

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, Hungary

†Dept. of Telecomm. and Media Informatics, Budapest Univ. of Technology and Economics

‡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 OpenFlow pipelines 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.

Categories and Subject Descriptors

C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*Packet-switching networks*

Keywords

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

1. INTRODUCTION

The OpenFlow switch is undoubtedly the most generic packet processing device ever conceived [1–3]. Depending on the configuration from the control plane, an OpenFlow switch can identify flows based on a broad combination of layer-2, layer-3, or layer-4 (L2–L4) protocol attributes and apply essentially any meaningful packet processing action on the packets of these flows. This facilitates to rapidly provision and centrally administer highly reconfigurable net-

work services on top of a potentially heterogeneous inventory of switches and thereby to provide Infrastructure-as-a-Service capabilities [4, 5]; it unlocks networks for radical innovation, allowing to mix circuit and packet switched infrastructures [6], heterogeneous control paradigms [7], and legacy with clean-slate protocols [1, 8]; and it still retains the familiar network operations mental model in expressive high-level declarative policies [9–11]. Accordingly, OpenFlow has found its use in a wide range of application areas, from enterprise networks [12], data centers [13] and multi-tenant clouds [14], optical transport networks [15], software-defined Internet exchanges [16], WANs [7] and WAN gateways [17], all the way to telco load-balancers, stateless firewalls, and mobile access gateways [18, 19]. Crucially, the OpenFlow dataplane must support all these diverse applications with reasonable efficiency.

Unfortunately, this genericity conflicts with performance; to quote [20], “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, despite decades of continuous research and development [21–25], *OpenFlow software packet classification still remains too expensive* for today’s line rates [20]. Reasons for this all trace back to the rich packet processing capabilities [26]; in particular, the need to perform basically arbitrary wildcard matches on a broad selection of packet header fields (40+ as of OpenFlow 1.4, [3]), flow priorities imposing a firm ordering on classification that, in certain cases, breaks elementary networking conventions like IP longest prefix matching (LPM), or the very fact that a pipeline may include dozens of stages and the corresponding classification tasks 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 [26]. With the rise of server virtualization in datacenter networking and network function virtualization, however, many OpenFlow appliances need to run on a stock x86 platform that lacks the necessary hardware-based packet classification components [6, 7, 12–14, 16–19].

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 [27–29]. 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, peer-to-peer protocols, MapReduce, or network monitoring [20,30]. Consequently, OpenFlow switches often exhibit abrupt performance regressions in various hard-to-predict combinations of flow tables and traffic patterns [30–34], opening the door to malicious denial-of-service-like attacks even over such innocently looking traffic patterns like port scans [20,30,35].

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 completely fails for large-scale IP routing (that would rather require LPM) or flow tables that heavily match across layer boundaries [20,30].

Instead of relying on an overly general-purpose fast path, we argue, *an OpenFlow switch should rather automatically 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 [37–39], or a fast, optimized packet classifier for L4 ACLs [21–25], and some 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 instead embraces a fully customized dataplane. We view OpenFlow as a declarative language to program the dataplane [9,10] 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 [40], 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. Crucially, 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, [41]). The dataplane is written entirely in assembly, hand-optimized to the x86 platform. We present extensive measurements to show that, compared

to OVS, ESWITCH *features superior and predictable performance, latency and multi-core scalability* with up to two orders of magnitude improvement on certain complex OpenFlow pipelines, *while supporting similar update intensities*. Our rudimentary prototype easily scales beyond 100 Gbps transmission speeds [42], downright beating many contemporary hardware OpenFlow switches by a large margin.

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 priority¹. 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 unchanged, while in the reverse direction only HTTP packets (`tcp_dst=80`) are admitted and the rest of the traffic is dropped.

Industry best practices recommend to split the pipeline

¹When not stated otherwise, we use the convention that flow entries are listed in decreasing order of priority, i.e., an entry higher up in the list has higher 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*).

into multiple consecutive stages, in order 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 [3, 18, 19]. For example, the VMware NVP network virtualization controller sets up more than a dozen stages in its packet processing pipeline [14]. 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 and, depending on the ingress interface, directs 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 lookup would entail iterating over this list priority-wise, possibly jumping to another table whenever a match is found and starting linear iteration anew. Easily, a direct datapath in the worst case loops through all flow entries and all match fields 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 [43], CPqD [44], xDPd [45], LINC [46]), or as a last resort for complex pipelines to which no fast specialized classification strategies apply [20].

