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 seven 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.

1 INTRODUCTION

The advent of programmable data planes [6, 7, 52] 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 [45]. A third example occurs in big data processing systems, which typically rely on message-oriented middleware, such as TIBCO Rendezvous [49], 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 abstract 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 middleboxes increase operational costs and are a frequent source of network failures [43, 44]. 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, 41]. However, in contrast to this prior work, we are not proposing a complete re-design of the Internet [20, 41]. Instead, we argue that 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. However, we believe our technique composes with encrypted data. We could add hardware to decrypt the part of the packet necessary for forwarding. Alternatively, a homomorphic scheme could be used to perform forwarding on encrypted packets.

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 over numbers (e.g., equality and ordering) and over strings ([§7]).

• 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]).

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:

\[
\text{ip.dst} \rightarrow 192.168.0.1
\]

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 \text{stock} field is the constant \text{GOOGL} and the \text{price} field is greater than 50:

\[
\text{stock} \rightarrow \text{GOOGL} \land \text{price} > 50
\]

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:

\[
\text{stock} \rightarrow \text{GOOGL} \land \text{avg(price)} > 50
\]

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

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).
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.

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, 52] 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.
Memory Reduction Policy (MR)

Traffic Reduction Policy (TR)

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

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 \( F_p^s \). 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 \( F_p^s \). 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 \( F_p^s \).

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, \( F_p^s \) 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 \( h \), \( F_p^h \) must match the exact set of packets to which \( h \) has subscribed (soundness). Different correct policies may then differ in how precisely each set \( F_p^h \) approximates the exact set of packets that must be forwarded from \( s \) through \( p \). Intuitively, a loose \( F_p^h \) 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 \( F_d^s \) that matches the exact (minimal) set of packets that are of interest to hosts reachable through \( d \), while \( F_{up}^s \) is the true filter that matches every packet. In the second policy, \( F_{up}^s \) 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 \( F_p^s \) 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 \( \alpha \). 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 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:

\[
stock == GOOGL : fwd(1)
\]
For each switch s and port p, we set 
\[ F_p^s = \emptyset \]

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

\[ \text{stock} \leftarrow \text{GOOGL} : \text{fwd}(1,2,3) \]

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 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.

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

\[ \text{stock} \leftarrow \text{GOOGL} : \text{fwd}(1,2,3) \]

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 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.

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

\[ \text{stock} \leftarrow \text{GOOGL} : \text{fwd}(1,2,3) \]

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 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.

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

\[ \text{stock} \leftarrow \text{GOOGL} : \text{fwd}(1,2,3) \]

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 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.

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

\[ \text{stock} \leftarrow \text{GOOGL} : \text{fwd}(1,2,3) \]

In this case, messages are forwarded to ports 1, 2, and 3.
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 \( \text{v} \leftarrow f(\text{args}) \) with a subscription action by associating that action to what amounts to pointers to \( \text{v} \), \( f \), and \( \text{args} \). 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 \( \text{shares} \) precede the conditions on field \( \text{stock} \). 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.

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 \( \text{shares} \) and \( \text{stock} \) 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...
Algorithm 2: Translating BDD to Tables

<table>
<thead>
<tr>
<th>Input:</th>
<th>The BDD graph, $G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>A set of tables $T_f : state \times dom(f) \rightarrow state$</td>
</tr>
<tr>
<td>foreach field $f$ do</td>
<td>$C_f \leftarrow$ subgraph of $G$ predicing on field $f$</td>
</tr>
<tr>
<td>$In_\leftarrow{n \in C_f$ with in-edges from outside $C_f}$</td>
<td></td>
</tr>
<tr>
<td>$Out_\leftarrow{n \notin C_f$ with in-edges from $C_f}$</td>
<td></td>
</tr>
<tr>
<td>foreach path $p = (u \in In, \ldots, v \in Out)$ in $C_f$ do</td>
<td>$range \leftarrow \top$ (all allowable values for field $f$)</td>
</tr>
<tr>
<td>foreach node $n \in p$ do</td>
<td>$range \leftarrow$ $range \cap$ predicate($n$)</td>
</tr>
<tr>
<td>$T_f \leftarrow T_f \cup {(u, range) \rightarrow v}$</td>
<td></td>
</tr>
</tbody>
</table>

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 if 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 $\text{fwd}(1)$ and $\text{fwd}(2)$ are merged into the single action $\text{fwd}(1,2)$. 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-bit).

