Life in the Fast Lane: A Line-Rate Linear Road

Theo Jepsen
Università della Svizzera italiana
Barefoot Networks

Masoud Moshref
Barefoot Networks

Antonio Carzaniga
Università della Svizzera italiana

Nate Foster
Barefoot Networks
Cornell University

Robert Soulé
Università della Svizzera italiana
Barefoot Networks

ABSTRACT

This paper explores the question: what abstractions are needed to support a more general form of stateful processing in programmable forwarding planes? It argues that we should look for clues from the domain of stream processing. As a case study, it describes an implementation of the Linear Road benchmark for stream processing systems written in P4. The artifact of our implementation, which runs on a programmable ASIC, provides a version of the benchmark that far exceeds the throughput of any prior work. More importantly, the experience provides perspective on the challenges for implementing stateful abstractions in P4.

CCS CONCEPTS

• **Networks** → **In-network processing**; • **Information systems** → *Database management system engines*;

KEYWORDS

Programmable switches; Stream processing

ACM Reference Format:
Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *SOSR '18: SOSR '18: Symposium on SDN Research, March 28–29, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3185467.3185494

1 INTRODUCTION

Until recently, it was commonly believed that network hardware needed to be simple and fixed; Performing computations in the data plane would make the network slow and the hardware expensive. However, this view appears to be changing, as a new generation of programmable switches which match the performance of fixed function devices has become commercially available [6, 32].

As a result, developers and network operators have begun to question widely held assumptions about the division of labor between the network and application layers [26]. Several recent projects have explored using this new hardware to offload or accelerate services that traditionally live outside the network. Some examples include consensus protocols [9, 10, 21], in-network caching [18], and conflict resolution for transaction processing [20].

One common feature of all these applications is that they depend on stateful computations to an extent that was not originally expected or intended by hardware and language designers focused primarily on the networking domain. If this trend continues—as appears likely—then it is worth identifying which abstractions are needed to support a more general form of stateful processing.

When trying to answer this question, a promising place to look for clues is the domain of stream processing. The database community, which developed stream processing, has decades of experience with "programming with tables" [8]. Stream processing naturally generalizes the basic model of computation offered by these devices to a graph, where the edges are streams and the nodes are operators. In this model, a *stream* is a continuous ordered sequence of data items (i.e., packets) and *operators* are stream transformers that may have state and side-effects beyond the output streams they produce (i.e., tables and actions) [15].

As a first step in this direction, we propose focusing on the Linear Road benchmark for stream processing systems [4]. Linear Road is a streaming application that monitors vehicles on a highway using five queries: three of which execute on streaming data, and two of which execute on historical data.

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185494

Linear Road provides a good case study for stateful data plane programming, since it is a relatively small, but semantically rich example of stateful streaming computation.

Linear Road has been implemented on a number of stream processing engines, including the STREAM data stream management system [5] from Stanford University; the Aurora system [2] from Brandeis University, Brown University and MIT; and IBM's Infosphere Streams [17].

In this paper, we present an implementation of Linear Road in P4. Implementing the query processing logic is challenging due to constraints imposed both in the language and in the target hardware. On the language side, P4 is, by construction, not a Turing-complete language; it excludes looping constructs, which are undesirable in hardware pipelines. This makes it difficult to implement common relational operators, such as join, which are used in the queries. At the hardware level, there are restrictions on state accesses, available storage, and depth of the processing pipeline.

We have implemented all five benchmark queries in a P4 program running on Barefoot Network's Tofino ASIC [6]. The implementation can process over 4 billion events/second. This is orders of magnitude faster than the most recent published implementation of Linear Road [17].

Beyond the artifact of our implementation of the benchmark in P4, the exercise provides a perspective on the challenges for implementing stateful abstractions in the P4 language. We hope that by describing our experiences, we will generate discussion within the larger community about the requirements for stateful computations in P4 and data-plane languages in general.

Overall, this paper makes the following contributions:

- It presents an implementation of the Linear Road stream processing benchmark that can run at line speed.
- It describes general techniques for implementing stateful processing on chips with an RMT architecture [7].
- It discusses the feasibility and limitations of the stateful abstraction for stream processing in the data plane.

We first provide background on Linear Road (§2), before describing the details of our implementation (§3). We then present a general discussion (§4), followed by a brief evaluation (§5), related work (§6), and concluding remarks (§7).