Indirect datapath. Indirect datapath designs adopt the venerable fast-path/slow-path separation principle [47]. Here,

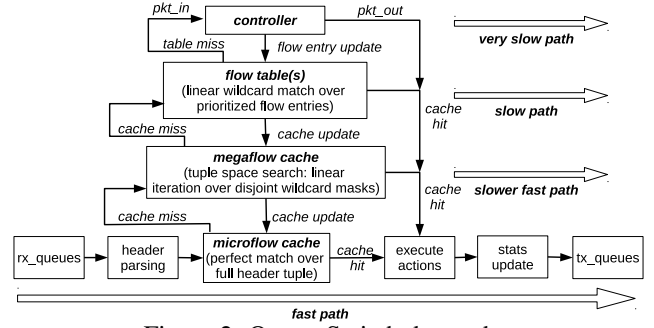


Figure 2: Open vSwitch datapaths.

the switch maintains multiple “views” of the pipeline: the fast path is constituted by possibly 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. 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 [27–29], contributing to increased performance without loss of expressiveness and genericity (see OVS [20], 6WINDGate [48]).

2.2 A Sophisticated Indirect Pipeline: OVS

The most popular OpenFlow software switch implementation today is undoubtedly Open vSwitch (OVS, [20]). 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 [24]: 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 multi-threading, 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 inherent 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 [27–29]. But only a single fine-grained rule is enough to “punch a hole” in all aggregates, leading to heavy megaflow fragmentation [14, 20]. Curiously, 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.

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

(a) flow table

(b) pkt port seq 1

(c) pkt port seq 2

(a) flow table

(b) pkt port seq 1 (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) (one cache entry to match each zero bit in the bit positions 1, 2, . . . , 7 counting from the LSB), while if destination port 191 arrives first as of seq 2 (c) then only a single entry arises (matching at position 7 and covering subsequent packets).

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 [30–34].

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 [30, 33]. 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 invalidation². 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 [49]. 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 caching brings 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 the worst possible *reactive* way, forcing packets to the slow path in order to populate the caches. But flow caching can also prove a hindrance to dataplane innovation; for instance, at the mo-

²OVS at the moment adopts the brute-force strategy to invalidate the entire flow cache after essentially all changes [20].

ment it is not clear how to fit flow caches to proposals like NOSIX [34] or P4 [50].

3. BEYOND FLOW CACHING: 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 concept of dataplane specialization and makes the datapath dependent on, and actually carefully tailored to, the configured OpenFlow pipeline.

We cast dataplane specialization as a 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 parser depends on the L2 header parser to find out the starting position of the L3 header and hence it will generally compose an L2 parser template inside, and similarly for layer 4.

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 separate 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 [3]. 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).

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, our current implementation fixes this parameter at 4.

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  FLOW_2       ; not an IP & TCP packet
    ; jump to next 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  FLOW_3       ; jump to next 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

Name: direct code Prerequisite: #flow-entries ≤ 4 Match type: arbitrary Implementation: on-the-fly assembled machine code Application: universal Fallback: compound hash	Name: compound hash Prerequisite: global mask Match type: exact match Implementation: perfect hash Application: MAC switching & port filtering Fallback: LPM	Name: LPM Prerequisite: prefix masks, consistent with priorities Match type: longest prefix match Implementation: DPDK LPM lib Application: IP routing Fallback: linked list	Name: linked list Prerequisite: none Match type: tuple space search Implementation: machine code with shared tuple matcher functions Application: complex pipelines Fallback: none
--	---	---	---

Figure 4: Flow table templates in ESWITCH.

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 the addition of the third entry violates 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 perfect hash. A perfect hash is a collision free hash; even though it may require 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 this 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 [24]: 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 on a certain port, flood, or modify packet header field) is a separate action template, and action templates are concatenated into composite action sets. Identical action sets are shared between flows with coinciding actions.

