Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks

Anurag Khandelwal, *UC Berkeley;* **Rachit Agarwal,** *Cornell University;*
Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/nsdi19/presentation/khandelwal

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks

Anurag Khandelwal
UC Berkeley

Rachit Agarwal
Cornell University

Ion Stoica
UC Berkeley

Abstract

Confluo is an end-host stack that can be integrated with existing network management tools to enable monitoring and diagnosis of network-wide events using telemetry data distributed across end-hosts, even for high-speed networks. Confluo achieves these properties using a new data structure — Atomic MultiLog— that supports highly-concurrent read-write operations by exploiting two properties specific to telemetry data: (1) once processed by the stack, the data is neither updated nor deleted; and (2) each field in the data has a fixed pre-defined size. Our evaluation results show that, for packet sizes 128B or larger, Confluo executes thousands of triggers and tens of filters at line rate (for 10Gbps links) using a single core.

1 Introduction

Recent years have witnessed tremendous progress on (the notoriously hard problem of) network monitoring and diagnosis by exploiting programmable network hardware [1–18]. This progress has been along two complementary dimensions. First, elegant data structures and interfaces have been designed that enable capturing increasingly rich telemetry data at network switches [1–6, 10, 13–17]. On the other hand, recent work [6–12] has shown that capitalizing on the benefits of above data structures and interfaces does not need to be gated upon the availability of network switches with large data plane resources — switches can store a small amount of state to enable in-network visibility, and can embed rich telemetry data in the packet headers; individual end-hosts monitor local packet header logs for monitoring spurious network events. When a spurious network event is triggered, network operator can diagnose the root cause of the event using switch state along with packet header logs distributed across end-hosts [7–10].

Programmable switches have indeed been the enabling factor for this progress — on design and implementation of novel interfaces to collect increasingly rich telemetry data, and on flexible packet processing to embed this data into the packet headers. To collect these packet headers and to use them for monitoring and diagnosis purposes, however, we need end-host stacks that can support:

- **monitoring of rich telemetry data** embedded in packet headers, *e.g.*, packet trajectory [7–11], queue lengths [1, 10], ingress and egress timestamps [10], etc. (§2.2);
- **low-overhead diagnosis** of network events by network operator, using header logs distributed across end-hosts;
- **highly-concurrent low-overhead read-write operations** for capturing headers, and for using the header logs for monitoring and diagnosis purposes using minimal CPU resources. The challenge here is that, depending on packet sizes, monitoring headers at line rate even for 10Gbps links requires 0.9-16 million operations per second!

Unfortunately, end-host monitoring and diagnosis stacks have not kept up with advances in programmable hardware and are unable to simultaneously support these three functionalities (§2.1, §6). Existing stacks that support monitoring of rich telemetry data (*e.g.*, OpenSOC [19], Tigon [20], Gigascope [21], Tribeca [22] and PathDump [8]) use general-purpose streaming and time-series data processing systems; we show in §2.1 that these systems are unable to sustain the target throughput even for 10Gbps links. This limitation has motivated design of stacks (*e.g.*, Trumpet [23]) that can monitor traffic at 10Gbps using a single core, but only by limiting the functionality — they do not support monitoring of even basic telemetry data like packet trajectory and queue lengths; we discuss in §2.1 that this is in fact a fundamental design constraint in these stacks.

Confluo is an end-host stack, designed and optimized for high-speed networks, that can be integrated with existing network management tools to enable monitoring and diagnosis of network-wide events using telemetry data distributed across end-hosts. Confluo simultaneously supports the above three functionalities by exploiting two properties specific to telemetry data and applications. First, telemetry data has a special structure: once headers are processed in the stack, these headers are not updated and are only aggregated over

long time scales. Second, unlike traditional databases where each record may have fields of arbitrary size, packet headers capture a precise protocol with fixed field sizes (*e.g.*, 32-bit IP addresses, 16-bit port numbers, 16-bit switchIDs [8–10], 16-bit queue lengths [1, 10], 32-bit timestamps [10], etc.)[1].

Confluo achieves its goals using a new data structure — Atomic MultiLog— that exploits the above two properties of telemetry data to trim down traditional lock-free concurrency mechanisms to a bare minimum without sacrificing correctness guarantees. A MultiLog, as the name suggests, generalizes traditional logs into a *collection* of lock-free logs. Atomic MultiLog uses a collection of such logs, one for each of the filters and aggregates (for monitoring purposes), one for each of the materialized views (for diagnosis purposes), and one for raw header logs. Atomic MultiLogs use the first property to efficiently maintain an updated view of these logs upon receiving new headers (each new header may incur multiple concurrent write operations on Atomic MultiLog for updating individual logs). Essentially, we show that the first property allows trimming down the traditional lock-free concurrency mechanisms to updating two integers per header (§3); using atomic hardware primitives readily available in commodity servers, Atomic MultiLog is able to ingest millions of headers per second using a single CPU core.

