Forwarding and Routing with Packet Subscriptions

Theo Jepsen
Università della Svizzera italiana

Ali Fattaholmanan
Università della Svizzera italiana

Masoud Moshref
Barefoot Networks / Intel

Nate Foster
Barefoot Networks / Intel
Cornell University

Antonio Carzaniga
Università della Svizzera italiana

Robert Soulé
Yale University
Barefoot Networks / Intel

ABSTRACT

In this paper, we explore how programmable data planes can naturally provide a higher-level of service to user applications via a new abstraction called packet subscriptions. Packet subscriptions generalize forwarding rules, and can be used to express both traditional routing and more esoteric, content-based approaches. We present strategies for routing with packet subscriptions in which a centralized controller has a global view of the network, and the network topology is organized as a hierarchical structure. We also describe a compiler for packet subscriptions that uses a novel BDD-based algorithm to efficiently translate predicates into P4 tables that can support O(100K) expressions. Using our system, we have built three diverse applications. We show that these applications can be deployed in brownfield networks while performing line-rate message processing, using the full switch bandwidth of 6.5Tbps.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**;

ACM Reference Format:
Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and Routing with Packet Subscriptions. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386367.3431315

1 INTRODUCTION

The advent of programmable data planes [6, 7, 53] is having a profound impact on networking, with clear benefits to network operators (e.g., increased visibility via fine-grained network telemetry) and to switch vendors (e.g., software development is faster and less expensive than hardware development). However, the benefits to users are still relatively unexplored, in the sense that today's programmable data planes offer the same forwarding abstractions that fixed-function devices have always provided—e.g., match on IP address, decrement TTL, and send to the next hop.

While the Internet is based on a well-motivated design [16], classic protocols such as TCP/IP provide a lower level of abstraction than modern distributed applications expect, especially in networks managed by a single entity, such as data centers. As a case in point, today it is common to deploy services in lightweight containers. Address-based routing for containerized services is difficult, because containers deployed on the same host may share an address, and because containers may move, causing its address to change. To cope with these networking challenges, operators are deploying identifier-based routing, such as Identifier Locator Addressing (ILA) [34]. These schemes require that name resolution be performed as an intermediate step. Another example is load balancing: to improve application performance and reduce server load, data centers rely on complex software systems to map incoming IP packets to one of a set of possible service end-points. Today, this service layer is largely provided by dedicated middleboxes. Examples include Google's Maglev [18] and Facebook's Katran [46]. A third example occurs in big data processing systems, which typically rely on message-oriented middleware, such as TIBCO Rendezvous [50], Apache Kafka [32], or IBM's MQ [24]. This middleware allows for a greater decoupling of distributed components, which in turn helps with fault tolerance and elastic scaling of services [19].

Although the current approach provides the necessary functionality—the middleboxes and middleware abstracts away the address-based communication fabric from the application—the impedance mismatch between the abstraction that networks offer and the abstraction that applications need adds complexity to the network infrastructure. Using middleboxes to implement this higher-level of network service limits performance, in terms of throughput and latency, as servers process traffic at gigabits per second, while ASICs can process traffic at terabits per second. Moreover, middleboxes increase operational costs and are a frequent source of network failures [44, 45]. *Given the existence of programmable devices, can't we do better?*

In this paper, we propose a new network abstraction called *packet subscriptions*. A packet subscription is a stateful predicate that, when evaluated on an input packet, determines a forwarding decision. Packet subscriptions generalize traditional forwarding rules; they are more expressive than basic equality or prefix matching and they can be written on arbitrary, user-defined packet formats. A packet subscription compiler generates both the data plane configuration and the control plane rules, providing a uniform interface for programming the network. Packet subscriptions easily express a range of higher-level network services, including pub/sub [19], in-network caching [29, 35], and identifier-based routing [34].

In some respects, packet subscriptions share a similar motivation to prior work on content-centric networking [11, 26, 31, 42]. However, in contrast to this prior work, we are *not* proposing a complete re-design of the Internet [20, 42]. Instead, we argue that

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431315

higher-level network abstractions are already used extensively by distributed applications, and this functionality can be naturally provided by the network data plane. Moreover, packet subscriptions can be implemented efficiently in controlled, data center deployments, in which the entire network is in a single administrative domain, and operators have the freedom to directly tailor the network to the needs of the applications. Packet subscriptions interoperate with other routing schemes (e.g. IP), so they are also suitable for brownfield deployments.

Furthermore, although security is an important concern for any network design, the controlled data center setting mitigates some of the most pressing security issues. Moreover, it is reasonable to assume that for applications that utilize a publish/subscribe style of communication, the data is public.

Supporting packet subscriptions as a network-level service requires addressing a series of challenges. At the network-wide level, the challenge is routing. Analogous to IP, routing on packet subscriptions amounts to placing and possibly combining rules throughout the network so as to induce the right flows of packets from publishers to subscribers. At the switch-local level, the main problem is forwarding, meaning efficiently matching packets against a set of local rules. Also at the switch-local level, there are technical challenges in efficiently parsing structured packets and allocating switch memory.

To address these challenges, we have designed a new network architecture, named Camus. Applications provide Camus with filters written in a packet subscription language. Camus provides a controller component that determines a global routing policy based on the subscriptions, and a compiler component that generates the control and data plane configurations for the local forwarding decisions that collectively realize the routing strategy.

At the routing level, Camus offers two different routing strategies. One strategy reduces the number of forwarding rules stored in switches at the expense of routing all traffic through the network core. The second strategy makes the opposite trade-off; it avoids sending traffic through the core, at the expense of greater storage requirements. Both strategies assumes a data-center network deployment, in which a centralized controller has a global view of the network, and the network topology is organized as a hierarchical structure, such as a Fat Tree or Clos architecture.

With respect to forwarding, naïvely translating packet subscriptions into FIB entries would require significant amounts of TCAM and SRAM memory, which is a scarce resource on network hardware. Instead, Camus uses an algorithm based on Binary Decision Diagrams (BDDs) [1, 9]. The Camus compiler translates logical predicates into P4 tables that can support O(100k) filter expressions within the limited resources of a programmable switch ASIC. Moreover, Camus provides functionality for parsing application-specific message formats, which requires reading deeply into the packet, and processing messages that have been batched together into a single network packet.

We have used Camus to provide communication for three applications: a financial application for filtering market feeds (i.e., the ITCH protocol provided by NASDAQ); video streaming services using Cisco's hybrid ICN (hICN) [15]; and in-band network telemetry (INT) event detection. This diversity of applications demonstrates the flexibility and expressiveness of Camus. Moreover, our prototype demonstrates substantial improvements in throughput over software based alternatives, while processing messages at line-rate.

Overall, this paper makes the following contributions:

- It introduces a high-level design of a packet subscription language targeting programmable ASICs (§2).
- It demonstrates a strategy to route via packet subscriptions in a hierarchical network topology.(§4).
- It presents an algorithm to efficiently compile packet subscription to P4 tables and control plane rules (§5).
- It describes techniques for parsing batches of application-level messages deep inside a packet (§6).
- It experimentally evaluates an implementation of in-network pub/-sub using packet subscriptions against software based alternatives (§7).

2 PACKET SUBSCRIPTIONS

A packet subscription is a filter that determines whether a packet is of interest, and therefore whether it should be forwarded to an application. So, when end-points submit a packet subscription to the global controller, they are effectively saying "send me the packets that match this filter". The following is an example of a stateless filter:

```
ip.dst == 192.168.0.1
```

It indicates that packets with the IP destination address `192.168.0.1` should be forwarded to the end-point that submitted this filter.

One can interpret this filter the traditional way: each host is assigned an IP address, and the switches forward packets toward their destinations. However, in this traditional interpretation, the network is responsible for assigning IP addresses to end-points. Instead, with packet subscriptions it is the application that assigns IP addresses. In other words, packet subscriptions empower applications with the ability to define the routing structure for the network.