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. For instance, in the firewall use case 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:

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

in which rules $F_{i,j}$ can be only of 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: an exact match template with a potential final catch-all rule. Our task is to transform T into a semantically equivalent [25, 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 is called first on T and then recursively on 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

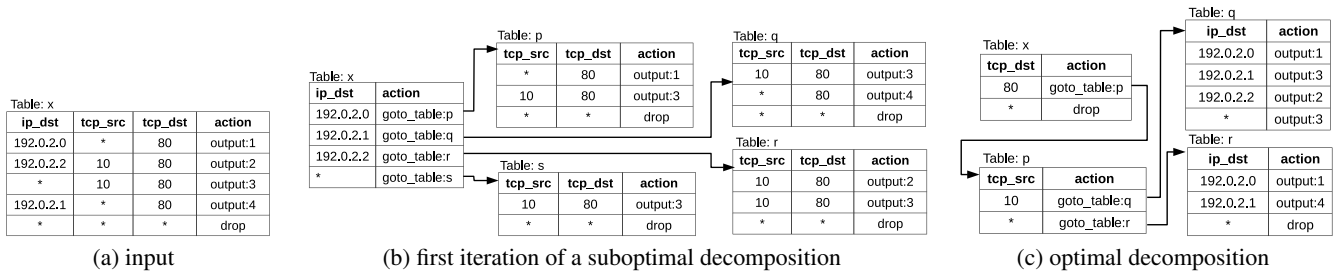


Figure 5: A sample flow table (a), first iteration of the heuristics after (suboptimally) decomposing along field `ip_dst` (b), and an optimal equivalent decomposition containing only 4 tables (c). Observe that each table in the optimal decomposition is consistent with the exact-match template; in fact,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

```

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:  $p = \underset{j \in \{1, \dots, n\}}{\operatorname{argmin}} |S_j|$ 
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  $i = 1, \dots, m$  do
     if  $F_{i,p} = *$  then
       for  $f \in S_p$  in descending order of priority do
          $T_f.add((F_{i,1}, \dots, F_{i,p-1}, F_{i,p+1}, \dots, F_{i,n}) \rightarrow a_i)$ 
     else
        $T_{F_{i,p}}.add((F_{i,1}, \dots, F_{i,p-1}, F_{i,p+1}, \dots, F_{i,n}) \rightarrow a_i)$ 
    $T.reset$ 
   for  $f \in S_p : S_p = \{f_1, f_2, \dots, *\}$  do
      $T.append(f \rightarrow \text{goto\_table } T_f)$ 
   for  $f \in S_p : \text{DECOMPOSE}(T_f)$ 

```

Figure 6: Table decomposition routine $\text{DECOMPOSE}(\tau)$.

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 produces 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 of tables produced (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 four-table pipeline, and hence promote the table to a pipeline of very fast hash templates.

Astute readers will recognize the set-pruning trie data structure, used extensively in packet classification [21], here, just doing matching field-wise and with a greedily optimized matching order. This scheme is then easy to extend to additional templates and arbitrary, possibly overlapping, masked keys; we omit the details.

Easily, no heuristics can beat fundamental (rather prohibitive) lower bounds [23]: for extremely complex flow tables our decomposer can not 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, 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.

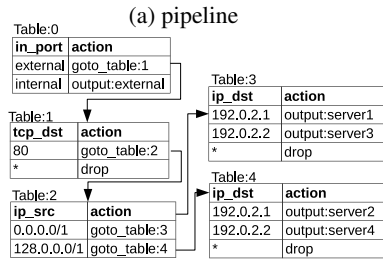
We evaluated the algorithm on a handful of real pipelines, collected from a production multi-tenant OpenStack cloud and a telco’s Border Network Gateway. Strikingly, in essentially all cases *our decomposer simply returned its input intact*, indicating that the *pipeline had already been decomposed optimally*. In fact, real-world controllers 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 [3, 18, 19]. 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 and, 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 templates still contain 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 use pointers to the

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



(b) decomposed pipeline

Figure 7: Load balancer use case.

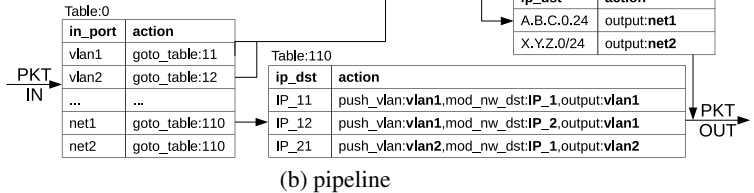
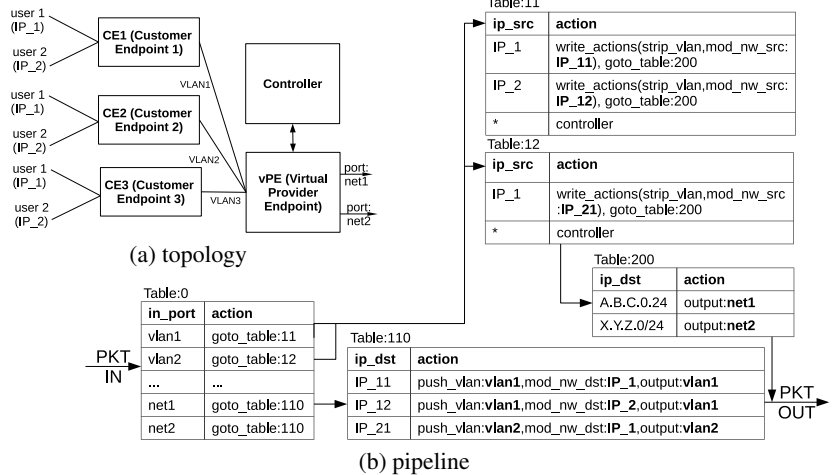


