESWITCH occupies just the opposite extreme of the spectrum: it fully embraces the con-

[2]OVS adopts the brute-force strategy to invalidate the entire cache after essentially all changes [19].

cept of dataplane specialization and makes the datapath dependent on, and actually carefully tailored to, the configured OpenFlow pipeline.

We cast dataplane specialization as the process to compile from a declarative description of the OpenFlow pipeline into an efficient machine code representation, together with a runtime that effectively realizes the switch using the compiled datapath as the fast path. ESWITCH builds on the insight that OpenFlow pipelines usually combine only a small selection of simple but generic patterns into complex dataplane programs and, accordingly, they can be represented in terms of a few simple *templates*. ESWITCH then uses template-based code generation to derive the customized datapath: it first applies *flow-table analysis* to decompose a pipeline into a series of templates, followed by *template specialization* to patch pre-compiled templates with flow keys, and finally a *linking* phase to relocate jump pointers, combining template code fragments into a single binary.

3.1 Templates

ESWITCH pipeline compilation revolves around the concept of templates. A template in this context is some unit of common OpenFlow packet processing behavior that admits a simple and composable machine code implementation out of which more complex functionality can be constructed. ESWITCH differentiates between packet parser templates, matcher templates, flow table templates, and action templates.

Packet parser templates are used, as the name suggests, to generate code that transforms packet headers into an internal representation that can be subjected to rule matching. ESWITCH separates header parsing at layer boundaries: it includes a separate L2, L3, and L4 parser. The motivation is to save on parsing for layers that do not participate in flow formation: e.g., for pure L2 MAC forwarding it is completely superfluous to parse L3 and L4 header fields for each packet, L3 routing in turn can omit parsing L4 headers altogether, etc. Note that parsing is incremental; the L3 header parser composes an L2 parser to find the starting position of the L3 header and the L4 parser composes both parsers.

The general functionality of parser templates is as below; a *protocol bitmask* (stored in register `r15`) is used to mark the presence of a particular protocol header in the packet and then each protocol's header position is saved into a register.

```
PROTOCOL_PARSER: <set protocol bitmask in r15>
L2_PARSER:  mov r12, <pointer to L2 header>
L3_PARSER:  mov r13, <pointer to L3 header>
L4_PARSER:  mov r14, <pointer to L4 header>
```

Matcher templates allow matching on header fields; a separate template belongs to every field defined in the OpenFlow specification [2]. For instance, the rule `ip_dst = ADDR/MASK` would be represented with the below template:

```
macro IP_DST_ADDR_MATCHER(ADDR,MASK):
  mov  eax,[r13+0x10]  ; IP dst address in eax
  xor  eax,ADDR        ; match ADDR
  and  eax,MASK        ; apply MASK
  jne  ADDR_NEXT_FLOW  ; no match: next entry
```

The rule `tcp_dst=PORT` is matched as follows:

```
macro TCP_DST_PORT_MATCHER(PORT):
  cmp  [r14+0x2],PORT  ; port field equals key?
  jne  ADDR_NEXT_FLOW  ; no match: next entry
```

Note that actual flow keys will be patched into the templates in the template specialization step, while jump pointers will be resolved during linking (see Section 3.3).

Flow table templates capture common flow table patterns. Our design has settled with four elemental table templates: the direct code, the compound hash, the LPM, and the linked list templates (see Fig. 4). Further table templates, like range search for port matches, can easily be added in the future.

The *direct code* template is a faithful machine code representation of the classification rules in a flow table. ESWITCH uses this template when the flow table does not contain enough entries to justify a complex data structure (like a hash or a trie). The maximum number of flow entries under which a table is directly compiled is controlled by a parameter; we will present CPU-level measurements to fine-tune this parameter later in Section 4.3.

Consider an example with the below flow entries:

```
ip_dst=ADDR_1/MASK_1,tcp_dst=PORT,action=ACTION_1
ip_dst=ADDR_2/MASK_2,action=ACTION_2
...
```

ESWITCH in this case takes the `ip_dst` and `tcp_dst` matcher templates and concatenates these into the direct code template below:

```
FLOW_1:                  ; first flow entry
  mov  eax,IP | TCP      ; ip and tcp?
  or   eax,r15d          ; check protocol bitmask
  cmp  eax,r15d          ;
  jne  ADDR_NEXT_FLOW    ; not an IP & TCP packet
                         ; jump to next flow entry
  IP_DST_ADDR_MATCHER(ADDR_1,MASK_1)
                         ; ip_dst=ADDR_1/MASK_1?
  TCP_DST_PORT_MATCHER(PORT)
                         ; tcp_dst=PORT?
  jmp ACTION_1           ; action=ACTION_1
FLOW_2:                  ; second flow entry
  bt   r15d, IP          ; ip?
  jae  ADDR_NEXT_FLOW    ; jump to next flow entry
  IP_DST_MATCHER(ADDR_2,MASK_2)
                         ; ip_dst=ADDR_2/MASK_2?
  jmp ACTION_2           ; action=ACTION_2
FLOW_3: ...
```

For larger flow tables, the *compound hash* template might be the fastest choice. This template is used for tables comprising one or more fields whereas every field is matched by exactly the same mask in each entry. For instance, in the below example the first two entries would lend themselves to the compound hash template (as `ip_dst` is masked with `/24` in both entries and `tcp_dst` is unmasked) while adding the third entry would violate the prerequisite (as `tcp_dst` is now a wildcard):

```
ip_dst=192.0.2.0/24,tcp_dst=80,action=ACTION_1
ip_dst=198.51.100.0/24,tcp_dst=21,action=ACTION_2
ip_dst=203.0.113.0/24,action=ACTION_3
```

The template works as follows: first, the code runs together relevant header fields into a single key, applies the global mask, and then looks up the key in a hash. Our implementation uses a collision free hash; even though it requires

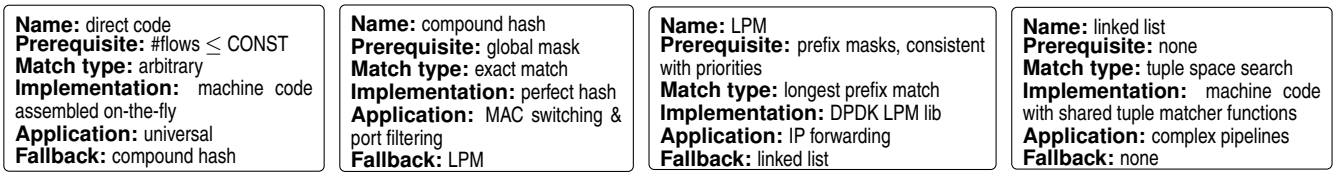


Figure 4: Flow table templates in ESWITCH.

more memory and more time to build, it supports fast constant time lookups, a key to a robust datapath performance.