2 THE LINEAR ROAD BENCHMARK

Linear Road is a simulation of a hypothetical application that computes tolls for vehicles on a highway system that consists of L expressways traveling from east to west. Each expressway is divided into 100 segments, each with five lanes, including an entrance and an exit ramp. Vehicles pay a toll when they drive on a congested highway (where the average speed of all vehicles is under 40 mph in a 5 minute span).

The benchmark queries receive input from both a set of continuous streams of data, and some pre-loaded, historical data referred to as relations. The four input data streams are:

- *Position Report*: a message periodically emitted by each vehicle with its current speed and location.
- *Account Balance Request*: a request for the sum of tolls assessed to a vehicle since the start of the simulation.
- *Daily Expenditure Request*: a request for the sum of tolls on a specific day from the historical data.
- *Travel Time Request*: a request for the estimated travel time between two segments.

The historical data includes the following two relations:

- *TollHistory*: for each day, for each vehicle, for each expressway, the total of tolls assessed.
- *SegmentHistory*: for every minute, the average speed and sum of tolls assessed in each segment of each expressway.

Each stream or relation has a schema that specifies the names of the attributes. For example, a *PositionReport* stream has the following schema:

```
(Time, VID, Spd, XWay, Seg, Lane, Dir)
```

where `Time` is the number of seconds since the start of the simulation, `VID` is the vehicle's identifier, `Spd` is the vehicle's current speed, and `XWay`, `Seg`, `Lane`, and `Dir` indicate the vehicles location (expressway, segment, lane, and direction, respectively). Arasu et al. provide a complete benchmark specification [4].

The benchmark defines five queries that compute the outputs from the input streams and relations. In prose, these queries are as follows:

- *Toll Notification:* Upon entering a segment of an expressway, a vehicle should be notified of the toll for that segment, which is based on the segment's level of congestion.
- *Accident Alert:* A vehicle travelling up to 4 segments upstream from an accident (detected as two or more vehicles stopped in the same lane) should be notified.
- *Account Balance:* Upon requesting an account balance, a vehicle should receive a response with the sum of tolls for that vehicle since the beginning of the simulation.
- *Daily Expenditures:* A request for the sum of tolls for a vehicle on a given day on a given expressway. This should be computed from the *TollHistory* historical data.
- *Travel Time Estimation:* Given a time of day and day of week, calculate the estimated travel time between two segments (computed from *SegmentHistory* historical data).

As a workload, we use sample input data available on the benchmark website [22].

3 P4 LINEAR ROAD

Using the P4 language, we implemented two versions of the benchmark: one that runs in the software Behavioral Model,

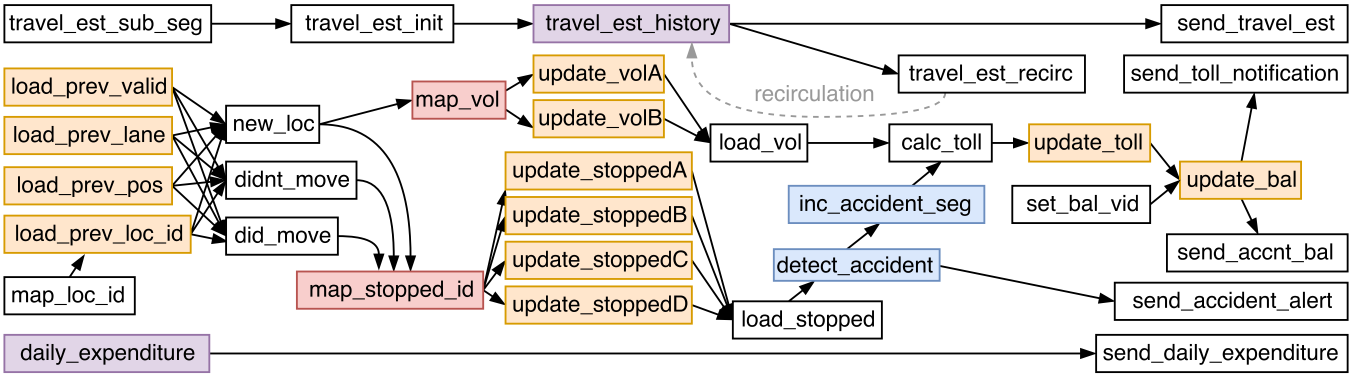

Figure 1: Tables and control flow of P4 Linear Road. Colors indicate a particular implementation technique.

and one that runs on the Tofino ASIC. Below, we describe the key implementation techniques, as well as some limitations in our implementation.

