Language-Directed Hardware Design for Network Performance Monitoring

Srinivas Narayana[1], Anirudh Sivaraman[1], Vikram Nathan[1], Prateesh Goyal[1],
Venkat Arun[2], Mohammad Alizadeh[1], Vimalkumar Jeyakumar[3], Changhoon Kim[4]
[1] *MIT CSAIL* [2] *IIT Guwahati* [3] *Cisco Tetration Analytics* [4] *Barefoot Networks*

ABSTRACT

Network performance monitoring today is restricted by existing switch support for measurement, forcing operators to rely heavily on endpoints with poor visibility into the network core. Switch vendors have added progressively more monitoring features to switches, but the current trajectory of adding specific features is unsustainable given the ever-changing demands of network operators. Instead, we ask what switch hardware primitives are required to support an expressive language of network performance questions. We believe that the resulting switch hardware design could address a wide variety of current and future performance monitoring needs.

We present a performance query language, Marple, modeled on familiar functional constructs like map, filter, groupby, and zip. Marple is backed by a new programmable key-value store primitive on switch hardware. The key-value store performs flexible aggregations at line rate (*e.g.,* a moving average of queueing latencies per flow), and scales to millions of keys. We present a Marple compiler that targets a P4-programmable software switch and a simulator for high-speed programmable switches. Marple can express switch queries that could previously run only on end hosts, while Marple queries only occupy a modest fraction of a switch's hardware resources.

CCS CONCEPTS

• **Networks** → **Network monitoring**; *Programmable networks*;

KEYWORDS

Network measurement; network hardware; network programming

ACM Reference format:
Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098829

1 INTRODUCTION

Effective performance monitoring of large networks is crucial to quickly localize problems like high queueing latency [12], TCP incast [61], and load imbalance across network links [27]. A common approach to network monitoring is to collect information from the endpoint network stack [53, 58, 62] or to use end-to-end probes [40] to diagnose performance problems. While endpoints provide application context, they lack visibility to localize performance problems at links deep in the network. For example, it is challenging to localize queue buildup to a particular switch or pinpoint traffic causing the queue buildup, forcing operators to infer the network-level root causes indirectly [40].

Switch-based monitoring could allow operators to diagnose problems with more direct visibility into performance statistics. However, traditional switch mechanisms like sampling [7, 21], mirroring [8, 42, 65], and counting [34, 49] are quite restrictive. Sampling and mirroring miss events of interest as it is infeasible to collect information on all packets, while counters only track traffic volume statistics. None of these mechanisms provides relevant performance data, like queueing delays.

Some upcoming technologies recognize the need for better performance monitoring using switches. In-band network telemetry [12] writes queueing delays experienced by a packet on the packet itself, allowing endpoints to localize delay spikes. The Tetration switching chip [9] provides a flow cache that measures flow-level performance metrics. These metrics are useful, but they are exposed at a fixed granularity (*e.g.,* per 5-tuple), and the metrics themselves are fixed. For example, the list of exposed metrics includes flow-level latency and packet size variation, but not latency variation, *i.e.*, jitter.

Operator requirements are ever-changing, and redesigning hardware is expensive. We believe that the trajectory of adding fixed-function switch monitoring piecemeal is unsustainable. Instead, we advocate building performance monitoring primitives that can be flexibly reused for a variety of needs. Programmable switches [3, 13, 25] now support flexible parsing [39], header processing [33, 56], and scheduling [57]. Our goal is to add monitoring to this list.

This paper applies *language-directed hardware design* to the problem of flexible performance monitoring, inspired by early efforts on designing hardware to support high-level languages [36, 50, 59]. Specifically, we design a language that can express a broad variety of performance monitoring use cases, and then design high-speed switch hardware primitives in service of this language. By designing hardware to support an expressive language, we believe the resulting hardware design can support a wide variety of current and future performance monitoring needs.

Fig. 1 provides an overview of our performance monitoring system. To use the system, an operator writes a query in a domain-specific language called *Marple,* either to implement a long-running

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098829

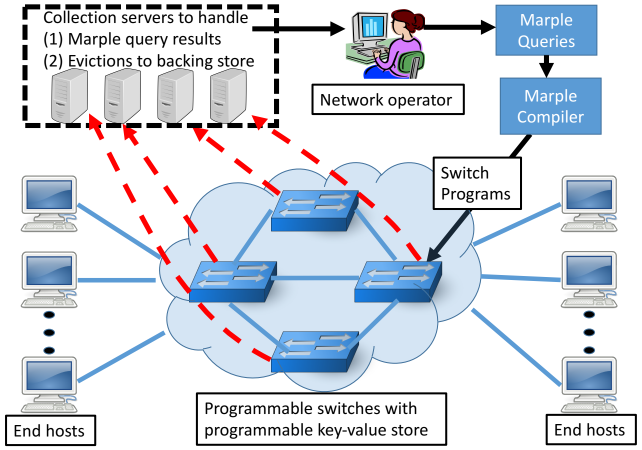

Collection servers to handle
(1) Marple query results
(2) Evictions to backing store
Network operator
Marple Queries
Marple Compiler
Switch Programs
Programmable switches with programmable key-value store
End hosts
End hosts

Figure 1: Operators issue Marple queries, which are compiled into switch programs for programmable switches augmented with our new programmable key-value store primitive. Switches stream results from this query to collection servers that also house the backing store for the key-value store.

monitor for a statistic (*e.g.,* detecting TCP timeouts), or to troubleshoot a specific problem (*e.g.,* incast [61]) at hand. The query is compiled into a switch program that runs on the network's programmable switches, augmented with new switch hardware primitives that we design in service of Marple. The switches stream results out to collection servers, where the operator can retrieve query results. We now briefly describe the three components of our system: the query language, the switch hardware, and the query compiler.

Performance query language. Marple uses familiar functional constructs like `map`, `filter`, `groupby` and `zip` for performance monitoring. Marple provides the abstraction of a stream that contains performance information for *every packet at every queue* in the network (§2). Programmers can focus their attention on traffic experiencing interesting performance using `filter` (*e.g.,* packets with high queueing latencies), aggregate information across packets in flexible ways using `groupby` (*e.g.,* compute a moving average over queueing latency per flow), compute new stateless quantities using `map` (*e.g.,* binning a packet's timestamp into an epoch), and detect simultaneous performance conditions using `zip` (*e.g.,* when the queue depth is large and the number of connections in the queue is high).

Hardware design for performance queries. A naïve implementation of Marple might stream every packet's metadata from the network to a central location and run streaming queries against it. Modern scale-out data-processing systems support 100K–1M operations per second per core [2, 4, 11, 43, 64], but processing every single packet (assuming a relatively large packet size of 1 KB) from a single 1 Tbit/s switch would need 100M operations per second — 2–3 orders of magnitude more than what existing systems support.

Instead, we leverage high-speed programmable switches [3, 13, 25, 33] as first-class citizens in network monitoring, because they can programmatically manipulate multi-Tbit/s packet streams. Early filtering and flexible aggregation on switches drastically reduce the number of records per second streamed out to a standard data-processing system running on the collection server.

While programmable switches support many of Marple's stateless language constructs that modify packet fields alone (*e.g.,* `map` and `filter`), they do not support aggregation of state across packets for a large number of flows (*i.e.*, `groupby`). To support flexible aggregations over packets, we design a programmable key-value store in hardware (§3), where the keys represent flow identifiers and the values represent the state computed by the aggregation function. This key-value store must update values at the line rate of 1 packet per clock cycle (at 1 GHz [6, 33]) and support millions of keys (*i.e.*, flows). Unfortunately, neither SRAM nor DRAM is simultaneously fast and dense enough to meet both requirements.

We split the key-value store into a small but fast on-chip cache in SRAM and a larger but slower off-chip backing store in DRAM. Traditional caches incur variable write latencies due to cache misses; however, line-rate packet forwarding requires deterministic latency guarantees. Our design accomplishes this by never reading back a value into the cache if it has already been evicted to the backing store. Instead, it treats a cache miss as the arrival of a packet from a new flow. When a flow is evicted, we *merge* the evicted flow's value in the cache with the flow's old value in the backing store. Because merges occur off the critical packet processing path, the backing store can be implemented in software on a separate collection server.

While it is not always possible to merge an aggregation function without losing accuracy, we characterize a class of affine aggregation functions, which we call *linear-in-state*, for which accurate merging is possible. Many useful aggregation functions are linear-in-state, *e.g.,* counters, predicated counters (*e.g.,* count only TCP packets that saw timeouts), exponentially weighted moving averages, and functions computed over a finite window of packets. We design a switch instruction to support linear-in-state functions, finding that it easily meets timing at 1 GHz, while occupying modest silicon area.

Query compiler. We implement a compiler that takes Marple queries and compiles them into switch configurations for two targets (§4): (1) the P4 behavioral model [19], an open source programmable software switch that can be used for end-to-end evaluations of Marple on Mininet [47], and (2) Banzai [56], a simulator for high-speed programmable switch hardware that can be used to experiment with different instruction sets. The Marple compiler detects linear-in-state aggregations in input queries and successfully targets the linear-in-state switch instruction that we add to Banzai.

Evaluation. We show that Marple can express a variety of useful performance monitoring examples, like detecting and localizing TCP incast and measuring the prevalence of out-of-order TCP packets. Marple queries require between 4 and 11 pipeline stages, which is modest for a 32-stage switch pipeline [33]. We evaluate our key-value store's performance using trace-driven simulations. For a 64 Mbit on-chip cache, which occupies about 10% of the area of a 64×10-Gbit/s switching chip, we estimate that the cache eviction rate from a single top-of-rack switch can be handled by a single 8-core server running Redis [20]. We evaluate Marple's usability through two Mininet case studies that use Marple to troubleshoot high tail latencies [26] and measure the distribution of flowlet sizes [27]. Marple is open source and available at http://web.mit.edu/marple.