As headers are processed in the stack, Confluo also needs to simultaneously execute monitoring and diagnosis queries that, in turn, require executing multiple concurrent read operations on Atomic MultiLogs. We show that having fixed field sizes in packet headers makes it extremely simple to handle race conditions for concurrent reads and writes over individual logs within an Atomic MultiLog. Finally, we show that these two properties allow Atomic MultiLog to not only achieve highly-concurrent read and write operations but to also support two strong distributed systems properties. First, updates to all the individual logs within an Atomic MultiLog are visible to the monitoring and diagnosis application atomically (formal proofs in [24]); and second, atomic snapshots of telemetry data distributed across the end-hosts can be obtained using a simple distributed algorithm (§4).

Confluo implementation is now open-sourced [25], with an API that is expressive enough to integrate Confluo with most existing end-host based monitoring and diagnosis systems [8–11, 23]. We have compiled an exhaustive list of monitoring and diagnosis applications from these systems; we show, in [24], that our implementation already supports all these applications. Evaluation of Confluo using packet traces from standard generators [26,27], and from real testbeds [8, 9] shows that, even for 128B packets, Confluo executes thousands of triggers and tens of filters at line rate (for 10Gbps links) using a single core. Moreover, for 40Gbps links and beyond, where multiple cores may be necessary, Confluo's performance scales well with number of cores.

[1]Packet headers can contain arbitrary number of fields, and the number of fields may vary across each packet; however, each field has a fixed size.

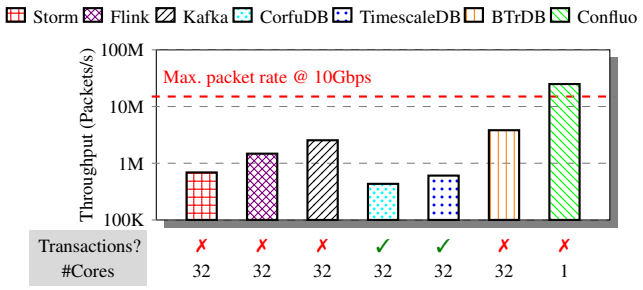


Figure 1: Header ingestion rates (no filters, aggregates, or indexes) for several open-sourced streaming and time-series data processing systems, and for Confluo, on a single end-host. The workload uses 64B TCP packets using DPDK's pktgen tool [28]. Unfortunately, existing systems are unable to sustain write rates for 10Gbps links, even when using 32 cores. Note that: (1) CorfuDB and TimescaleDB tradeoff write rates for stronger semantics; (2) BTrDB results use 16B packet prefixes since it does not support larger entries; (3) Storm and Flink results use Kafka as a data sink since these systems do not store data. See §2.1 for discussion.

2 Confluo Overview

This section provides an overview of Confluo. We start by elaborating on the observation that end-host monitoring and diagnosis stacks have not kept up with increasing network bandwidths and with advances in programmable network hardware (§2.1). We then outline Confluo interface, along with an example on how a network operator can use this interface for monitoring and diagnosis (§2.2). We conclude the section with a high-level overview of Confluo design (§2.3).

2.1 Motivation

Existing end-host stacks fall short of simultaneously supporting the three functionalities outlined in the introduction either because they cannot scale to large network bandwidths (10Gbps and beyond), or do not support monitoring of rich telemetry data (*e.g.*, packet trajectory, queue lengths, ingress and egress timestamps, and many others outlined in [10]). We discuss these challenges next.

Challenges with larger network bandwidths. Existing end-host monitoring stacks that support rich telemetry data (*e.g.*, Time Machine [29], Gigascope [21], Tribeca [22]) were designed for 1Gbps links, with reported performance of 180-610 Mbit/sec [21] and 20-30k headers/sec [22]. While these systems are not available for evaluation, they are unlikely to scale to 10Gbps and higher link bandwidths since this would require processing 10-100× more headers. To overcome this limitation, recently developed stacks [8, 9, 19, 20] use open-source streaming and time-series data processing systems. However, as shown in Figure 1, these systems are unable to support write rates at 10Gbps even when using 32 cores. We believe that the fundamental reason behind this limitation is that these systems are targeting data types that are too general — supporting the three functionalities outlined in the introduction with minimal CPU resources requires exploiting the

Table 1: Confluo's End-Host API. In addition, Confluo exposes certain API to the coordinator to facilitate distributed snapshot (§4). All supported operations are guaranteed to be atomic. See §2.2 for definitions and detailed discussion.

	API	Description
	`setup_packet_capture(fExpression, sampleRatio)`	Capture packet headers matching filter `fExpression` at `sampleRatio`.
Monitoring	`filterId = add_filter(fExpression)`	Add filter `fExpression` on incoming packet headers.
	`aggId = add_aggregate(filterId, aFunction)`	Add aggregate `aFunction` on headers filtered by `filterId`.
	`trigId = install_trigger(aggId,condition,period)`	Install trigger over aggregate `aggId` evaluating `condition` every `period`.
	`remove_filter(filterId), remove_aggregate(aggId), uninstall_trigger(trigId)`	Remove or uninstall specified filter, aggregate or trigger.
Diagnosis	`add_index(attribute)`	Add an index on a packet header `attribute`.
	`Iterator<Header> it = query(fExpression,tLo,tHi)`	Filter headers matching `fExpression` during time `(tLo,tHi)`.
	`agg = aggregate(fExpression,aFunction,tLo,tHi)`	Compute aggregate `aFunction` on headers matching `fExpression` during time `(tLo,tHi)`.
	`remove_index(attribute)`	Remove index for specified packet header `attribute`.