Another interpretation is that the subscription is equivalent to joining a multicast group with a given IP address. However, with packet subscriptions, the IP address has no particular global meaning, and instead it is just another attribute of the packet. Applications can use other attributes for routing, and in particular they can express their interests by combining multiple conditions on one or more attributes.

For example, suppose that a trading application is interested in ITCH messages about Google stock. The following filter matches ITCH messages where the `stock` field is the constant `GOOGL` and the `price` field is greater than 50:

```
stock == GOOGL  ∧ price > 50
```

Packet subscriptions may also be stateful—i.e., their behavior may depend on previously processed data packets. To specify a stateful packet subscription, we can use operations on variables in the switch data plane, such as computing a moving average over the value contained in a header field:

```
stock == GOOGL ∧ avg(price) > 60
```

In addition to checking the `stock` field, this filter requires that the moving average of the `price` field exceeds the threshold value 60. The macro `avg` stores the current average, which is updated whenever the rest of the filter matches.

$h \in$ *Packet headers*	$n \in$ *Numbers*	
$f \in$ *Header fields*	$s \in$ *Strings*	
$v \in$ *State variables* (e.g., `my_counter`, see Figure 4)		
$g \in$ *State aggregation functions* (e.g., `avg`)		
$c ::= c_1 \wedge c_2 \mid c_1 \vee c_2 \mid !\, c \mid e$		Filter: logical expression
$e ::= a > n \mid a < n \mid a == n \mid \ldots$		Numeric constraint
$\mid a$ *prefix* $s \mid a == s \mid \ldots$		String constraint
$a ::= h.f \mid v \mid g(v_0 \ldots v_n)$		Attributes

Figure 1: Packet subscription language abstract syntax.

In general, a packet subscription is a logical expression of constraints on individual attributes of a packet or on state variables (see Figure 1). Each constraint compares the value of an attribute or a state variable (or an aggregate thereof) with a constant, using a specified relation. The Camus subscription language supports basic relations over numbers (e.g., equality and ordering) and over strings (e.g., equality and prefix).

The packet subscription language is designed to be expressive while also allowing for an efficient realization in the network [10]. In particular, the simple structure and semantics is easy to understand, since it corresponds to a very basic query language such as a subset of the WHERE clause of an SQL expression. The structure and semantics is also versatile and expressive, since it can represent non-trivial conditions over application-defined data within packets. And, crucially, subscriptions can be aggregated, using exact or approximate reductions, and then compiled into appropriate table structures for fast evaluation in network switches. As we will see later in Section 5, the aggregation and reduction algorithms exploit the simple structure of subscriptions, as well as the semantics of the numeric and string relations.

The language also supports stateful predicates, to a limited extent. First, it can only evaluate predicates that reason about local state. It cannot filter on global state (e.g., the sum of values at more than one device). Second, re-evaluating stateful predicates on multiple devices can lead to unexpected results (e.g., the average of the average of the average). Therefore, it only evaluates stateful functions at the last hop switch before a subscriber. And, third, due to the underlying hardware constraints, the types of computations it can perform is limited. For these reasons, the stateful functions that it supports are restricted to basic aggregations over tumbling windows, including count, sum, and average. This is similar to systems such as Linear Road [27] and Sonata [22].

3 NETWORK ARCHITECTURE

Adopting packet subscriptions as a new network abstraction requires that we re-think the network architecture. Figure 2 illustrates the architecture design. Subscribers express interest in messages, and publishers send messages. The switches running the Camus pipeline process the messages and forward them to interested subscribers.

Camus assumes a logically centralized controller with an omniscient view of the network (i.e., the current topology and device state). One could imagine a decentralized version of Camus, whereby each switch control plane runs the Camus compiler, and subscription information is disseminated through the network (à la conventional routing protocols). However, we leave such a design for future work.

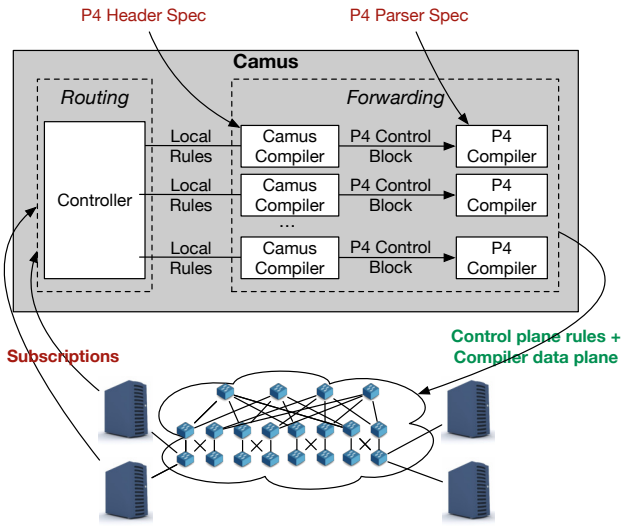


Figure 2: Overview of Camus.

Applications provide the controller with a set of filters written in the Camus subscription language. The application domain is characterized by a set of headers and corresponding packet formats. Camus requires that headers and packet formats be specified through user-provided P4 code.

The Camus controller combines the end-point subscriptions and computes a global routing policy. We assume a static, hierarchical network topology, such as a Fat Tree architecture. This architecture is common in data-center networks [2], which is our expected deployment for packet subscriptions. It also simplifies the job of the controller, as the topology naturally forms tree-structures by simply distinguishing links that go up or down the hierarchy. We discuss how routing is handled by Camus in Section 4.

To implement the routing policy, the controller emits a set of local rules that are compiled to run on the individual switches in the network. These rules determine the runtime control plane configuration of the switch, whereas the static data plane is configured once using the user-provided packet format specification. Camus relies on programmable switch hardware [7, 53] to realize an application-specific packet processing pipeline. Our prototype implementation uses the P4 compiler to program the switch.

More specifically, the Camus compiler takes the generated rules together with the P4 packet header specification, and generates two outputs: (i) a P4 control block that specifies the control-flow and match-action tables in the pipeline, and (ii) a set of control-plane rules to populate the tables. The P4 compiler then takes the P4 parser specification (packet format) and the control block generated by the Camus compiler to generate the switch image for the packet processing pipeline. We discuss the details of compiling and forwarding in Section 5.

Forwarding with packet subscriptions requires that the generated pipeline can parse application-specific message formats, which are often deep in the packet header. In Section 6, we present techniques to read deep into the packet, and process messages that have been batched together into a single network packet.

4 ROUTING ON SUBSCRIPTIONS

Routing, in general, is a complex issue that raises a number of challenges. In this paper, we evaluate two possible approaches to routing with packet subscriptions on an arbitrary hierarchical datacenter topology. These two strategies explore trade-offs between memory and traffic. Neither of these schemes is "better", in the sense that the choice between them depends on the needs and resources of the specific network.

Expected Datacenter Deployment. We assume a static, hierarchical network such as a Fat Tree [2] topology. This removes some of the complexity of routing since the topology already enforces tree-like structures in which all simple paths are shortest paths. Figure 3 illustrates an example with three levels in the hierarchy: a top-of-rack layer (ToR), an aggregate layer, and a core layer.

Considering the chosen context (datacenters), Camus currently relies on a centralized controller with a global view of the network. However, there is nothing inherent in the design of packet subscriptions that prevents the use of distributed routing protocols, like BGP or OSPF.

Routing Policies. The main task of the controller is to convert the subscriptions into a global routing policy, and then generate local rules for every switch in the network to realize the policy.

A routing policy associates each port p in a switch s with a set of filters F^s_p. Switch s forwards an incoming packet to all the ports p, other than the ingress port, such that the packet matches at least one filter in F^s_p. Figure 3 shows these associations for two policies that we discuss below. The diagrams focus on the router along a particular set of paths (a tree) taken by messages originating from an ITCH publisher on the left-hand side of the network, and going to two subscribers on the right-hand side. For each switch s along those paths, the diagrams show the local forwarding rules derived from the corresponding sets F^s_p.