Whenever a flow table does not fulfill the prerequisite for compound hashes, ESWITCH falls back to an *LPM template*. The LPM template applies to single-field tables that contain prefix rules: each mask is such that it wildcards the last consecutive bits of the field and whenever rules overlap the more specific one has higher priority. For instance, the below example would violate the latter principle:

```
priority=100,ip_dst=192.0.2.0/24,action=ACTION_1
priority=20,ip_dst=192.0.2.12/30,action=ACTION_2
```

Our prototype uses the Intel DPDK built-in `rte_lpm` library for implementing the LPM template.

As a final fallback, ESWITCH includes a *linked list* table template for doing tuple space search [23]: for every relevant combination of fields a separate matcher function is constructed (out of the matcher templates introduced above) and these matchers are called iteratively with subsequent flow entry keys as input. Since this template is rarely seen in practice, we omit the details for brevity.

Action templates, finally, are used to construct packet processing functionality. In this regard, ESWITCH takes a refreshingly simple approach: every action type (e.g., output to a port, flood, or modify header field) is a separate action template and action templates are collapsed into composite action sets. Identical action sets are shared across flows.

3.2 Flow Table Analysis

Template-based code generation, understandably, relies on some mechanism to recognize the very templates in the input. This is the responsibility of the flow table analysis pass in ESWITCH pipeline compilation.

Recognizing packet parser, matcher, and action templates is fairly simple, but deciding on table templates is more involving. In ESWITCH this occurs incrementally during code generation: ESWITCH always attempts to compile into the most efficient table template available; whenever it detects that the prerequisite no longer applies it gradually falls back to the next most efficient representation and rebuilds the table with the new template (see Fig. 4 for template fallbacks).

In order to avoid that complex OpenFlow pipelines end up with inefficient templates, ESWITCH actively tries to transform tables that may not fit well with our templates into ones that do, thereby promoting such "difficult" flow tables towards faster templates. In the firewall example, for instance, the single-stage pipeline (Fig. 1a) would fit only the slow linked list template (disregarding the direct code template for a moment), while the multi-stage pipeline (Fig. 1b) would admit a series of two hash-templates, which is much faster.

Pipeline transformation is done in the *flow table decomposition* pass. For brevity, herein we limit ourselves to a heavily simplified exposition, which we believe is still sufficient to demonstrate the main points. In this setting, we are given a flow table T with m flow entries defined on n fields:

$$T = \{(F_{i,1}, F_{i,2}, \ldots, F_{i,n}) \rightarrow a_i : i \in [1, m]\} \ ,$$

in which rules $F_{i,j}$ can be of only two types: either an exact-match or a full wildcard (arbitrary masks are disallowed for now). Further suppose that we have only a single table template available: a single-field exact-match (hash) template with a potential final catch-all rule. Our task is to transform T into a semantically equivalent [24, 37] pipeline comprising the minimal possible number of flow tables, each one compliant with the template. An example is given in Fig. 5.

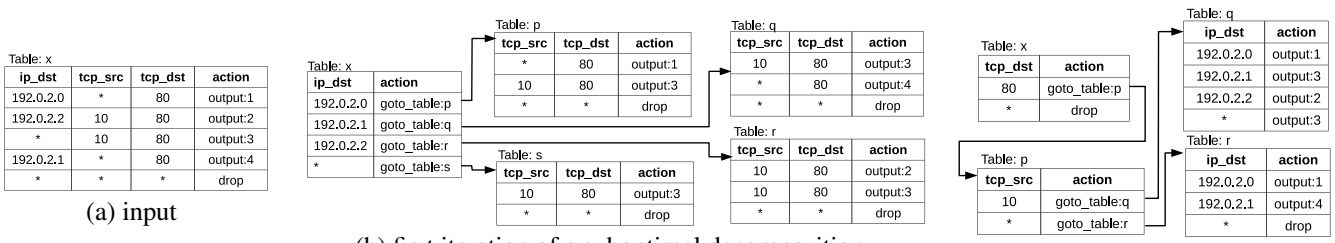
In the Appendix, we show that this simplified problem is already intractable. Correspondingly, our flow table decomposition algorithms rely on heuristics, focusing on speed instead of efficiency. The main iterative step is DECOMPOSE(τ) given in Fig. 6; this routine is called first on T and then recursively on all the tables produced.

The key to the algorithm is step (4), which decomposes a table T along one of its columns, in this case p, into a new table that replaces T and matches only on p, plus a set of further tables T_f for each separate key f in column p of T. Note that the new table for T is already compliant with the exact-match template and the rest of the tables will also become compliant after the recursion terminates. The procedure processes the entries in T one by one and takes the p-th column: for each non-wildcard key f it merely just strips column p from the rule and appends it to the table T_f, while rules with a wildcard will go (stripped) to *all* tables.

One easily sees that the procedure terminates with a semantically equivalent representation, with as many new tables as there are different keys in column p. On that ground, *decomposing along the column of minimal diversity* gives a heuristic algorithm that greedily minimizes the number or tables produced (as of step (1) and (2) in Fig. 6).

An example is shown in Fig. 5: given the table in Fig. 5a, decomposition along column `ip_dst` with 3 distinct keys (plus the wildcard) would yield the tables at Fig. 5b at the first iteration and eventually terminate with 9 tables, while if we chose column `tcp_dst` first (of diversity 2) we would end up with only 4 tables (Fig. 5c). ESWITCH will then automatically substitute the initial table, and the ensuing inefficient linked list template, with the decomposed pipeline and hence promote it to a sequence of fast hash templates.

Astute readers will recognize here a decision-tree-based packet classifier scheme, each node of which is a separate



(a) input

(b) first iteration of a suboptimal decomposition

(c) optimal decomposition

Figure 5: Flow table decomposition: (a) a sample flow table, (b) first iteration of the heuristics after (suboptimally) decomposing along field `ip_dst`, and (c) an optimal decomposition containing only 4 tables. Observe that each table in the optimal decomposition is consistent with the exact-match template. Table ids will be resolved after the algorithm terminates.

Procedure DECOMPOSE(T):

1. Find distinct keys in each column: $S_j = \bigcup_{i=1}^{m} F_{i,j}, \forall j \in [1, n]$
2. Find column of minimal diversity: $p = \underset{j \in \{1,\ldots,n\}}{\operatorname{argmin}} |S_j|$
3. Initialize an empty table T_f for each field value $f \in S_p$
4. Decompose table T along column p, suppose $n > 1$

for $i \in 1, \ldots, m$ **do**
  if $F_{i,p} = *$ **then**
    for $f \in S_p$ in descending order of priority **do**
      $T_f.add((F_{i,1}, ..., F_{i,p-1}, F_{i,p+1}, ..., F_{i,n}) \rightarrow a_i)$
  else
    $T_{F_{i,p}}.add((F_{i,1}, ..., F_{i,p-1}, F_{i,p+1}, ..., F_{i,n}) \rightarrow a_i)$
$T.reset$
for $f \in S_p : S_p = \{f_1, f_2, \ldots, *\}$ **do**
  $T.append(f \rightarrow$ goto_table $T_f)$
for $f \in S_p$: DECOMPOSE(T_f)

Figure 6: Table decomposition routine.

flow table, organized similarly to the the set-pruning trie data structure [20] and HyperCuts [50] but doing matching fieldwise and with a greedily optimized matching order. This is then easy to extend to additional templates and arbitrary, possibly overlapping, masked keys; we omit the details.