Table 2: Elements of Confluo filters, aggregates and triggers.

	Operator	Examples
Relational	**Equality**	`dstPort==80`
	Range	`ipTTL<3, srcIP in 10.1.3.0/24`
	Wildcard	`dstIP like 192.*.*.1`
Boolean	**Conjunction**	`srcIP=10.1.3.2 && pktSize<100B`
	Disjunction	`dstPort==80 || dstPort==443`
	Negation	`protocol!=TCP`
Aggregate	**AVG**	`AVG(ipTTL)`
	COUNT, SUM	`COUNT(ecn), SUM(pktSize)`
	MAX, MIN	`MIN(ipTOS), MAX(tcpRxWin)`

specific structure in network packet headers, especially for 40-100Gbps links where multiple cores may be necessary to process packet headers at line rate.

Challenges with monitoring rich telemetry data. The aforementioned limitations of streaming and time-series data processing systems have motivated custom-designed end-host monitoring stacks [23, 30–34]. State-of-the-art among these stacks (*e.g.*, Trumpet [23] and FloSIS [34]) can operate at high link speeds — Trumpet enables monitoring at line rate for 10Gbps links using a single core; similarly, FloSIS can support offline diagnosis for up to 40Gbps links using multiple cores. However, these systems achieve such high performance either by giving up on online monitoring (*e.g.*, FloSIS) or by applying filters only on the first packet in the flow (*e.g.*, Trumpet). This is a rather fundamental limitation and severely limits how rich telemetry data embedded in the packet headers is utilized — for instance, since header state (*e.g.*, trajectories or timestamps) may vary across packets, monitoring and diagnosing network events requires applying filters to each packet [6, 8, 9, 18]. For instance, if a packet is rerouted due to failures or bugs, its trajectory in the header could be used to raise an alarm [8, 9, 18]; however, if this is not the first packet in the flow, optimizations like those in Trumpet will fail to trigger this network event[2]. On the other hand, if filters were applied to each and every packet, these systems will observe significantly worse performance.

2.2 Confluo Interface

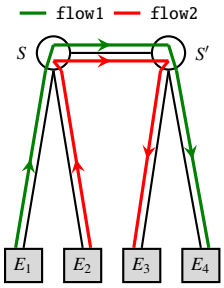
We now describe Confluo interface. Confluo is designed to integrate with existing tools that require a high-performant end-host stack [8, 9, 11, 12, 23]. To that end, Confluo exposes an interface that is expressive enough to enable integration with most existing tools; we discuss, in [24], that Confluo interface already allows implementing all applications from recent end-host monitoring and diagnosis systems.

Confluo operates on packet headers, where each header is associated with a number of attributes that may be protocol-specific (*e.g.*, attributes in TCP header like `srcIP, dstIP, rwnd, ttl, seq, dup`) or custom-defined (*e.g.*, packet trajectories [8, 9, 11], or queue lengths [1, 10], timestamps [10], etc.). Confluo does not require packet headers to be fixed; each header can contain arbitrary number of fields, and the number of fields may vary across each packet.

API. Table 1 outlines Confluo's end-host API. While Confluo captures headers for all incoming packets by default, it can be configured to only capture headers matching a filter `fExpression`, sampled at a specific `sampleRatio`.

Confluo uses a match-action language similar to [8, 23] with three elements: *filters*, *aggregates* and *triggers*. A filter is an expression `fExpression` comprising of relational and boolean operators (Table 2) over an arbitrary subset of header attributes, and identifies headers that match `fExpression`. An aggregate evaluates a computable function (Table 2) on an attribute for all headers that match a certain filter expression. Finally, a trigger is a boolean `condition` (*e.g.*, `<, >, =`, etc.) evaluated over an aggregate.

[2]For some applications, detecting such cases may be necessary due to privacy laws. The canonical example here is that of a bug leading to incorrect packet forwarding and violating isolation constraints in datacenters storing patient information — patient data from two healthcare providers must never share the same network element due to HIPAA laws [35, 36]



Scenario	Monitoring	Diagnosis

flow1 rate+flow2 rate > bandwidth, flow1 priority = flow2 priority
Packet drops for flow1, flow2 at S
	Tracking retransmissions (rtms):
	`MAXSEQ((maxSeq, maxTs), pkt) {`
	`  if (pkt.seqNo > maxSeq)`
	`  return (pkt.seqNo, pkt.ts)`
	`  else return (maxSeq, maxTs)`
	`}`
	`SEQ,TS=add_aggregate(flow,MAXSEQ)`
	`cond = seqNo<SEQ && ts>TS+`t_{delay}
	`rtms = add_filter(cond)`
	`R = add_aggregate(rtms, COUNT)`
	`T = add_trigger(R, R>T, 1ms)`
		t = `T.timestamp`,
		p_1 = `flow1` priority, p_2 = `flow2` priority
		r_1 = `flow1` retransmits, r_2 = `flow2` retransmits,
		c_1 = `aggregate(`r_1`,COUNT,t-1ms,t)`,
		c_2 = `aggregate(`r_2`,COUNT,t-1ms,t)`,
		check if $c_1 \approx c_2 > 0$ && $p_1 = p_2$

