Packet Subscriptions for Programmable ASICs

Theo Jepsen[*†] Masoud Moshref[†]
Antonio Carzaniga[*] Nate Foster[§†] Robert Soulé[*†]
[*]*Università della Svizzera italiana* [§]*Cornell University* [†]*Barefoot Networks*

ABSTRACT

In this paper, we explore how programmable data planes can 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 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 compiler, we've built a proof-of-concept pub/sub financial application for splitting market feeds (e.g., Nasdaq's ITCH protocol) with line-rate message processing, using the full switch bandwidth of 6.5Tbps.

1 INTRODUCTION

The advent of programmable data planes [5, 6, 40] 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 [11], 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 (e.g., Identifier Locator Addressing [24]), that requires 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 [13] and Facebook's Katran [21]. A third example occurs in big data processing systems, which typically rely on message-oriented middleware, such as TIBCO Rendezvous [38], Apache Kafka [23], or IBM's MQ [17]. This middleware allows for a greater decoupling of distributed components, which in turn helps with fault tolerance and elastic scaling of services [14].

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 thoughput 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 [33, 34]. *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 communication [14], in-network caching [20, 25], and identifier-based routing [24].

In some respects, packet subscriptions share a similar motivation to prior work on content-centric networking [8, 18, 22, 29]. However, in contrast to this prior work, we are *not* proposing a complete re-design of the Internet [15, 29]. Instead, we argue that higher-level network abstractions are

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286092

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.

Packet subscriptions can be used to program any network hardware. Our prototype compiler targets programmable ASICs, since they are already deployed in data centers, and do not require adding additional hardware. Moreover, this paper focuses on the challenge of compiling subscriptions to a single device. Routing over multiple switches would require a different set of techniques [30].

Implementing packet subscriptions is challenging; a naïve approach would require significant amounts of TCAM and SRAM memory, which is a scarce resource on network hardware. Compilation is further complicated by the need to support application-specific message formats and stateful predicates, such as filtering based on some aggregate.

To address these challenges, we have developed a packet-subscription compiler that uses a novel algorithm based on Binary Decision Diagrams (BDDs) [1, 7]. Our compiler translates logical predicates into P4 tables that can support O(100k) filter expressions within the limited resources of a programmable switch ASIC. Using our compiler, we have built a proof-of-concept pub/sub application for splitting financial market feeds (e.g., Nasdaq's ITCH protocol). Our evaluation demonstrates message processing at line rate using the full switch bandwidth of 6.5Tbps.

Overall, this paper makes the following contributions:

- It describes a high-level design of a packet subscription language targeting programmable ASICs (§2).
- It presents an algorithm to efficiently compile packet subscription to P4 tables and control plane rules (§3).
- It experimentally evaluates an implementation of in-network pub/sub using packet subscriptions against software based alternatives (§4).

2 PACKET SUBSCRIPTIONS

To make our presentation of packet subscriptions more concrete, we introduce a running example motivated by the financial domain. Nasdaq publishes market data feeds using 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. For our experiments, we have implemented `add-order` messages, which indicate 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

$h \in$ *Packet headers*	
$f \in$ *Header fields*	
$v \in$ *State variables*	
$n \in \mathbb{N}$	
$r ::= c : a$	Condition-Action Rule
$c ::= c_1 \wedge c_2 \mid c_1 \vee c_2 \mid ! c_1 \mid e$	Logical Expression
$e ::= p > n \mid p < n \mid p == n$	Relational Expression
$p ::= h.f \mid v$	Variables
$a ::= a_1 ; a_2 \mid fwd(n_0 \ldots n_i) \mid g$	Action
$g ::= v \leftarrow f(v_0 \ldots v_j, h)$	State-update Function

Figure 1: Packet subscription language abstract syntax.

subscriptions can refer to fields in the traditional header stack, or in the application-specific message format.

Subscription Language. Packet subscriptions are filter rules that identify a packet and indicate an action. The rule:

```
ip.dst == 192.168.0.1 : fwd(1)
```

indicates that packets with the IP destination address `192.168.0.1` should be forwarded out port 1 of a switch:

One can interpret this filter rule the traditional way—i.e., each host is assigned an IP addresses, and the switches forward packets toward their destinations. An alternative interpretation of the same phenomenon is that hosts express an "interest" in subscribing to packets with a given IP address, as in pub/sub-style communication. Packet subscriptions are general enough that they can support push-based or pull-based communication, depending on whether the servers publish content and clients subscribe, or vice-versa.