6 EFFICIENT PACKET PARSEING

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 naively, 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 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.
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?

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.

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 add-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 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, 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 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 [36] (e.g., Spark [54] 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.

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, 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:

\[
\text{int.switch_id} = 2 \text{ and int.hop_latency} > 100
\]

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) [39] 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. 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/DPDK [50] 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.

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 1\µs.

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 [46], which has been used to
evaluate prior work in pub/sub systems [12]. Figure 13 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 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.

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>TCAM</th>
<th>MCast Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>hICN (1M filters)</td>
<td>74.9%</td>
<td>0.69%</td>
<td>0%</td>
</tr>
<tr>
<td>INT (100K filters)</td>
<td>0.73%</td>
<td>3.45%</td>
<td>0%</td>
</tr>
<tr>
<td>ITCH (100K filters)</td>
<td>3.7%</td>
<td>4.17%</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 1: Switch resource usage for three applications.

Figure 12: Switch memory usage for two routing policies and approximation.

Figure 13: Compiler BDD memory efficiency

Figure 14: Compile time

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 [37] 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.

Figures 12a and 12b show the memory usage for the two policies with an increasing number of filters, where each filter 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 12a and 12c show that memory usage is reduced 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 12d illustrates the correlation between the discretization unit (a) 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 14a shows the time to recompile the BDD tables for the MR policy with $\alpha = 10$. These results are similar for TR (Figure 14b) 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 Fre- netic [21], Pyretic [38], Merlin [48], 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 [40] 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. [47].

Publish/Subscribe Messaging Systems. Packet subscriptions are comparable to application-level middleware messaging services, such as Kafka, ActiveMQ [4], and Sienna [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) [42, 51, 53]. 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 hiICN streams. These applications demonstrate predictable, low-latency packet processing using the full switch bandwidth.

REFERENCES


CoNEXT ’20, December 1-4, 2020, Barcelona, Spain

A ARTIFACT

We have made the packet subscriptions compiler publicly available in the camus-compiler repository. The Camus compiler takes two inputs: a P4 specification of the application; and subscription rules (see Section 2). Camus outputs a new P4 program that performs the filtering and the table entries that represent the subscription rules.

The camus-compiler repository includes some examples in examples/. You can generate a P4 filtering pipeline by providing an application specification:

```
s./camus.exe -prog-out out.p4 examples/itch.p4
```

You can generate the table entries by providing the application specification, as well as some subscriptions:

```
s./camus.exe -rules examples/itch_rules.txt examples/itch.p4
```

You can download the Camus compiler from the camus-compiler repository.
A.1 ITCH Filtering Demo

To demonstrate packet subscriptions in action, we created a demo\footnote{https://github.com/usisystems/packet-subscriptions-demo} that uses Camus to filter an ITCH market feed in a Fat Tree topology. It runs an ITCH feed publisher and three subscribers. The subscribers have different filters, which are compiled with the Camus compiler to generate P4Runtime forwarding rules for all the switches in the topology.

To try the demo, follow the instructions in the README for cloning the repository, installing the dependencies and building the compiler. Then, you can run the demo with p4app:

```
./p4app/p4app run itch.p4app
```

This will setup a Fat Tree topology with $K = 4$, three subscribers, and one publishers. The subscription filtering rules are specified on lines 14-16 of `itch.p4app/main.py`. They are compiled to produce the table entries. Then, the subscribers are started (lines 103-105) and the publisher sends a single ITCH `add_order` message (line 108). The subscribers print the message upon receiving it. Since all three subscribers have filters that match the message, they all print the same message.