Construct	Description
`pktstream`	Stream of packet performance metadata.
`filter(R, pred)`	Output tuples in `R` satisfying predicate `pred`.
`map(R, [exprs], [fields])`	Evaluate expressions, `[exprs]`, over fields of `R`, emitting tuples with new fields, `[fields]`.
`groupby(R, [fields], fun)`	Evaluate function `fun` over the input stream `R` partitioned by `fields`, producing tuples on `emit()`.
`zip(R, S)`	Merge fields in incoming `R` and `S` tuples.

Figure 2: Summary of Marple language constructs.

2 THE MARPLE QUERY LANGUAGE

This section describes the Marple query language. §3 then covers the switch implementation of the language constructs, while §4 describes the compiler. Marple provides the abstraction of a network-wide stream of performance information. The tuples in the stream contain performance metadata, such as queue lengths and timestamps when a packet entered and departed queues, for each packet at each queue in the network. Network operators write queries on this stream as if the entire stream is processed by a single hypothetical server running the query. In reality, the compiler partitions the query across the network and executes each part on individual switches.

Marple programs process the performance stream using familiar functional constructs (`filter`, `map`, `groupby`, and `zip`), all of which take streams as inputs and produce a stream as output. This functional language model is expressive enough to support diverse performance monitoring use cases, but still simple enough to implement in high-speed hardware. Marple's language constructs are summarized in Fig. 2.

Packet performance stream. As part of the base input stream, which we call `pktstream`, Marple provides one tuple for each packet at each queue with the following fields.

```
(switch, qid, hdrs, uid, tin, tout, qsize)
```

`switch` and `qid` denote the switch and queue at which the packet was observed. A packet may traverse multiple queues even within a single switch, so we provide distinct fields. The regular packet headers (Ethernet, IP, TCP, *etc.*) are available in the `hdrs` set of fields, with a `uid` that uniquely determines a packet.[1]

The packet performance stream provides access to a variety of performance metadata: `tin` and `tout` denote the enqueue and dequeue timestamps of a packet, while `qsize` denotes the queue depth when a packet is enqueued. It is beneficial to have two timestamps to detect co-habitation of the queue by packets belonging to different flows. Additionally, it is beneficial to have a queue size, since we cannot always determine the queue size from the two timestamps: a link may service multiple queues, and the speed at which a queue drains may not be known.

Tuples in `pktstream` are processed in order of packet dequeue time (`tout`), since this is the earliest time at which all tuple fields in `pktstream` are known.[2] If a packet is dropped, `tout` and `qsize` are `infinity`. Tuples corresponding to dropped packets may be processed in an arbitrary order.

Restricting packet performance metadata of interest. Consider the example of tracking packets that experience high queueing latencies at a specific queue (`Q`) and switch (`S`). This is expressed by the query:

```
result = filter(pktstream, qid == Q and switch == S
                and tout - tin > 1ms)
```

The `filter` operator restricts the user's attention to those packets with the relevant performance metadata. A filter has the form `filter(R, pred)` where `R` is some stream containing performance metadata (*e.g.,* `pktstream`), and the filter predicate `pred` may involve packet headers, performance metadata, or both. The result of a `filter` is another stream that contains only tuples satisfying the predicate.

Computing stateless functions over packets. Marple lets users compute functions of the fields available in the incoming stream, to express new quantities of interest. A simple example is rounding packet timestamps to an 'epoch':

```
result = map(pktstream, [tin/epoch_size], [epoch]);
```

The `map` operator evaluates the expression `tin/epoch_size`, written over the fields available in the tuple stream, and produces a new field `epoch`. The general form of this construct is `map(R, [expression], [field])` where a list of `expressions` over fields in the input stream `R` creates a list of new `fields` in the `map` output stream.

Aggregating statefully over multiple packets. Marple allows aggregating statistics over multiple tuples at user-specified granularities. For example, the following query counts packets belonging to each transport-level flow (*i.e.*, 5-tuple):

```
result = groupby(pktstream, [5tuple], count)
```

Here, the `groupby` partitions the incoming `pktstream` into substreams based on the transport 5-tuple, and then applies the aggregation function `count` to count the number of tuples in each substream. Marple allows users to write flexible order-dependent aggregation functions over the tuples of each substream. For example, a user can track latency spikes for each connection by maintaining an exponentially weighted moving average (EWMA) of queueing latencies:

```
result = groupby(pktstream, [5tuple, switch], ewma);
def ewma([avg], [tin, tout]):
  avg = ((1-alpha)*avg) + (alpha*(tout-tin));
```

Here the aggregation function `ewma` evolves an EWMA `avg` using the current value of `avg` and incoming packet timestamps. Unlike the previous `count` example, the EWMA aggregation function depends on the order of packets being processed.

`groupby`s take the general form `groupby(R, [aggFields], fun)`, where the aggregation function `fun` operates over tuples sharing attributes in a list `aggFields` of headers and performance metadata. This construct is inspired by folds in functional programming [45]. Such order-dependent folds are challenging to express in existing query languages. For instance, SQL only allows order-independent commutative aggregations, whether built-in (*e.g.,* count, average, sum) or user-defined.

The aggregation function `fun` is written in an imperative form, with two arguments: a list of state variables and a list of relevant

[1] It is usually possible to use a combination of the 5-tuple and IP ID field as the `uid`.

[2] We assume clock synchronization to let us compare `tin` and `tout` values from different switches. Without synchronization, the programmer can still write queries that do not compare time-valued fields `tin` and `tout` across switches.

incoming tuple fields. Each statement in fun can be an assignment to an expression (x = ...), a branching statement (if pred {...} else {...}), or a special emit() statement that controls the output stream of the groupby. Below, we show an example of an aggregation that detects a new connection:

```
result = groupby(pktstream, [5tuple], new_flow);
def new_flow([fcount], []):
  if fcount == 0:
    fcount = 1
    emit()
```

The output of a groupby is a stream containing the aggregation fields (*e.g.,* 5-tuple) and the aggregated values (*e.g.,* fcount). The output stream contains only tuples for which the emit() statement is encountered during execution of the aggregation function. For example, the output stream of new_flow consists of the first packet of every new transport-level connection. If the function has no emit()s, the user can still read the aggregated fields and their current aggregated state values as a table.

Chaining together multiple queries. Because all Marple constructs produce and consume streams, Marple allows users to write queries that take in the results of previous queries as inputs. A stream of tuples flows from one query to the next, and each query may add or filter out information from the incoming tuple, or even drop the tuple entirely. For example, the program below tracks the size distribution of flowlets, *i.e.*, bursts of packets from the same 5-tuple separated by more than a fixed time amount delta.

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
def fl_detect([last_time, size], [tin]):
  if (tin - last_time > delta):
    emit()
    size = 1
  else:
    size = size + 1
  last_time = tin
```

The function fl_detect detects new flowlets using the last time a packet from the same flow was seen. Because of the emit() statement's location, the flowlet size from fl_track is only streamed out to other operators upon seeing the first packet of a new flowlet.

```
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

The map fl_bkts bins the flowlet size emitted by fl_track into a bucket index, which is used to count the number of flowlets in the corresponding bucket in fl_hist.

Joining results across queries. Marple provides a zip operator that "joins" the results of two queries to check whether two conditions hold simultaneously. Consider the example of detecting the fan-in of packets from many connections into a single queue, characteristic of TCP incast [61]. This can be checked by combining two distinct conditions: (1) the number of active flows in a queue over a short interval of time is high, and (2) the queue occupancy is large.

A user can first compute the number of active flows over the current epoch using two aggregations:

```
R1 = map(pktstream, [tin/epoch_size], [epoch]);
R2 = groupby(R1, [5tuple, epoch], new_flow);
R3 = groupby(R2, [epoch], count);
```

The number of active flows in this epoch can be combined with the queue occupancy information in the original packet stream through the zip operator:

```
R4 = zip(R3, pktstream);
result = filter(R4, qsize > 100 and count > 25);
```

The result of a zip operation over two input streams is a single stream containing tuples that are a concatenation of all the fields in the two streams, whenever both input streams contain valid tuples processed from the same original packet tuple. A zip is a special kind of stream join where the result can be computed without having to synchronize the two streams, because tuples of both streams originate from pktstream. The result of the zip can be processed like any other stream: the filter in the result query checks the two incast conditions above.

We did not find a need for more general joins akin to joins in streaming query languages like CQL [30]. Streaming joins have semantics that can be quite complex and may produce large results, *i.e.*, $O(\#pkts^2)$. Hence, Marple restricts users to simple zip joins.

We show several examples of Marple queries in Fig. 7. For instance, Marple can express measurements of simple counters, TCP reordering of various forms, high-loss connections, flows with high end-to-end network latencies, and TCP fan-in.

Restrictions on Marple queries. Some aggregations are challenging to implement over a network-wide stream. For example, consider an EWMA over some packet field across all packets seen anywhere in the entire network, while processing packets in the order of their tout values. Even with clock synchronization, this aggregation is hard to implement because it requires us to either coordinate between switches or stream all packets to a central location.