flow1 rate+flow2 rate > bandwidth, flow1 priority < flow2 priority Packet drops for flow1 at S		$t, r_1, r_2, c_1, c_2, p_1, p_2 \rightarrow$ Same as above check if $c_1 \approx 0$ && $c_2 > 0$ && $p_1 < p_2$ or, $c_2 \approx 0$ && $c_1 > 0$ && $p_2 < p_1$
flow1 rate+flow2 rate < bandwidth, Bug at S drops based on packet timing, Packet drops for flow1, flow2 at S		t_i = Timestamp buckets of packets in rtms, $\delta_i = t_i - t_{i-1}$ and σ_δ = STDEV on δ_i check if AVG(δ_i) $\approx$ 100ms && $\sigma_\delta <$ 1ms

Figure 2: Examples of monitoring and diagnosis of network events in Confluo. See §2.2 for details.

Confluo supports ad-hoc filter queries and aggregates via indexes on arbitrary packet header attributes. These indexes serve to speed up diagnostic queries when filters or aggregates have not been pre-defined. We describe the design and implementation of Confluo indexes, filters, aggregates and triggers in §3.2 and §3.3.

Examples. Figure 2 shows Confluo functionality using a simple example comprising three scenarios where switch S is dropping packets. This example assumes that the monitoring and diagnosis application employing Confluo uses TCP retransmissions as an indicator of packet loss. A network operator can use Confluo to maintain an aggregate to determine the latest TCP sequence number SEQ and the corresponding packet timestamp TS in a flow. The operator then filters out packets that have TCP sequence number smaller than SEQ and timestamp larger than TS by a delay threshold (t_{delay}) as probable retransmissions. Confluo can then be configured to trigger an alarm if estimated retransmission count exceeds a limit. Confluo also allows the operator to issue diagnostic queries to the relevant end-hosts to determine priorities of involved flows, their retransmission counts, and periodicity of retransmissions during the relevant time-period to distinguish between the three scenarios based on observed values.

2.3 Confluo Design Overview

We now provide an overview of Confluo design (Figure 3), that comprises a central coordinator interface and an end-host module at each end-host in the network.

Coordinator Interface. Confluo's coordinator interface allows monitoring and diagnosing network-wide events by delegating monitoring responsibilities to Confluo's individual end-host modules, and by providing the diagnostic information from individual modules to the network operator. An operator submits *control programs* composed of Confluo API calls to the coordinator, which in turn contacts relevant end-host modules and coordinates the execution of Confluo API calls via RPC. The coordinator API also allows obtaining distributed atomic snapshots of telemetry data distributed across the end-hosts (§4).

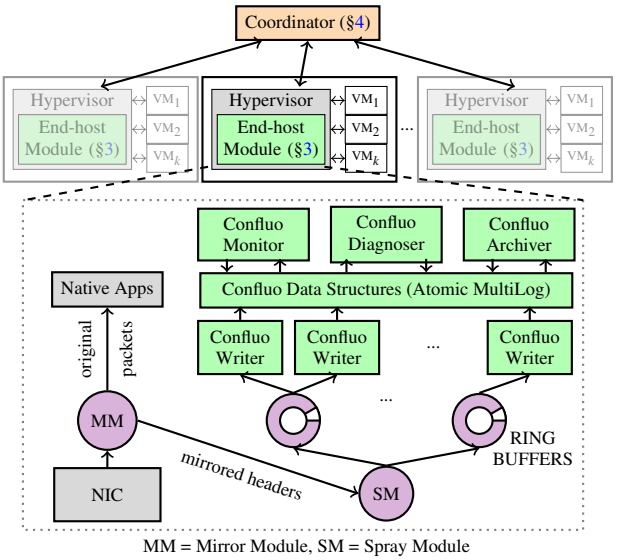


Figure 3: High-level Confluo Architecture (§2).

End-host Module. Confluo conducts bulk of monitoring and diagnosis operations at the end-hosts. Confluo captures and monitors packets in the hypervisor, where a software switch could deliver packets between NICs and VMs. A *mirroring module* mirrors packet headers to a *spray module*, that writes these headers to one of multiple ring buffers in a round-robin manner. Confluo currently uses DPDK [37] to bypass the kernel stack, and Open vSwitch [38] to implement the mirror and spray modules. This choice of implementation is merely to perform our prototype evaluation without the overheads of existing cloud frameworks (*e.g.*, KVM or Xen); our implementation on OVS trivially allows us to integrate Confluo with these frameworks.