Since we focus on specific paths, in the diagrams we refer to each specific port numbers. However, Camus treats the upward ports of a switch—those that link to higher-layer switches—as a single logical *up* port. For example, ports 1 in the ToR and aggregate layers are all *up* ports. When forwarding a packet to the *up* port, Camus actually chooses one of the corresponding physical ports, at random or round-robin (ECMP could be used for flow-based protocols). Also, a packet received on one of the upward ports is never forwarded to the *up* port.

As a general correctness condition, F^s_p must match a *superset* of the packets identified by the subscriptions of the hosts reachable from switch s through port p (completeness). And when port p leads directly to a host h, F^s_p must match the *exact* set of packets to which h has subscribed (soundness). Different correct policies may then differ in how precisely each set F^s_p approximates the exact set of packets that must be forwarded from s through p. Intuitively, a loose F^s_p would require fewer rules and therefore less switch memory, but would also generate unnecessary traffic.

Camus implements two policies: one that favors memory (MR) and one that favors traffic (TR), illustrated in Figure 3. In the first policy, every downward port d is associated with a set F^s_d that matches the *exact* (minimal) set of packets that are of interest to hosts reachable through d, while F^s_{up} is the *true* filter that matches every packet.

Algorithm 1: Routing in a Fat Tree network

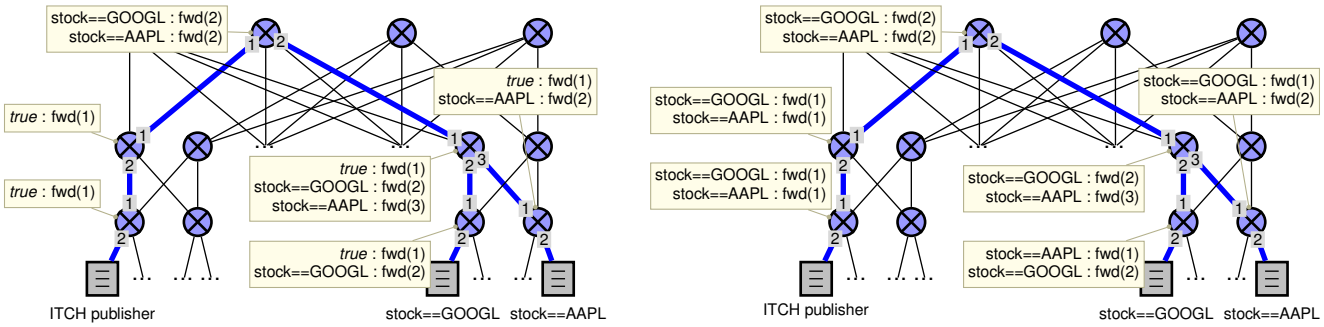
Input: $Network = (Hosts, Switches, access, up, down)$
Input: $access : Hosts \rightarrow (Ports \times Switches)$
Input: $up : Switches \rightarrow (Ports \times Switches)^*$
Input: $down : Switches \rightarrow (Ports)^*$
Input: $Subscriptions : Hosts \rightarrow Filter^*$
Output: A set of sets of subscriptions $F = \{F^s_p : s \in S, p \in P\}$

foreach *switch s and port p* **do**
  $F^s_p \leftarrow \emptyset$
foreach $h \in Hosts$ **do**
  $s, p \leftarrow access(h)$ ▷ h connects to s on port p
  $F^s_p \leftarrow F^s_p \cup Subscriptions(h)$
foreach *src* $\in$ *Switches, bottom up* **do**
  $F^{src} \leftarrow \emptyset$ ▷ local, temporary set
  foreach $p \in down(src)$ **do**
    ▷ bottom up, so F^{src}_p already computed
    $F^{src} \leftarrow F^{src} \cup F^{src}_p$
  foreach $dst, q \in up(src)$ **do** *src* connects to *dst* on *dst*'s local port q
    $F^{dst}_q \leftarrow F^{dst}_q \cup F^{src}$
if *memory policy* **then** Memory Reduction
  foreach *src* $\in$ *Switches* **do**
    $F^{src}_{up} \leftarrow \{true\}$
else if *traffic policy* **then** Traffic Reduction
  foreach *src* $\in$ *Switches* **do**
    $dst, q \leftarrow$ first up link $\in up(src)$
    $F^{src}_{up} \leftarrow \emptyset$
    foreach $p \in down(dst)$ **do**
      if $p \neq q$ **then**
        $F^{src}_{up} \leftarrow F^{src}_{up} \cup F^{dst}_q$

In the second policy, F^s_{up} also matches the exact and therefore minimal set of packets that are of interest to hosts reachable through (one of) the *up* port. The controller uses Algorithm 1 to compute the filter sets F^s_p for all switches and ports.

Filter Approximation Scheme. In addition to the somewhat crude approximation used in the memory reduction policy that simply replaces all the filters associated with an *up* port with a single *true* filter, we also develop a more refined approximation based on filter rewriting. This rewriting is specifically designed to control the amount of false positives, and at the same time to favor *aggregation* of filters. This aggregation is particularly beneficial in combination with the optimizations performed by the Camus compiler for each local switch.

The general idea of this approximation scheme is to rewrite individual constraints so as to reduce the number of *unique* constraints. One way to do that, is to discretize and therefore cluster the comparison constants used in the constraints. In particular, Camus rewrites all numeric constants as multiples of a chosen discretization unit α. For example, choosing $\alpha = 10$, Camus would rewrite constraints `price > 53` and `price > 57` as `price > 50`, and correspondingly constraints `price < 53` and `price < 57` as `price < 60`. As we show experimentally in Section 7, this simple scheme leads



Memory Reduction Policy (MR) Traffic Reduction Policy (TR)

Figure 3: Routing on subscriptions in the Fat Tree topology: routing policies.

to significant improvements in the compilation time and also in the aggregation of filters, at the expense of only a modest increase in traffic.

Once the sets of filters are computed for each link, the Camus controller turns these sets of subscriptions into an intermediate representation, which is then compiled and installed onto the switch. The intermediate representation appends a forwarding directive to the subscription filter.

Returning to the running example, if the trading application running on a server connected to port 1 of a switch is interested in ITCH messages about Google stock, then the localized rule at the last hop switch would be:

```
stock == GOOGL : fwd(1)
```

The rule asserts if the field `stock` is equal to the constant `GOOGL`, then the message should be forwarded to port 1. A forwarding action may be unicast or multicast:

```
stock == GOOGL : fwd(1,2,3)
```

In this case, messages are forwarded to ports 1, 2, and 3.

The routing schemes we describe require that predicates are re-evaluated at every switch that packets pass through. We considered an alternative design in which paths through the network would be enumerated, and predicates evaluated at the edge would attach a tag indicating that a path that the packet would travel. This design seemed like an appealing way to reduce the work done by each switch. However, since multiple predicates can match on a given packet, one would need to attach multiple path "tags". Then the switch would need to check for equality on all of these tags. So, there would not really be an advantage, we would just need to generate an equivalent rule that matched on tags instead of application header fields. Nevertheless, we expect that there should be methods to optimize the storage of rules, since there could be overlap in rules at higher levels in the hierarchy.

5 FORWARDING WITH SUBSCRIPTIONS

With local subscriptions assigned by the controller to a switch, Camus sets up the forwarding structures on that switch. The key challenge is to compile subscriptions to forwarding structures that are memory efficient and run at line-rate.