Marple's compiler rejects queries with aggregations that need to process *multiple packets* at *multiple switches* in *order of their* tout *values*. Concretely, we only allow aggregations that relax one of these three conditions, and thus either

(1) operate independently on each switch, in which case we naturally partition queries by switch (*e.g.,* a per-flow EWMA of queueing latencies on a particular switch), or
(2) operate independently on each packet, in which case we have the packet perform the coordination by carrying the aggregated state to the next switch on its path (*e.g.,* a rolling average link utilization seen by the packet along its path), or
(3) are associative and commutative, in which case independent switch-local results can be combined in any order to produce a correct overall result for the network [15], *e.g.,* a count of how many times packets from a flow appeared throughout the network. In this case, we rely on the programmer to annotate the aggregation function with the assoc and comm keywords.

3 SCALABLE AGGREGATION AT LINE RATE

How should switches implement Marple's language constructs? We require instructions on switches that can aggregate packets into per-flow state (groupby), transform packet fields (map), stream only packets matching a predicate (filter), or merge packets that satisfy two previous queries (zip).

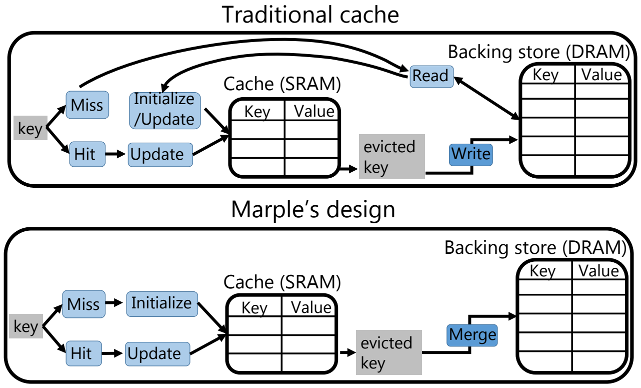


Figure 3: Marple's key-value store vs. a traditional cache

Of the four language constructs, `map`, `filter`, and `zip`, are *stateless*: they operate on packet fields alone and do not modify switch state. Such stateless manipulations are already supported on emerging programmable switches that support programmable packet header processing [3, 13, 25, 33]. On the other hand, the `groupby` construct needs to maintain and update state on switches.

Stateful manipulation on a switch for a `groupby` is challenging for two reasons. First, the time budget to update state before the next packet arrives can be as low as a nanosecond on high-end switches [56]. Second, the switch needs to maintain state proportional to the number of aggregated records (*e.g.,* per flow), which may grow unbounded with time. We address both challenges using a programmable key-value store in hardware, where the keys represent aggregation fields and values represent the state being updated by the aggregation function. Our key-value store has a 'split' design: a small and fast on-chip key-value store on the switch processes packets at line rate, while a large and slow off-chip backing store allows us to scale to a large number of flows.

High-speed switch ASICs typically feature an ingress and egress pipeline shared across multiple ports, running at a 1 GHz clock rate to support up to a billion 64-byte packets per second of aggregate capacity [33]. To handle state updates from packets arriving at 1 GHz, the on-chip key-value store must be in SRAM on the switch ASIC. However, the SRAM available for monitoring on an ASIC (§5.2) is restricted to tens of Mbits (about 10K–100K flows).

To scale to a larger number of flows, the on-chip key-value store serves as a cache for the larger off-chip backing store. In traditional cache designs, cache misses require accessing off-chip DRAM with non-deterministic latencies [37] to read off the stored state. Because the aggregation operation requires us to read the value in order to update it, the entire state update operation incurs non-deterministic latencies in the process. This results in stalls in the switch pipeline. Unfortunately, pipeline stalls affect the ability to provide guarantees on line-rate packet processing (10–100 Gbit/s) on all ports.

We design our key-value store to process packets at line rate even on cache misses (Fig. 3). Instead of stalling the pipeline waiting for a result from DRAM, we treat the incoming packet as the first packet from a new flow and initialize the flow's state to an initial value. Subsequent packets from the same flow are aggregated within the newly created flow entry in the key-value store, until the flow is evicted. When it is evicted, we merge the flow's value just before eviction with its value in the backing-store using a *merge function*, and write the merged result to the backing store. In our design, the switch only writes to the backing store but never reads from it, which helps avoid non-deterministic latencies. The backing store may be stale relative to the on-chip cache if there have been no recent evictions. We remedy this by forcing periodic evictions.

To merge a flow's new aggregated value in the switch cache with its old value in the backing store correctly, the cache needs to maintain and send *auxiliary state* to the backing store. A naïve usage of auxiliary state is to store relevant fields from every packet of a flow, so that the backing store can simply run the aggregation function over the entire packet stream when merging. However, in a practical implementation, the auxiliary state should be bounded in size and not grow with the number of packets in the flow. Over the next four subsections, we describe two classes of queries that are *mergeable* with a small amount of auxiliary state (§3.1 and §3.2), discuss queries that are not mergeable (§3.4), and provide a general condition for mergeability that unifies the two classes of mergeable queries and separates them from non-mergeable queries (§3.5).

3.1 The associative condition

A simple class of mergeable aggregations is associative functions. Suppose the aggregation function on state s is $s = op(s, f)$, where op is an associative operation and f is a packet field. Then, if op has an identity element I and a flow's default value on insertion is $s_0 = I$, it is easy to show that this function can be merged using the function $op(s_{backing}, s_{cache})$, where $s_{backing}$ and s_{cache} are the value in the backing store and the value just evicted from the cache, respectively. The associative condition allows us to merge aggregation functions like addition, max, min, set union, and set intersection.

3.2 The linear-in-state condition

Consider the EWMA aggregation function, which maintains a moving average of queueing latencies across all packets within a flow. The aggregation function updates the EWMA s as follows:

$$s = (1-\alpha)\cdot s + \alpha \cdot (t_{out} - t_{in})$$

We initialize s to s_0. Suppose a flow F is evicted from the on-chip cache for the first time and written to the backing store with an EWMA of $s_{backing}$.[3] The first packet from F after F's eviction is processed like a packet from a new flow in the on-chip cache, starting with the state s_0. Assume that N packets from F then hit the on-chip cache, resulting in the EWMA going from s_0 to s_{cache}. Then, the correct EWMA $s_{correct}$ (*i.e.*, for all packets seen up to this point) satisfies:

$$s_{correct} - (1-\alpha)^N s_{backing} = s_{cache} - (1-\alpha)^N s_0$$
$$s_{correct} = s_{cache} + (1-\alpha)^N (s_{backing} - s_0)$$

So, the correct EWMA can be obtained by: (1) having the on-chip cache store $(1-\alpha)^N$ as auxiliary state for each flow after each update, and (2) adding $(1-\alpha)^N(s_{backing} - s_0)$ to s_{cache} when merging s_{cache} with $s_{backing}$.

We can generalize this example. Let **p** be a vector with the headers and performance metadata from the last k packets of a flow, where k

[3] When a flow is first evicted, it does not need to be merged.

is an integer determined at query compile time (§4.3). We can merge any aggregation function with state updates of the form $S = A(\mathbf{p}) \cdot S + B(\mathbf{p})$, where S is the state, and $A(\mathbf{p})$ and $B(\mathbf{p})$ are functions of the last k packets. We call this condition the *linear-in-state* condition and say that $A(\mathbf{p})$ and $B(\mathbf{p})$ are functions of *bounded packet history.*

The requirement of bounded packet history is important. Consider the TCP non-monotonic query from Fig. 7, which counts the number of packets with sequence numbers smaller than the maximum sequence number seen so far. The aggregation can be expressed as

```
count = count + (maxseq > tcpseq) ? 1 : 0
```

While the update superficially resembles $A(\mathbf{p}) \cdot S + B(\mathbf{p})$, the coefficient $B(\mathbf{p})$ is a function of `maxseq`, the maximum sequence number so far, which could be arbitrarily far back in the stream. Intuitively, since $B(\mathbf{p})$ is not a function of bounded packet history, the auxiliary state required to merge `count` is large. §3.5 formalizes this intuition.

In contrast, the slightly modified TCP out-of-sequence query from Fig. 7 *is* linear-in-state because it can be written as

```
count = count + (lastseq > tcpseq) ? 1 : 0
```

where `lastseq`, the previous packet's sequence number, depends only on the last 2 packets: the current and the previous packet. Here, $A(\mathbf{p})$ and $B(\mathbf{p})$ are functions of bounded packet history, with $k = 2$.

Merging queries that are linear-in-state requires the switch to store the first k and most recent k packets for the key since it (re)appeared in the key-value store; details are available in the accompanying tech report [15]. An aggregation function is linear-in-state if, for every variable in the function, the state update satisfies the linear-in-state condition. A query is linear-in-state if all its aggregation functions are linear-in-state.

3.3 Scalable aggregation functions

A `groupby` with no `emit()` and a linear-in-state (or associative) aggregation function can be implemented scalably without losing accuracy. Examples of such aggregations (from Fig. 7) include tracking successive packets within a TCP connection that are out-of-sequence and counting the number of TCP timeouts per connection.

If a `groupby` uses an `emit()` to pass tuples to another query, it cannot be implemented scalably even if its aggregation function is linear-in-state or associative. An `emit()` outputs the current state of the aggregation function, which assumes the current state is always available in the switch's on-chip cache. This is only possible if flows are never evicted, effectively shrinking the key-value store to its on-chip cache alone.

3.4 Handling non-scalable aggregations

While the linear-in-state and associative conditions capture several aggregation functions and enable a scalable implementation, there are two practical classes of queries that we cannot scale: (1) queries with aggregation functions that are neither associative nor linear-in-state and (2) queries where the groupby has an `emit()` statement.