Our forwarding-plane version of Linear Road depends on the abstractions offered by the P4 language, which reflects the structure of the target hardware—i.e., the RMT architecture [7]. A RMT architecture has a pipeline of logical match-action units with local memory. Each match-action unit imposes a strict ordering on operations; all data reads must occur before all writes. There are also a number of physical constraints, e.g., a fixed number of match units in a pipeline; a limited amount of available SRAM and TCAM; and each TCAM can only return a single result from a match (i.e., the highest priority match).

Our implementation runs on a single switch which receives and emits streams of UDP packets. A stream tuple is encoded as a P4 header with fixed-width fields, including a field that specifies the tuple type (e.g. a *PositionReport* or *AccidentAlert*). Figure 1 illustrates the tables and control flow of our P4 program. The arrows indicate the direction packets flow through the pipeline. The colors indicate locations in the pipeline where we use different implementation techniques.

3.1 Implementation Techniques

To cope with the above constraints, our implementation relies on several techniques, which we describe below.

Incremental Operator Computations. All queries must perform their computations incrementally. That is, for every input tuple (i.e., packet), the operator computes the differences in state relative to the previous operator invocation. This approach reduces memory usage, as the query only maintains a limited amount of incremental state. This technique is used in the orange tables in Figure 1.

For example, the *Toll Notification* query checks the number of vehicles in a segment, as well as their average velocity. This requires storing two state aggregates per segment: a counter for the number of vehicles, and an average of their speeds. If a *Position Report* indicates that a car crossed into a segment, then the previous segment's counter is decremented and the next segment's is incremented.

Explicit Loop Unrolling. P4 excludes looping constructs, which are undesirable in hardware pipelines. Therefore, all loops in our queries must be explicitly unrolled. For example, the *Accident Alert* and *Toll Notification* queries must explicitly check the next four segments for stopped cars, which is done in the blue tables in Figure 1.

Multiple Register Arrays. Based on the report from Sharma et al. [27], we assume that a single index of the same register can be accessed (first a read, then a write) in a stage. Therefore, to implement queries that need to access multiple indexes, we partitioned the data into multiple registers (highlighted red in Figure 1). For example, when a car crosses from segment 5 to 6, the query must decrement the volume of cars in the previous segment, and increment the next. Rather than express this as `segment[5]--`; `segment[6]++`, we keep two register arrays, one for even and one for odd segments: `segmentOdd[5/2]--`; `segmentEven[6/2]++`.

Pre-computed Historical State. The purple tables in Figure 1 store data for answering historical requests. Using tables, rather than registers, simplifies look-ups, since the match already implements the logic for reading by keys. The controller inserts the tuples from *TollHistory* and *SegmentHistory* into two separate tables. When the switch evaluates a historical query, the row matching the query is selected.

The *Travel Time Estimation* query selects multiple rows (one for each segment in a path). However, the match-action paradigm only returns one table entry per lookup. To find all the entries, our implementation recirculates the packet through the egress pipeline: on each recirculation the sum of estimated travel time is incremented with the next entry in the table, and finally the sum is sent in the output packet. This is indicated by the gray dashed arrow in Figure 1.

Another technique for selecting all the entries could be to create multiple replicas of the packet, assigning each a different key, and multicasting them to the egress pipeline. Each replica would match a different entry in the table, and update a common register. The final replica would read and output the accumulated value from that register. This technique would use the same amount of bandwidth as the one we implemented, but have lower latency, since the replicas could be processed in parallel. However, this would require an additional stage and a register to accumulate the partial result from each replica. It is only applicable if the accumulation operation is commutative (due to parallelization).

We assume memory is local to a pipe and not shared between pipes. To increase the amount of memory available, values can be partitioned among multiple pipes; if a query arrives in a certain pipe, but requires values stored in another pipe, the query can be recirculated to that pipe, as done in NetCache [18]. Of course, this requires re-circulation, and would therefore reduce throughput.

Over-allocation of Resources. Our implementation stores vehicle state in registers. To lookup the state of a vehicle, the VID is used as an index into the registers. If VIDs are sparse, then register space will be wasted.