Confluo's end-host module makes two important architectural choices. First, as outlined in §1, Confluo optimizes for highly-concurrent operations, potentially from multiple cores processing different packet streams, at the end-host. To that end, Confluo uses multiple ring buffers so that downstream modules can keep up with incoming headers. Multiple Confluo writers read headers from these ring buffers and write them to Confluo data structures. Achieving high throughput with multiple Confluo writers requires highly

concurrent write operations. This is where Confluo's new data structure — Atomic MultiLog — makes its key contribution. Recall from §1 that Atomic MultiLog exploits two unique properties of network logs — append-only workload and fixed field sizes for each header attribute — to minimize the overheads of traditional lock-free concurrency mechanisms while providing atomicity guarantees. We describe the design and implementation of Atomic MultiLogs in §3.

The second architectural decision is to separate threads that "read" from, and that "write" to Atomic MultiLog. Specifically, read threads in Confluo implement monitoring functionality (that requires evaluating potentially thousands of triggers on each header) and on-the-fly diagnosis functionality (that requires evaluating ad-hoc filters and aggregates using header logs and materialized views). The write threads, on the other hand, are the Confluo writers described above. This architectural decision is motivated by two observations. First, while separating read and write threads in general leads to more concurrency issues, Atomic MultiLog provides low-overhead mechanisms to achieve highly concurrent reads and writes. Second, separating read and write threads also require slightly higher CPU overhead (less than 4% in our evaluation even for a thousand triggers per packet); however, this is a good tradeoff to achieve on-the-fly diagnosis, since interleaving reads and writes within a single thread may lead to packet drops when complex ad-hoc filters need to be executed (§3).

Atomic MultiLogs guarantee that all read/write operations corresponding to an individual header become visible to the application atomically. However, due to a number of reasons (*e.g.*, different queue lengths on the NICs during packet capturing, random CPU scheduling delays, etc.), the ordering of packets visible at an Atomic MultiLog may not necessarily be the same as ordering of packets received at the NIC. One easy way to overcome this problem, that Confluo naturally supports, is to use ingress/egress NIC timestamps to order the updates in Atomic MultiLog to reflect the ordering of packets received at the NIC; almost all current generation 10Gbps and above NICs support ingress and egress packet timestamps at line rate. Without exploiting such timestamps or any additional information about packet arrival ordering at the NIC, unfortunately, this is an issue with any end-host based monitoring and diagnosis stack.

Distributed Diagnosis. Confluo supports low-overhead diagnosis of spurious network events even when diagnosing the event requires telemetry data distributed across multiple end-hosts [8–11]. Diagnosis using telemetry data distributed across multiple end-hosts leads to the classical consistency problems from distributed systems — unless all records (packets in our case) go through a central sequencer, it is impossible to achieve an absolutely perfect view of the system state. Confluo does not attempt to solve this classical problem, but rather shows that by exploiting the properties of telemetry data, it is possible to simplify the classical distributed atomic snapshot algorithm to a very low-overhead one (§4). This is indeed the strongest semantics possible without all packets going through a central sequencer.

3 Confluo Design

We now describe the design for Confluo end-host module (see Figure 3), that comprises of packet processing (mirror and spray) modules, multiple concurrent Confluo writers, the Atomic MultiLog, Confluo monitor, diagnoser and archival modules. We discussed the main design decisions made in the packet processing and writer modules in §2.3. We now focus on the Atomic MultiLog (§3.1, §3.2) and the remaining three modules (§3.3, §3.4).

3.1 Background

We briefly review two concepts from prior work that will be useful in succinctly describing the Atomic MultiLog.

Atomic Hardware Primitives. Most modern CPU architectures support a variety of atomic instructions. Confluo will use four such instructions: `AtomicLoad`, `AtomicStore`, `FetchAndAdd` and `CompareAndSwap`. All four instructions operate on 64 bit operands. The first two permit atomically reading from and writing to memory locations. `FetchAndAdd` atomically obtains the value at a memory location and increments it. Finally, `CompareAndSwap` atomically compares the value at a memory location to a given value, and only if they are equal, modifies the value at the memory location to a new specified value.

Concurrent Logs. There has been a lot of prior work on design of efficient, lock-free concurrent logs [39–42] that exploit the append-only nature in many applications to support high-throughput writes. Intuitively, each log maintains a "writeTail" that marks the end of the log. Every new append operation increments the writeTail by the number of bytes to be written, and then writes to the log. Using the above hardware primitives to atomically increment the writeTail, these log based data structure support extremely high write rates.

It is easy to show that by additionally maintaining a "readTail" that marks the end of completed append operations (and thus, always lags behind the writeTail) and by carefully updating the readTail, it is possible to guarantee *atomicity* for concurrent reads and writes on a single log (see [24] for a formal proof). Using atomic hardware primitives to update both readTail and writeTail, it is possible to achieve high throughput for concurrent reads *and* writes for such logs.

