Normal Forms for Match-Action Programs

Felicián Németh
MTA-BME Network Softwarization Research Group
nemethf@tmit.bme.hu

Marco Chiesa
KTH Royal Institute of Technology
mchiesa@kth.se

Gábor Rétvári
MTA-BME Information Systems Research Group
retvari@tmit.bme.hu

ABSTRACT

Packet processing programs may have multiple semantically equivalent representations in terms of the match-action abstraction exposed by the underlying data plane. Some representations may encode the entire packet processing program into one large table allowing packets to be matched in a single lookup, while others may encode the same functionality decomposed into a pipeline of smaller match-action tables, maximizing modularity at the cost of increased lookup latency. In this paper, we provide the first systematic study of match-action program representations in order to assist network programmers in navigating this vast design space. Borrowing from relational database and formal language theory, we define a framework for the equivalent transformation of match-action programs to obtain certain irredundant representations that we call "normal forms". We find that normalization generally improves the capacity of the control plane to program the data-plane and to observe its state, at the same time having negligible, or positive, performance impact.

ACM Reference Format:
Felicián Németh, Marco Chiesa, and Gábor Rétvári. 2019. Normal Forms for Match-Action Programs. In *CoNEXT '19: International Conference On Emerging Networking Experiments And Technologies, December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3359989.3365417

1 INTRODUCTION

The match-action paradigm, describing general packet processing programs in terms of a sequence of classifier tables of user-defined (rule, action) pairs, has recently emerged as the prevalent model for exposing the functionality of reconfigurable data-plane devices to the network programmer [3]. This is on the one hand thanks to the convenient abstraction of "flows", traffic aggregates defined by wildcard rules on header fields, making it possible to formulate data-plane programs in the mental model familiar from L2/L3 forwarding tables, firewalls, QoS classifiers, etc. This abstraction, furthermore, ended up being simple enough to be efficiently implemented in data plane devices [16]. Correspondingly, the match-action paradigm has become the major driver for data-plane programming languages (OpenFlow 1.1+ [26], P4 [5]), reconfigurable hardware ASICs (RMT [6]), software switches (OVS [28], ESwitch [24]), programmable kernel datapaths (eBPF/XDP [32]), and network-function virtualization frameworks (FastClick [2], BESS [13], NetBricks [27]).

Any nontrivial packet processing program may have multiple semantically equivalent representations in terms of match-action tables. Certain functionality may be best represented as a single match-action table, yielding a complex data-plane classifier matching on possibly many header fields simultaneously, while others may lend themselves to be modularized into a pipeline of successive tables [7, 12, 16, 23, 24]. Different representations in turn may have different operational properties, and currently there is very little understanding as to which representation is optimal for a particular purpose. In this paper, we take the first steps at studying the *systematic transformations of match-action programs* between single-table and multi-table forms and understanding how choosing a representation *affects the complexity of control-plane–data-plane interactions and the raw packet-processing performance.*

Our main contributions are as follows:

Redundancy in match-action programs. We observe that certain intrinsic dependencies that may exist between different match and action fields may introduce considerable redundancy, depending on the particular representation (single-table or multi-table pipeline) chosen. This redundancy may then unnecessarily bloat data-plane footprint and adversely affect the capacity of the control plane to program and monitor the data plane. While earlier work hinted at the possibility of "flow state explosion" phenomena to arise in various match-action program representations [24], to the best of our knowledge we are the first to uncover the systematic reasons behind such redundancy (§2).

Normal forms. We categorize the types of dependencies that may exist in match-action programs and we show certain multi-table representations that are guaranteed to be irredundant. Borrowing from relational database theory, we call these irredundant representations *normal forms* and we show in real-life use cases that such normal forms naturally arise in well-known applications. We note that our normal forms are orthogonal to existing approaches for minimizing packet classifiers [21, 23] (§3).

Program transformation. We define a formal framework for the equivalent transformation of match-action programs (decomposing a match-action table into multiple tables and vice versa) and we give sufficient and necessary conditions for the existence of such semantically equivalent decompositions. Our framework provides the first formal methodology for network programmers to reason about the redundancy in packet processing programs and to systematically convert between different representations (§4).