In line with the fundamental (rather prohibitive) lower bounds for packet classification [22], for very complex flow tables our decomposer cannot help but output an immense number of tables. However, the depth of the hierarchy, and thus the time it takes to send a packet through the pipeline, is constrained by the number of header fields in the input table only, which is usually not too large. Note that we are not restricted by OpenFlow's limit on maximum flow table number (255) here, since decomposition is internal to ESWITCH. Further note that decomposition does not necessarily occur over layer boundaries (even though in many practical cases it just happens to be the most efficient decomposition strategy); later we shall show an example in Fig. 7.

We evaluated the algorithm on a handful of real pipelines, collected from a production multi-tenant OpenStack cloud and a telco's Border Network Gateway. Strikingly, in essentially all cases *our decomposer simply returned its input intact*, indicating that the *pipeline had already been decomposed optimally*. In fact, real-world controllers often emit optimally decomposed pipelines out of the box for reasons we alluded to in earlier sections: modularization, layer-based processing, and avoidance of cross-product effects [2, 17, 18]. Correspondingly, we see table decomposition as an optional feature for ESWITCH, which can be freely disabled for most "well-behaved" control programs.

To still stress the algorithm to its limits, we fed it with a complete firewall setup, consisting of arbitrarily wildcarded five-tuple ACLs ("snort community rules v2.9", stripped to OpenFlow compatible rules): with the active 72 rules we obtained only 50 separate tables in the decomposition, while adding obsolete rules resulted in 197 tables on an input of 369 ACLs. This shows that, thanks to table decomposition, ESWITCH can efficiently implement complex firewall and intrusion detection functionality entirely in OpenFlow, without having to recur to middleboxes.

3.3 Template Specialization & Linking

ESWITCH keeps the templates as a library of precompiled object code fragments to avoid online assembling/compiling. After the flow table analysis pass, it builds the skeleton of the compiled datapath by simply hoarding all the necessary template objects into a single binary. At this point the binary still contains placeholders for the flow keys, which will be patched into the code in the *template specialization* pass. Note that this step is not necessarily inevitable; the code could as well include pointers to the memory locations of the flow keys instead of the keys themselves. We still decided to patch keys right into the code; we found that standard OpenFlow datapath processing burdens the CPU data caches extensively, but compiling match keys right *into* the code directs some of this load to the CPU instruction caches, which gives greater locality, better distribution of CPU cache load, and hence faster processing (see Section 4.3).

The code still contains many dangling jump pointers, like `ADDR_NEXT_FLOW`. These are resolved in the final *linking* pass; jump pointers are again statically compiled into the code, with the exception of `goto_table` action jumps which go via a trampoline. The reason for this indirection is to improve the granularity of datapath updates.

3.4 Updates

For templates that support incremental updates (compound hash, LPM, and linked list), ESWITCH performs updates non-destructively: whenever the controller modifies a flow ESWITCH simply updates the data structure underlying the template. Complete rebuilding happens only for the direct

code template (unconditionally), for the hash template to minimize hash collisions (periodically), or when the modification would violate the prerequisite for the current template and a fallback must be constructed. But even then ESWITCH can continue processing packets during the rebuild, as the new template representation is constructed side by side with the running datapath and then inserted into the pipeline by atomically redirecting all referring `goto_table` jumps to the address of the new code via the trampoline.

This update mechanism gives two important benefits. First, datapath updates in ESWITCH are *transactional*, with partial updates automatically rolled back, which eliminates inconsistent behavior common to many OpenFlow switches [51]. Second, updates are of *per-flow-table* granularity, which is certainly an improvement over OVS that needs to invalidate the entire megaflow cache (shared across the pipeline) on any update and re-populate it, again reactively, from scratch.

4. EVALUATION

Next we turn to discuss the practical aspects of ESWITCH and make our case for dataplane specialization. We take four illustrative use cases from operational OpenFlow deployments [17, 18, 30], which will serve for demonstrating the ESWITCH pipeline compilation process and also as basic scenarios for performance evaluations. First, we present the use cases themselves, then we give a detailed description of our ESWITCH prototype and present the measurement studies, and finally we derive a rudimentary performance model and we show that this simple model can already supply useful performance characterizations.

In what follows, we shall weigh ESWITCH against OVS, the flagship OpenFlow softswitch, and show major performance improvements. The subsequent discussion, however, is nowhere meant to be a critique on OVS *per se*; in fact, during working on this paper we have come to truly admire the amount of engineering that went into OVS datapaths; our goal is, accordingly, not to call out OVS but rather to point out the advantages of dataplane specialization over a flow-caching-centered architecture.

4.1 Use Cases

The first two use cases are *Layer-2 switching*, i.e., packet forwarding by exact-matching on a MAC table, and *Layer-3 routing*, i.e., longest-prefix-matching IP addresses from a routing table. These use cases model pure run-to-completion pipelines, whereas all processing is specified in a single flow table (see e.g., L2 in [12], L3 in [13]). Notably, ESWITCH attains optimal dataplane specialization in both cases: the L2 pipeline compiles into the hash table template, effectively reducing into a conventional Ethernet software switch, while the L3 pipeline is compiled into the LPM template yielding a datapath identical to that of an IP softrouter. For each use case we prepared a suite of flow tables with different number of entries in order to measure switches' robustness to pipeline complexity and we also generated a sequence of traffic mixes to test robustness against the number of active flows. The L2 flow tables contained random MAC addresses and the L2 destination addresses in the flow mix were adequately aligned to avoid frequent table misses; for the L3 use case routing tables were randomly sampled from a real Internet router and again the traces were adjusted accordingly.

The rest of the use cases, a load balancer and an access gateway, model multi-stage pipeline applications [17, 18].

The *load balancer* use case captures the functionality of a web frontend, which distributes HTTP traffic for different web services, available at different IP addresses, between backend servers. Load distribution happens based on the first bit of the source IP address in the incoming packets. In the ingress direction only web traffic is allowed, while traffic is forwarded unconditionally in the other direction (see Fig. 7a). A naive compiler would represent this single-table pipeline with the linked list template, leading to an inefficient datapath. However, our table decomposition algorithm can infer an equivalent multi-stage pipeline (see Fig. 7b), whereas all tables fit the direct code template as well as the hash template. This gives rise to a more efficient fast path, demonstrating the power of table decomposition. Flow table complexity was set by varying the number of web services between 1 and 100 and traffic traces were generated so that half of the packets go to a random web service and the rest of the traffic be dropped.