Figure 8: Access gateway use case.

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.

Updates in ESWITCH occur non-destructively: whenever the controller modifies a flow, ESWITCH either adds the new entry to the template (in the case of the compound hash, the LPM, and the linked list templates) or it rebuilds it from zero (for the direct code template or when the perfect hash is rebuilt). In the latter case, it constructs the new representation side by side with the running datapath and, when ready, it just adjusts the trampoline to the address of the new code, updating the datapath without interfering with packet forwarding in any ways. Correspondingly, ESWITCH can modify the datapath on a 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 [18, 19], 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, we give a detailed description of our ESWITCH prototype, and then we present the measurement

studies. Finally, we also show 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 inevitable problems with the underlying 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 [13], L3 in [14]). Notably, ESWITCH attains optimal dataplane specialization in both cases: the L2 pipeline compiles into the hash flow table template, effectively reducing into a classical Ethernet switch, while the L3 pipeline is compiled into the LPM template yielding a datapath identical to that of an IP softrouter.

The rest of the use cases, a load balancer and an access gateway, model multi-stage pipeline applications [18, 19].

The *load balancer* use case captures the functionality of a web frontend, which distributes HTTP traffic of 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 Table 0, 1, and 2 fit the direct code template and Table 3 and 4, depending on the number of web services, would either end up with the direct code template (for 4 or less services) or the hash template (for more). Thus, the new pipeline permits a more efficient fast path, demonstrating the power of table decomposition.

The most complex use case is a telco *access gateway*, which models a virtual Provider Endpoint (vPE) providing users Internet access through 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 into an efficient datapath, using the hash template for each table with the exception of Table 110 that is mapped to the LPM store.

4.2 Prototype & Evaluation Platform

We implemented a proof-of-concept ESWITCH prototype on top of the Intel DataPlane Development Kit (DPDK, [41]). 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.

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 DPDK *pktgen* packet generator in loopback mode. All evaluations were done with minimum sized (64 byte) packets on a single CPU core; we found that both switches scale to larger packet sizes and more cores as expected (but see later) [51]. The maximum packet rate attainable with DPDK on this platform is 15.7 million packets per second (Mpps), measured in port-forward mode with the DPDK *12fwd* tool; we shall set this metric as a benchmark for the measurements.

4.3 Measurement Results

Packet rate. The raw packet throughput for ESWITCH (ES) vs Open vSwitch (OVS) is given for the L2 use case in Fig. 9, for L3 in Fig. 10, for the load balancer in Fig. 11, and for the gateway in Fig. 12, respectively. We generated

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

increasingly more complex flow tables for each use case, by growing the MAC table and the routing table in the L2 and L3 use cases, the number of web services in the load balancer use case, etc. Meanwhile, we also varied the number of active flows in the input traffic mix. For the L2 use case, this was done by setting as many separate MAC destination addresses in the traffic trace as there were active flows, for L3 we adjusted the number of distinct destination IP addresses, while in the load balancer case we increased the number of web sessions and for the gateway we extended the range of per user flows. For the latter, we provisioned 10 CEs with 20 users per CE and the IP routing table contained 10K IP prefixes randomly sampled from a real Internet router.

The main observations are as follows. As the number of active flows increases, that is, as traffic locality is gradually 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. 13, 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. 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 (which is 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, strikingly, 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 throughput, suggesting that it is not susceptible to such attacks. This can be attributed to the compact datapath code and the stable and small working set size; see the CPU cache access patterns in Fig. 14.