Camus compiles the rules into two steps: *static* and *dynamic*. The *static* step is performed once per application, and generates the packet processing pipeline (i.e., packet parsers and a sequence of match-action tables) deployed on the switch. The *dynamic* compilation step is performed whenever the subscription rules are updated, and generates the control-plane entries that populate the tables in the pipeline.

```
1  header itch_order {
2      bit<16> stock_locate;
3      ...
4      bit<32> shares;
5      bit<64> stock;
6      bit<32> price;
7  }
8  @pragma query_field(itch_order.shares)
9  @pragma query_field(itch_order.price)
10 @pragma query_field_exact(itch_order.stock)
11 @pragma query_counter(my_counter, 100, 1024)
```

Figure 4: Specification for ITCH message format.

This compilation strategy assumes long-running, mostly stable filters. Supporting highly dynamic filters would require an incremental algorithm. Prior work has demonstrated that such incremental algorithms are feasible. BDDs—our primary internal data structure—can leverage memoization [48], and state updates can benefit from table entry re-use [28].

5.1 Compiling the Static Pipeline

In general, a packet processing pipeline includes a packet parsing stage followed by a sequence of match-action tables. The compiler installs a different pipeline for each application, as different applications require different protocol headers, packet parser, and tables to match on header fields.

To generate the static plane, users provide a message format specification, based on data packets structured as a set of named attributes. Each attribute has a typed atomic value. For example, a particular ITCH data packet representing a financial trade would have a string attribute called `stock`, and two numeric attributes called `shares` and `price`.

Figure 4 shows the specification for the ITCH application. The message format specification extends a P4 header specification with annotations that indicate state variables and fields that will be used by the filters. In the figure, lines 8–11 contain annotations indicating that

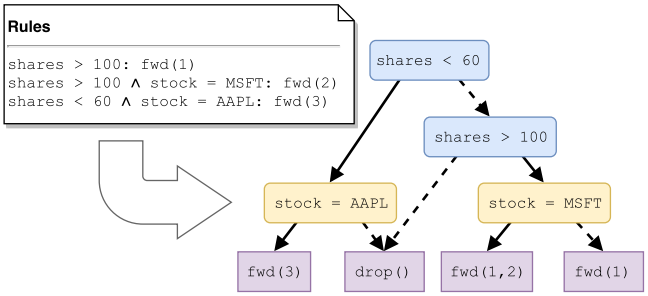


Figure 5: BDD for three rules. Solid and dashed arrows represent true and false branches, respectively.

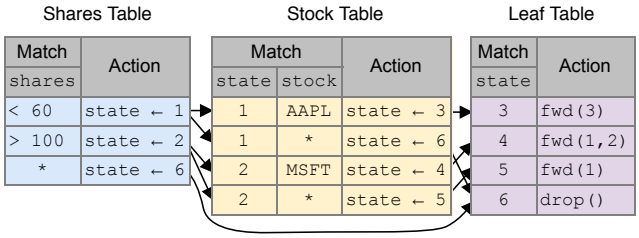


Figure 6: Table representation of the BDD in Figure 5.

the fields `shares`, `price`, and `stock` from the `itch_order` header will be used in subscriptions. Thus, the compiler should generate P4 code that matches on those fields. As an optimization, users may specify the match type. The annotation on line 10 specifies that the match should be exact by appending the suffix `_exact`. The annotation on line 11 declares a counter state variable. The first argument is the name of the counter (`my_counter`) and the second is its window size (100μs).

To support state variables, the compiler statically pre-allocates a block of registers that are then assigned to specific variables dynamically. The compiler also outputs the necessary code to update state variables in response to subscription actions at periodic intervals—e.g., to implement the tumbling window used on line 11 in Figure 4. Notice that the use (read/write) of state variable is determined by subscription rules, which are not known statically. Therefore, the static compiler outputs generic code for various update functions, and the dynamic compiler effectively links subscription actions to that code. In particular, the dynamic compiler links an update action of the general form $v \leftarrow f(args)$ with a subscription action by associating that action to what amounts to pointers to v, f, and *args*. However, the dynamic compiler in our current prototype only supports actions without arguments.

5.2 Compiling Dynamic Filters

A naïve approach for representing subscription rules would use one big match-action table containing all the rules—each rule would be encoded using a single table entry. However, this approach would be inefficient because the table would require a wide TCAM covering all headers but containing only a few unique entries per header. Furthermore, programmable switch ASICs only support matching a single entry in a table, but a packet might satisfy multiple rules. Hence, we would require a table entry for every possible *combination* of rules, resulting in an exponential number of entries in the worst case.

Instead, our compiler generates a pipeline with multiple tables to effectively compress the large but sparse logical table used by the program. To do this, the compiler represents the subscription rules using a binary decision diagram (BDD) [1, 9]. BDDs are often used to obtain compact representations of functions on a wide input domain for which a single table would be too large. A BDD is a rooted acyclic graph in which non-terminal nodes encode conditions on the input and terminal nodes encode the result (see Figure 5).

The evaluation of the overall function of the BDD that encodes all subscription rules starts at the root node and recursively evaluates the conditions (if) at each node, proceeding to the true (then) or false (else) branch as appropriate. Evaluation terminates when it reaches a terminal node (actions).

We now briefly describe the algorithm for building a BDD out of subscriptions rules. What is important for our purposes is to define the structure of the BDD, so we can implement the BDD evaluation as a sequence of table lookups.

Representing Rules with a BDD. The subscription rules are first normalized into disjunctive form, yielding a set of independent rules in which the condition in each rule consists of a conjunction of atomic predicates. An atomic predicate is defined by an equals, greater-than, or less-than relationship between a field and a constant. For example, the rules in Figure 5 are in disjunctive normal form. The compiler then builds the BDD incrementally by evaluating the condition at each node using the Shannon expansion and adding nodes for the predicates in the condition as needed.

The compiler reduces the BDD using a combination of standard and domain-specific transformations. (i) If two nodes are isomorphic, one is deleted. The incoming edges of the deleted node are updated to point to the remaining copy. (ii) If both outgoing edges of a node point to the same successor, then that node is deleted. The incoming edges of the deleted node are updated to point to the successor. (iii) If any ancestors n' of a new node n implies that n is always true or always false, then n is not added; instead, it reduces to a direct connection to its true or false branch, respectively. The overall effect is to share common structure and remove redundant nodes and unsatisfiable paths [14].

As is standard in ordered BDDs, the conditions in the BDD are arranged in a fixed order. For example, every path in the BDD of Figure 5 consists of a sequence of atomic predicates such that the conditions on field `shares` precede the conditions on field `stock`. This is essential for the representation and evaluation of the BDD as a sequence of table lookups, as we discuss next. The choice of an order can significantly impact the size of a BDD. Determining an optimal field order is NP-hard, but simple heuristics often work well in practice.

BDDs to Tables. The BDD can be seen as a state machine, where each state corresponds to a predicate, and the transition function is the evaluation of the predicate on the input packet. However, this naïve evaluation would require an excessively long sequence of evaluation steps. We instead implement BDD evaluation using a fixed-length pipeline.

Algorithm 2: Translating BDD to Tables

Input: The BDD graph, G
Output: A set of tables $T_f : state \times dom(f) \rightarrow state$

```
foreach field f do
    C_f ← subgraph of G predicating on field f
    In ← {n ∈ C_f with in-edges from outside C_f}
    Out ← {n ∉ C_f with in-edges from C_f}
    foreach path p = (u ∈ In, ..., v ∈ Out) in C_f do
        range ← ⊤                ▷ all allowable values for field f
        foreach node n ∈ p do
            range ← range ∩ predicate(n)
        T_f ← T_f ∪ {(u, range) ↦ v}
```

Since every path in the BDD traverses predicates that consider fields in order, and that order is the same for every path, we use that ordering to effectively slice the BDD into a fixed number of field-specific components. Each component is a subgraph of the BDD that contains all and only those nodes that predicate on a particular field. By extension, we also consider the set of terminal nodes as a component. For example, the BDD in Figure 5 has three components consisting of the blue, yellow, and red nodes, corresponding to the `shares` and `stock` fields, and to actions, respectively.