Performance. Finally, we study the implications of match-action program transformations to data-plane performance. In our preliminary benchmarks conducted on several real programmable switches, we find that normalization generally allows for more

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365417

table T_0

ip_src	ip_dst	tcp_dst	out
0/1	192.0.2.1	80	vm_1
1/1	192.0.2.1	80	vm_2
00/2	192.0.2.2	443	vm_3
01/2	192.0.2.2	443	vm_4
1/1	192.0.2.2	443	vm_5
*	192.0.2.3	22	vm_6

(a)

table T_0

ip_dst	tcp_dst	goto
192.0.2.1	80	T_1
192.0.2.2	443	T_2
192.0.2.3	22	T_3

table T_1

ip_src	out
0/1	vm_1
1/1	vm_2

table T_2

ip_src	out
00/2	vm_3
01/2	vm_4
1/1	vm_5

table T_3

ip_src	out
*	vm_6

(b)

table T_0

ip_dst	tcp_dst	wr_meta
192.0.2.1	80	1
192.0.2.2	443	2
192.0.2.3	22	3

↓

table T_1

meta	ip_src	out
1	0/1	vm_1
1	1/1	vm_2
2	00/2	vm_3
2	01/2	vm_4
2	1/1	vm_5
3	*	vm_6

(c)

table T_0

ip_dst	tcp_dst
192.0.2.1	80
192.0.2.2	443
192.0.2.3	22

↓

table T_1

ip_dst	ip_src	out
192.0.2.1	0/1	vm_1
192.0.2.1	1/1	vm_2
192.0.2.2	00/2	vm_3
192.0.2.2	01/2	vm_4
192.0.2.2	1/1	vm_5
192.0.2.3	*	vm_6

(d)

Figure 1: Cloud gateway & load-balancer pipeline, ingress direction: (a) universal table T_0 and equivalent decompositions chained with (b) with goto instructions, (c) explicit metadata tags, and (d) repeated matches on the ip_dst field (the last two omitting the goto jumps).

efficient data-plane programmability and observability, while it has very small performance impact. We find one particular software switch [24], however, where normalization also brings 1.5× throughput improvement and significant latency reduction (§5). We summarize our findings in §6.

2 MOTIVATION

Consider the sample *cloud access-gateway & load-balancer* pipeline depicted in Fig. 1a, routing different tenants' services available on specific public IP address/TCP port combinations to the VMs running the corresponding workload.

Packets destined to the web service of tenant 1 at IP address 192.0.2.1 and TCP port 80 are handled by one of the first two entries. Packets whose source IP address matches 0.0.0.0/1 are sent to vm_1 (first entry in Fig. 1a, source address prefix is marked in binary notation) and the rest, matching 128.0.0.0/1, to vm_2 (second entry), distributing load roughly evenly between the backends. Similarly, entries 3–5 distribute load for the second tenant's HTTPS service at 192.0.2.2 across vm_3, vm_4, and vm_5 in proportion 1:1:2, while entry 6 simply routes SSH requests for 192.0.2.3 to vm_6 without splitting.

Packets are matched successively against the entries (order implies priority), by applying each entry's wildcard rules (ip_src, ip_dst, tcp_dst) on the corresponding bits of the packet header and, once a matching rule is found, the corresponding action (out) is executed; packets missing all entries are handled by the table's default action (drop, send to the controller, etc.). We call such a single-table pipeline the *universal match-action table representation.*

Fig. 1b specifies the same functionality, but this time decomposed into two stages of match-action tables. Here, the first table T_0 matches only the ip_dst and tcp_dst fields and then directs execution to per-tenant second-stage tables T_1, T_2, and T_3, which in turn perform load-balancing for each tenant separately and send packets to the proper backends. Fig. 1c gives the same pipeline using opaque metadata tags instead of explicit goto_table instructions to chain match-action tables, and Fig. 1d shows an alternative decomposition. We call these multi-table representations the *normal forms* of a match-action program (in particular, the second normal form) for reasons that will be made clear in the next section.

Next, we compare the universal table representation and the normal form considering various operational aspects.

Controllability. Suppose that tenant 1, identified by the IP address/TCP port pair 192.0.2.1:80, wishes to move its service from unencrypted HTTP (tcp_dst=80) to the more secure HTTPS version (tcp_dst=443). In this case, the controller needs to update both of the *two* entries that relate to tenant 1 in the universal table (the first two flows), whereas in the normal form modifying only *one* entry is enough to reach the same effect. Similarly, changing the public IP address would require two updates in the universal table and if any of these updates gets lost (either because the data-plane incorrectly implements atomic updates [12, 18] or it does not support atomic updates at all [13]) then the service may remain halfway-exposed on the new and the old IP addresses; since the same functional modification requires only one update in the normal form, the service is guaranteed to always remain in a consistent state. In general, we see that the normal form leads to a more *reactive* data plane [4, 10, 15], in that the controller can apply more changes on the normal form than on the universal table with the same effort (i.e., changing the same number of rule-action pairs).

Monitorability. Suppose now that the task is to monitor the aggregate traffic of tenant 2. This requires the installation of 3 counters into the universal table (for entries 3–5) and then to add up the readings in a separate step in the controller; in contrast, the normal form allows to monitor at a single point as all traffic of tenant 2 flows through the second entry of table T_0. In general, observing and verifying a data-plane program in the normal form requires less effort than for the single-table representation.