The most complex use case is a telco *access gateway*, which models a virtual Provider Endpoint (vPE) providing users Internet access via Customer Endpoints (CEs). Each CE is identified by a unique VLAN tag and each user is assigned a per-CE unique private IP address (see Fig. 8). The gateway pipeline is as follows. Table 0 separates user–network traffic on a per-CE basis from network–user traffic; user–network traffic in turn goes to separate per-CE tables that identify users and swap the (private) source IP address with a unique public address (realizing a simple NAT) and then to the Internet based on an IP routing table (Table 110). Packets missing the per-CE tables are passed to the controller that does admission control, allocates a public IP, and installs per-user "NAT" rules into the proper tables. In the reverse direction, packets are mapped from the public IP back to the adequate combination of VLAN tag and user private address. ESWITCH compiles this pipeline using the hash template for each table except for Table 110 that is mapped to the LPM store. In each measurement we provisioned 10 CEs with 20 users per CE and the IP routing table contained 10K IP prefixes, and the traffic mix was generated by varying the number of per-user flows.

4.2 Prototype & Evaluation Platform

We implemented a proof-of-concept ESWITCH prototype on top of the Intel DataPlane Development Kit (DPDK, [40]). The DPDK provides a highly efficient user space networking toolchain, with NIC polling drivers, batch processing, direct cache access, and NUMA optimized memory pools, as well as some prefab flow table templates (LPM). Our prototype implements a useful subset of OpenFlow 1.3 along with the main ESWITCH features with one notable exception: at the moment it defaults to a combined L2–L4 packet parser. Adding full support for parser templates is underway.

in_port	ip_src	ip_dst	tcp_dst	action
external	0.0.0.0/1	192.0.2.1	80	output:server1
external	128.0.0.0/1	192.0.2.1	80	output:server2
external	0.0.0.0/1	192.0.2.2	80	output:server3
external	128.0.0.0/1	192.0.2.2	80	output:server4
...	...	...	...	...
external	*	*	*	drop
internal	*	*	*	output:external

(a) pipeline

Table:0

in_port	action
external	goto_table:1
internal	output:external

Table:1

tcp_dst	action
80	goto_table:2
*	drop

Table:2

ip_src	action
0.0.0.0/1	goto_table:3
128.0.0.0/1	goto_table:4

Table:3

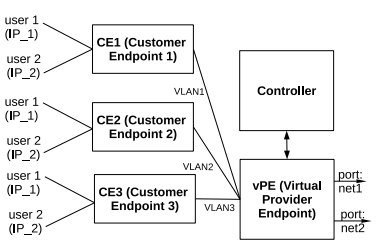
ip_dst	action
192.0.2.1	output:server1
192.0.2.2	output:server3
*	drop

Table:4

ip_dst	action
192.0.2.1	output:server2
192.0.2.2	output:server4
*	drop

(b) decomposed pipeline

Figure 7: Load balancer use case.



(a) access gateway setup

Table:11

ip_src	action
IP_1	write_actions(strip_vlan,mod_nw_src: **IP_11**), goto_table:200
IP_2	write_actions(strip_vlan,mod_nw_src: **IP_12**), goto_table:200
*	controller

Table:12

ip_src	action
IP_1	write_actions(strip_vlan,mod_nw_src :**IP_21**), goto_table:200
*	controller

Table:200

ip_dst	action
A.B.C.0.24	output:**net1**
X.Y.Z.0/24	output:**net2**

Table:0

in_port	action
vlan1	goto_table:11
vlan2	goto_table:12
...	...
net1	goto_table:110
net2	goto_table:110

PKT IN

Table:110

ip_dst	action
IP_11	push_vlan:**vlan1**,mod_nw_dst:**IP_1**,output:**vlan1**
IP_12	push_vlan:**vlan1**,mod_nw_dst:**IP_2**,output:**vlan1**
IP_21	push_vlan:**vlan2**,mod_nw_dst:**IP_1**,output:**vlan2**

PKT OUT

(b) pipeline

Figure 8: Access gateway use case.

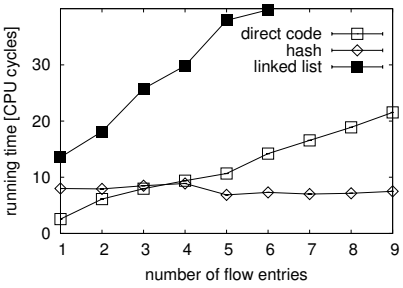


Figure 9: Running time as the function of the number of flow entries for the different flow table templates.

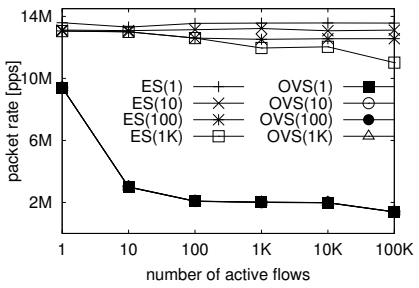


Figure 10: Packet rate for L2 switching over MAC tables of size 1, 10, 100, and 1K entries, as the active flow set grows.

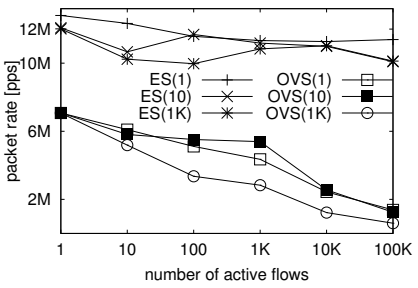


Figure 11: Packet rate for L3 routing over 1, 10, and 1K IP prefixes, as the active flow set grows.

When not stated otherwise, the system-under-test (SUT) was set up as of Table 1, connected via a 40 Gbps interface back to back to a similarly configured system that ran the Network Function Performance Analyzer (NFPA, [52]), a home-grown measurement platform using the DPDK *pktgen* packet generator, configured in loopback mode. All evaluations were done using the DPDK datapath of OVS compiled with the same DPDK version as ESWITCH, with minimum sized (64 byte) packets on a single CPU core; we found that both ESWITCH and OVS scale to larger packet sizes and more cores as expected (but see later). The maximum single-core packet rate attainable with DPDK on this platform is 15.7 million packets per second (Mpps), measured in port-forward mode with the DPDK `l2fwd` tool; we shall set this metric as a benchmark for the measurements.

Table 1: System-under-test datasheet.

CPU: Intel Xeon CPU E5-2620 @ 2.00GHz, Sandy Bridge
Caches: 32k L1i and L1d, 256 KB L2, 15 MB L3
Cache latency: L1 = 4 cycles, L2 = 12 cycles, L3 = 29 cycles
Memory: 64 GB DDR3 @ 1333 MHz, 4-channels
NIC: Intel XL710, PCI Express 3.0/x8, 40 Gb
DPDK v2.2.0, Open vSwitch (DPDK datapath) version 2.5.90

4.3 Measurement Results

Fine-tuning template application. Our first series of measurements were aimed at calibrating the code generation process. Of particular interest were designating the flow table template fallbacks and adjusting the rules of when to invoke these fallbacks. For a series of increasingly larger synthetic flow tables with the N-th entry set to

```
vlan_vid=3,ip_src=10.0.0.3,ip_proto=17,udp_dst=N ,
```

the execution time (in terms of CPU cycle count at 99% confidence level) needed to perform a flow lookup by the direct code, the compound hash, and the linked list template, is shown in Fig. 9 (the LPM template does not apply to this flow table). Until about 4 entries the direct code template is the most efficient choice, but from that point the hash template becomes faster thanks to its constant lookup time. Accordingly, we fixed the fallback constant for the direct code template at 4; tables with at most that many entries are directly compiled while for larger tables the hash template is preferred. We set the linked list as the last-resort fallback for complex flow tables despite being consistently slower than the direct code, since it supports fast incremental updates.