Passing State Through Stages. A register array is stored in a specific pipeline stage, and thus can only be accessed in that stage. Since computation in a stage may require state that is read in a previous stage, P4 metadata is used to pass state between stages. After reading/writing state to a register, the state is also written to a metadata field, so that it can be used in a subsequent stage (e.g. for determining whether a query evaluation has been triggered).

3.2 Deviations from Specification

Our hardware implementation diverges from the original Linear Road specification in some relatively minor aspects:

Lane detection. The original specification requires that an accident should be detected when two or more cars are stopped in the same *lane*; our implementation checks whether they are stopped in the same *segment*. This is due to the restriction on the number of stages in a RMT machine.

Time-based Average. The original specification requires that the queries calculate the average speed in a 5 minute window. Doing so would require us to maintain values for 5 minutes. Maintaining a sliding window on an ASIC is difficult. So, we use a hardware-supported low-pass filter (LPF) to calculate the average using single exponential smoothing [1]. Since a LPF is not window-based, we cannot compare it to the specification's window approach for accuracy.

4 TOWARDS A GENERAL QUERY LANGUAGE

In the previous section, we described an implementation of the Linear Road benchmark in P4. We chose to focus on Linear Road for several reasons: (i) it has small scale in input data stream and queries, making a switch-based deployment feasible; (ii) it has a clearly defined semantics [4], allowing us to verify the correctness of our implementations; and (iii) it has been generally adapted to a number of streaming engines (e.g., [2, 3, 5, 17]), indicating that it is representative of stream processing applications.

However, the more general question this work addresses is: *how feasible would it be to implement general abstractions from streaming languages targeting the RMT architecture via P4*? In this section, we expand our discussion to more general abstractions from the domain of stream processing systems that could be adapted for use in a programmable data plane.

There are, of course, many stream processing languages (e.g., [3, 13, 24, 25, 29, 33]). Although they are all different, we make two broad generalizations that we believe are useful. First, many stream languages distinguish between two types of inputs: data from time varying streams (which is updated continuously) and data from relations (which is mostly static). Second, many of them are based on streaming-specific extensions to SQL, which in term is based on relational algebra [8]. Below, we organize our discussion along these lines.

4.1 Input Data

Because the handling and storage requirements for transient, continuous data may differ significantly from historical data, it is useful to distinguish between two types of inputs: time varying streams and static relations.

Time Varying Streams. Time-varying streams are data that arrives continuously in online fashion. Stream processing systems try to process this data with high throughput and low latency (average and tail). Programmable data planes are well-suited to processing this type of data, as it is similar to processing network packets. This data is typically associated with a timestamp, which must be either an explicit attribute (i.e., packet header field), or acquired from the hardware. P4 does not provide a built-in function for accessing a hardware timestamp, but this can be exposed by the architecture, such as Portable Switch Architecture (PSA).

[1] $\text{avg}_n = \alpha\text{avg}_{n-1} + (1-\alpha)x_n$, where weight $0 < \alpha < 1$ for the nth observation x_n.

Static Relations. Static Relations are used to store historical data. Depending on the needs of the query, this data may or may not be pre-aggregated. There are two methods for storing static data with different trade-offs: (i) as pre-loaded data in action parameters stored in SRAM/TCAM match tables, or (ii) in registers, index by a key coming from a match table or a hash.

Match tables can only be programmed from the control-plane which has limited throughput. Although they cannot be used for instantaneous data, they perfectly suit the historical data where the pre-loading latency is not critical.

Registers are more general and can be programmed and queried from data-plane and control-plane. However, the number of registers and the bitwidth that can be read in each stage is an order of magnitude smaller than tables.

Beyond storing the data, there must be ways to quickly access the data with one or more *keys*. For both methods, keys can be generated from a match table or a hash function. Match tables do not need to save all possible combinations for historical data. Since the control-plane can pre-process the data, it can program only the keys used in the match tables, thus reducing memory consumption. On the other hand, if we use the hash for key, we do not need the match table. But, if key indexes are sparse, then memory might be over-allocated. Depending on the hash function and key space, there may be hash collisions. Depending on the needs of the application, such collisions may or may not be tolerable.

Note that the ASIC cannot guarantee that historical data is persistent. However, in most streaming systems, the application needs only to consult historical data when answering a query (i.e., persistent storage is not a requirement). So, data can be re-loaded into the device in the event of a memory failure or device restart.