An example of the first class is the TCP non-monotonic query discussed earlier. An example of the second class is the flowlet size histogram query from Fig. 7, where the first `groupby` emits flowlet sizes, which are grouped into buckets by the second `groupby`.

There are workarounds for non-scalable queries. One is to rewrite queries to remove `emit()`s. For instance, we can rewrite the loss rate query (Fig. 7) to independently record the per-flow counts for dropped packets and total number of packets in separate key-value stores, and have an operator consult both key-value stores every time they need the loss rate. Each key-value store can be scaled, but the implementation comes at a transient loss of accuracy relative to precisely tracking the loss rate after every packet using a `zip`. Second, an operator may be content with flow values that are accurate for each time period between two evictions, but not across evictions (Fig. 10b). Third, an operator may want to run a query to collect data until the on-chip cache fills up and then stop data collection. Finally, if the number of keys is small enough to fit in the cache (*e.g.,* if the key is an application type), the system can provide accurate results without evicting any keys.

3.5 A unified condition for mergeability

We present a general condition that separates mergeable functions from non-mergeable ones. Informally, mergeable aggregation functions are those that maintain auxiliary state linear in the size of the function's state itself. This characterization also has the benefit of unifying the associative and linear-in-state conditions. We now formalize our results in the form of several theorems without proofs; an accompanying technical report [15] contains the proofs.

Let n denote the size of state (in bits) tracked in a Marple query: it must be bounded and should not increase with the number of packets. When merging state s_{cache} in the on-chip cache with state $s_{backing}$ in the backing store, the switch may maintain and send auxiliary state *aux* for the backing store to perform the merge correctly. In the EWMA example, the value $(1-\alpha)^N$ is auxiliary state. Then, a *merge function m* for an aggregation function f is a function satisfying:

$$m(s_{cache}, aux, s_{backing}) = f(s_0, \{p_1, \ldots, p_N\})$$

for any N and sequence of packets $p_1, \ldots, p_N$. The application of f to a list is shorthand for folding f over each packet in order.

First, we show that *every* aggregation function has a merge function, provided it is allowed to use a large amount of auxiliary data.

THEOREM 3.1. *Every aggregation function has a corresponding merge function that uses $O(n2^n)$ auxiliary bits.*

Unfortunately, memory is limited and Marple should not use much more state than indicated by the user's aggregation function. We say an aggregation function is *mergeable* if the auxiliary state has size $O(n)$ for any sequence of packets. This characterization is consistent with what we have described so far: the linear-in-state and associative conditions are indeed mergeable by this definition, while queries that we cannot merge (*e.g.,* TCP non-monotonic in Fig. 7) violate it.

THEOREM 3.2. *If an aggregation function is either linear-in-state or associative, it has a merge function that uses $O(n)$ bits of auxiliary state.*

THEOREM 3.3. *The TCP non-monotonic query from Fig. 7 requires $\Theta(n2^n)$ auxiliary bits in the worst case.*

This raises the question: can we determine whether an aggregation function is mergeable with $O(n)$ auxiliary bits? We provide an algorithm (described in the tech report) that computes the minimum auxiliary state size needed to merge a given aggregation function. Our current algorithm uses brute force and is doubly exponential in n.

However, a polynomial time algorithm is unlikely. We demonstrate a hardness result by considering a decision version of a simpler version of this problem where the merge function m is given as input: given an aggregation function f and merge function m, does m successfully merge f for all possible packet inputs?

THEOREM 3.4. *Determining whether a merge function successfully merges an aggregation function is co-NP-hard.*

The practical implication of this result is that there is unlikely to be a general and efficient procedure to check if an arbitrary aggregation function can be merged using a small amount of auxiliary state. Thus, identifying specific classes of functions (*e.g.,* linear-in-state and associative) and checking if an aggregation function belongs to these classes is the best we can hope to do.

3.6 Hardware feasibility

We optimize our stateful hardware design for linear-in-state queries and break it down into five components. Each component is well-known; our main contribution is putting them together to implement stateful queries. We now discuss each component in detail.

The on-chip cache is a hash table where each row in the hash table stores keys and values for a certain number of flows. If a packet from a new flow hashes into a row that is full, the least recently used flow within that row is evicted. Each row has 8 flows and each flow stores both its key and value.[4] Our choice of 8 flows is based on 8-way L1 caches, which are very common in processors [14]. This cache eviction policy is close to an ideal but impractical policy that evicts the least recently used (LRU) flow across the whole table (§5).

Within a switch pipeline stage, the on-chip cache has a logical interface similar to an on-chip hash table used for counters: each packet matches entries in the table using a key extracted from the packet header, and the corresponding action (*i.e.*, increment) is executed by the switch. An on-chip hash table may be used as a path to incrementally deploying a switch cache for specific aggregations (*e.g.,* increments), on the way to supporting more general actions and cache eviction logic in the future.

The off-chip backing store is a scale-out key-value store such as Redis [20] running on dedicated collection servers within the network. As §5 shows, the number of measurement servers required to support typical eviction rates from the switch's on-chip cache is small, even for a 64×100-Gbit/s switch.

Maintaining packet history. Before a packet reaches the pipeline stage with the on-chip cache, we use the preceding stages to precompute $A(\mathbf{p})$ and $B(\mathbf{p})$ (the functions of bounded packet history) in the state-update operation $S = A(\mathbf{p}) \cdot S + B(\mathbf{p})$. Our current design only handles the case where S, $A(\mathbf{p})$, and $B(\mathbf{p})$ are scalars. Say $A(\mathbf{p})$ and $B(\mathbf{p})$ depend on packet fields from the last k packets. Then, these preceding pipeline stages act like a shift register and store fields from the last k packets. Each stage contains a read/write register, which is read by a packet arriving at that stage, carried by the packet as a header, and written into the next stage's register. Once values from the last k packets have been read into packet fields, $A(\mathbf{p})$ and $B(\mathbf{p})$ can be computed with stateless instructions provided by programmable switch architectures [33, 56].

```
def oos_count([count, lastseq], [tcpseq, payload_len]):
  if lastseq != tcpseq:
    count = count + 1
    emit()
  lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                 and (switch == S1 or switch == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                  [5tuple, switch, epoch],
                  oos_count);
```

Figure 4: Running example for Marple compiler (§4).

Carrying out the linear-in-state operation. Once $A(\mathbf{p})$ and $B(\mathbf{p})$ are known, we use a multiply-accumulate (MAC) instruction [17] to compute $A(\mathbf{p}) \cdot S + B(\mathbf{p})$. This instruction is very cheap to implement: our circuit synthesis experiments show that a MAC instruction meets timing at 1 GHz and occupies about 2000 μm^2 in a recent 32 nm transistor library. A switching chip with an area of a few hundred mm^2 can easily support a few hundred MAC instructions.

Queries that are not linear-in-state. We use the set of stateful instructions developed in Domino [56] for queries that are not linear-in-state. Our evaluations show that these instructions are sufficient for our example queries that are not linear-in-state.

4 QUERY COMPILER

We compile Marple queries to two targets: the P4 behavioral model [19], configured by emitting P4 code [18], and the Banzai machine model, configured by emitting Domino code [56]. In both cases, the emitted code configures a switch pipeline, where each stage is a match-action table [33] or our key-value store.[5]

A preliminary pass of the compiler over the input query converts the query to an abstract syntax tree (AST) of functional operators (Fig. 5a). The compiler then:

(1) produces switch-local ASTs from a global AST (§4.1);
(2) produces P4 and Domino pipeline configurations from switch-local ASTs (§4.2); and
(3) specifically recognizes linear-in-state aggregation functions, and sets up auxiliary state required to merge such functions for a scalable implementation (§4.3). To scalably implement associative aggregation functions (§3.1), we use the programmer annotation `assoc` to determine if an aggregation is associative. If it is associative, the merge function is the aggregation function itself.

We use the query shown in Fig. 4 as a running example to illustrate the details in the compiler. The query counts the number of out-of-sequence TCP packets over each time epoch, measured at two switches `S1` and `S2` in the network.

[4] The LRU policy is actually implemented across 3-bit pointers that point to the keys and values in a separate memory. So we shuffle only the 3-bit pointers for the LRU, not the entire key and value.

[5] In this paper, we do not consider the problem of reconfiguring the switch pipeline on the fly as queries change.

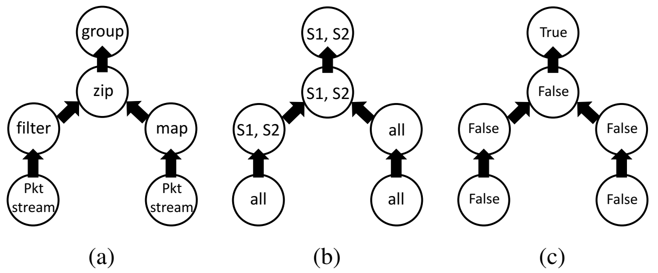


Figure 5: Abstract Syntax Tree (AST) manipulations for the running example (§4). (a) Operator AST. (b) Stream location (set of switches). (c) Stream switch-partitioned (boolean).

4.1 Network-wide to switch-local queries

The compiler partitions a network-wide query written over all packets at all queues in the network (§2) into switch-local queries to generate switch-specific configurations. We achieve this in two steps. First, we determine the *stream location*, *i.e.*, the set of switches that contribute tuples to a stream, for the final output stream of query. For instance, the output stream of a query that filters by switch id s has a stream location equal to the singleton set s. Second, we determine how to partition queries with aggregation functions written over the entire network into switch-local queries.