Packet rate. The raw packet throughput for ESWITCH (ES) vs Open vSwitch (OVS) is given for the L2 use case in Fig. 10, for L3 in Fig. 11, for the load balancer in Fig. 12, and for the gateway in Fig. 13, respectively, over different pipeline complexities and increasingly more diverse traffic mixes. The main observations are as follows. As the number of active flows increases, that is, as traffic locality is gradu-

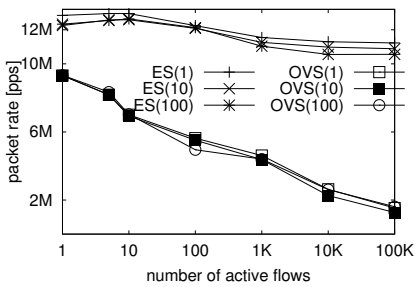


Figure 12: Packet rate for the load balancer use case over 1, 10, and 100 web services, as the active flow set grows.

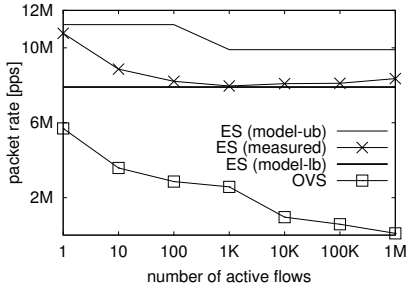


Figure 13: Packet rate for the gateway use case with 10 CEs, 20 user/CE, and 10K IP prefixes, with estimated lower and upper bounds.

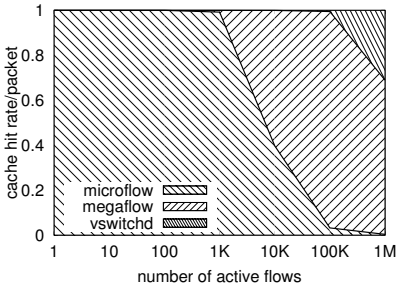


Figure 14: Fraction of packets forwarded at different levels of the OVS cache hierarchy as the active flow set grows (gateway use case).

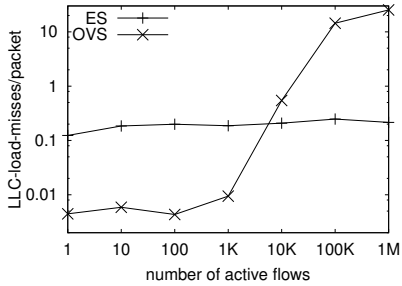


Figure 15: Last-level CPU cache (LLC) misses per packet measured with the `perf` tool, as the active flow set grows (gateway use case).

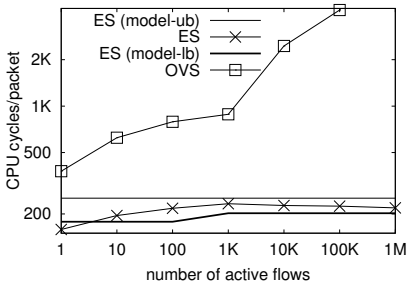


Figure 16: Latency in terms of mean CPU cycles/packet on the gateway pipeline as the active flow set grows, with estimated lower and upper bounds.

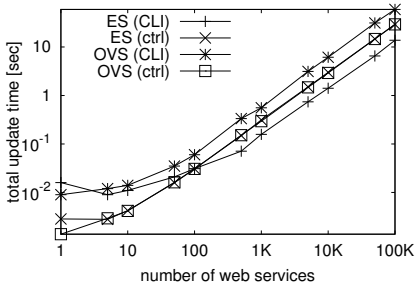


Figure 17: Total time to set up the load-balancer pipeline with a command line tool (CLI) and OpenFlow controller (ctrl), as the number of services grows.

ally removed, so the performance of OVS flow caching deteriorates. We already experience major performance drops at as few as 10 active flows, and for 100 flows the packet rate essentially halves (or even worse). The reason is revealed in Fig. 14, which gives a rundown on cache hit intensities experienced at different levels of the OVS flow cache hierarchy: as the active flow set grows packet processing gradually shifts from the very fast microflow cache to the slower megaflow cache and finally to the `vswitchd` slow path. This goes hand in hand with a degradation of CPU cache affinity (Fig. 15): as long as packet processing occurs entirely inside the microflow cache (up to ∼1K flows) OVS basically never misses the CPU cache, while the megaflow cache and `vswitchd` make excess out-of-cache memory references. ESWITCH, on the other hand, sidesteps flow caching and *exhibits robust and high packet rate over essentially all OpenFlow pipelines and all traffic mixes examined*, consistently reaching 12–14 Mpps packet rate (close to the platform benchmark of 15 Mpps) when the active flow set is not too large, and 9–12 Mpps with many flows.

In summary, ESWITCH generally achieves 2–7 times higher packet throughput than OVS but the factor can grow up to two orders of magnitude(!) for complex pipelines with many active flows. For the gateway use case, OVS throughput drops hundredfold to a mere 90K packets per second at 1M flows, which, if exploited by a malicious user, means a full-blown denial of service to the entire user population. Meanwhile, ESWITCH robustly delivers over 9 Mpps packet rate, suggesting that it is not susceptible to such attacks. Since both switches use the DPDK, the slowdown of OVS is clearly attributed to some overhead in the datapath code; we suggest that the culprit is generic flow-caching. In contrast, ESWITCH's compact custom datapaths deliver high switch performance and stable and small working set size.

Latency. The mean time for a packet to traverse the datapath is given in Fig. 16. For ESWITCH, we get about 0.1 μsec packet processing time independently of the active flow set, while latency for OVS varies between 0.2–13 μsec. This shows that *a compiled datapath yields smaller and predictable latency* compared to a flow-caching-based switch.

Update processing. For the first sight, one would consider template-compilation prohibitive for update-intensive workloads, like cloud hypervisor switches. To show that this is not the case, we measured the time it takes for OVS and ESWITCH to set up the complete load-balancer use case at different scales of pipeline complexity. The switches were fed first from a command line tool (`ovs-ofctl`, CLI) and then using the `Ryu` OpenFlow controller (we got similar results with OpenDaylight). The results are in Fig. 17 (note the log-log scale). Both switches scale linearly, but in general it takes just one fifth the time for ESWITCH to set up the use case than for OVS, when using the CLI tool. With the controller the two perform similarly. Overall, this indicates that *it is the OpenFlow controller, rather than* ESWITCH *itself, that bottlenecks update rates*, which justifies our choice of template-based compilation over slower methods that may

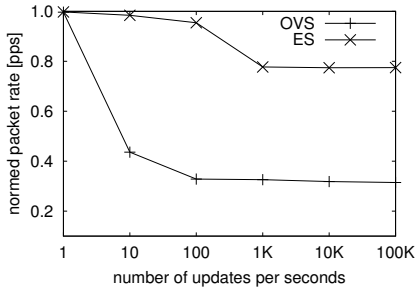


Figure 18: Normed packet rate (relative to the unloaded case) in the gateway use case (1K active flows), as the update intensity grows to 100K/sec.