Redundancy. The operational benefits of multi-table representations trace back to the single reason that *the universal representation encodes the packet-processing functionality in a redundant way, whereas the normal form has all these redundancies eliminated.* In our example, the fact that "tenant 1 provides its service at 192.0.2.1:80" is encoded *twice* in the universal representation while the normal form uses only a single entry to state this fact, and the IP address/ TCP port association for tenant 3 is stated thrice in the universal table. The normal form is accordingly smaller; for our example, the universal table in Fig. 1a contains 24 match-action fields while the normalized pipeline in Fig. 1b contains only 21. In general, for N services and M backends per service the universal table contains $4MN$ fields while the normalized pipeline contains only $N(3 + 2M)$ in the decomposition of Fig. 1c, which yields roughly half the data-plane encoding size (i.e., TCAM space [21, 23]) for M large enough.

Performance. Normalization involves the decomposition of a match-action program into a hierarchy of modular tables, and thereby it may lead to substantially reduced throughput and increased latency due to the larger number of subsequent match-action tables packets need to be traced through. The general rule of thumb, analogously to database theory, is to *denormalize when performance is critical*. Our measurement studies on 4 programmable switches cast a much richer performance landscape though, revealing intricate interactions between match-action program representations and packet classification speed and latency. We defer the discussion of the measurement results to §5.

3 NORMAL FORMS FOR MATCH-ACTION PROGRAMS

In order to model pipeline transformations, we adopt the formalism from the NetKAT network programming language [1]. While NetKAT can be used to reason about network-wide packet processing policies in the "one big-switch" abstraction, in this paper we use the framework in a severely restricted setting to describe simplified per-switch *local* policies only. Note further that NetKAT also defines various constructs called "normal forms"; our use of the term will be completely different below.

In NetKAT, a packet header is represented as a record of *header fields* $f_1, \ldots, f_k$. A packet processing program contains *predicates* to filter packets (e.g., the predicate ip_dst = 192.0.2.1 would match only packets containing the destination IP address 192.0.2.1) and *policies*, which may combine predicates and *actions* (e.g., meta $\leftarrow 2$ would modify the abstract metadata field meta to 2, goto(q) would invoke the NetKAT program q on a packet, out(r) would schedule the packet to be forwarded to r, etc.) into complex packet processing programs. The combination of predicates and actions may occur using either the *sequential* composition operator $a; b$ (which applies first the expression a to the packet and then b to the result) or the *parallel* composition operator $a + b$ (which applies a and b simultaneously). Header fields and actions will be collectively called *attributes*. We assume that predicates define exact-match filters only; see [1] on how to relax this assumption.

Given a match-action program T, our task is to remove redundancy from T by equivalent transformations. We call this process *normalization* and the intermediate representations we obtain along the way will be called normal forms.

First normal form. A match-action program T is in the first normal form (1NF), if T is a set of *entries* composed into a table using the parallel operator, with each entry defined as a set of match expressions $f_i = x$ composed sequentially with a set of actions a_j taken from a fixed attribute set:

$$\begin{aligned} T = \; & (f_1 = x_{11}; \ldots; f_k = x_{1k}; a_{11}; \ldots; a_{1n}) + \\ & (f_1 = x_{21}; \ldots; f_k = x_{2k}; a_{21}; \ldots; a_{2n}) + \ldots \end{aligned} \tag{1}$$

We further require that each entry is uniquely identified by the match field values $x_{i1}, \ldots, x_{ik}$ (cf. order-independence [17]). A match-action program in the first normal form generally corresponds to the "local OpenFlow normal form" from [1] with priorities removed and the "universal table representation" from the previous sections. The latter name comes from the *universal relation assumption* in database theory, which states that any database can be designed as a single, possibly very wide table [14]. In the context of packet processing programs, this corresponds to the analogous assumption that all match-action programs can be equivalently formulated as a single universal match-action table.

In order to define the second and the third normal forms, we borrow some further terminology from database theory. A *superkey* is a set of attributes that together uniquely identify an entry in T. Note that we let keys to contain both header fields f_i and actions a_j; for instance, in the universal table in Fig. 1a (ip_src, ip_dst, out) is a superkey even though out is an action. A *key* is a minimal superkey and a *non-prime attribute* is an attribute that does not appear in any of the keys; e.g., in the above example (ip_src, ip_dst) and (out) are (minimal) keys and tcp_dst is a non-prime attribute.

Second normal form. The second normal form (2NF) requires that the pipeline be rid of certain types of redundancy. A key concept in this context is *functional dependencies*: given a match-action program T in 1NF, a set of attributes X in T is said to functionally determine another set of attributes Y (denoted as $X \rightarrow Y$) if each X value is associated with precisely one Y value in T [14]. For instance, in the universal table in Fig. 1a both ip_dst $\rightarrow$ tcp_dst and out $\rightarrow$ ip_dst are functional dependencies, but the latter is a trivial dependency because out is a superkey.