Determining stream location for the final output stream. The stream location of `pktstream` is the set of all switches in the network. The stream location of the output of a `filter` is the set of switches implied by the filter's predicate. Concretely, we evaluate the set of switches contributing tuples to the output of a `filter` operation through basic syntactic checks of the form `switch == X` on the `filter` predicate. We combine switch sets for boolean combinators (`or` and `and`) inside filter predicates using set operations (union and intersection respectively). The stream location of the output of a `zip` operator is the intersection of the stream locations of the two inputs. Stream locations are unchanged by the `map` and `groupby` operators.

The stream locations for the running example are shown in Fig. 5b. The stream location of `pktstream` is the set of all network switches, but is restricted to just `S1` and `S2` by the `filter` in the query (left branch). This location is then propagated to the root of the AST through the `zip` operator in the query.

Partitioning network-wide aggregations. As described in §2, we only permit aggregations that satisfy one of three conditions: they operate independently on each switch, operate independently on each packet, or are associative and commutative. We describe below how we check the first condition, failing which we simply check the last two conditions syntactically: either the `groupby` aggregates by `uid` (condition 2) or contains programmer annotations `assoc` and `comm` (condition 3).

To check if an aggregation operates independently on each switch, we label each AST node with an additional boolean attribute, *switch-partitioned*, corresponding to whether the output stream has been partitioned by the switch at which it appears. Intuitively, if a stream is switch-partitioned, we allow packet-order-dependent aggregations over multiple packets of that stream; otherwise, we do not.

Determining and propagating *switch-partitioned* through an AST is straightforward. The base `pktstream` is not switch-partitioned. The `filter` and `zip` operators produce a switch-partitioned stream if their output only appears at a single switch. The `groupby` produces a switch-partitioned stream if it aggregates by `switch.` In all other cases, the operators retain the operands' switch-partitioned attribute.

The switch-partitioned attributes for our running example are shown in Fig. 5c. The `filter` produces output streams at two switches, hence is not switch-partitioned. The `groupby` aggregates by `switch` and hence is switch-partitioned. After the partitioning checks have succeeded, we are left with a set of independent switch-local ASTs corresponding to each switch location that the AST root operator appears in, *i.e.*, `S1, S2`.

4.2 Query AST to pipeline configuration

This compiler pass first generates a sequence of operators from the switch-local query AST of §4.1. This sequence of operators will then be used in the same order to generate a switch pipeline configuration. There are two aspects that require care when constructing a pipeline structure: (1) the pipeline should respect read-write dependencies between different operators, and (2) repeated subqueries should not create additional pipeline stages. We generate a sequence through a post-order traversal of the query AST, which guarantees that the operands of a node are added into the pipeline before the operator in the node. Further, we deduplicate subquery results from the pipeline to avoid repeating stages in the final output. For the running example, the algorithm produces the sequence of operators: `tcps` (`filter`) → `tslots` (`map`) → `joined` (`zip`) → `oos` (`groupby`).

Next, the compiler emits P4 code for a switch pipeline from the operator sequence. The `filter` and `zip` configuration just involves checking a predicate and setting a "valid" bit on the packet metadata. The `map` configuration assigns a packet metadata field to the computed expression. The `groupby` configuration uses a register that is indexed by the aggregation fields, and is updated through the action specified in the aggregation function. We transform Marple aggregation functions into straight-line code consisting of C-style conditional operators through a standard procedure known as if-conversion [29]. This allows us to fit the aggregation function into the body of a single P4 action.

To target the Banzai switch pipeline simulator [56], the Marple compiler emits Domino code by concatenating C-like code fragments from all pipeline stages into a single Domino program. The Domino compiler then takes this program and compiles it to a pipeline of Banzai *atoms.* Atoms are ALUs representing a programmable switch's instruction set. Atoms implement either stateless (*e.g.,* incrementing a packet field) or stateful (*e.g.,* atomically incrementing a switch counter) computations.

4.3 Handling linear-in-state aggregations

We now consider the problem of detecting if an aggregation function is linear-in-state (*i.e.*, updates to all state variables within the aggregation function can be written as $S = A(\mathbf{p}) \cdot S + B(\mathbf{p})$). A general solution to this problem is challenging because the aggregation function can take varied forms. For instance, the assignment $S = \frac{S^2-1}{S-1}$ is linear-in-state but needs algebraic simplifications to be detected.

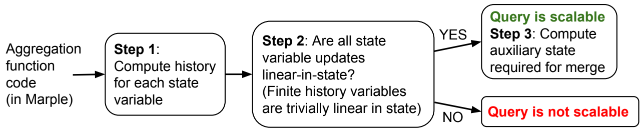

Aggregation function code (in Marple)
Step 1: Compute history for each state variable
Step 2: Are all state variable updates linear-in-state? (Finite history variables are trivially linear in state)
YES
Query is scalable
Step 3: Compute auxiliary state required for merge
NO
Query is not scalable

Figure 6: Steps for compiling linear-in-state updates.

We take a pragmatic approach and sacrifice completeness, but still cover useful functions. Specifically, we only detect linear-in-state state updates through simple syntactic pattern matching in the compiler (*i.e.*, without algebraic transformations). Despite these simplifications, the Marple compiler correctly identifies all the linear-in-state aggregations in Fig. 7 and targets the multiply-accumulate instruction that we added to the Banzai pipeline.

To describe how linear-in-state detection works, we introduce some terminology. Recall that an aggregation function takes two arguments (§2): a list of state variables (*e.g.,* a counter) and a list of tuple fields (*e.g.,* the TCP sequence number). We use the term variable in this subsection to refer to either a state variable or a tuple field. These are the only variables that can appear within the body of the aggregation function.[6]

We carry out a three-step procedure for linear-in-state detection, summarized in Fig. 6. First, for each variable in an aggregation function we assign a *history*. This history tells us how many previous packets we need to look at to determine a variable's value accurately (history = 1 means the current packet). For instance, for the value of a byte counter, we need to look back to the beginning of the packet stream (history = ∞), while for a variable that tracks the last TCP sequence number we need to only look back to the previous packet (history = 2). Consistent with the definition of history, constants are assigned a history value of 0, and variables in the tuple field list are assigned a history of 1. For state variables, we use Alg. 1 to determine each variable's history.

Second, once each variable has a history, we look at the history of each state variable s. If the history of s is a finite number k, then s only depends on the last k packets and the state update for that variable is trivially linear-in-state, by setting A to 0 and B to the aggregation function itself.[7] If s has an infinite history, we use syntactic pattern matching to check if the update to s is linear-in-state.

Third, if all state variables have linear-in-state state updates, the aggregation function is linear-in-state, and we generate the auxiliary state that permits merging of the aggregation function (§3). If not, we use the set of stateful instructions developed in Domino [56] to implement the aggregation function. We now describe each of the three steps in detail.

Determining history of variables. To understand Alg. 1, observe that if all assignments to a state variable only use variables that have a finite history, then the state variable itself has a finite history. For instance, in Fig. 4, right after it is assigned, `lastseq` has a history of 1 because it only depends on the current packet's fields `tcpseq` and `payload_len`. To handle branching in the code, *i.e.*, `if (predicate) {...}` statements, we generalize this observation. A state variable has finite history if (1) it has finite history in all its assignments in all branches of the program, and (2) each branching condition `predicate` itself only depends on variables with a finite history.

Concretely, ComputeHistory (line 2) assigns each variable a history corresponding to an upper bound on the number of past packets that the state variable depends on. We track the history separately for each branching *context*, *i.e.*, the sequence of branches enclosing any statement.[8] The algorithm starts with a default large pessimistic history (*i.e.*, an approximation to ∞) for each state variable (line 1), and performs a fixed-point computation (lines 3–20), repeatedly iterating over the statements in the aggregation function (line 7–16).

For each assignment to a state variable in the aggregation function, the algorithm updates the history of that state variable in the current branching context (lines 7–9). For each branch in the aggregation function, the algorithm maintains a new branching context and a history for the branching context itself (lines 10–14). At the end of each iteration, the algorithm increments each variable's history to denote that the variable is one packet older (line 18). The algorithm returns a conservative history k for each state variable, including possibly max_bound (line 1, Alg. 1) to reflect an infinite history.

Algorithm 1 Determining history of all state variables

```
 1: hist = {state = {true: max_bound}}              ▷ Init. hist. for all state vars.
 2: function ComputeHistory(fun)
 3:     while hist is still changing do                        ▷ Run to fixed point.
 4:         hist ← {}
 5:         ctx ← true                                   ▷ Set up outermost context.
 6:         ctxHist ← 0                                       ▷ History value of ctx.
 7:         for stmt in fun do
 8:             if stmt == state = expr then
 9:                 hist[state][ctx] ← GetHist(ctx, expr, ctxHist)
10:             else if stmt == if predicate then
11:                 save context info (restore on branch exit)
12:                 newCtx ← ctx and predicate
13:                 ctxHist ← GetHist(ctx, newCtx, ctxHist)
14:                 ctx ← newCtx
15:             end if
16:         end for
17:         for ctx, var in hist do                     ▷ Make history one pkt older.
18:             hist[var][ctx] ← min(hist[var][ctx] + 1, max_bound)
19:         end for
20:     end while
21: end function
22: function GetHist(ctx, ast, ctxHist)
23:     for xi ∈ LeafNodes(ast) do
24:         hi = hist[xi][ctx]
25:     end for
26:     return max(h1, ... , hn, ctxHist)
27: end function
```

Now we show precisely how the histories are updated as each statement of the aggregation function is processed using the helper function GetHist. Consider a statement assigning a variable to an expression, `x = expr`, within a branching context `ctx`. Then the history of `x` is the maximum of the history of the predicates in `ctx` and the history of the expression `expr`. This is because if either is a function of the last k packets, then `x` is a function of at least the

[6] Marple supports local variables within the function body, but the more general algorithm is not materially different from the simpler version we present in this paper.