Latency. The mean time for a packet to traverse the datapath is given in Fig. 15. For ESWITCH, we get about 0.1 μ sec packet processing time independently of the active flow set, while latency for OVS varies between 0.2–13 μ sec. This shows that *a compiled datapath yields smaller and predictable latency* compared to a flow-caching-based switch.

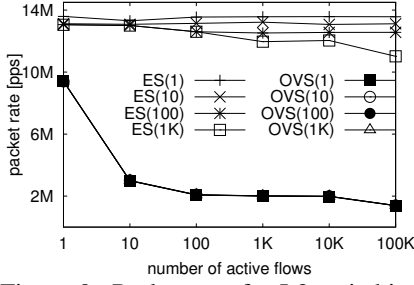


Figure 9: Packet rate for L2 switching over MAC tables of size 1, 10, 100, and 1K entries, as the active flow set grows.

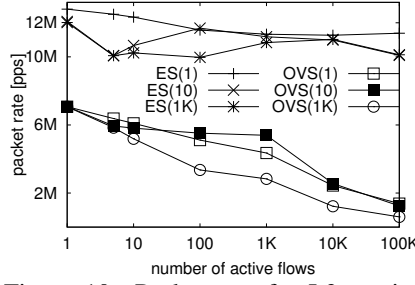


Figure 10: Packet rate for L3 routing over 1, 10, and 1K IP prefixes, as the active flow set grows.

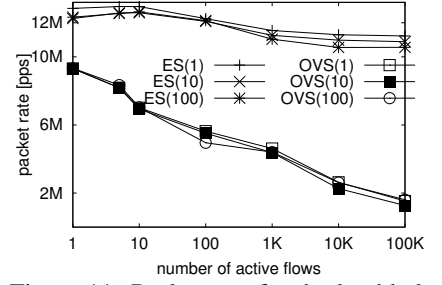


Figure 11: Packet rate for the load balancer use case over 1, 10, and 100 web services, as the active flow set grows.

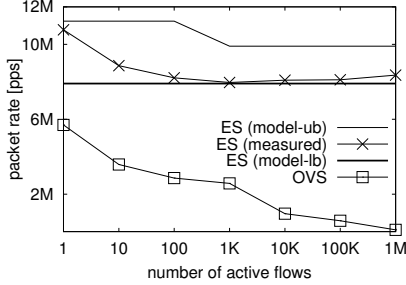


Figure 12: Packet rate for the gateway use case with 10 CEs, 20 user/CE, and 10K IP prefixes, with estimated lower and upper bounds.

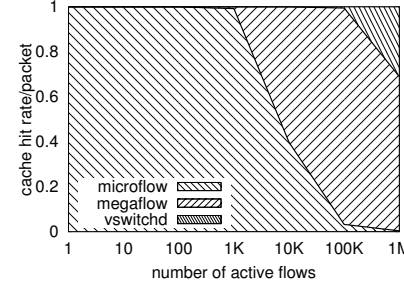


Figure 13: Fraction of packets forwarded at different levels of the OVS cache hierarchy as the active flow set grows (gateway use case).

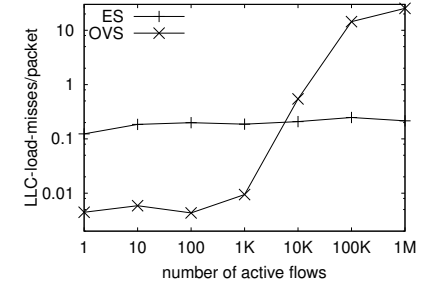


Figure 14: Last-level CPU cache (LLC) misses per packet measured with the perf tool, as the active flow set grows (gateway use case, 1K active flows).

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. 16 (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 as 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 potentially generate better code. Further, we observed that ESWITCH tolerates updates more robustly; we saw at most 3% change in ESWITCH packet rate when applying a batch of 20 flow add and delete operations in the gateway use case, while for OVS the throughput drop reached 23% in this setup. This is because in ESWITCH pipeline modifications are per-table and usually non-destructive, while OVS flushes the entire datapath on updates.