A nontrivial functional dependency in a match-action table T is a telltale sign of redundancy, since the fact that some $x \in X$ maps uniquely to some $y \in Y$ needs to be encoded as many times in T as x appears in T. Then, *a match-action program is in 2NF if it is in 1NF and there is no functional dependency from any proper subset of any minimal key to a non-prime attribute* [14]. For instance, our universal table in Fig. 1a is not in 2NF as there is a functional dependency ip_dst $\rightarrow$ tcp_dst where the left-hand side ip_dst is a subset of the key (ip_src, ip_dst) and the right-hand side is a non-prime attribute.

In order to remove the redundancy introduced by a functional dependency, the match-action table needs to be decomposed into a pipeline of tables. Different programmable data planes expose different "join" abstractions for the network programmer to compose multi-table pipelines, like the goto_table instruction in OpenFlow [26], the apply construct in P4 [5], or connecting input/output gates in BESS [13] or Click [2]. For instance, the decomposed pipeline in Fig. 1b uses the goto_table operator; Fig. 1c chains match-action tables by communicating the tenant identifier from the first stage table to the second stage in explicit metadata tags; and Fig. 1d reaches the same end by re-matching the ip_dst field in the second table. In the sequel we treat all these "join" constructs unified under the umbrella of an abstract operation $T \gg S$, with a slight abuse of the notation for the analogous high-level sequential composition operator from Pyretic [29].

Exactly how to obtain such a decomposition is not important for the moment (an initial theoretical framework will be revealed in the next section), for now it is enough to know that, given a non-trivial functional dependency, there is a systematic way to decompose a universal table irrespective of the "join" abstraction used by the underlying data plane so that the decomposed pipelines will encode equivalent semantics. Furthermore, herein we leave the intriguing question of *how* functional dependencies are defined, and known during decomposition, for further study; we merely note that dependencies may exist inherently encoded into the high-level data plane model (e.g., in our example the functional dependency ip_dst $\rightarrow$ tcp_dst is an intrinsic consequence of the way the access gateway service is defined) or they may be transient data-level

table T_0					
eth_type	ip_dst	mod_ttl	mod_smac	mod_dmac	out
0x800	P_1	1	S_A	D_1	A
0x800	P_2	1	S_A	D_2	A
0x800	P_3	1	S_B	D_3	B
0x800	P_4	1	S_A	D_1	A

(a)

table T_0		
eth_type	ip_dst	group
0x800	P_1	1
0x800	P_2	2
0x800	P_3	3
0x800	P_4	1

$\longrightarrow$

table T_1				
group	mod_ttl	mod_smac	mod_dmac	out
1	1	S_A	D_1	A
2	1	S_A	D_2	A
3	1	S_B	D_3	B

(b)

table T_0	
eth_type	mod_ttl
0x800	1

$\longrightarrow$

table T_1	
ip_dst	group
P_1	1
P_2	2
P_3	3
P_4	1

$\downarrow$

table T_2			
group	mod_dmac	out	wr_meta
1	D_1	A	M_A
2	D_2	A	M_A
3	D_3	B	M_B

$\longrightarrow$

table T_3	
meta	mod_smac
M_A	S_A
M_B	S_B

(c)

Figure 2: An L3 forwarding pipeline: (a) "universal" table T_0, (b) equivalent two-stage decomposition $T_0 \gg T_1$ violating 3NF; and (c) equivalent pipeline $T_0 \times T_1 \gg T_2 \gg T_3$ in 3NF.

dependencies that may happen in a given ephemeral data-plane configuration but may easily disappear during the next update.

Third normal form. Consider the sample L3 pipeline in Fig. 2a. The match fields {eth_type, ip_dst} check the protocol version and match on the (disjoint) prefixes P_1–P_4, while the action attributes {mod_ttl, mod_smac, mod_dmac, out} encode standard IP packet processing actions: decrement the TTL field, rewrite the source and the destination MAC addresses, and forward on some port. This single-table representation is often chosen in IP router ASICs due to its simplicity (minus the matching on eth_type that happens separately); see, e.g., [16, Section 2: L2L3 use case].

Observe that (ip_dst) is a minimal key for the L3 pipeline and all other attributes are non-prime. Further note that multiple prefixes may map to the same next-hop with the same MAC address (e.g., P_1 and P_4 to D_1). This gives rise to the functional dependency mod_dmac $\rightarrow$ (mod_ttl, mod_smac, out). Therefore, our L3 pipeline violates 2NF. Decomposition along this functional dependency over the metadata-based join semantics yields the pipeline in Fig. 2b, where packet processing actions with respect to each unique next-hop are encoded in a separate second-stage table. It is noteworthy how this decomposition reproduces the group-table abstraction in OpenFlow [26] and Broadcom' OF-DPA switches [9], or the neighbor-table in OS IP stacks [30].