[7] More precisely, the parts of the aggregation function that update s.

[8] Currently, Marple forbids multiple `if ... else` statements at the same nesting level; hence, the enclosing branches uniquely identify a code path through the function. This restriction is not fundamental; the more general form can be transformed into this form.

last k packets. To determine the history of `expr`, suppose the AST of `expr` contains the variables `x1, x2, ..., xn` as its leaves. Then, the history of `expr` is the maximum of the histories of the `xi`. For example, the history for `lastseq` after its assignment in `oos_count` is the maximum of 1 (`tcpseq` and `payload_len` are functions of the current packet), and 0 (for the enclosing outermost context `true`).

Determining if a state variable's update is linear-in-state. For each state variable S with an infinite history, we check whether the state updates are linear-in-state as follows: (1) each update to S is syntactically affine, *i.e.*, $S \leftarrow A \cdot S + B$ with either A or B possibly zero; and (2) A, B and every branch predicate depend on variables with a finite history. This approach is sound, but incomplete: it misses updates such as $S = \frac{S^2-1}{S-1}$.

Determining auxiliary state. For each state variable with a linear-in-state update, we initialize four pieces of auxiliary state for a newly inserted key:[9] (1) a running product $S_A = 1$; (2) a packet counter $c = 0$; (3) an entry log, consisting of relevant fields from the first k packets following insertion; and (4) an exit log, consisting of relevant fields from the last k packets seen so far. After the counter c crosses the packet history bound k, we update S_A to $A \cdot S_A$ each time S is updated.[10] When the key is evicted, we send S_A along with the entry and exit logs to the backing store for merging (details are in our technical report [15]).

5 EVALUATION

We evaluate Marple in three ways. In §5.1, we quantify the hardware resources required for Marple queries. In §5.2, we measure the memory-eviction tradeoff for the key-value store. In §5.3 and §5.4, we show two case studies that use Marple compiled to the P4 behavioral model running on Mininet: debugging microbursts [44] and computing flowlet size distributions.

5.1 Hardware compute resources

Fig. 7 shows several Marple queries. Alongside each query, we show (1) whether all its aggregations are linear-in-state, (2) whether it can be scaled by merging correctly with a backing store, and (3) the switch resources required, measured through the pipeline depth (number of stages), width (maximum number of parallel computations per stage), and number of Banzai atoms (total number of computations) required.

Fig. 7 shows that many useful queries contain only linear-in-state aggregations, and most of them scale to a large number of keys (§3.2). Notably, the flowlet size histogram and lossy connection queries are not scalable despite being linear-in-state, since they contain `emit()` statements. In §3.4, we showed how to rewrite some of these queries (*e.g.,* lossy connections) to scale, at the cost of losing some accuracy.

We compute the pipeline's depth and width by compiling each query to the Banzai switch pipeline simulator. Banzai is supplied with stateless atoms, which perform binary operations (arithmetic, logic, and relational) on pairs of packet fields, and one stateful atom. For the linear-in-state operations, we use the multiply-accumulate atom as the stateful atom, while for the other operations, we use Banzai's own NestedIf atom [56]. The Domino compiler determines whether the input program can be mapped to a pipeline with the specified atoms. As expected, all the linear-in-state queries map to a pipeline with the multiply-accumulate atom.

The computational resources required for Marple queries are modest. All queries in Fig. 7 require a pipeline shorter than 11 stages. This is feasible, *e.g.,* the RMT architecture offers 32 stages [33]. Further, functionality other than measurement can run in parallel because the number of atoms required per stage is at most 6, while programmable switches provide ~100 parallel instructions per stage (*e.g.,* RMT provides 224 [33]).

5.2 Memory and bandwidth overheads

In this section, we answer the following questions:

(1) What is a good size for the on-chip key value store?
(2) What are the eviction rates to the backing store?
(3) How accurate are queries that are not mergeable?

Experimental setup. We simulate a Marple query over three unsampled packet traces: two traces from 10 Gbit/s core Internet routers, one from Chicago (~150M packets) from 2016 [24] and one from San Jose (~189M packets) from 2014 [23]; and a 2.5 hour university data-center trace (~100M packets) from 2010 [32]. We refer to these traces as Core16, Core14, and DC respectively.

We evaluate the impact of memory size on cache evictions for a Marple query that aggregates by 5-tuple. As discussed in §3.6, our hardware design uses an 8-way LRU cache. We also evaluate two other geometries: a hash table, which evicts the incumbent key upon a collision, and a fully associative LRU. Comparing our 8-way LRU with other hardware designs demonstrates the tradeoff between hardware complexity and eviction rate.

Eviction ratios. Each evicted key-value pair is streamed to a backing store. This requires the backing store to be able to process packets as quickly as they are evicted, which depends on the incoming packet rate and the eviction ratio, *i.e.*, the ratio of evicted packets to incoming packets. The eviction ratio depends on the geometry of the on-chip cache, the packet trace, and the cache size (*i.e.*, the number of key-value pairs it stores). Hence, we measure eviction ratios over (1) the three geometries for the Core16 trace (Fig. 8b), (2) the three traces using the 8-way LRU geometry (Fig. 8a), and (3) for caches sizes between 2^{16} (65K) and 2^{21} (2M) key-value pairs.

Fig. 8b shows that a full LRU has the lowest eviction ratios, since the entire LRU must be filled before an eviction occurs. However, the 8-way associative cache is a good compromise: it avoids the hardware complexity of a full LRU while coming within 2% of its eviction ratio. Fig. 8a shows that the DC trace has the lowest eviction ratios. This is because it has much fewer unique keys than the other two traces and these keys are less likely to be evicted.

The reciprocal of the eviction ratio (as a fraction) is the reduction in server data collection load relative to a *per-packet collector* that processes per-packet information from switches. For example, for the Core14 trace with a 2^{19} key-value pair cache, the server load reduction is 25× (corresponding to an eviction ratio of 4%).

Eviction rates. Eviction ratios are agnostic to specifics of the switch, such as link speed, link utilization, and on-chip cache size in bits. To translate eviction ratios (evictions per packet) to eviction

[9] This can happen either when a key first appears or reappears following an eviction.

[10] This stateful update itself can be implemented through a multiply-accumulate atom.

Example	Query code	Description	Linear in state?	Scales?	Pipe depth	Pipe width	# of atoms

Packet counts	`def count([cnt], []): cnt = cnt + 1; emit()`
	`result = groupby(pktstream, [srcip], count);`
		Count packets per source IP.	Yes	Yes	5	2	7

EWMA over latencies	`def ewma([avg], [tin, tout]):`
	`  avg = (1-alpha)*avg + (alpha)*(tout-tin)`
	`ewma_q = groupby(pktstream, [5tuple], ewma);`
		Maintain a moving EWMA over packet latencies per flow.	Yes	Yes	6	4	11

TCP out-of-sequence	`def oos([lastseq, cnt], [tcpseq, payload_len]):`
	`  if lastseq != tcpseq: cnt = cnt + 1`
	`  lastseq = tcpseq + payload_len`
	`oos_q = groupby(pktstream, [5tuple], oos);`
		Count the number of packets per connection arriving with a sequence number that is non-consecutive with the last packet.	Yes	Yes	7	4	14

TCP non-monotonic	`def nonmt([maxseq, cnt], [tcpseq]):`
	`  if maxseq > tcpseq: cnt = cnt + 1`
	`  else: maxseq = tcpseq`
	`nm_q = groupby(pktstream, [5tuple], nonmt);`
		Count the number of packets per connection with sequence numbers lower than the maximum so far.	No	No	5	2	6

Flowlet size histogram	`def fl_detect([last_time, size], [tin]):`
	`  if tin - last_time > delta:`
	`    emit(); size = 1`
	`  else: size = size + 1`
	`  last_time = tin`
	`R1 = groupby(pktstream, [5tuple], fl_detect);`
	`fl_hist = groupby(R1, [size], count);`
		Compute a histogram over the lengths of flowlets. This statistic is useful to evaluate network load balancing schemes, *e.g.*, [27].	Yes	No	11	6	31

High E2E latency	`def sum_lat([e2e_lat], [tin, tout]):`
	`  e2e_lat = e2e_lat + tout - tin`
	`e2e = groupby(pktstream, [uid], sum_lat);`
	`high_e2e = filter(e2e, e2e_lat > 10);`
		Capture packets experiencing high end-to-end queueing latency, by adding time spent in the queue at each hop.	Yes	Yes	5	3	8

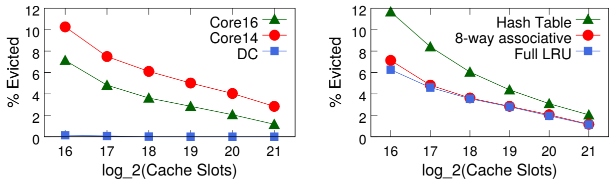
Count concurrently active connections	`def new_flow([cnt], []):`
	`  if cnt == 0: emit(); cnt = 1`
	`R1 = map(pktstream, [tin/128], [epoch]);`
	`R2 = groupby(R1, [5tuple, epoch], new_flow);`
	`num_conns = groupby(R2, [epoch], count);`
		Count the number of active connections in a queue over a period of time ("epoch").	No	No	4	3	10