4.2 Query Operators

Many streaming languages are based on SQL, which is based on relational algebra [8]. The standard relational operators include *set operations*, such as union and difference; *join operations* to correlate data; *filter operations*, such as selection and projection, and *aggregations*. Streaming languages extend these operators with *window operators*, which convert an input stream into a relation.

Set Operators. Some operators perform set operations, such as union, difference or Cartesian product. Implementing these set-based operators would require some form of iteration. For example, one naïve implementation strategy would be to store data as "tables" in an array of registers. In this strategy, each "row" could be implemented as a P4 metadata structure, and operators would iterate over these structures to perform their computations. However, this design is impractical on a switch for two reasons. First, storing the individual tuples of the input stream requires a prohibitive amount of memory (SRAM). Second, iterating over the table rows would be difficult to express in P4 and could not be done at line-rate, as it would likely require re-circulation through the pipeline.

	Time-based	Count-based	Event-based
Sliding		✓	
Tumbling	✓	✓	✓

Table 1: Categories of windowed operators. Checks indicate windowing that is implementable in P4.

Join Operators. Performing a join in hardware can be implemented if the sets being joined have constant sizes. In our Linear Road implementation, the vehicle state registers are joined on the VID. In this case, each register contains exactly one entry for each VID, so this operation can be implemented with a fixed amount of memory and without iterating.

Filtering Operators. Selections and projections are straightforward to implement as a single transformation. A selection simply matches on a field, for which switch hardware is specialized. Projections can be implemented by modifying fields, or adding or removing new fields (i.e. with the P4 `add_header()` primitive). Depending on the computational resources of the hardware (e.g. arithmetic operations supported by the ALU), some projections may not be possible.

Aggregation Operators. Aggregates, such as average, count, sum and min/max, are computed over a window. The aggregate operators that require a fixed-size window can be implemented incrementally by updating the latest value, which is stored in a register array. To index into the register array, the input tuple's key (or hash of the key) can be used as an index. If the queries can tolerate bounded error, an approximate data structure (sketch) can be used. For example, to count the number of unique items, a cardinality estimator such as linear counting [31] or hyperloglog [12] can be implemented using hashing and registers in data plane [16, 28].

Window Operators. Many streaming queries require window operations, which intuitively convert a stream into a relation. In general, there are many types of windows. Table 1 provides a summary of the main categories. A window may be *sliding* or *tumbling*. A sliding window is a series of fixed-sized, overlapping, contiguous time slices. The window advances by a *slide size*, and events outside of the slide are evicted, while new events are added. A tumbling window is a series of fixed-sized, non-overlapping, contiguous time slices. At the end of each interval, the window "tumbles" and all data items are evicted. Both sliding and tumbling windows may be time-based, count-based, or event based.

In P4/Tofino, it is feasible to implement count-based and tumbling windows as they both have a static size. Sliding time-based and event-based windows, however, require a dynamic window size, because the window can grow to an arbitrary size during the interval or before the event. They are thus impractical to implement, because of the memory and iteration constraints already described.

Linear Road makes use of sliding time-based windows (e.g. for calculating average speed in a 5 minute interval), and count-based tumbling windows (e.g. counting vehicles in the same segment). To approximate the sliding time-based window for average speed, in our implementation we use single exponential smoothing.

4.3 Summary

P4 primitives and data structures can be used to express many operators, albeit some more cumbersomely (e.g. with loop unrolling). We found that data planes are well-suited for streaming operations that do not require looping and that require a bounded amount of state. Overall, the stream processing model maps surprisingly well onto the P4 programmable switch hardware abstraction. That we were able to implement the whole Linear Road benchmark demonstrates the expressive power of this new programmable substrate.

5 EVALUATION

We implemented two versions of the benchmark using $P4_{14}$: one that runs in the software Behavioral Model (BMv2) with 1,263 lines of code (LoC), and one that runs on the Tofino ASIC with 1,335 LoC. The code could be easily converted to $P4_{16}$. We created a Python library (560 LoC) that parses the sample workload data available on the Linear Road website [22], and outputs packets to be sent to the switch. The BMv2 version of the code is publicly available[2].