Nevertheless, there remains further redundancy: groups that use the same outgoing port map to the same source MAC (e.g., group 1 and 2), yielding the functional dependency out $\rightarrow$ mod_smac. None of these attributes belongs to a minimal key, thusly the pipeline may be in 2NF. Yet, it violates the stronger notion of *3NF, which rules out transitive functional dependencies between non-prime attributes.*

The pipeline in Fig. 2c has this dependency eliminated, and hence satisfies the requirements of 3NF. Note that we organized the eth_type and mod_ttl attributes into a separate table T_0; since these attributes take the same value for each entry the join with T_0 simplifies into a Cartesian product, yielding the pipeline $T_0 \times T_1 \gg T_2 \gg T_3$. This example also demonstrates that while the $\gg$ operation is not commutative this Cartesian product $\times$ is (we could as well append T_0 at the end of the pipeline or anywhere in between) and, accordingly, that normal forms may not be unique.

Although relational database theory defines further useful normal forms (e.g., the Boyce–Codd normal form requires *all* functional dependencies to be removed), we stop at 3NF as we find this notion to capture most practical cases. The interested reader is referred to the Appendix for a discussion of an industrial use case for our normalization framework that goes beyond 3NF.

4 EQUIVALENT TRANSFORMATIONS

One of the postulates of relational database theory is that the decomposition of a relation R_{XYZ} with attributes XYZ into relations $R_{XY} \bowtie R_{XZ}$ is lossless if and only if $X \rightarrow Y$ is a functional dependency (Heath's theorem, [14]). This theorem then marks functional-dependencies as the main driver for the normalization of match-action programs, where the fact that Y depends on X is stated in a separate match-action table T_{XY}, chained before another table T_{XZ} that specifies the rest of the pipeline logics: $T_{XY} \gg T_{XZ}$.

Next, we cast the decomposition $T_{XY} \gg T_{XZ}$ using the metadata-based join abstraction (see Fig. 1c and Fig. 2 for examples). We may introduce a new "write-metadata" action A_X into T_{XY} and a new metadata match-field M_X into T_{XZ} to communicate match results for X from one table to another, which yields the NetKAT policy $(T_{XYA_X}; T_{M_XZ})$. Another alternative to implement $T_{XY} \gg T_{XZ}$ would be to simply re-match on X in the second table (as per Fig. 1d): $(T_{XY}; T_{XZ})$. This may result in larger data-plane footprint though, since X may involve matching on multiple header fields. Finally, the goto_table abstraction may also be used to encode $T_{XY} \gg T_{XZ}$ (see Fig. 1b); this join abstraction results the smallest aggregate space in general. Exactly which join abstraction to use is highly implementation specific; certain switches will work best with the metadata-based join, others may have limited support for metadata-based matches and hence will use the goto or the "rematching" strategy, while there are switches that are completely agnostic to normalization (see the next section for some initial benchmarks).

First, we consider decomposition along a functional dependency $X \rightarrow Y$ for the case when the attribute sets X and Y contain header fields exclusively. For instance, the functional dependency ip_dst $\rightarrow$ tcp_dst of Fig. 1 is of this type. The below result states that, with this assumption, decomposition along a functional dependency generates a semantically equivalent match-action program.

Theorem 1. *Let T be a match-action program in 1NF over the attributes XYZ and suppose that T contains a functional dependency*

$X \longrightarrow Y$ where X and Y are header fields. Then, the decomposition $T_{XY} \gg T_{XZ}$ is equivalent to T.

Proof. Let T be a match-action program in 1NF with the disjoint attribute set $\{X, Y, Z\}$, where X and Y are header fields. Suppose there is dependency $X \longrightarrow Y$, denoted by the function $D \colon X \mapsto Y$. Using the NetKAT axiom BA-Seq-Comm from [1], Eq. (1) can be rearranged as $T = \sum_i x_i; y_i; z_i$, where x_i and y_i are predicates and z_i are policies for each i. Then, further applying the NetKAT axioms we write:

$$
\begin{aligned}
T &= \sum_i x_i; y_i; z_i \\
&= \sum_i x_i; D(x_i); z_i && \text{(by } X \longrightarrow Y\text{)} \\
&= \sum_i x_i; x_i; D(x_i); z_i && \text{(by BA-Seq-Idem)} \\
&= \sum_i x_i; D(x_i); x_i; z_i && \text{(by BA-Seq-Comm)} \\
&= \sum_i \left(x_i; \left(\sum_{j : x_i = x_j} D(x_j) \right); x_i; z_i \right) && \text{(by KA-Plus-Idem)} \\
&= \sum_i \left(\left(\sum_{j : x_i = x_j} x_i; D(x_j) \right); x_i; z_i \right) && \text{(by KA-Seq-Dist-L)} \\
&= \sum_i \left(\left(\sum_j x_i; x_j; D(x_j) \right); x_i; z_i \right) \\
& && \text{(by BA-Contra and KA-Plus-Zero)} \\
&= \sum_i \left(x_i; \left(\sum_j x_j; D(x_j) \right); x_i; z_i \right) && \text{(by KA-Seq-Dist-L)} \\
&= \sum_i \left(\left(\sum_j x_j; D(x_j) \right); x_i; x_i; z_i \right) && \text{(by BA-Seq-Comm)} \\
&= \sum_i \left(\sum_j x_j; D(x_j) \right) x_i; z_i && \text{(by KA-Plus-Zero)} \\
&= \left(\sum_j x_j; D(x_j) \right); \left(\sum_i x_i; z_i \right) && \text{(by KA-Seq-Dist-L)} \\
&= T_{XY} \gg T_{XZ} \quad . && \text{(by Eq. (1))} \quad \square
\end{aligned}
$$

Similar ideas may be used to extend the theory to the cases when $X \longrightarrow Y$ maps predicates to actions or actions to actions. Certain cases, however, require special caution. Consider the universal table in Fig. 3a and the decomposition in Fig. 3b along a functional dependency `output` $\longrightarrow$ `vlan` (using the metadata-based join abstraction) where the left-hand side is an action and the right-hand size in a match field. In this decomposition, the first table violates the 1NF "order-independence" assumption as the first two entries both contain the same match predicate (`in_port=1`). Consequently, a packet with `in_port=1` would match *two* entries, a fact that we cannot communicate across match-action tables using our join abstractions (see [31] for concurrent joins).

This observation leads us to conclude that the area of match-action table normalization is richer than relational database normalization theory: a naïve decomposition along certain type of functional dependencies, namely dependencies $X \longrightarrow Y$ where X contains actions and Y includes predicates, does not result 1NF sub-tables, despite that relational database theory would suggest so. Such "action-to-match" dependencies are somewhat artificial though, as the whole point in a match-action program is to associate actions with matches, that is, to describe a "match-to-action" dependency. This discussion suggests that, using the ideas above, one could equivalently normalize any 1NF match-action program all the way into 3NF analogously to how relational database theory would postulate; uncovering the precise formal relations with database normalization and a mathematical proof are, however, beyond the space limits of this paper.

table T_0		
`in_port`	`vlan`	`out`
1	1	1
1	2	2
2	1	1
3	1	3

(a)

table T_1	
`in_port`	A_{out}
1	1
1	2
2	1
3	3

$\longrightarrow$

table T_2		
M_{out}	`vlan`	`out`
1	1	1
2	2	2
3	1	3

(b)

Figure 3: Decomposition on the functional dependency `output` $\longrightarrow$ `vlan` may not yield sub-tables in 1NF: (a) universal table T_0, (b) incorrect two-stage decomposition $T_1 \gg T_2$ where T_1 is not in 1NF.

5 EVALUATION

Next, we analyze the data-plane impacts of normalization. For this, we implemented the gateway & load-balancer pipeline from Fig. 1 with the universal table and the decomposed pipelines in 3 popular programmable software switches, Open vSwitch (OVS, [28]), ESwitch [24], and Lagopus [19], and a hardware NoviFlow 2128 OpenFlow switch [25] with 28x10Gbps ports. We measured the raw performance (packet rate and latency) with traffic of 64 byte-long packets, 20 random services, and 8 backends per service, using an open-source data-plane measurement framework [20, 33][1].

Reactiveness. First, we checked whether normalization indeed results in more reactive pipelines. Recall, we observed in §2 that updating a normalized pipeline needs to touch fewer table entries than updating the universal pipeline. Our measurements with the NoviFlow switch clearly reproduced this premise: compared to the nominal case (no updates) atomically *updating a random service port 100 times per second in the universal table results in $20\times$ throughput loss, whereas no performance drop is visible for the normalized pipeline* (see Fig. 4). The reason is the $8\times$ greater control plane churn in the universal representation compared to the normalized one, which takes a toll on forwarding speed. Meanwhile, we see minor latency increase for normalization (roughly 25%), mostly independently from the control plane churn. Note that operational switches may experience similar, or larger, control plane churn during update bursts [8].

Static performance. The nominal performance results (no runtime updates) are given in Table 1.

First, we see that Lagopus and OVS are agnostic to normalization. For OVS, the reason is that the datapath collapses OpenFlow tables into a single flow cache [28]; in other words, OVS explicitly denormalizes the pipeline prior to encoding it into the datapath.

[1]The configuration for each of the evaluations can be found at https://github.com/hsnlab/tipsy/tree/master/module/gwlb.