We can now consider the evaluation of the BDD as a state-machine at the level of the field-specific components. Thus the transition function out of the component of field f depends on the value of field f in the packet. However, since the component of field f is a macro-state corresponding to potentially many states of the BDD, the transition function must also depend on the BDD state in which we enter the component. This entry BDD state and the value of field f are necessary and sufficient to determine the path through the component of field f and therefore the transition function for that component. We represent this transition function as a match-action table where we match on the entry state and on the value of field f, and where the action points to the next component and BDD state.

Figure 6 shows all the component-specific match-action tables corresponding to the transition functions for the BDD of Figure 5. The three tables also define the three-stage processing pipeline. The evaluation through the pipeline stores the current BDD state in metadata. The initial state is set to 0 and can be omitted entirely from the first table. The actions define the entry state for the next stage, except for the *Leaf* table where the action corresponds to the overall evaluation of the BDD. For example, the rightmost path through the BDD in Figure 5 corresponds to the path through the 2nd, 4th, and 3rd entries of the *Shares*, *Stock*, and *Leaf* tables in Figure 6.

Notice that it is possible for multiple rules to match the same packet. For example, in Figure 5, the first two rules could match the same packet, so the actions `fwd(1)` and `fwd(2)` are merged into the single action `fwd(1,2)`. The compiler translates this to forwarding to a multicast group that comprises ports 1 and 2.

We compute the transition tables with Algorithm 2. In essence, for each field-specific component C_f in the BDD, Algorithm 2 identifies a set of *In* nodes within C_f that are the destinations of all the edges that enter C_f from components of preceding fields, and a set of *Out* nodes outside C_f that are the destinations of all the edges that exit from C_f to components of succeeding fields. Then Algorithm 2 computes the transition table by iterating over all the paths that connect *In* and *Out* nodes. In general, a BDD could have an exponential number of such paths. However, the domain-specific optimizations we use guarantee that there is at most one path between any pair of *In* and *Out* nodes, which in turn guarantees that the number of paths is at most quadratic.

Resource Optimizations. One of the scarce resources in switching ASICs are TCAM memories that allow matching on a subset of bits in headers but consume large area of die and high power. The compiler uses three techniques to reduce TCAM usage. First, by default the compiler generates P4 code that implements range matches, which usually require an expensive TCAM lookup. However, the user can guide the compiler by specifying a matching type for each field that may not require a TCAM lookup. Second, matching on a range in TCAM is not scalable to hundreds of thousands of ranges as each range-match requires multiple TCAM entries ($O(\#bits)$). To cope with this, the compiler uses exact matches instead of range when possible, allowing it to leverage SRAM while saving TCAM. Third, some fields, like `shares`, will probably have only a few unique range predicates. The compiler can map values for that field and the corresponding range predicates onto a lower-resolution domain (e.g., 8-bits).

6 EFFICIENT PACKET PARSING

There are two major challenges that Camus must address with respect to packet parsing: (1) generating multiple copies of a packet with different subsets of messages, and (2) parsing deep inside the packet to handle all application messages.

There are three key observations about the functionality of the hardware that we leverage for efficient parsing. First, a switch may advance arbitrarily deep in a packet in the parsing stage if the packet bytes are not written to the Packet Header Vector (PHV) that is sent through the programmable pipeline. Second, packets may only be replicated in the cross-bar between the ingress and egress pipelines. Third, a number of ports may be dedicated as recirculation or loop-back ports, which re-send packets back through the pipeline. Recirculating a packet effectively increases the depth of the processing pipeline, allowing for additional processing at the expense of slightly increased latency and reduced overall throughput.

Per-subscriber Subsets of Messages. To send different subsets of a message to different destinations, Camus uses the following strategy. First, the ingress pipeline creates a port mask that indicates which messages should be sent to which port on a switch. Next, in the crossbar, Camus replicates the packet, creating a separate copy for each output port. Finally, at egress, Camus uses the port mask to prune different messages from each of the replicas, filtering appropriately for each port. To avoid sending the port mask from ingress to egress naïvely, which would add bandwidth overhead, Camus uses a domain-specific optimization. It stores the mask in an unused packet header field (e.g. `ethernet.srcAddr`, which will be overwritten at the end of the pipeline anyway).

Parsing Deep. Since hardware switches have limited memory for carrying packet data through the programmable pipeline, it may not be possible to parse all application-level messages in a single pass. To parse deep, Camus processes the messages in multiple parallel

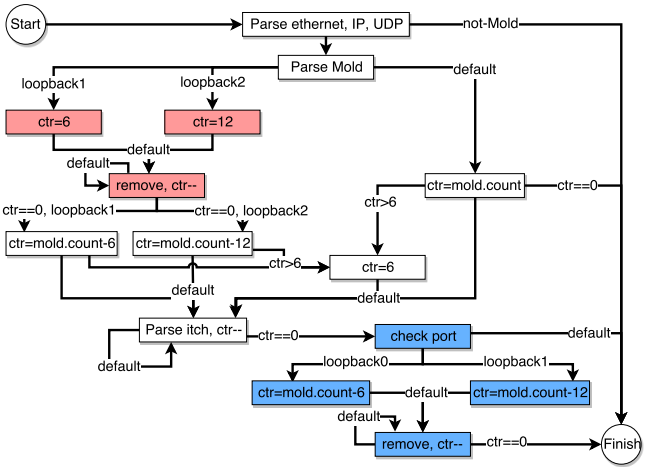


Figure 7: ITCH ingress parser for 3 loopback ports.

passes after the first pass. In the first pass, it multicasts packets on recirculation ports to make multiple copies. When the copies return to ingress, each recirculation port starts parsing at different offsets of each copy. This technique is implemented in a parser loop: the red boxes in Figure 7 show that in each iteration the parser matches on a counter; updates the counter; and shifts the parser buffer without extracting any headers. Then, the parser reads messages and finally truncates the outgoing packets using another loop to remove the messages that cannot be parsed (blue boxes). This design is also extensible to multiple recirculation passes.

7 EVALUATION

The central thesis of this paper is that the network can and should provide efficient pub/sub communication. To evaluate this claim, we have deployed three applications on a network running our packet subscriptions prototype, Camus. We relate our experiences running the applications to demonstrate qualitatively that packet subscriptions are expressive and beneficial. The evaluation focuses on operational aspects, demonstrating that the abstraction can be practically and efficiently implemented and deployed.

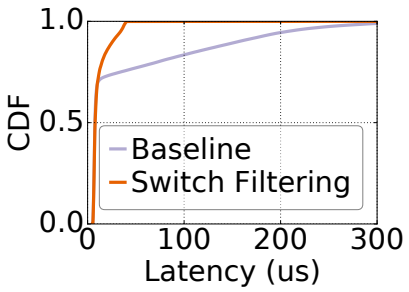
Overall, the evaluation focuses on three main questions:

(1) Do applications benefit from the use of packet subscriptions?
(2) Are packet subscriptions efficient, in terms of performance and memory?
(3) Is routing with packet subscriptions efficient, in terms of traffic load and FIB memory?

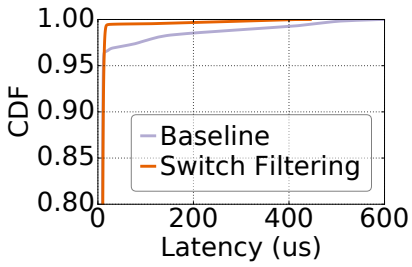
Implementation. The evaluation uses our prototype compiler implementation, which was written in OCaml, and is publicly available [1]. The compiler parses the application specifications written in $P4_{16}$ using the p4v library [36], patched to support our custom annotations. We use our own implementation of a multi-terminal BDD library with reduction optimizations.