We deployed our Linear Road implementation on a 64-port ToR switch, with Barefoot Network's Tofino ASIC [6]. As per standard practice in industry for benchmarking switch performance, we used a snake test: each port is looped-back to the next port, so a packet passes through every port before being sent out the last port. This is equivalent to receiving 64 replicas of the same packet. To generate and receive traffic, we used an Ixia XGS12-H hardware packet tester, connected to the switch with 100G QSFP+ direct-attached copper cables. We verified that P4 Linear Road can process over 4 billion events/second. Furthermore, the P4 Linear Road packet processing pipeline has a fixed latency that is orders of magnitude lower than that of software implementations.

For comparison, the most recent published implementation of Linear Road [17] running on a single node (dual-core 3GHz Xeon CPU with 2GB RAM) could handle around 2M events/second with 1.67 seconds of latency. A more recent streaming engine, Drizzle [30], does not evaluate Linear Road, but reports 100M events/second for the Yahoo streaming benchmark, using 128 `r3.xlarge` Amazon EC2 instances. We report these numbers simply to give context for our performance; this is not an apples-to-apples comparison because these other systems have different capabilities (e.g. persistent memory, complex transformations, and fault-tolerance).

6 RELATED WORK

P4 Benchmarks. Whippersnapper [11] is a P4 benchmark. This paper describes an implementation of a benchmark as a case study for stateful dataplane programming.

Implementations of Linear Road. Linear Road was first described in a language-agnostic logical specification [4]. It has since been implemented for various streaming systems. Notably, a version written in CQL [3] ran on the Stanford STREAM data stream management system [5]. It was used to benchmark the Aurora [2] and Borealis [1] streaming engines. Jain et al. describe an implementation written in SPL [14] running on IBM's Infosphere Streams [17].

Stateful Dataplane Applications. Several recent projects have made heavy use of state in the forwarding plane to offload or accelerate systems services. Marple [23] uses stream processing techniques to process telemetry data on switches. NetCache [18] implements a key-value store. NetPaxos [9, 10] offers consensus as a network service. Eris [20] orders transactions in an optimistic-concurrency control system to avoid aborts. Jose et al. [19] describe a congestion control mechanism that leverages switch statistics.

7 CONCLUSION

This paper argues for using stream processing as a model for the types of abstractions we will need to support general stateful computations in programmable network hardware. As a case study, it describes an implementation of Linear Road benchmark in P4. This exercise not only provides a line-rate implementation of Linear Road, but also helps to identify constraints and challenges for stateful processing. As developers and network operators continue to explore ways to leverage this new hardware to offload or accelerate services, this work highlights the pressing need for new language abstractions.

Acknowledgments. We thank the anonymous SOSR reviewers and our shepherd, Anirudh Sivaraman, for their valuable feedback. This work is supported in part by SNF 167173. We thank Changhoon Kim for helpful discussions.

[2] https://github.com/usi-systems/p4linearroad

REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, Jan. 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. In *The VLDB Journal*, volume 12, pages 120–139, Aug. 2003.

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. In *The VLDB Journal*, volume 15, pages 121–142, June 2006.

[4] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *30th International Conference on Very Large Data Bases*, pages 480–491. VLDB, Aug. 2004.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Principles of Database Systems (PODS)*, pages 1–16, June 2002.

[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 *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, Aug. 2013.

[7] 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 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, Aug. 2013.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[9] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, Apr. 2016.

[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 59–73, June 2015.

[11] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon. Whippersnapper: A p4 language benchmark suite. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 95–101, Apr. 2017.

[12] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156, June 2007.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1123–1134, June 2008.

[14] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development (IBMRD)*, 57(3/4):7:1–7:11, May/July 2013.

[15] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), Apr. 2014.

[16] In-Network DDoS Detection. https://barefootnetworks.com/use-cases/in-nw-DDoS-detection, 2017.

[17] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, June 2006.

[18] 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)*, pages 121–136. ACM, Oct. 2017.

[19] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High speed networks need proactive congestion control. In *Workshop on Hot Topics in Networks (HotNets)*, Nov. 2015.

[20] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120. ACM, Oct. 2017.

[21] 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.

[22] Linear Road. http://www.cs.brandeis.edu/~linearroad/.

[23] 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 *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 85–98. ACM, Aug. 2017.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, June 2008.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, pages 277–298, 2005.

[26] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2:277–288, Nov. 1984.

[27] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 67–82, Mar. 2017.

[28] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 67–82, Mar. 2017.

[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, pages 179–196, Apr. 2002.

[30] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 374–389. ACM, Oct. 2017.

[31] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

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

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, Dec. 2008.