CPU scalability. Finally, we show that previous single-core results extend nicely to multiple CPUs. Strangely, ESWITCH proved just too fast for this experiment: it saturates the 40 Gb ports even with just two cores and fails to produce any

meaningful result afterwards. To still make packet forwarding CPU-bounded rather than IO-bounded, we had to downgrade to a much 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. 17). 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

Curiously, the compiled datapaths are simple enough to lend themselves to coarse-grained performance models; after all, a compiled datapath is nothing more than 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 be useful for providing operators with quick performance promises, supporting virtualized 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. Table 2 gives a rundown on the templates used

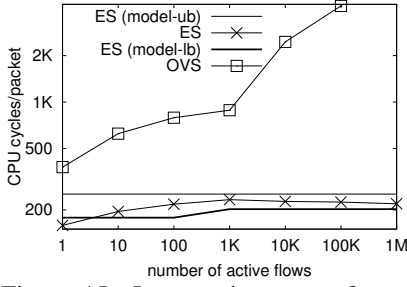


Figure 15: Latency in terms of mean CPU cycles/packet on the gateway pipeline as the active flow set grows, with estimated lower and upper bounds.

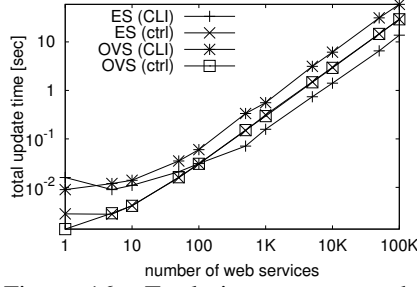


Figure 16: Total time to set up the load-balancer pipeline with a command line tool (CLI) and OpenFlow controller (ctrl), as the number of services grows.

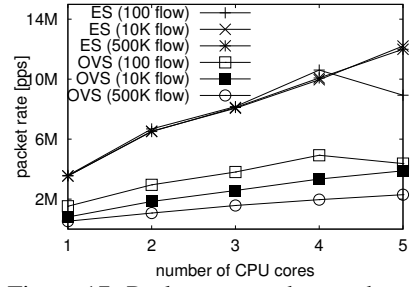


Figure 17: Packet rate as the number of CPU cores dedicated to packet forwarding grows; L3 routing 100, 10K, resp. 500K active flows over 2K IP prefixes.

Table 2: Performance model for the gateway use case.

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 processing
hash template 2	$8 + Lx$	Per-CE table processing
LPM template	$13 + 2 * Lx$	Routing table LPM
action templates	25	Action set processing
PKT_OUT	40	DPDK packet transmit IO

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 cache misses (one for every 5th–10th packet, see Fig. 14). 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 occupying a single 64 byte cache line, from where it can be 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 comprehen-

sive, as CPU pipeline semantics, branch prediction misses, cache collisions, etc., greatly influence the real performance. That being the case, the best use of these models is to obtain coarse performance estimates. For instance, we could presume optimistically that all cache accesses succeed from the L1 cache, which would give 178 cycles/packet and 11.2 Mpps packet rate. 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 packet rate achievable with ESWITCH in the gateway use case. 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, the model turns out to provide surprisingly useful performance hints, both in terms of packet rates (Fig. 12) and per-packet processing cost (Fig. 15). 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 the the gateway use case. We have seen cases, however, when the predicted performance was off by an order of magnitude, indicating that we failed to consider some crucial performance factor. Further research is therefore needed to make our models reliable and to eliminate, or to prognosticate, such pathologic cases.

5. RELATED WORK

Recent work takes a programming-language approach to leverage compilers and runtime systems to optimize OpenFlow performance [9–11]. These compilers, however, are to provide high-level human-centric abstractions to complex network configurations, 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 [50], and RMT [52], aimed at finding a better match between controller programs and the underlying dataplane. The closest to our approach is perhaps network stack specialization [53], but instead of con-

centrating on endhosts herein we focus on intermediate systems.

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 and extended this design for softswitches (but see also P4 [50]). Template-based program specialization, a technique to dynamically link copies of precompiled code fragments into machine code [54, 55], has for a long time been the preferred choice for writing compilers for embedded domain-specific languages, like SQL [56] or LINQ [57], 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 [58]. The analogous JIT compilation of “hot flows” by ESWITCH seems a particularly intriguing future research direction.

Performance modeling and prediction for network functions was initiated in [59] and has been shown to provide useful characterizations for real-life use cases in e.g., [60, 61]. 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. Intriguingly, ESWITCH could be easily taught to derive such models automatically, parallel with the compilation process, by programmatically composing template model “atoms” using the primitives introduced in Atomix [62]. This would make it possible to not only produce efficient specialized datapaths but to also deliver reliable performance promises for these datapaths in real time.