Experimental Setup. We ran Camus in our cluster with three 32-port Barefoot Tofino switches and four servers, with varying topologies. Each server was running Ubuntu 16.04 on 12 cores (dual-socket Intel Xeon E5-2603 CPUs @ 1.6GHz), 16GB of 1600MHz DDR4 memory, and Intel 82599ES 10Gb or Mellanox ConnectX-5 100Gb Ethernet controllers.

[1] https://github.com/usi-systems/camus-compiler



(a) Nasdaq trace, 1 msgs/pkt



(b) 5% GOOGL, 1-12 msgs/pkt

Figure 8: ITCH experiments with two different workloads.

7.1 Market Data Filter

Financial exchanges, such as the Nasdaq stock market and the Chicago Mercantile Exchange, publish price and trade-related information in market data feeds. Different exchanges use different message formats. Nasdaq publishes data in the ITCH format.

ITCH data is delivered to subscribers as a stream of IP multicast packets, each containing a UDP datagram. Inside each UDP datagram is a MoldUDP header containing a sequence number, a session ID, and a count of the number of ITCH messages inside the packet. There are several ITCH message types. An `order` message indicates a new order that has been accepted by Nasdaq. It includes the stock symbol, number of shares, price, message length and a buy/sell indicator. In the descriptions below, packet subscriptions can refer to fields in the traditional header stack, or in the application-specific message format.

The first application is an implementation of the Nasdaq ITCH Market data feed filter and router. The feed is delivered as a stream of IP multicast packets. The switch splits ITCH packets into multiple messages, and forwards them to back-end servers based on the subscription.

We evaluated latency in the context of processing ITCH messages. In the experiment, a publisher sends a feed of ITCH messages to a subscriber that filters the feed for *add-order* messages with

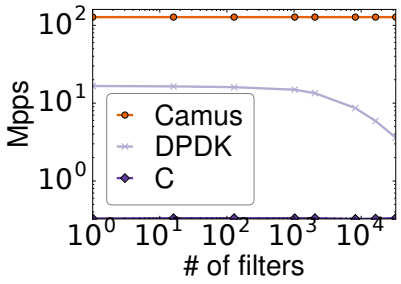


Figure 9: Filtering INT packets from a 100G link.

stock symbol GOOGL. We measured the end-to-end latency, between publication and delivery.

The publisher and subscriber (both using DPDK) are collocated on a server for accurate timestamping, and communicate using separate NICs connected via the switch. We ran the experiment in two configurations: in the baseline configuration, the subscriber performs the filtering; in the second configuration, the filtering is done on the switch with Camus. We used two workloads: a Nasdaq trace from August 30th 2017 and a synthetic feed with multiple ITCH messages per packet (Zipf distribution). The number of messages of interest (i.e. for GOOGL) is 0.5% of the Nasdaq trace, and 5% of the synthetic feed. We sent the feeds at 8.25 Mpps, which is 90% of the maximum filtering throughput of the subscriber. To be clear, Camus can run at a much higher throughput, but we slowed down the feed rate for the subscriber.

Figure 8 shows the latency CDF for both workloads. For the Nasdaq trace, all messages arrived within 50us with Camus, compared to 300us for the baseline. For the synthetic workload, 99.5% of the messages arrived within 20us with Camus, compared to 96.5% with the baseline. Overall, filtering messages on the switch with Camus reduces the tail latency, allowing applications to meet their latency requirements under high throughput.

7.2 Network Telemetry Analytics

One recent approach to network monitoring is to collect fine grained statistics on every packet that passes through a switch [23, 25]. Once the data is collected, it is sent to an analytics system for processing, such as Barefoot Networks Deep Insight [5] or Broadcom's BroadView Analytics [8]. These analytics systems are usually built following the Lambda architecture design [37] (e.g., Spark [55] for anomaly detection, Cassandra [33] for storage, and Kafka [32] for communication), which requires each of the components to scale out to cope with the input load. We used Camus to filter and route interesting (i.e., anomalous) events from the stream of in-band network telemetry (INT) data. For example, the subscription can select events indicating flows that experience high latency. In this application, Camus performs the work normally done by Kafka and Spark. It is also worth mentioning that while systems like Kafka and Spark offer features not provided by Camus (e.g., fault tolerance), for high-throughput analytics processing, these features are often disabled to achieve better performance.

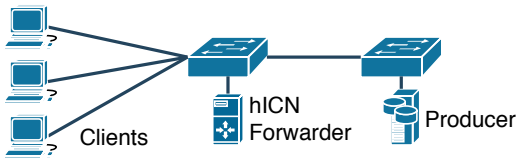


Figure 10: Topology for hICN video streaming experiment.

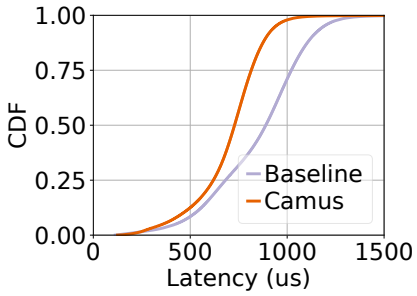


Figure 11: Lower tail latency for uncached content in hICN.

For the INT filtering application, we compared with a C program running in userspace and a C program using DPDK, both running on a commodity server. To generate a high-bandwidth packet stream, as one would expect when collecting data from a realistic datacenter, we used a switch to generate a stream of INT packets on a 100G link. We used filters that match less than 1% of the packets. The filters check that the INT packet passed through a switch with a latency above a threshold, for example:

```
int.switch_id = 2 and int.hop_latency > 100
```

Figure 9 shows the results. DPDK has better performance than the plain C program, but is fundamentally limited by the CPU clock speed: at 1.6GHz, spending about 100 instructions per packet, DPDK can process 16 Mpps. Camus, on the other hand, processes the whole 100G stream at line rate. Moreover, we found that the software based filtering does not scale with the number of filters: the latency for DPDK drastically increases after 10K filters. Camus installs the filters in hardware memory, so it has low latency, regardless of the number of filters.

7.3 Streaming Video

Video streaming is a powerful use case for pub/sub communication, since there is a single publisher and an unknown number of subscribers. Because many subscribers want the same content, network bandwidth can be reduced by caching copies in the network.

Prior work on Information Centric Networking (ICN) made this same observation, and several systems have implemented some combination of pub/sub communication and in-network caching [13, 30]. Notably, Cisco has recently developed an ICN-style network architecture to address the problem of streaming to clients in unknown locations in 5G networks [17]. Their system, named hybrid ICN (hICN) [40] embeds a content identifier in an IP address, allowing content to be routed in a heterogeneous (hybrid) network of standard L2 hardware switches and software-based hICN forwarders.

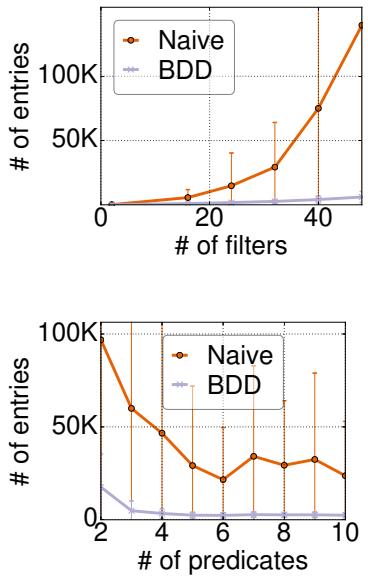


Figure 12: Compiler BDD memory efficiency

The hICN forwarders serve as content caches. While the caches can reduce latency and improve bandwidth utilization, they are only effective for "hot" content that is likely to reside in the cache. If there are many cache misses, then the software-based forwarder can become a bottleneck.

The design of the system could be improved if packets were only sent to a forwarder if a cache hit were likely. We used Camus to implement this improved design. We wrote Camus subscription filters that refer to meter state and content identifier. The filters only route packets to the software forwarders if the meter rate exceeds a threshold (i.e., a cache hit is likely). Otherwise, packets are sent upstream through other hardware switches, on the path to the original producer of the content.