3.2 Atomic MultiLog

An Atomic MultiLog uses a collection of concurrent lock-free logs to store packet header data, packet attribute indexes, aggregates and filters defined in §2.2 (see Figure 4). As outlined earlier, Atomic MultiLog exploit two unique properties of network logs to facilitate this:

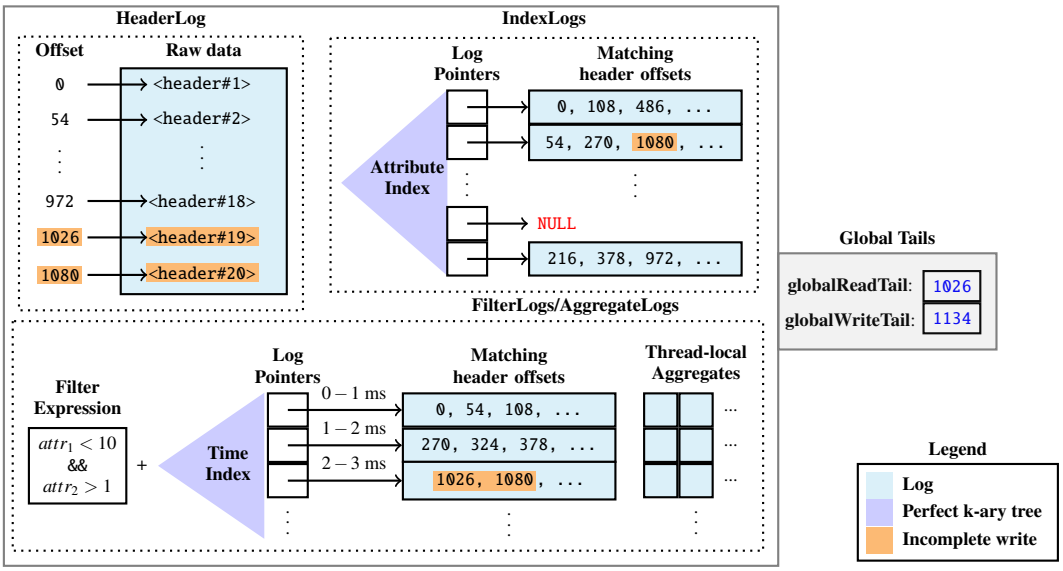


Figure 4: The Atomic MultiLog uses a collection of concurrent lock-free logs to store packet headers, indexes, aggregates and filters (as defined in §2.2) and efficiently updates these data structures as a single atomic operation as new packet headers arrive. See §3.2 for details.

- **Property 1:** Packet headers, once processed by the stack, are not updated and only aggregated over long time scales.
- **Property 2:** Each packet header attribute has a fixed size (number of bits used to represent the attribute)

HeaderLog. This concurrent append-only log stores the raw data for all captured packet headers in Confluo. Each packet header in the HeaderLog has an offset, which is used as a unique reference to the packet across all data structures within the Atomic MultiLog. We will discuss in §3.2.1 how this simplifies guaranteeing atomicity for operations that span multiple data structures within the Atomic MultiLog.

IndexLog. An Atomic MultiLog stores an IndexLog for each indexed packet attribute (*e.g.*, `srcIP`, `dstPort`), that maps each unique attribute value (*e.g.*, `srcIP=10.0.0.1` or `dstPort=80`) to corresponding packet headers in HeaderLog. IndexLogs efficiently support concurrent, lock-free insertions and lookups using two main ideas.

Protocol-defined fixed attribute widths in packet headers allow IndexLogs to use a perfect k-ary tree [43] (referred to as an attribute index in Figure 4) for high-throughput insertions upon new data arrival. Specifically, an n-bit attribute is indexed using a k-ary tree with a depth of $\lceil \frac{n}{log_2 k} \rceil$ nodes, where each node indexes $log_2 k$ bits of the attribute. For instance, Figure 5 shows an example of a 2^{16}-ary tree for IP addresses, where the root node has 2^{16} child pointers corresponding to all possible values of the 16-bit IP prefix, and each of its children have 2^{16} pointers for the 16-bit IP suffix.

The use of a perfect k-ary tree greatly simplifies the write path. All child pointers in a k-ary tree node initially point to `NULL`. When a new packet attribute value (*e.g.*, `srcIP=10.0.0.1`) is indexed, all unallocated nodes along

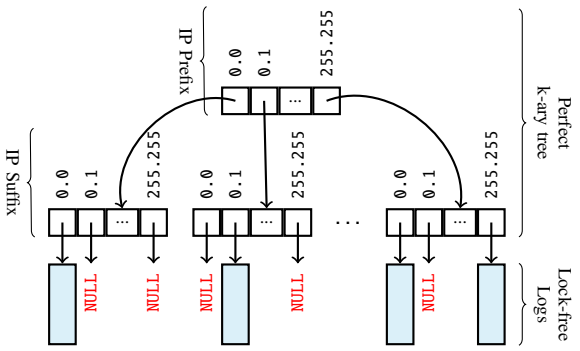


Figure 5: 2^{16}-ary IndexLog for 32-bit IP address. Each node in the tree (depth=2) has k=2^{16} children and indexes 16 bits (2 bytes) of the IP address.

the path corresponding to the attribute value are allocated. This is where an IndexLog uses the second idea — since the workload is append-only, HeaderLog offsets for attribute value to packet header mapping are also append-only; thus, traditional lock-free concurrent logs can be used to store this mapping at the leaves of the k-ary tree.