6. CONCLUSIONS

OpenFlow software switches are 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 specializing flow tables for the underlying data plane it is rather the dataplane that should be customized with respect to the intended 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 data paths then were shown to give major gains over flow-caching-based alternatives, with several times higher raw packet rates, much smaller latency, and, perhaps most importantly, robust and predictable performance even with widely varying, or straight-out 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.

7. REFERENCES

- [1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [2] ONF Foundation. Software-Defined Networking: The New Norm of Networks. *ONF White Paper*, 2012.
- [3] The Open Networking Foundation. *OpenFlow Switch Specifications v1.4.0*, 2013.
- [4] Paolo Costa, Matteo Migliaiavacca, Peter Pietzuch, and Alexander L Wolf. NaaS: network-as-a-service in the cloud. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE*, volume 12, pages 1–1, 2012.
- [5] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. In *SIGCOMM*, pages 62–73, 2011.
- [6] Abhinava Sadasivarao, Sharfuddin Syed, Ping Pan, Chris Liou, Andrew Lake, Chin Guok, and Inder Monga. Open Transport Switch: A software defined networking architecture for transport networks. In *HotSDN*, pages 115–120, 2013.
- [7] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *SIGCOMM*, pages 43–56, 2015.
- [8] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: Efficient support for evolvable internetworking. In *USENIX NSDI*, 2012.
- [9] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *ICFP*, pages 279–291, 2011.
- [10] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: a language for provisioning network resources. In *CoNEXT*, pages 213–226, 2014.
- [11] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, pages 87–98, 2013.
- [12] Lalith Suresh, Julius Schulz-Zander, Ruben Merz, Anja Feldmann, and Teresa Vazao. Towards programmable enterprise WLANS with Odin. In *HotSDN*, pages 115–120, 2012.
- [13] Chen Chen, Changbin Liu, Pingkai Liu, Boon Thau Loo, and Ling Ding. A scalable multi-datacenter layer-2 network architecture. In *SOSR*, pages 1–12, 2015.
- [14] Teemu Koponen et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI 14*, pages 203–216, 2014.
- [15] Norberto Amaya, S Yan, Mayur Channegowda, BR Rofoee, Y Shu, Mehdi Rashidi, Y Ou, Geogios Zervas, Reza Nejabati, Dimitra Simeonidou, et al. First demonstration of software defined networking (SDN) over space division multiplexing (SDM) optical networks. In *European Conference and Exhibition on Optical Communication (ECOC2013)*, 2013.
- [16] Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Hyojoon Kim, Russ Clark, Nick Feamster, Jennifer Rexford, and Scott Shenker. SDX: a software defined Internet Exchange. In *SIGCOMM*, pages 551–562, 2014.
- [17] Netronome. SDN Gateway: Reference design. <https://netronome.com/sdn-gateway>.
- [18] Intel. Network function virtualization: Virtualized BRAS with Linux and Intel architecture. https://networkbuilders.intel.com/docs/Network_Builders_RA_vBRAS_Final.pdf.
- [19] 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.
- [20] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin

- Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.
- [21] P. Gupta and N. McKeown. Algorithms for packet classification. *Netw. Mag. of Global Internetwkg.*, 15(2):24–32, 2001.
- [22] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.
- [23] A. Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *INFOCOM*, volume 3, pages 1193–1202, 2000.
- [24] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *SIGCOMM*, pages 135–146, 1999.
- [25] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. SAX-PAC: scalable and expressive packet classification. In *SIGCOMM*, pages 15–26, 2014.
- [26] Sajad Shirali-Shahreza and Yashar Ganjali. ReWiFlow: restricted wildcard OpenFlow rules. *SIGCOMM Comput. Commun. Rev.*, 45(5):29–35, 2015.
- [27] Martin Casado, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Rethinking packet forwarding hardware. In *HotNets*, 2008.
- [28] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *PAM*, pages 3–12, 2009.
- [29] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient FIB caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, 2013.
- [30] Nick Shelly, Ethan J. Jackson, Teemu Koponen, Nick McKeown, and Jarno Rajahalme. Flow caching for high entropy packet fields. *SIGCOMM Comput. Commun. Rev.*, 44(4):1–1, 2014.
- [31] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: an open framework for OpenFlow switch evaluation. In *PAM*, pages 85–95, 2012.
- [32] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, pages 43–48, 2013.
- [33] A. Bianco, R. Birke, L. Giraud, and M. Palacin. OpenFlow switching: Data plane performance. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5, 2010.
- [34] Minlan Yu, Andreas Wundsam, and Muruganantham Raju. NOSIX: a lightweight portability layer for the SDN OS. *SIGCOMM Comput. Commun. Rev.*, 44(2):28–35, 2014.
- [35] Junyuan Leng, Yadong Zhou, Junjie Zhang, and Chengchen 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] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *CoNEXT*, pages 97–108, 2013.
- [37] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zsolt Heszberger. Compressing IP forwarding tables: Towards entropy bounds and beyond. In *SIGCOMM*, pages 111–122, 2013.
- [38] Tong Yang, Gaogang Xie, Yanbiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. Guarantee IP lookup performance with FIB explosion. In *SIGCOMM*, pages 39–50, 2014.
- [39] Hirochika Asai and Yasuhiro Ohara. Poptrie: a compressed trie with population count for fast and scalable software IP routing table lookup. In *SIGCOMM*, pages 57–70, 2015.
- [40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, elements of reusable object-oriented software*. Addison Wesley, 1994.
- [41] Intel. Data plane development kit. <http://dppk.org>.
- [42] A. Császár and G. Pongrácz. SDN virtual switching innovation (demo), 2015. Mobile World Congress, Barcelona.
- [43] Openflow reference switch. [git://gitosis.stanford.edu/openflow.git](https://github.com/openflow/openflow.git).
- [44] CPqD OpenFlow repository. <https://github.com/CPqD/ofsoftswitch13>.
- [45] The xDPd project. <http://xdpd.org>.
- [46] LINC software repository. <https://github.com/FlowForwarding/LINC-Switch>.
- [47] Peter Newman, Greg Minshall, and Thomas L. Lyon. IP switching – ATM under IP. *IEEE/ACM Trans. Netw.*, 6(2):117–129, 1998.
- [48] 6WINDGate virtual switch. <http://www.6wind.com/6windgate-performance/virtual-switching>.
- [49] A Canteaut, C. Lauradoux, and Sez nec A. Understanding cache attacks, 2006.
- [50] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.
- [51] Levente Csikor, Márk Szalay, Balázs Sonkoly, and László Toka. NFPA: Network function performance analyzer. In *IEEE Conference on Network Function Virtualization and Software Defined Networks Demo Track (NFV-SDN)*, pages 17–19, 2015.
- [52] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, pages 99–110, 2013.
- [53] Ilias Marinou, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *SIGCOMM*, pages 175–186, 2014.
- [54] F. Noel, L. Hornof, C. Consel, and J.L. Lawall. Automatic, template-based run-time specialization: implementation and experimental study. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 132–142, 1998.
- [55] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.
- [56] Marcel Bittorf et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.
- [57] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416, 2013.
- [58] Michael P. Plezbert and Ron K. Cytron. Does “just in time” = “better late than never”? In *POPL*, pages 120–131, 1997.
- [59] D. Joseph and I. Stoica. Modeling middleboxes. *Network, IEEE*, 22(5):20–25, 2008.
- [60] A. Sapio, M. Baldi, and G. Pongrácz. Cross-platform estimation of network function performance. In *EWSDN*, pages 73–78, 2015.
- [61] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet IO. In *ANCS*, pages 29–38, 2015.
- [62] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: a framework for deploying signal processing applications on wireless infrastructure. In *USENIX NSDI*, pages 173–188, 2015.

Appendix

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

DEFINITION 1. $\text{REGDECOMP}(T, k)$: given flow table T and integer k , is there a semantically equivalent pipeline \mathcal{T} so that $|\mathcal{T}| \leq k$ and $\forall \tau \in \mathcal{T} : \tau$ is regular?

THEOREM 1. $\text{REGDECOMP}(T, k)$ is coNP-hard.

SKETCH OF THE PROOF: We reduce 3SAT to $\text{REGDECOMP}(T, 1)$. We are given a 3SAT instance on n variables X_1, \dots, X_n in conjunctive normal form

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

such that no variable appears in *all* clauses as an all positive (un-negated) or all negative (negated) literal. We construct a flow table $T = \{F_{i,j}\}$ of exactly n columns (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. 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 , 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	
0	*	1	0	<i>false</i>
1	0	0	*	<i>false</i>
*	*	*	*	<i>true</i>

Then, asking whether the 3SAT instance is *not* satisfiable is equivalent to asking whether the above $\text{REGDECOMP}(T, 1)$ instance is true with the (regular) table $T = \{(*, *, \dots, *) \rightarrow false\}$ as result. \square