There are two benefits of our approach: it reduces the load on the hICN forwarder and reduces the latency for "cold" content by bypassing the cache. To evaluate this claim, we deployed the VPP/D-PDK [51] implementation of hICN with the topology in Figure 10. On two clients we ran the hICN performance measurement tool, `hiperf`, to stream content for the same identifier, while in parallel another client pulled content for many different identifiers, which are unlikely to be cached. First we ran a baseline IPv6 setup where all the requests from the clients pass through the hICN forwarder, which processes packets at about 3.5 Gbps. Then, we ran with the Camus stateful predicates, which only sends "hot" requests to the forwarder. Figure 11 shows the latency for receiving data that is unlikely to be cached. Camus reduces the 95th percentile latency by 21%, because it detects requests for uncached data, and thus avoids the latency of the forwarder. Furthermore, the reduced load on the hICN forwarder allows it to stream to the other clients at a throughput 3% higher than the baseline.

	SRAM	TCAM	MCast Groups
ITCH (100K filters)	3.7%	4.17%	46%
INT (100K filters)	0.73%	3.45%	0%
hICN (1M filters)	74.9%	0.69%	0%

Table 1: Switch resource usage for three applications.

7.4 Efficiency of Forwarding

We evaluate the efficiency of forwarding with Camus in terms of performance and memory usage.

Performance. Camus can support the full switch bandwidth of 6.5Tbps. The latency of the pipeline, which depends on the application, is less than 1μs.

Memory. We compare the memory usage of our compiler to a baseline of naïvely representing all the filters in one big table (see Section 5). One big table has entries for all permutations of overlapping queries; workloads with similar queries cause an explosion in the table size. We chose this baseline because it is an intuitive representation of query forwarding logic. We generated workloads using the Siena Synthetic Benchmark Generator [47], which has been used to evaluate prior work in pub/sub systems [12]. Figure 12 shows the total size of the tables (number of entries times entry width) as we vary: (a) the number of subscriptions; and (b) the selectiveness of subscriptions (number of predicates).

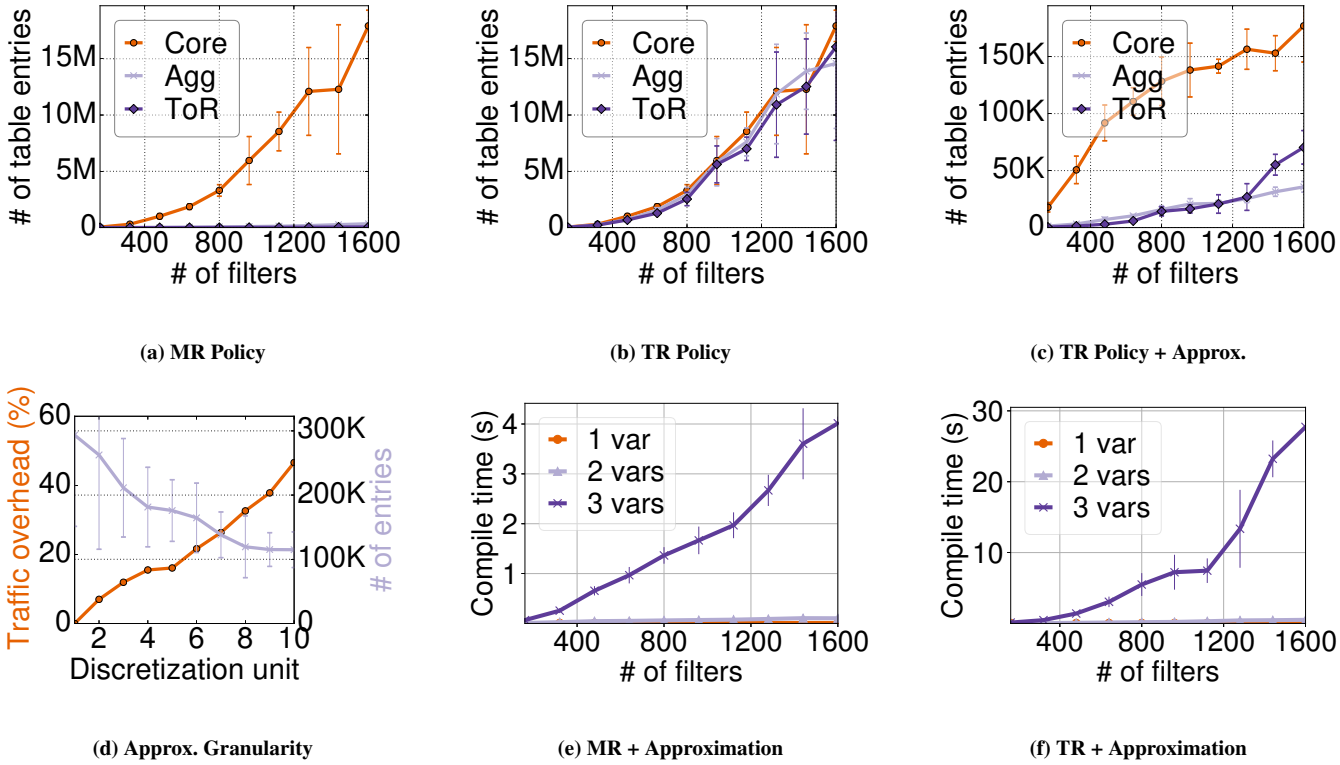
Given the low growth rate of table entries as workloads become more complex, the experiments show that Camus uses available space effectively, especially compared to the baseline. More selective subscription conditions (i.e. more predicates per filter) require fewer entries, because they result in fewer paths in the BDD.

To understand the memory implications for applications, we measured the switch resource usage for the three applications with different filter sizes. For ITCH we generated filters of the form "`stock == S ∧ price > P: fwd(H)`", where `S` is one of a 100 stock symbols, `P` is in the range (0, 1000) and `H` is one of 200 end-hosts. For INT we generated the filters described in Section 7.2 with 100 switches and 1000 `hop_latency` ranges. For hICN (described in Section 7.3) we used 1M unique content identifiers. Table 1 shows that these applications are well within the limits of the switch resources. In fact, these applications can be deployed together, or with other other standard network functionality. ITCH is the only application that makes heavy use of multicast groups, which is because many end-hosts have overlapping filters.

7.5 Efficiency of Routing

We evaluate the routing efficiency, both in terms of memory usage and compile time, for different layers in the network.

Memory. We compare the switch memory usage at various layers of a Fat Tree topology for different routing policies and with the approximation described in Section 4. We used Mininet [38] to emulate the topology depicted in Figure 3 with 20 switches and 16 hosts that publish and subscribe to an ITCH feed. The filters were generated with the synthetic benchmark generator described in the previous section.



(a) MR Policy (b) TR Policy (c) TR Policy + Approx.

(d) Approx. Granularity (e) MR + Approximation (f) TR + Approximation

Figure 13: Switch memory usage and compile time for two routing policies and approximation.

Figures 13a and 13b show the memory usage for the two policies with an increasing number of filters, where each filters checks three variables. With the MR policy, the Agg and ToR layer only need to store southbound filters, which uses less memory. On the other hand, the TR policy requires storing the filters from the whole network. The TR policy uses more memory but can utilize the bandwidth more efficiently which results in less congested links in the network.

Figures 13a and 13c show that memory usage is reduced by by discretizing the filters while aggregating filters at different layers in the network. Although this approximation reduces both compile time and memory usage, it causes extra traffic in the network. Figure 13d illustrates the correlation between the discretization unit (α) and the percentage of extra traffic forwarded in the core layer of the network.