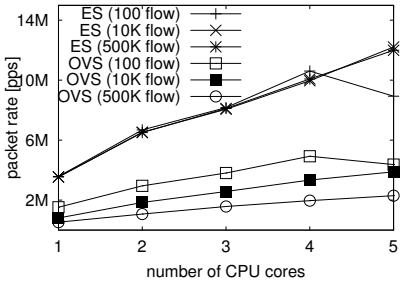


Figure 19: Packet rate as the number of packet processing CPU cores grows; L3 routing 100, 10K, resp. 500K active flows over 2K IP prefixes.

Pipeline stage	CPU cycles	Comment
`PKT_IN`	40	DPDK packet receive IO
parser template	28	Parse header fields
hash template 1	$8 + L1$	Table 0 lookup
hash template 2	$8 + Lx$	Per-CE table lookup
LPM template	$13 + 2 * Lx$	Routing table LPM
action templates	25	Action set processing
`PKT_OUT`	40	DPDK packet transmit IO

Figure 20: Performance model for the gateway use case: estimated number of CPU cycles spent per pipeline stage (Lx denotes access time for CPU cache level x).

potentially generate better code. Further, we observed that ESWITCH *packet processing is more robust in the face of updates*; ESWITCH churns out 95% of its nominal packet rate when the last level IP routing table in the gateway use case (Table 110) is updated 100 times per second and even at 100K update/sec intensity it maintains 80% of its unloaded performance; contrarily, OVS throughput falls by more than 65% even for 100 updates/sec due to deteriorating flow cache hit rates (Fig. 18). For batched updates (20 flow add and delete operations periodically), we saw at most 3% change in ESWITCH packet rate (most probably due to cold CPU caches) while for OVS the throughput drop reached 23%. This is because in ESWITCH datapath updates are, in contrast to OVS, per-table and usually non-destructive.

CPU scalability. Finally, we show that previous single-core results extend nicely to multiple CPUs. Since the Intel XL710 NIC in the SUT supports only about 23 Mpps packet rate with 64-byte packets [53, 54], ESWITCH proves too fast for this experiment: it saturates the NIC even with just two cores. To still make packet forwarding CPU-bounded rather than IO-bounded, we had to downgrade to a slower 2.40GHz Intel Atom platform. The evaluations were performed on the L3 use case, with a subset of 2K routes obtained from a real router (see Fig. 19). Both OVS and ESWITCH show strong linear CPU scaling (apart from a small reproducible glitch with 5 cores), but ESWITCH *consistently outperforms* OVS *roughly* 5*-fold and the gap increases with more flows and more CPU cores*. Again, ESWITCH performance is robust against the number of active flows.

4.4 Performance Estimation

A crucial advantage of dataplane specialization is that the compiled datapaths are simple enough to lend themselves to coarse-grained performance models. After all, a compiled datapath is just a handful of templates linked into a binary and so we can define elementary performance "atoms" to characterize each template and track down the template generation process to combine these atoms into composite datapath models. Such models can then yield coarse performance estimates, providing operators with quick performance promises, supporting network function placement, etc.

We demonstrate the derivation of such a model on the gateway pipeline. Our measurements indicated that it is the user-network direction, involving a costly longest prefix match on a largish routing table, that dominates performance for this use case. Fig. 20 gives a rundown on the templates used for compiling this pipeline direction.

A quick analysis of the assembly code suggests that the most expensive operations in the compiled datapath are the memory fetches. We also observe that ESWITCH performs very few last-level CPU cache misses (roughly one for every 10th packet, Fig. 15). Based on these considerations, we can divide per-packet cost into two components, a fix cost that is invariant for each packet and a variable cost component that changes with the distribution of accesses between the L1, L2 and L3 CPU caches and, correspondingly, with the working set size (the amount of total data accessed by the datapath) determined by the number of active flows. The fix cost can come from assembly code analysis, CPU cycle-count measurements, and basic common sense, while the variable cost component is shaped by CPU memory access speed.

Static code analysis yields that a generic DPDK packet IO takes about 40-50 CPU cycles (the NIC loads the packet directly into the L3 cache, from where the first 64 bytes containing the header is fetched by the CPU in a single L3 load), packet parsing takes 28 cycles, and applying actions another 25. Table 0 will compile into the hash template but the size is small enough to warrant a safe L1 CPU cache access, taking $8 + L1$ CPU cycles where Lx denotes the number of cycles needed to access the cache at level x. The per-CE tables again use the hash template but, being variable size, may access the L2 or the L3 cache, which takes $8 + Lx$ cycles. Finally, the LPM stage, using DPDK's built-in `DIR-24-8` data structure, runs in $13 + 2 * Lx$ cycles, assuming that each LPM search needs two memory accesses. Eventually, we get $166 + 3 * Lx$ cycles per packet.

Of course, such models can never aim to be comprehensive, as CPU pipeline semantics, branch prediction misses, cache collisions, etc., greatly influence real performance. That being the case, we can still use the model to obtain simple best-case and worst-case throughput estimates. The optimistic presumption that all cache accesses succeed from the L1 cache would give 178 cycles/packet and 11.2 Mpps packet rate. A slightly less optimistic assumption is that,

at roughly 1K active flows, the working set size grows large enough to shift memory accesses to the L2 CPU cache, which gives 202 cycles/packet and 9.9 Mpps throughput. These optimistic assumptions suggest a rude upper bound on achievable packet rate. Conversely, a pessimistic assumption will constrain all memory accesses to the L3 cache, yielding 253 cycles/packet and 7.9 Mpps lower bound on the throughput.

When validated against real measurements, these bounds turn out to provide surprisingly useful performance hints, both in terms of packet rates (Fig. 13) and per-packet processing cost (Fig. 16). Our experiments so far with similar performance models have given promising results. For most scenarios the models deliver close performance estimates for at least a limited regime of the configuration space, even on workloads as complex as the gateway use case. We have seen cases, however, when the predicted performance was off by an order of magnitude; further research is therefore needed to make our models reliable and to eliminate, or to at least prognosticate, such pathologic cases.

5. RELATED WORK

Recent work takes a programming-language approach to leverage compilers and runtime systems to optimize OpenFlow performance [8–10]. These compilers, however, are to provide high-level human-centric abstractions to network configuration, whereas ESWITCH shoots at a lower-level of the OpenFlow food chain: compiling a pipeline specification to the bare metal. In this regard, ESWITCH is closer to proposals like NOSIX [34], P4 [55], and RMT [56], aimed at finding a better match between controller programs and the underlying dataplane. The closest to our approach is perhaps network stack specialization [57], but instead of concentrating on endhosts herein we focus on intermediate systems.