Packet subscriptions can also refer to application-specific packet formats. Returning to our running example, suppose that a trading application running on the server connected to port 1 is interested in ITCH messages about Google stock. The following rule states that if the `stock` field is the constant `GOOGL`, the message should be forwarded to port 1.

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

Forwarding actions may be unicast or multicast:

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

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

The rules we have seen so far are stateless: the condition does not depend on previously processed data packets. However, packet subscriptions may also be stateful. A stateful rule may read or write variables inside the switch data plane:

```
stock == GOOGL ∧ avg(price) > 50 : fwd(1)
```

In addition to checking equality on the `stock` field, this rule requires that the moving average of the `price` field exceeds the threshold value 50. The macro `avg` stores the current average, which is updated when the rest of the rule matches.

Syntax and Semantics. Packet subscriptions are evaluated according to event-condition-action semantics. The event is the arrival of a data packet of a given format; the condition specifies a set of constraints on the values of the attributes in the data packet; and the action defines the processing of the data packet, typically consisting of a forwarding action. In

```
1  header_type itch_add_order_t {
2      fields {
3          shares: 32;
4          stock: 64;
5          price: 32;
6          ...
7      }
8  }
9  header itch_add_order_t add_order;
10
11 @query_field(add_order.shares)
12 @query_field(add_order.price)
13 @query_field_exact(add_order.stock)
14 @query_counter(my_counter, 100)
```

Figure 2: Specification for ITCH message format.

processing a data packet, the switch executes the actions of all matching rules, in no particular order.

Figure 1 shows the language syntax. Each filter rule has a *condition* and an *action*. Conditions are logical expressions that may be combined using conjunction ($\wedge$), disjunction ($\vee$), and negation (`!`). The language includes the standard relational operators (i.e., < and >). Atomic predicates of the form h.f == n denote the set of packets whose header field $h.f$ is equal to n. An action may forward packets (`fwd(1)`) or perform a computation that updates a state variable.

3 COMPILING SUBSCRIPTIONS

Compilation is divided 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.

Note that this compilation strategy assumes long-running, mostly stable queries. Highly dynamic queries would require an incremental algorithm, both to reduce compilation time and to minimize the number of state updates in the network. Prior work has demonstrated that such incremental algorithms are feasible. BDDs—our primary internal data structure—can leverage memoization [36], and state updates can benefit from table entry re-use [19].

3.1 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 must provide a message format specification. The specification is 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 2 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 12-13 contain annotations indicating that the fields `shares`, `price`, and `stock` from the `add_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 13 specifies that the match should be exact by appending the suffix `_exact`. The annotation on line 14 declares a counter state variable. The first argument is the name of the counter (`my_counter`) and the second is its window size (100us).

To support state variables, the compiler statically preallocates a block of registers that are then assigned to specific variables dynamically. The compiler also outputs the code to update state variables in response to subscription actions at periodic intervals—e.g., to implement the tumbling window used on line 14 in Figure 2. 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 implemented in our current prototype only supports actions without arguments.

3.2 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 incredibly 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, 7]. 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 (i.e., the packet headers), and terminal nodes encode the result (i.e., a set of actions). See the example in Figure 3.

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

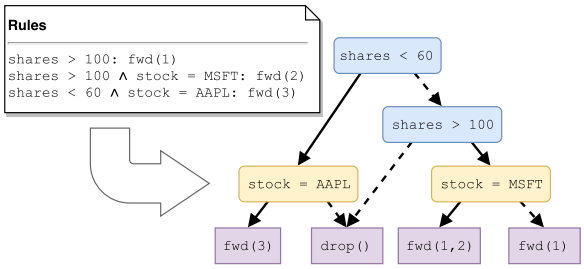


Figure 3: BDD for three rules. Solid and dashed arrows represent true and false branches, respectively.

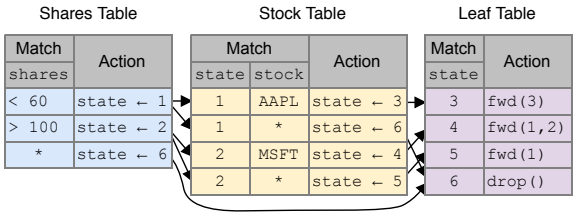


Figure 4: Table representation of the BDD in Figure 3.