Dynamic Reconfiguration. When subscriptions change, or if there is a change in network topology (e.g., caused by a link or switch failure), the runtime table entries need to be recompiled. We evaluate how the compile time of runtime table entries is affected by the number of subscriptions and the number of variables each subscription contains. Figure 13e shows the time to recompile the BDD tables for the MR policy with $\alpha = 10$. These results are similar for TR (Figure 13f) and are two orders of magnitude faster than without approximation (i.e. $\alpha = 1$). The bottleneck is compiling the ToR layer, since it stores all the original (i.e. unapproximated) subscriptions. Whereas the MR policy only needs to recompile entries for 8 out of the 20 switches, the TR policy must recompile entries for all switches. Moreover, the compile time is negligible for 1 to 2 variables, which is a reasonable number of variables for applications like INT and ITCH.

8 RELATED WORK

Packet subscriptions can be seen as a domain-specific language for networking. Another perspective is to see packet subscriptions as a network-level implementation of publish/subscribe messaging system.

Network Programming Languages. Several languages support the configuration of networks of programmable switches, including Frenetic [21], Pyretic [39], Merlin [49], and NetKAT[3]. Packet Subscriptions differs from this work in that it is more than a control-plane language for network designers or administrators. In particular, it provides stateful filtering rules that realize a form of in-network processing, and therefore amount to data-plane programs. The Marple [41] language also evaluates queries in-network, but for the domain-specific application of network telemetry. The packet subscription compilation algorithm is similar to the FDD compilation algorithm by Smolka et al. [48].

Publish/Subscribe Messaging Systems. Packet subscriptions are comparable to application-level middleware messaging services, such as Kafka, ActiveMQ [4], and Siena [12]. Eugster et al. provide a comprehensive survey of pub/sub systems [19]. Packet subscriptions are also comparable to the large body of prior work on information-centric networking (ICN) [43, 52, 54]. ICN is founded on the idea of addressing data packets using symbolic names rather

than network addresses. Prior work reports throughput limits that are well below those of packet subscriptions. Also, notice that these systems implement a stateless prefix matching, which is a problem that is significantly simpler than the content-based and stateful filtering of packet subscriptions.

9 CONCLUSION

Today, networks provide a lower level of abstraction than what is expected by modern distributed applications. This paper argues that the emergence of programmable data planes has created an opportunity to resolve this incongruity, by allowing the network to offer a more expressive interface.

The core technical contributions of this paper include the design of an expressive filter language that generalizes traditional forwarding rules; a set of algorithms for routing with packet subscriptions; and techniques for compiling complex filters to reconfigurable network hardware using BDDs.

These techniques are widely applicable to a range of network services. As a demonstration, we have used our prototype controller and compiler to build a diverse set of applications, including a financial application for filtering market feeds; detecting network events using INT; and stateful forwarding of hICN streams. These applications demonstrate predictable, low-latency packet processing using the full switch bandwidth.

ACKNOWLEDGMENTS

We thank the anonymous shepherd and reviewers, who helped us improve this paper. We acknowledge the support from the Swiss National Science Foundation (SNF) (project 200021_166132).

REFERENCES

[1] S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Transactions on Computers (TC)* 27, 6 (June 1978).

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Symposium on Principles of Programming Languages (POPL)*.

[4] Apache. 2019. Apache ActiveMQ. http://activemq.apache.org/. (2019).

[5] Barefoot Networks Deep Insight 2019. Barefoot Networks Deep Insight. https://www.barefootnetworks.com/products/brief-deep-insight/. (2019).

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014).

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[8] Broadcom BroadView Analytics 2019. Broadcom BroadView Analytics. https://www.broadcom.com/products/ethernet-connectivity/software/broadview-analytics. (2019).

[9] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers (TC)* 35, 8 (Aug. 1986).

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2000. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *19th ACM Symposium on Principles of Distributed Computing (PODC)*. Portland, Oregon.

[11] Antonio Carzaniga and Alexander L. Wolf. 2001. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop Infrastructure for Mobile and Wireless Systems (IMWS)*.

[12] Antonio Carzaniga and Alexander L. Wolf. 2003. Forwarding in a Content-based Network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[13] Wei Koong Chai, Diliang He, Ioannis Psaras, and George Pavlou. 2012. Cache "Less for More" in Information-Centric Networks. In *Networking*.

[14] William Chan, Richard Anderson, Paul Beame, and David Notkin. 1997. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *International Conference on Computer Aided Verification (CAV)*.

[15] Cisco Systems 2017. *Mobile Video Delivery with Hybrid ICN*. Cisco Systems. https://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/ultra-services-platform/mwc17-hicn-video-wp.pdf

[16] David Clark. 1988. The Design Philosophy of the DARPA Internet Protocols. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[17] Michael Cooney. 2018. Cisco, Verizon take Information-Centric Networking for a real-world spin. *Network World* (March 2018). https://www.networkworld.com/article/3264650/cisco-verizon-take-information-centric-networking-for-a-real-world-spin.html

[18] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)* 35, 2 (June 2003).

[20] Darleen Fisher. 2014. A Look Behind the Future Internet Architectures Efforts. *SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014).

[21] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *International Conference on Functional Programming (ICFP)*.

[22] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[23] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[24] IBM MQ 2019. IBM MQ. https://www-03.ibm.com/software/products/en/ibm-mq. (2019).

[25] int 2017. Inband Network Telemetry (INT). https://github.com/p4lang/p4factory/tree/master/apps/int. (2017).

[26] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.

[27] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*.

[28] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, CA, 87–101.

[29] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[30] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. 2009. LIPSIN: Line Speed Publish/Subscribe Inter-networking. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[31] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A Data-oriented (and Beyond) Network Architecture. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[32] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB)*. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

[33] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010).

[34] Petr Lapukhov. 2016. Internet-scale Virtual Networking Using Identifier-Locator Addressing. https://www.nanog.org/sites/default/files/20161018_Lapukhov_Internet-Scale_Virtual_Networking_v1.pdf. (2016).

[35] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[36] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4V: Practical Verification for Programmable Data Planes. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[37] Nathan Marz. 2013. *Big data: principles and best practices of scalable realtime data systems*. O'Reilly Media. http://www.amazon.de/Big-Data-Principles-Practices-Scalable/dp/1617290343

[38] mininet 2019. Mininet. http://mininet.org. (2019).

[39] Christopher Monsanto et al. 2013. Composing Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[40] Luca Muscariello, Giovanna Carofiglio, Jordan Auge, and Michele Papalini. 2020. Hybrid Information-Centric Networking. https://datatracker.ietf.org/doc/html/draft-muscariello-intarea-hicn-04. (May 2020).

[41] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[42] Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer, Matvey Arye, Steven Y. Ko, Jennifer Rexford, and Michael J. Freedman. 2012. Serval: An End-host Stack for Service-centric Networking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[43] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue. 2014. Caesar: A Content Router for High-speed Forwarding on Content Names. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[44] Rahul Potharaju and Navendu Jain. 2013. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *ACM SIGCOMM Internet Measurement Conference (IMC)*.

[45] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[46] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer — Facebook Engineering. (May 2018). https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[47] Siena Synthetic Benchmark Generator 2019. Siena Synthetic Benchmark Generator. https://www.inf.usi.ch/carzaniga/cbn/forwarding/. (2019).

[48] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. In *International Conference on Functional Programming (ICFP)*.

[49] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.

[50] TIBCO Rendezvous 2019. TIBCO Rendezvous. https://www.tibco.com/products/tibco-rendezvous. (2019).

[51] vpp 2019. Vector Packet Processing (VPP). https://fd.io/. (2019).

[52] Yi Wang, Boyang Xu, Dongzhe Tai, Jianyuan Lu, Ting Zhang, Huichen Dai, Beichuan Zhang, and Bin Liu. 2014. Fast name lookup for Named Data Networking. In *IEEE International Symposium of Quality of Service (IWQoS)*.

[53] xpliant 2019. XPliant Ethernet Switch Product Family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html. (2019).

[54] Haowei Yuan and Patrick Crowley. 2015. Reliably Scalable Name Prefix Lookup. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[55] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM (CACM)* 59, 11 (Oct. 2016).