Conflicts among concurrent attribute index nodes and log allocations are resolved using the `CompareAndSwap` instruction, thus alleviating the need for locks. Subsequent packet headers with the same attribute value are indexed by traversing the tree to the relevant leaf, and appending the headers's offset to the log. To evaluate range queries on the index, Confluo identifies the sub-tree corresponding to the attribute range (*e.g.*, `10.0.0.0/24`); the final result is then the union of header offsets across logs in the sub-tree leaves.

FilterLog. A FilterLog is simply a filter expression (*e.g.*, `srcIP==10.0.0.1 && dstPort==80`), and a time-indexed

collection of logs that store references to headers that match the expression (bucketed along user-specified time intervals). The logs corresponding to different time-intervals are indexed using a perfect k-ary tree, similar to IndexLogs.

AggregateLog. Similar to FilterLogs, an AggregateLog employs a perfect k-ary tree to index aggregates (*e.g.*, `SUM(pktSize)`) that match a filter expression across user-specified time buckets. However, atomic updates on aggregate values is slightly more challenging — it requires reading the most recent version, modifying it, and writing it back. Maintaining a single concurrent log for aggregates requires handling complex race conditions to guarantee atomicity.

Confluo instead maintains a collection of *thread-local* logs, with each writer thread executing read-modify-write operations on its own aggregate log. The latest version of an aggregate is obtained by combining the most recent thread-local aggregate values from individual logs. We note that the use of thread-local logs restricts aggregation to associative, commutative operations, that are sufficient to implement network monitoring and diagnosis functionalities.

3.2.1 Atomic Operations on Collection of Logs

End-to-end Atomic MultiLog operations may require updating multiple logs across HeaderLog, IndexLogs and FilterLogs. Even if individual logs support atomic operations, end-to-end Atomic MultiLog operations are not guaranteed to be atomic by default. Fortunately, it is possible to extend the readTail/writeTail mechanism for concurrent logs to guarantee atomicity for Atomic MultiLog operations; however, this requires resolving two challenges.

First, in order to guarantee total order for Atomic MultiLog operations, its component logs must agree on an ordering scheme. Confluo uses HeaderLog as single source of ground truth, and designates its readTail and writeTail as globalReadTail and globalWriteTail for the Atomic MultiLog. Before packet headers are written to different ring buffers, Confluo first atomically increments globalWriteTail by the size of the packet header using `FetchAndAdd`. This atomic instruction resolves potential write-write conflicts, since it assigns a unique HeaderLog offset to each header. When Confluo writers read headers from different ring buffers, they update all relevant logs in Atomic MultiLog, and finally update the globalReadTail to make the data available to subsequent queries.

The globalReadTail imposes a *total order* among Atomic MultiLog write operations based on HeaderLog offsets: Confluo only permits a write operation to update the globalReadTail after all write operations writing at smaller HeaderLog offsets have updated the globalReadTail, via repeated `CompareAndSwap` attempts. This ensures that there are no "holes" in the HeaderLog, and allows Confluo to ensure atomicity for queries via a simple globalReadTail check. In particular, queries first atomically obtain globalReadTail value using `AtomicLoad`, and only access headers and their references (across IndexLogs, FilterLogs and AggregateLogs) if the header lies within the globalReadTail in HeaderLog. Note that since queries do not modify globalReadTail, they cannot conflict with other queries or write operations.

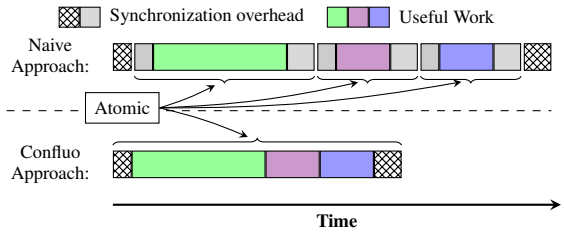


Figure 6: Confluo relaxes atomicity guarantees of individual logs, guaranteeing atomicity only for end-to-end Atomic MultiLog operations. Different colors correspond to operations on different logs.

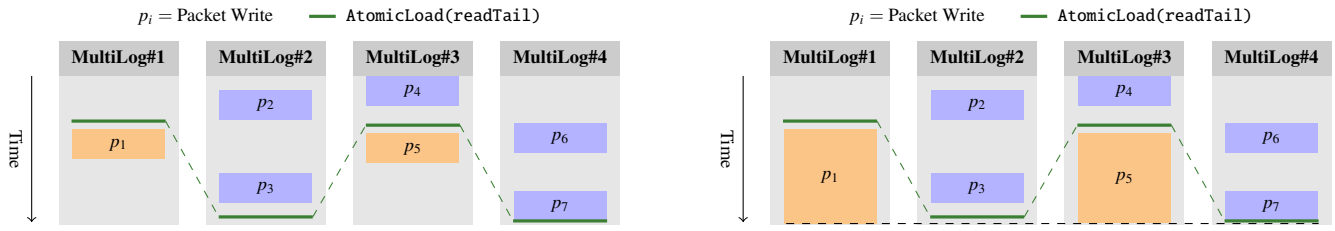
The second challenge lies in preserving atomicity for operations on Confluo aggregates, since they are not associated with any single packet header that lies within or outside the globalReadTail. To this end, aggregate values in AggregateLogs are versioned with the HeaderLog offset of the write operation that updates it. To get the final aggregate value, Confluo obtains the aggregate with the largest version smaller than the current globalReadTail for each of the thread-local aggregates. Since each Confluo writer thread modifies its own local aggregate, and queries on aggregates only access versions smaller than the globalReadTail, operations on pre-defined aggregates are rendered atomic.