Of particular interest here is P4 [49, 55]. Both P4 and ESWITCH are datapath compilers, emitting efficient fast-paths from an abstract, declarative description of a switch's packet processing functionality. P4, however, is much more generic than ESWITCH as it offers a complete language to customize switch behavior, all the way from header formats and supported actions to the semantics of flow tables and the control of flow among them; ESWITCH is, in contrast, limited to OpenFlow, with hard-coded header formats, actions, matching semantics, etc. On the other hand, ESWITCH can potentially generate a more efficient fast-path than an equivalent P4 program would do; in contrast to P4 that has only an abstract dataplane description available at compile time but not the flow entries themselves, ESWITCH also knows the *content* of the pipeline; correspondingly, P4 is constrained to produce the datapath statically while ESWITCH can apply sophisticated run-time optimizations in full knowledge of the pipeline, like upgrading small tables to the direct code template, template specialization with full constant inlining and direct jump pointers, etc. Of course, run-time optimization does not come for free, as ESWITCH needs to partially recompile the datapath from time to time but, as we have shown experimentally, this is very cheap thanks to the efficiency of template-based code generation.

Flow table templates are common in hardware OpenFlow switches, which usually include separate pipeline stages supporting varying sorts of match semantics [34]; herein, we simply adopted this design for softswitches. Template-based program specialization, a technique to dynamically link precompiled code fragments into machine code [58, 59], has for a long time been the preferred choice for writing compilers for embedded domain-specific languages, like SQL [60] or LINQ [61], thanks to the fast compilation cycles. Notably, it also allows to eliminate looping overhead and fold constants into instructions. ESWITCH heavily builds on these features. Recently runtime code generation has received renewed interest, for just-in-time (JIT) compiling hot code paths from intermediate representations into machine code [62]. The analogous JIT compilation of "hot flows" by ESWITCH seems a particularly intriguing research direction.

Performance modeling and prediction for network functions was initiated in [63] and has been shown to provide useful characterizations for various real-life use cases [64, 65]. Nevertheless, as far as we know this is the first time that a, however simple, datapath performance model for a complex OpenFlow pipeline appears in the literature. In the future ESWITCH could be easily taught to derive such models automatically, by programmatically composing template model "atoms" using the primitives introduced in Atomix [66]. This would make it possible to not only produce efficient specialized datapaths but also to deliver reliable performance promises for these datapaths in real time.

6. CONCLUSIONS

OpenFlow software switches are indeed true masterpieces of genericity, supporting a broad spectrum of packet forwarding semantics with considerable efficiency. Unfortunately, genericity comes at a high cost: despite increasingly powerful hardware operators still often need to manually tweak their pipelines to work around softswitch performance regressions. In this paper, we argued that this should be the other way around: instead of customizing flow tables for the underlying data plane it is rather the dataplane that should be specialized with respect to the workload.

We introduced ESWITCH, a novel switch architecture capitalizing on the observation that OpenFlow pipelines are sufficiently structured to admit efficient machine code representations constructed out of simple packet processing templates. The resultant specialized datapaths then were shown to give major performance gains over flow-caching-based alternatives, with several times higher raw packet rates, much smaller latency, and, perhaps most importantly, robust and predictable throughput even on widely varying, or borderline malicious, workloads. The proposed switch architecture easily scales to hundreds of flow tables and hundreds of thousands of traffic flows, while supporting updating the fast path at similar, or higher, intensity.

Acknowledgements

The authors would like to thank the support for the HSNLab at BME-TMIT and the MTA-BME Future Internet Research

Group. G.R. was visiting TrafficLab, Ericsson Research, Hungary, while working on this paper. Corresponding author: Gábor Rétvári <retvari@tmit.bme.hu>.

7. REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] The Open Networking Foundation, *OpenFlow Switch Specifications v.1.4.0*, 2013.

[3] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NaaS: network-as-a-service in the cloud," in *Hot-ICE*, vol. 12, pp. 1–1, 2012.

[4] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary, "NetLord: a scalable multi-tenant network architecture for virtualized datacenters," in *SIGCOMM*, pp. 62–73, 2011.

[5] A. Sadasivarao, S. Syed, P. Pan, C. Liou, A. Lake, C. Guok, and I. Monga, "Open Transport Switch: A software defined networking architecture for transport networks," in *HotSDN*, pp. 115–120, 2013.

[6] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *SIGCOMM*, pp. 43–56, 2015.

[7] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, "XIA: Efficient support for evolvable internetworking," in *NSDI*, 2012.

[8] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in *ICFP*, pp. 279–291, 2011.

[9] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: a language for provisioning network resources," in *CoNEXT*, pp. 213–226, 2014.

[10] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *SIGCOMM*, pp. 87–98, 2013.

[11] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao, "Towards programmable enterprise WLANS with Odin," in *HotSDN*, pp. 115–120, 2012.

[12] C. Chen, C. Liu, P. Liu, B. T. Loo, and L. Ding, "A scalable multi-datacenter layer-2 network architecture," in *SOSR*, pp. 1–12, 2015.

[13] T. K. et al., "Network virtualization in multi-tenant datacenters," in *NSDI*, pp. 203–216, 2014.

[14] N. Amaya, S. Yan, M. Channegowda, B. Rofoee, Y. Shu, M. Rashidi, Y. Ou, G. Zervas, R. Nejabati, D. Simeonidou, *et al.*, "First demonstration of software defined networking (SDN) over space division multiplexing (SDM) optical networks," in *ECOC*, 2013.

[15] A. Gupta, M. Shahbaz, L. Vanbever, H. Kim, R. Clark, N. Feamster, J. Rexford, and S. Shenker, "SDX: a software defined Internet Exchange," in *SIGCOMM*, pp. 551–562, 2014.

[16] Netronome, "SDN Gateway: Reference design." https://netronome.com/sdn-gateway.

[17] Intel, "Network function virtualization: Virtualized BRAS with Linux and Intel architecture." https://networkbuilders.intel.com/docs/Network_Builders_RA_vBRAS_Final.pdf.

[18] Intel, "Network function virtualization: Quality of Service in Broadband Remote Access Servers with Linux and Intel architecture." https://networkbuilders.intel.com/docs/Network_Builders_RA_NFV_QoS_Aug2014.pdf.

[19] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *NSDI*, pp. 117–130, 2015.

[20] P. Gupta and N. McKeown, "Algorithms for packet classification," *Netwrk. Mag. of Global Internetwkg.*, vol. 15, no. 2, pp. 24–32, 2001.

[21] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *SIGCOMM*, pp. 147–160, 1999.

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

[23] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM*, pp. 135–146, 1999.

[24] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC: scalable and expressive packet classification," in *SIGCOMM*, pp. 15–26, 2014.

[25] S. Shirali-Shahreza and Y. Ganjali, "ReWiFlow: restricted wildcard OpenFlow rules," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 29–35, 2015.

[26] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking packet forwarding hardware," in *HotNets*, 2008.

[27] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting route caching: The world should be flat," in *PAM*, pp. 3–12, 2009.