TCP incast	`R3 = zip(num_conns, pktstream);`
	`ic_q = filter(R3, qin > 100 and cnt < 25);`
		Detect when many connections use a long queue. Uses the query above.	No	No	7	4	14

Lossy connections	`total = groupby(pktstream, [5tuple], count);`
	`R1 = filter(pktstream, tout == infinity);`
	`lost = groupby(R1, [5tuple], count);`
	`Z = zip(total, lost);`
	`lc_q = filter(Z, lost.cnt > p*total.cnt);`
		Compute packet loss rate per connection, reporting connections with packet drop rate higher than a threshold `p`.	Yes	No	8	4	19

TCP timeouts	`def timeout([cnt], [last_time, tin]):`
	`  timediff = tin - last_time`
	`  if timediff > 280ms and timediff < 320ms:`
	`    cnt = cnt + 1`
	`  last_time = tin`
	`to_q = groupby(pktstream, [5tuple], timeout);`
		Count the number of timeouts for each TCP connection, by checking for packet inter-arrival times around 300 ms (retransmission timer).	Yes	Yes	8	3	15

Figure 7: Examples of performance queries. We report that a query *scales* to a large number of keys either if (1) there are no stateful updates involved, or (2) all its stateful updates are linear-in-state *and* there are no `emit()`s. We use Domino [56] to report the hardware resources, *i.e.*, atom count and pipeline depth and width. Linear-in-state queries use the multiply-accumulate atom (§3); others use a NestedIf atom [56] that supports updates predicated on the state value itself.



(a) By trace **(b) By cache geometry (Core16)**

Figure 8: Eviction ratios to the backing store.

rates (evictions per second), we first compute the average packet size (700 Bytes) and link utilization (30%) from the Core16 trace.

Next, we estimate the on-chip cache size for a 64×10-Gbit/s switch and a 64×100-Gbit/s switch. On a 64×10-Gbit/s switch, SRAM densities are $\approx$ 3–4 $^{Mbit}/_{mm^2}$ [1], and the smallest switching chips occupy 200 mm^2 [39]. Therefore, a 64 Mbit cache in SRAM costs around 10% additional area, which we believe is reasonable. For recent 64×100-Gbit/s switches [5], SRAM densities are $\approx 7^{Mbit}/_{mm^2}$ [22], and the switches occupy $\approx$ 500 mm^2,[11] making a 256 Mbit cache (7.3% area overhead) a reasonable target.

For a given query, we divide these cache sizes by the size of the aggregation's key-value pair to get the number of key-value pairs. We then look up this number in Fig. 8 to get the eviction ratio for that query, which we translate to an eviction rate using the network utilization and packet size mentioned earlier.

Eviction rates for some sample queries are shown in Fig. 9. For a 64×10-Gbit/s switch with a 64 Mbit cache, we observe eviction rates of $\approx$ 1M packets per second. For a 64×100-Gbit/s switch with a 256 Mbit cache and the same average packet size and utilization,

[11] S. Chole. Cisco Systems. Private communication. June 2017.

Query	State size (bits)	Eviction rate at 64×10-Gbit/s (packets/s)	Eviction rate at 64×100-Gbit/s (packets/s)
Packet count	32	1.0M (34×)	4.29M (81×)
Lossy connections	64	1.08M (32×)	5.18M (66×)
TCP out-of-sequence	128	1.21M (28×)	6.72M (52×)
Flowlet size histogram (Stage 1)	160	1.26M (27×)	7.17M (48×)

Figure 9: Eviction rates and reduction in collection server load for queries from Fig. 7. Each key-value pair occupies the listed state size plus 104 bits for a 5-tuple key. The 10-Gbit/s and 100-Gbit/s switches have a 64 Mbit and 256 Mbit cache, respectively.

the eviction rates can reach 7.17M packets per second. Relative to a per-packet collector, Marple reduces the server load by 25–80×.

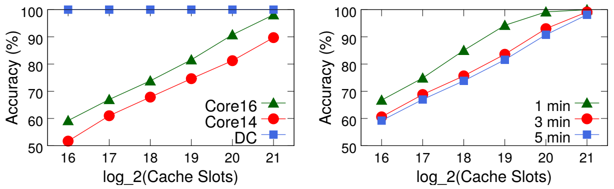
The eviction rates for both the 10 Gbit/s and 100 Gbit/s switches are under 10M packets per second, well within the capabilities of multi-core scale-out key-value stores [2, 11, 43], which typically handle 100K–1M operations per second per core. For instance, for a single 64×10-Gbit/s switch running an aggregation with a 64-bit state size, a single 8-core server is sufficient to handle the eviction rate of 1.08M packets per second. For a single 64×100-Gbit/s switch running the same aggregation, the eviction rate goes up to 5.18M packets per second, requiring four such servers.

Generalizing to other scenarios. Fig. 8 also generalizes to multiple aggregations and aggregations of different state sizes. First, coarsening the aggregation key by picking a subset of the 5-tuple reduces the eviction ratio, since there are fewer keys in the working set. We believe that the 5-tuple may well be the most fine-grained and still practically useful aggregation level; hence, our results show the worst-case eviction ratios for a single `groupby`. Second, variations in the size of the `groupby` value simply result in a different number of key-value pairs for a given memory size. Third, running multiple `groupby` queries with the same number of key-value pairs, and aggregating by the same key, results in synchronized evictions across all queries. Hence, the eviction rate can be read off Fig. 8 at the correspondingly reduced memory size.

Accuracy of non-mergeable queries. Queries that are neither linear-in-state nor associative cannot be merged in the backing store. If a key from such a query is evicted multiple times, Marple cannot guarantee its correctness and marks it as invalid. However, these keys' values are still valid if they are either never evicted or are evicted once and never reappear. We quantify a query's accuracy as the fraction of keys with valid values over the query's lifetime. Fig. 10a shows query accuracy using the three traces, with the DC trace being near-perfect since it has fewer unique keys, and hence, evictions. If the query is run over a shorter time interval, its accuracy is typically higher, since the cache may not be full and a smaller fraction of keys are evicted. Fig. 10b shows this tradeoff for a range of cache sizes and geometries using the Core16 trace. Shortening the query from 5 minutes to 1 minute boosts accuracy by 10%.

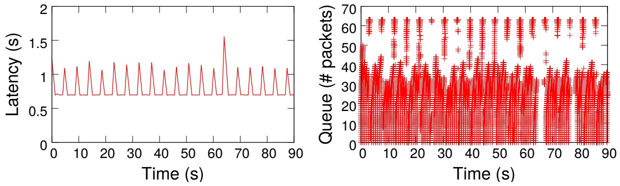
5.3 Case study #1: Debugging microbursts

To demonstrate Marple in practice, we present a case study of diagnosing microbursts, *i.e.*, spikes in latency caused by bursty transmissions of packets into a queue. Our setup in Mininet [47] consists of four hosts (`h1, h2, h3, h4`) and two switches (`S1, S2`) in a dumbbell topology. Switch `S1` is connected to `h1` and `h3`, and `S2` to `h2` and `h4`. The switches are connected via a single link and programmed in P4 [19] with queries compiled by Marple.



(a) By trace **(b) By query duration (Core16)**

Figure 10: Query accuracy for non-mergeable aggregations.



(a) TCP request latency **(b) Queue depth at egress port 2**

Figure 11: Microburst case study measurements.

src → dst	protocol	# Bursts	Time (μs)	# Packets
h3:34573 → h4:4888	UDP	19	8969085	6090
h4:4888 → h3:34573	UDP	18	10558176	5820
h1:1777 → h2:58439	TCP	1	72196926	61584
h2:58439 → h1:1777	TCP	1	72248749	33074

Figure 12: Per-flow burst statistics from Marple.

Host `h2` repeatedly downloads a 1MB objects over TCP from `h1`. Meanwhile, `h3` sends `h4` bursts of UDP traffic, which `h4` acks. Suppose a network operator notices the irregular latency spikes for the downloads (Fig. 11a). She suspects a queue buildup in the switches and measures the queue depths seen by the traffic by writing: `result = filter(pktstream, srcip == h1 and dstip == h2)`.

The results are streamed out on each packet to a collection server. After plotting the queue latencies, she notices spikes in queue size at egress port 3 on the switch (Fig. 11b) matching the periodicity of the latency spikes. To isolate the responsible flow(s), she divides the traffic into "bursts," which she defines as a series of packets separated by a gap of at least 800ms, as determined from the gap between latency spikes. She issues the following Marple query:

```
def burst_stats([last_time, nburst, time], [pkts, tin]):
    if tin - last_time > 800000:
        nbursts++;
        emit();
    else:
        time = time + tin - last_time;
    pkts = pkts + 1;
    last_time = tin;
result = groupby(R1, 5tuple, burst_stats)
```

She runs the query for 72 seconds and sees the result in Fig. 12. She concludes, correctly, that UDP traffic between `h3` and `h4` is responsible for the latency spikes. There are 18 UDP bursts, with an average packet size and duration that matches our emulation setup.

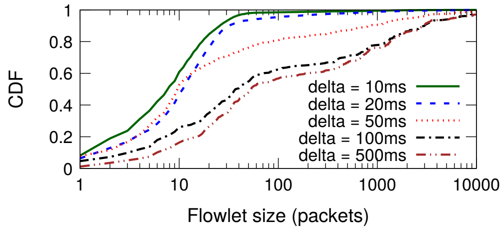


Figure 13: CDF of flowlet sizes for different flowlet thresholds.