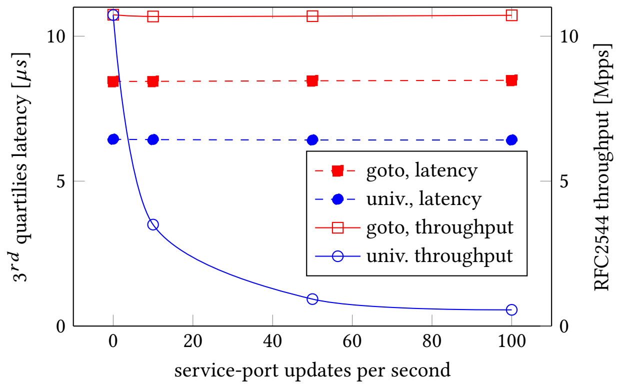

Figure 4: Reactiveness on the gateway & load-balancer pipeline, Noviflow switch: universal table and normalized representation using goto_table based join.

Table 1: Static performance on the gateway & load-balancer pipeline with the universal table (Fig. 1a) and the normalized pipelines using goto_table based join (Fig. 1b). Packet rate is specified in [Mpps] and delay is specified as the 3^{rd} quartiles latency [μs].

	OVS		ESwitch		Lagopus		Noviflow	
	Rate	Delay	Rate	Delay	Rate	Delay	Rate	Delay
universal	4.7	426	9.6	426	1.4	731	10.73	6.4
goto	4.8	422	15.0	247	1.4	728	10.74	8.4

The NoviFlow switch produces line-rate throughput irrespectively of the particular representation, with a slight latency impact due to the longer pipeline.

For ESwitch, however, we see completely different results: using the goto_table join on the normal form improves raw throughout by more than 50% and, at the same time, latency halves. This stems from the datapath architecture of ESwitch, which carefully instantiates each match-action table with the most efficient packet classifier template possible. The universal table can be encoded only with the slowest wildcard matching template, leading to poor performance and large delay. In the decomposed pipeline, however, the first table will be compiled to the very fast exact-match template and the second table to an efficient longest-prefix-matching template, resulting in substantial throughput boost and significant latency reduction. In general, *decomposition usually yields much simpler match-action sub-tables* (less fields, simpler match-type), *which is expected to transform into improved performance* on a data plane device that can dynamically optimize the datapath to the given pipeline representation.

6 CONCLUSIONS

In this paper, we presented a formal framework for the equivalent transformation of match-action programs between single-table and multi-table representations. We showed that match-action programs can be normalized into a form that is guaranteed to be free from certain type of redundancies, which we described using the notion of functional dependencies. We argued that these irredundant normal forms simplify the control-plane–data-plane interface, which transforms into substantial performance boost when control plane churn is of concern. On the other hand, the denormalized form may be more efficient on static workloads, since it generally results smaller latency. We found, however, that on switches that can dynamically optimize the data-plane to the pipeline representation (ESwitch) normalization may result considerable performance boost.

We have stopped redundancy-elimination at the third normal form. Database theory, however, recognizes several normal forms that go beyond 3NF by removing so called *multi-valued dependencies* from relations. Understanding the landscape beyond 3NF in match-action programs is currently a compelling open research problem; the Appendix shows an interesting industrial use case for such future research.

ACKNOWLEDGEMENTS

G. Rétvári has been supported by project no. 123957, 129589 and 124171 with the support provided from the National Research, Development and Innovation Fund of Hungary under the FK-17, KH-18 and K-17 funding schemes. G. Rétvári and F. Németh are also with the Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, and HSNLab.

REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. Technical report, Cornell University, 2013. available online: https://ecommons.cornell.edu/bitstream/handle/1813/34445/netkat-tech-report.pdf.
[2] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.
[3] R. Bifulco and G. Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *IEEE HPSR*, 2018.
[4] O. Bonaventure, C. Filsfils, and P. François. Achieving Sub-50 Milliseconds Recovery Upon BGP Peering Link Failures. In *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2005, Toulouse, France, October 24-27, 2005*, pages 31–42, 2005.
[5] 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.*, 44(3):87–95, 2014.
[6] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.
[7] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *SIGCOMM*, 2016.
[8] A. Elmokashfi and A. Dhamdhere. Revisiting bgp churn growth. *SIGCOMM Comput. Commun. Rev.*, 44(1):5–12, Dec. 2013.
[9] S. Feldman. Rocker: switchdev prototyping vehicle. *Proceedings of Netdev 0.1*, 2015.
[10] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever. An industrial-scale software defined internet exchange point. In *USENIX NSDI*, 2016.
[11] 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*, pages 551–562, 2014.
[12] J. Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. Moore, and P. Neumann. Blueswitch: enabling provably consistent configuration of network switches. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 17–27, 5 2015.
[13] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: a software NIC to augment hardware. available online: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.pdf.
[14] I. T. Hawryszkiewycz. *Database Analysis and Design*. Macmillan Press Ltd., 1991.
[15] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever. SWIFT: Predictive Fast Reroute. In *SIGCOMM*, 2017.