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 3 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 replaced with the successor. (iii) If any ancestor 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 [10].

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 3 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.

Algorithm 1: Translating BDD to Tables

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

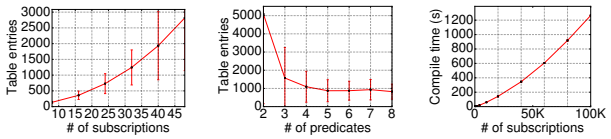
foreach *field* f **do**
  $C_f \leftarrow$ subgraph of G predicating on field f
  $In \leftarrow \{n \in C_f$ with in-edges from outside $C_f\}$
  $Out \leftarrow \{n \notin C_f$ with in-edges from $C_f\}$
  foreach *path* $p = (u \in In, \ldots, v \in Out)$ *in* C_f **do**
    $range \leftarrow \top$ ▷ all allowable values for field f
    foreach *node* $n \in p$ **do**
      $range \leftarrow range \cap$ predicate(n)
    $T_f \leftarrow T_f \cup \{(u, range) \mapsto v\}$

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 3 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 4 shows all the component-specific match-action tables corresponding to the transition functions for the BDD of Figure 3. 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 from the first table. The actions set the entry state for the following stage, except for the *Leaf* table where the action corresponds to the overall BDD evaluation. For example, the rightmost path through the BDD in Figure 3 corresponds to the path through the 2nd, 4th, and 3rd rows of the *Shares*, *Stock*, and *Leaf* tables in Figure 4, respectively.



(a) Subscriptions **(b) Predicates** **(c) Compile time**

Figure 5: Compiler efficiency

It is possible for multiple rules to match the same packet. For example, in Figure 3, the first two rules could match the same packet, so the actions `fwd(1)` and `fwd(2)` are merged into one action: `fwd(1,2)`. The compiler translates this to forwarding to a multicast group with ports 1 and 2.

We compute the transition tables with Algorithm 1. In essence, for each field-specific component C_f in the BDD, Algorithm 1 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 1 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 that are application-agnostic 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).

4 EVALUATION

We have implemented a prototype compiler in OCaml. The compiler parses the application specifications written in $P4_{14}$ using the P4V library [26], patched to support our custom annotations. We use our own implementation of a multi-terminal BDD library with reduction optimizations.

There are three parts to our evaluation: (i) we explore the space/time efficiency of the compiler; (ii) we demonstrate the efficacy of packet subscriptions by implementing an in-network publish/subscribe system; and (iii) we compare the end-to-end system latency and throughput for in-network publish/subscribe to a baseline software implementation.

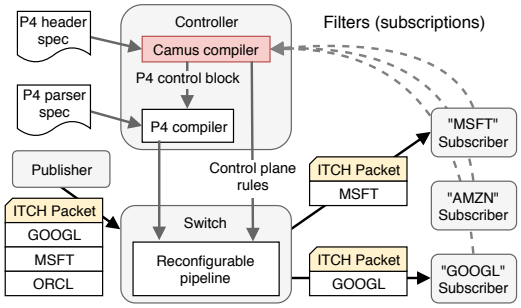


Figure 6: Overview of Camus.

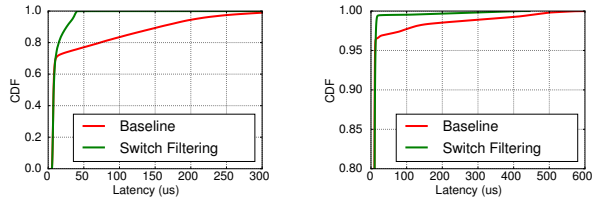
Efficiency of the compiler. To measure the space efficiency of the compiler, we generated workloads using the Siena Synthetic Benchmark Generator [35], which has been used to evaluate prior work in pub/sub systems [9]. Figure 5 shows the number of table entries required on the switch as we vary key parameters: (a) number of subscriptions; and (b) selectiveness of subscriptions (number of predicates).