Marple's flexibility makes this diagnosis simple. By contrast, localizing the root cause of microbursts with existing monitoring approaches is challenging. Packet sampling and line-rate packet captures would miss microbursts because of heavy undersampling, packet counters from switches would have disguised the bursty nature of the offending UDP traffic, and probing from endpoints would not be able to localize queues with bursty contending flows.

Marple builds on In-band Network Telemetry [12, 26] (INT) that exposes queue lengths at switches. However, Marple's on-switch aggregation goes beyond INT to determine contending flows at the problematic queue through a customized query, without prior knowledge of those flows. In comparison, a pure INT-based solution may require a network operator to manually identify flows that could contend at the problematic queue, and then collect data from the INT endpoints for those flows.

5.4 Case study #2: Flowlet size distributions

We demonstrate another use case for Marple in practice: computing the flowlet size distribution as a function of the flowlet *threshold*, the time gap above which subsequent packets are considered to be in different flowlets. This analysis has many practical uses, *e.g.,* for configuring flowlet-based load balancing strategies [27, 60]. In particular, the performance of LetFlow [60] depends heavily on the distribution of flowlet sizes.

Our setup uses Mininet with a single switch connecting five hosts: a single client and four servers. Flow sizes are drawn from an empirical distribution computed from a trace of a real data center [28]. The switch runs the "flowlet size histogram" query from Fig. 7 for six values of `delta`, the flowlet threshold.

Fig. 13 shows the CDF of flowlet sizes for various values of `delta`. Note that the actual values of `delta` are a consequence of the bandwidth allowed by the Mininet setup; a data center deployment would likely use much lower `delta` values.

6 RELATED WORK

Endpoint-based monitoring. Owing to limited switch support for measurement, many systems monitor network performance from endpoints alone [16, 31, 53, 58, 62]. While endpoint solutions are necessary for application context (*e.g.,* socket calls), they are insufficient to debug all network problems, since endpoints lack visibility into the network core. With switch-augmented endpoint solutions such as INT, the data is scattered over multiple endpoints. We believe networks will need both endpoint and switch-based systems because each sees something the other cannot.

Switch-based monitoring. Most switch-based monitoring has focused on per-flow counts (*e.g.,* NetFlow [7]) and sampling (*e.g.,* sFlow [21]), not performance measurement. Packet sampling can miss important events due to heavy undersampling [55]. Hardware implementations of NetFlow do not keep a record of every flow, since the flow lookup table cannot insert new flows at line rate in the presence of hash collisions [46]. Marple solves exactly this problem through its cache design and merging.

Line-rate packet capture devices, *e.g.,* [10] record all packet traffic at high data rates, providing valuable data for posthoc analyses of network traffic. Ideally, performance monitoring should be possible everywhere in a network, but the high data collection and storage requirements make it impossible to run packet captures pervasively. The same concern limits other strategies that mirror traffic or collect packet digests from switches [8, 42, 65]. In comparison, Marple's flexible language and switch-based aggregation provide network-wide performance monitoring at low data processing overhead, by collecting only what is needed.

Sketches [34, 48, 49, 52, 63] and programmable switch counters [38, 54] expose traffic volume statistics using summary data structures and flow counters on switches. Marple enables monitoring performance statistics much broader than the flow-counter-based statistics from these prior works (Fig. 7). Unlike sketches, which trade off accuracy with memory, Marple implements counters with full accuracy, since counting is a linear-in-state aggregation. Instead, Marple trades off cache eviction rate with cache memory size.

In-band Network Telemetry (INT) [12, 44] exposes queue lengths and other performance metadata from switches by piggybacking them on the packet. Marple uses INT-like performance metadata, but provides flexible aggregations directly on switches. Marple's data aggregation on switches provides three advantages relative to INT. First, without aggregation, each INT endpoint needs to process per-packet data at high data rates. Second, on-switch aggregation saves the bandwidth needed to bring together per-packet data distributed over many INT endpoints. Third, on-switch aggregation can handle cases where INT packets are dropped before they reach endpoints.

Network query languages. Prior network query languages [35, 38, 41, 54] allow users to ask questions primarily about traffic volumes and count statistics, since their input data is collected using NetFlow and match-action rule counters [51]. In contrast, Marple allows operators to ask richer performance questions by designing new switch hardware to support Marple queries. Marple shares some functional and relational constructs with Gigascope [35] and Sonata [41], but supports aggregations directly in the switch. Marple allows operators to program online queries over traffic, enabling the collection of fine-grained customized statistics at low overhead. It is complementary to offline query systems that answer post-facto questions over historical data collected by sampling or packet captures.

7 CONCLUSION

Performance monitoring is a crucial part of the upkeep of any large system. Marple's network visibility and query language demystify network performance for applications, enabling operators to quickly diagnose problems. We want network operators to query for whatever they need, and not be constrained by limited switch feature sets. Marple presents a step forward in enabling fine-grained and programmable network monitoring, putting the network operator—and not the switch vendor—in control of the network.

Acknowledgments. This work was supported by NSF grants CNS-1563826, CNS-1526791, and CNS-1617702, DARPA I2O Award No. HR0011-15-2-0047, and a gift from the Cisco Research Center. We thank Jennifer Rexford, Fadel Adib, Amy Ousterhout, our shepherd Marco Canini, and the anonymous SIGCOMM reviewers for their thoughtful feedback.

REFERENCES

[1] 45 nanometer - Wikipedia, Technology demos. https://en.wikipedia.org/wiki/45_nanometer#Technology_demos.

[2] An Update on the Memcached/Redis Benchmark. http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html.

[3] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf.

[4] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[5] Broadcom First to Deliver 64 Ports of 100GE with Tomahawk II 6.4Tbps Ethernet Switch. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[6] Cavium XPliant Switches and Microsoft Azure Networking Achieve SAI Routing Interoperability. http://www.cavium.com/newsevents-Cavium-XPliant-Switches-and-Microsoft-Azure-Networking-Achieve-SAI-Routing-Interoperability.html.

[7] Cisco IOS NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[8] Configuring SPAN. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2940/software/release/12-1_19_ea1/configuration/guide/2940scg_1/swspan.html.

[9] Data center flow telemetry. http://www.cisco.com/c/en/us/products/collateral/data-center-analytics/tetration-analytics/white-paper-c11-737366.html.

[10] Gigamon. https://www.gigamon.com/products/visibility-nodes/visibility-appliances.html.

[11] How Fast is Redis? http://redis.io/topics/benchmarks.

[12] In-band Network Telemetry. https://github.com/p4lang/p4factory/tree/master/apps/int.

[13] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[14] Intel64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[15] Marple proofs. http://web.mit.edu/marple/marple_tr.pdf.

[16] Microsoft bets big on SDN. https://azure.microsoft.com/en-us/blog/microsoft-showcases-software-defined-networking-innovation-at-sigcomm-v2/.

[17] Multiply-accumulate operation. https://en.wikipedia.org/wiki/Multiply-accumulate_operation.

[18] P4-16 Language Specification. http://p4.org/wp-content/uploads/2016/12/P4_16-prerelease-Dec_16.html.

[19] P4 Behavioral Model. https://github.com/p4lang/behavioral-model.

[20] Redis. http://redis.io/.

[21] sFlow. https://en.wikipedia.org/wiki/SFlow.

[22] SRAM - ARM. https://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php.

[23] The CAIDA UCSD Anonymized Internet Traces 2014 - June. http://www.caida.org/data/passive/passive_2014_dataset.xml.

[24] The CAIDA UCSD Anonymized Internet Traces 2016 - April. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[25] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[26] The Future of Network Monitoring with Barefoot Networks. https://youtu.be/Gbm7kDHXR-o, 2017.

[27] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[28] Alizadeh, Mohammad. Empirical Traffic Generator. https://github.com/datacenter/empirical-traffic-gen, 2017.

[29] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[30] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 2006.

[31] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, 2016.

[32] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *ACM International Measurement Conference*, Nov. 2010.

[33] 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 *SIGCOMM*, 2013.

[34] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 2005.

[35] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[36] D. R. Ditzel and D. A. Patterson. Retrospective on High-level Language Computer Architecture. In *ISCA*, 1980.

[37] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-processing Platforms. In *NSDI*, 2012.

[38] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[39] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.

[40] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[41] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network Monitoring is a Streaming Analytics Problem. In *HOTNETS*, 2016.

[42] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[43] S. Hart, E. Frachtenberg, and M. Berezecki. Predicting Memcached Throughput Using Simulation and Modeling. In *Symposium on Theory of Modeling and Simulation*, 2012.

[44] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *SIGCOMM*, 2014.

[45] S. P. Jones and P. Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, 2007.

[46] M. Kumar and K. Prasad. Auto-learning of MAC addresses and lexicographic lookup of hardware database. US Patent App. 10/747,332.

[47] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *HotNets*, 2010.

[48] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*, 2016.

[49] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[50] W. M. McKeeman. Language directed computer design. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, 1967.

[51] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[52] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *SIGCOMM*, 2014.

[53] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, 2016.

[54] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *NSDI*, 2016.

[55] P. Phaal. SFlow sampling rates, 2016. http://blog.sflow.com/2009/06/sampling-rates.html.

[56] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[57] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[58] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *OSDI*, 2016.

[59] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of SOAR: Smalltalk on a RISC. In *ISCA*, 1984.

[60] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[61] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, 2009.

[62] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, 2011.

[63] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[64] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[65] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, 2015.