[16] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *USENIX NSDI*, pages 103–115, 2015.

[17] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (scalable and expressive packet classification). In *ACM SIGCOMM*, 2014.

[18] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about SDN flow tables. In *PAM*, pages 347–359, 2015.

[19] The Lagopus switch project. http://www.lagopus.org.

[20] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, 2018.

[21] A. X. Liu, C. R. Meiners, and E. Torng. TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2010.

[22] R. MacDavid, R. Birkner, O. Rottenstreich, A. Gupta, N. Feamster, and J. Rexford. Concise encoding of flow attributes in SDN switches. In *ACM SOSR*, pages 48–60, 2017.

[23] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel. Split: Optimizing space, power, and throughput for TCAM-based classification. In *ACM/IEEE ANCS*, pages 200–210, 2011.

[24] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári. Dataplane specialization for high performance OpenFlow software switching. In *ACM SIGCOMM*, 2016.

[25] NoviFlow. NoviSwitch 2128 - High Performance OpenFlow Switch, 2018. https://noviflow.com/wp-content/uploads/NoviSwitch-2128-Datasheet.pdf.

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

[27] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: taking the V out of NFV. In *USENIX OSDI*, pages 203–216, 2016.

[28] 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*, pages 117–130, 2015.

[29] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN programming with Pyretic. *USENIX ;login*, 38(5), 2013.

[30] R. Rosen. *Linux kernel networking: Implementation and theory*. Apress, 2014.

[31] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From policies to pipelines. In *ACM ICFP*, pages 11–24, 2014.

[32] A. Starovoitov and T. Herbert. eXpress Data Path, 2016. available online: https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf.

[33] Tipsy: Telco pipeline benchmarking system. https://github.com/hsnlab/tipsy.

APPENDIX

BEYOND THE THIRD NORMAL FORM

A match-action pipeline may often contain redundancy beyond what normal forms can capture. We take the example of Software-Defined Internet eXchange (SDX) as an example [10, 11]. A simplified SDX interconnects a set of member networks that exchange traffic according to their configured routing policies. In the example of Fig. 5, one IXP member A receives routes for two IP prefixes P_1 and P_2 from members C and D, where C only announces P_1 while D announces both prefixes. AS A specifies its *outbound policy* as preferring C over D for HTTP traffic for IP prefixes that are actually announced by C (i.e., P_1 only). AS C specifies its *inbound policy* as balancing ingress load across its two edge routers C_1 and C_2. The rest of the traffic follows BGP ranking where we assume D is preferred over C.

The original SDX controller would collapse all policies into a single universal match-action table T (see Fig. 5a).[2] One could try to normalize T into 3 tables T_{an}, T_{out}, and T_{in} for the announcement, outbound, and inbound policies, so that their *join* would produce the original table (see Fig. 5b):

$$T = T_{an} \underset{an.N=out.N}{\gg} T_{out} \underset{out.M=in.M}{\gg} T_{in} \quad .$$

This decomposition belongs to the *fourth and the fifth normal forms* as it cannot be derived from functional dependencies alone.

[2]For simplicity, we do not model Forwarding Equivalence Classes.

table T

in port	ip src	ip dst	tcp dst	output
A	0/1	P_1	80	C_1
A	1/1	P_1	80	C_2
$*$	$*$	P_1	$*$	D
$*$	$*$	P_2	$*$	D

(a)

tab. T_{an}

ip dst	N
P_1	C
P_1	D
P_2	D

$\rightarrow$

table T_{out}

N	in port	tcp dst	M
C	A	80	C
D	$*$	$*$	D

$\rightarrow$

table T_{in}

N	ip src	out
C	0/1	C_1
C	1/1	C_2
D	$*$	D

(b)

table T_{an}

ip_dst	all	best
P_1	$\{C, D\}$	D
P_2	$\{D\}$	D

$\rightarrow$

table T_{out}

in_port	$x \in$ all	tcp_dst	N
A	C	80	C
$*$	$*$	$*$	best

$\rightarrow$

table T_{in}

N	ip_src	out
C	0/1	C_1
C	1/1	C_2
D	$*$	D

(c)

Figure 5: (a) Q simplified universal SDX table, (b) the individual tables, and (c) the metadata-based pipeline.

The resultant pipeline $T_{ann} \gg T_{out} \gg T_{in}$ would however be incorrect as T_{in} is not order-independent, *i.e.*, a packet to P_1 has to choose between the 1st and 2nd entries without any knowledge whether the outbound policy in the next stage will be matched or not.

One way to communicate multiple matches across consecutive tables would be to encode the multiple match results into a metadata/header field and then let the next table match on this field, as shown in Fig. 5c where `all` is the extra field added in the pipeline. This is also the solution proposed in [10], which has been cleverly generalized in [22], and begs for a systematic approach to fully characterize its intrinsics.