Given the low growth rate of table entries as workloads become more complex (5a), the experiments show that Camus uses available space effectively. Figure 5b shows that more selective subscription conditions (i.e. more predicates in the conjunction) require fewer table entries, which is because they result in fewer paths in the BDD.

To measure our compiler's runtime, we used a synthetic workload generator to create ITCH subscriptions 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. Figure 5c shows the results. Compiling 100K subscriptions resulted in 21,401 table entries and 198 multicast groups, which can easily fit in switch memory.

Case Study: In-Network Pub/Sub. As an example use-case for packet subscriptions, we have implemented an in-network pub/sub system. Figure 6 illustrates the design of our pub/sub system, which we call Camus. Camus takes the subscription filters together with the message format 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 (i.e., the packet format) and the control block generated by the Camus compiler to generate the switch image for the packet processing pipeline. At runtime, publishers send messages. The switches running the Camus pipeline process the messages and forward them to interested subscribers.

We used our Camus pub/sub system to do in-network filtering of market data feeds. Many financial companies subscribe to the Nasdaq feed and broadcast it to all of their servers in order to execute trading strategies. Typically, each server is only interested in a very small subset of stocks. For example, one trading strategy might only depend on data related to Google stock, while another might depend on data related to Apple.



(a) Nasdaq trace **(b) Synthetic trace**

Figure 7: ITCH experiments on hardware.

Therefore, broadcasting the feed wastes resources. Moreover, broadcasting all packets to servers builds queues at switches and servers, which increases delay and the chances of packet drops. Any increase in latency can have a significant impact on the user, as high-frequency trading strategies depend on speed to gain an advantage in arbitraging price discrepancies.

Throughput and Latency. Our experimental setup resembles Figure 6, except for that the publisher and subscriber are collocated for accurate timestamping. We ran our packet subscription pipeline on a 32-port Barefoot Tofino switch, which can process packets at 3.25Tbps (on the 64-port version of the switch, we would support 6.5Tbps). The ITCH publisher and subscriber are implemented with DPDK [12], running on a server with an 8-core Intel Xeon E5-2620 v4 @ 2.10GHz CPUs, 256GB DDR4-2133 RAM and 25Gb/s NICs (Mellanox ConnectX-4 Lx and Intel XXV710-DA2). We ran the same workloads under two configurations. In the baseline configuration, the subscriber filters the feed for `add-order` messages with stock symbol `GOOGL`. In the second configuration, the filtering is done with Camus.

We used two workloads: a Nasdaq trace from August 30th 2017 and a synthetic feed. The number of messages of interest (i.e. for `GOOGL`) is 0.5% of the Nasdaq trace, and 5% of the synthetic feed. We measured the latency between the publisher sending a message with `GOOGL`, and it being received by the subscriber. Figure 7 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.

Other applications. In this paper, we have focused on one running example, financial market data. We chose this application because it demonstrates many of the requirements that motivate packet subscriptions: routing based on application-specific content, stringent latency demands, and filtering based on expressive predicates. Because packet subscriptions can be used to implement pub/sub communication, they could be used as an alternative for frameworks like Kafka or ActiveMQ [3]. However, packet subscriptions are not a one-for-one replacement. They are limited, in that they do not provide features such as reliable communication and persistence. Rather, packet subscriptions are best suited for application domains with high-throughput workloads that can tolerate some loss, such as streaming analytics. In on-going work, we are exploring other use cases, related to load-balancing, elastic scaling of services, and security. Packet subscriptions would also be a useful abstraction for in-network caching, which routes based on content identifier (e.g., NetCache [20]).

5 RELATED WORK

Packet subscriptions are related to pub/sub messaging, network languages, and information-centric networking.

Publish/subscribe messaging system. Many application-level middleware messaging services provide pub/sub communication, such as Kafka, ActiveMQ, and Siena [9]. Eugster et al. provide a comprehensive survey [14].

Network programming languages. Several languages support the control-plane configuration of switches, including Frenetic [16], Pyretic [27], Merlin [37], and NetKAT[2]. In contrast to this work, packet subscriptions provide stateful filtering rules that realize a form of in-network processing, and therefore amount to data-plane programs. Marple [28] evaluates telemetry queries in-network. BDDs have long been used as a compressed representation of relations. In networking, BDDs have been used to verify network properties [41], check network configurations [4], and optimize compilation of OpenFlow rules [36]. Our compiler also uses BDD as an efficient internal representation, but differs from this prior work in that it generates a switch pipeline configuration.