[28] Y. Liu, S. O. Amin, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 1, pp. 14–21, 2013.

[29] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, 2014.

[30] EANTC, "Validating Cisco's NFV infrastructure Pt. 1." http://www.lightreading.com/nfv/nfv-tests-and-trials/validating-ciscos-nfv-infrastructure-pt-1/d/d-id/718684.

[31] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: an open framework for OpenFlow switch evaluation," in *PAM*, pp. 85–95, 2012.

[32] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *HotSDN*, pp. 43–48, 2013.

[33] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "OpenFlow switching: Data plane performance," in *IEEE ICC*, pp. 1–5, 2010.

[34] M. Yu, A. Wundsam, and M. Raju, "NOSIX: a lightweight portability layer for the SDN OS," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 28–35, 2014.

[35] J. Leng, Y. Zhou, J. Zhang, and C. Hu, "An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network." http://arxiv.org/abs/1504.03095.

[36] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," in *CoNEXT*, pp. 97–108, 2013.

[37] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," in *SIGCOMM*, pp. 111–122, 2013.

[38] H. Asai and Y. Ohara, "Poptrie: a compressed trie with population count for fast and scalable software IP routing table lookup," in *SIGCOMM*, pp. 57–70, 2015.

[39] E. Gamma, R. Helm, R. Johnon, and J. Vlissides, *Design Patterns, elements of reusable object-oriented software*. Addison Wesley, 1994.

[40] Intel, "Data Plane Development Kit." http://dpdk.org.
[41] A. Császár and G. Pongrácz, "SDN virtual switching innovation (demo)," 2015. Mobile World Congress, https://twitter.com/ericssonhungary/status/573087639080972288.
[42] "Openflow reference swtich." git://gitosis.stanford.edu/openflow.git.
[43] "CPqD OpenFlow repository." https://github.com/CPqD/ofsoftswitch13.
[44] "The xDPd project." http://xdpd.org.
[45] "LINC software repository." https://github.com/FlowForwarding/LINC-Switch.
[46] P. Newman, G. Minshall, and T. L. Lyon, "IP switching – ATM under IP," *IEEE/ACM Trans. Netw.*, vol. 6, no. 2, pp. 117–129, 1998.
[47] "6WINDGate virtual switch." http://www.6wind.com/6windgate-performance/virtual-switching.
[48] A. Canteaut, C. Lauradoux, and S. A., "Understanding cache attacks," 2006.
[49] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: a programmable, protocol-independent software switch," in *SIGCOMM*, 2016.
[50] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *SIGCOMM*, pp. 213–224, 2003.
[51] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Passive and Active Measurement*, pp. 347–359, 2015.
[52] L. Csikor, M. Szalay, B. Sonkoly, and L. Toka, "NFPA: Network function performance analyzer," in *IEEE NFV-SDN, Demo Track*, pp. 17–19, 2015.
[53] "Intel Ethernet Controller XL710 10/40 GbE – Product Brief," 2014.
[54] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: a scriptable high-speed packet generator," in *IMC*, pp. 275–287, 2015.
[55] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.
[56] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *SIGCOMM*, pp. 99–110, 2013.
[57] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *SIGCOMM*, pp. 175–186, 2014.
[58] F. Noel, L. Hornof, C. Consel, and J. Lawall, "Automatic, template-based run-time specialization: implementation and experimental study," in *ICCL*, pp. 132–142, 1998.
[59] F. Smith, D. Grossman, G. Morrisett, L. Hornof, and T. Jim, "Compiling for template-based run-time code generation," *Journal of Functional Programming*, vol. 13, no. 3, pp. 677–708, 2003.
[60] M. B. et al., "Impala: A modern, open-source SQL engine for Hadoop," in *CIDR*, 2015.
[61] J. Cheney, S. Lindley, and P. Wadler, "A practical theory of language-integrated query," in *ICFP*, pp. 403–416, 2013.
[62] M. P. Plezbert and R. K. Cytron, "Does "just in time" = "better late than never"?," in *POPL*, pp. 120–131, 1997.
[63] D. Joseph and I. Stoica, "Modeling middleboxes," *Network, IEEE*, vol. 22, no. 5, pp. 20–25, 2008.
[64] A. Sapio, M. Baldi, and G. Pongrácz, "Cross-platform estimation of network function performance," in *EWSDN*, pp. 73–78, 2015.
[65] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *ANCS*, pp. 29–38, 2015.
[66] M. Bansal, A. Schulman, and S. Katti, "Atomix: a framework for deploying signal processing applications on wireless infrastructure," in *NSDI*, pp. 173–188, 2015.

Appendix

We are given a flow table $T = \{(F_{i,j} : j \in [1,n]) \to a_i : i \in [1,m]\}$, where each key $F_{i,j}$ is either constant or a wildcard. We call a flow table *regular* if it matches on only a single field that contains no masks/wildcards except the last catch-all rule. Consider the below problem formulation:

REGDECOMP(T, k): *given flow table T and integer k, is there a semantically equivalent pipeline $\mathcal{T}$ so that $|\mathcal{T}| \leq k$ and each flow table $\tau \in \mathcal{T}$ is regular?*

THEOREM 1. REGDECOMP*(T, k) is coNP-hard.*

SKETCH OF THE PROOF: We show that REGDECOMP(T, k) is difficult already for $k = 1$ by reducing 3SAT to REGDECOMP(T, 1).

We are given a 3SAT instance on n variables $X_1, \ldots, X_n$ in conjunctive normal form

$$f(X_1, X_2, \ldots, X_n) = \bigwedge_{i=1}^{m} \Big((\neg) X_{i_1} \vee (\neg) X_{i_2} \vee (\neg) X_{i_3} \Big)$$

such that no variable appears in *all* clauses as an all positive (un-negated) or all negative (negated) literal.

First, we construct a flow table $T = \{F_{i,j}\}$ of n fields (one for each variable) and m rows (one for each clause) as follows: $F_{i,j} = 0$ if X_j is positive in the i-th clause, $F_{i,j} = 1$ if it is negative, and $F_{i,j} = *$ if X_j does not appear at all. We add an extra field Y and set it to 1 for all rows. For each row $i \in [1,m]$ we set the action $a_i =$ *false* and we also add a low-priority catch-all rule with action *true*.

Then for any choice of variables X and $Y = 1$, T effectively evaluates $f(X)$; the i-th row matches, yielding action *false*, if and only if the i-th clause is not satisfied. For the 3SAT example $(X_1 \vee \neg X_3 \vee X_4) \wedge (\neg X_1 \vee X_2 \vee X_3)$ we get the table:

X_1	X_2	X_3	X_4	Y	
0	*	1	0	1	*false*
1	0	0	*	1	*false*
*	*	*	*	*	*true*

Then, asking whether the 3SAT instance is *not* satisfiable is equivalent with deciding whether T can be decomposed into the pipeline with the single regular table:

Y	
1	*false*
0	*true*

This is because the 3SAT is not satisfiable if and only if T returns *false* for $Y = 1$ independently of the input X. □