Information-centric networking. With information-centric networking (ICN), packets are routed using symbolic names rather than network addresses. Some ICN architectures support the rich pub/sub semantics of packet subscriptions [30], but the mainstream architectures (CCN and NDN) are based on a "pull" model and on a stateless prefix matching that is significantly less expressive than the content-based and stateful filtering of packet subscriptions. In any case, prior work in ICN achieves a maximum throughput that is well below the line-rate throughput of packet subscriptions [31, 32, 39, 42].

6 CONCLUSION

Today, networks provide a lower level of abstraction than what is expected by modern distributed applications. The main argument of this paper is 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 contribution of this paper is a set of algorithms for compiling complex filter expressions to reconfigurable network hardware using BDDs. These techniques are widely applicable to a range of network services. As a systems artifact, we have used these techniques to build an in-network publish-subscribe system that demonstrates predictable, low-latency packet processing using the full switch bandwidth.

REFERENCES

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers (TC)*, 27(6):509–516, June 1978.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Symposium on Principles of Programming Languages (POPL)*, pages 113–126, Jan. 2014.

[3] Apache activemq. http://activemq.apache.org/.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 155–168, Aug. 2017.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, Aug. 2013.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers (TC)*, 35(8):677–691, Aug. 1986.

[8] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop Infrastructure for Mobile and Wireless Systems (IMWS)*, pages 59–68, Oct. 2001.

[9] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 163–174, Aug. 2003.

[10] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *International Conference on Computer Aided Verification (CAV)*, pages 316–327, June 1997.

[11] D. Clark. The design philosophy of the DARPA Internet Protocols. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 106–114, Aug. 1988.

[12] DPDK. http://dpdk.org/.

[13] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, Mar. 2016.

[14] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003.

[15] D. Fisher. A look behind the future internet architectures efforts. *SIGCOMM Computer Communication Review (CCR)*, 44(3):45–49, July 2014.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *International Conference on Functional Programming (ICFP)*, pages 279–291, Sept. 2011.

[17] IBM MQ. https://www-03.ibm.com/software/products/en/ibm-mq.

[18] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 1–12, Dec. 2009.

[19] X. Jin, J. Gossels, J. Rexford, and D. Walker. Covisor: A compositional hypervisor for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–101, Oakland, CA, May 2015.

[20] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[21] Open-sourcing Katran, a scalable network load balancer. https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[22] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-oriented (and Beyond) Network Architecture. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 181–192, Aug. 2007.

[23] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, June 2011.

[24] P. Lapukhov. Internet-scale virtual networking using identifier-locator addressing. https://www.nanog.org/sites/default/files/20161018_Lapukhov_Internet-Scale_Virtual_Networking_v1.pdf.

[25] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44, Mar. 2016.

[26] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster. P4v: Practical verification for programmable data planes. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 490–503, Aug. 2018.

[27] C. Monsanto et al. Composing Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–13, Apr. 2013.

[28] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 85–98, Aug. 2017.

[29] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.

[30] M. Papalini, A. Carzaniga, K. Khazaei, and A. L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking (HotICN)*, pages 17–26, Sept. 2014.

[31] M. Papalini, K. Khazaei, A. Carzaniga, and D. Rogora. High throughput forwarding for ICN with descriptors and locators. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 43–54, 2016.

[32] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 137–148, Oct. 2014.

[33] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 9–22, Oct. 2013.

[34] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 13–24, Aug. 2012.

[35] Siena Synthetic Benchmark Generator. http://www.inf.usi.ch/carzaniga/cbn/forwarding/.

[36] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *International Conference on Functional Programming (ICFP)*, pages 328–341, Sept. 2015.

[37] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2014.

[38] Tibco rendezvous. https://www.tibco.com/products/tibco-rendezvous.

[39] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for named data networking. In *IEEE International Symposium of Quality of Service (IWQoS)*, pages 198–207, May 2014.

[40] XPliant Ethernet Switch Product Family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[41] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE International Conference on Network Protocols (ICNP)*, pages 1–11, Oct. 2013.

[42] H. Yuan and P. Crowley. Reliably scalable name prefix lookup. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 111–121, May 2015.