While the operations above enable end-to-end atomicity for Atomic MultiLog operations, we note that readTail updates for each individual log in the Atomic MultiLog may add up to a non-trivial amount of overhead (Figure 6). Confluo alleviates this overhead by observing that in any Atomic MultiLog operation, the globalReadTail is only updated after each of the individual log readTails are updated. Therefore, any query that passes the globalReadTail check trivially passes the individual readTail checks, obviating the need for maintaining individual readTails. Removing individual log readTails relaxes unnecessary ordering guarantees for them, while enforcing it only for end-to-end operations. This significantly reduces contention among concurrent operations.

3.3 Monitor & Diagnoser Modules

We now describe Confluo monitor and diagnoser modules.

Monitor Module. This module is responsible for online evaluation of Confluo triggers via a dedicated monitor thread. Confluo triggers operate on pre-defined aggregates (§2.2) in the Atomic MultiLog. Since the aggregates are updated for every packet, trigger evaluation itself involves little work. The monitor thread wakes up at periodic intervals, and first obtains relevant aggregates for intervals since the trigger was last evaluated, performing coarse aggregations over multiple stored aggregates over sliding windows. It then checks if the trigger predicate (*e.g.*, `SUM(pktSize)>1GB`) is satisfied, and if so, generates an alert.



(a) Naive approach may lead to inconsistent snapshots

(b) Atomic snapshots with delayed packet writes

Figure 7: Simply obtaining (global) readTails for a collection Atomic MultiLogs can yield inconsistent snapshots, as shown in (a), where `AtomicLoad` on readTails at different Atomic MultiLogs are skewed in time, and packets p_1, p_5 appear to be written after p_3, p_7 (inconsistent). (b) We can render the same snapshot consistent by delaying completion of p_1, p_5 until *after* `AtomicLoad` on on Atomic MultiLog #4.

Diagnoser Module. Confluo's diagnoser module serves ad-hoc queries on packet headers captured by the Atomic MultiLog. Recall from Table 1 that Confluo allows a diagnostic query to provide a filter expression `fExpression` as well as a time range. If there already exists a filter `fExpression`, query execution is fairly straightforward — since FilterLogs are time-indexed (Figure 4), Confluo simply looks up the FilterLog(s) to extract packet header offsets corresponding to the specified time interval, drops the offsets that are greater than the globalReadTail value, and returns packet headers corresponding to the remaining offsets. Confluo allows nested queries; Confluo can apply additional filters on these packet headers or obtain attribute aggregates for them.

If a filter for `fExpression` specified in the query does not already exist, Confluo first performs IndexLog lookups for individual packet attributes in the filter expression (§3.2), and then combines their results based on the boolean operators in the expression (Table 2). This can be an expensive operation; to that end, Confluo uses several optimizations. For instance, Confluo first converts the filter expression to its canonical disjunctive normal form (DNF) [44], where the resulting filter expression is a disjunction (`OR`) of conjunction (`AND`) clauses. The DNF form yields the most selective filter sub-expressions in its conjunction clauses. In order to minimize the number of packet references scanned for a specific conjunction clause, Confluo uses the tail value for individual attributes IndexLog as an estimate of their selectivity; Confluo then evaluates the conjunction clause by scanning through IndexLog entries for the most selective attribute, dropping all packet headers that occur after the globalReadTail, or do not satisfy the remaining predicates in the clause. The results for individual conjunction clauses are combined using a simple set union for the disjunction operator.

3.4 Archival Module

Confluo stores network logs with rich telemetry data, along with materialized views, pre-defined filters and aggregates to support low-overhead monitoring and diagnostic queries. Storing these logs and materialized views in their raw form over long time periods would lead to tremendous storage requirements. Confluo overcomes this via periodic archival of Atomic MultiLog data. Our current implementation employs a basic approach — an archival thread periodically flushes packet header entries up to a certain offset in the HeaderLog to secondary storage, along with associated IndexLog, FilterLog and AggregateLog entries, and ensures that the in-memory footprint does not exceed a user-configured threshold. While Confluo data structures are amenable to several approaches that exist for log archival (*e.g.*, periodically summarizing older data with aggregated statistics, log compression [45–47], compaction [48–50], etc.), a detailed treatment of the archival process is an interesting future work.

4 Distributed Diagnosis

Confluo Coordinator interface (Figure 3) facilitates monitoring and diagnosis of network-wide events. Recall from §2.3 that operators express monitoring and diagnosis tasks via *control programs* composed of Confluo API calls (Table 1). Based on the control program, the coordinator interface delegates tasks to individual end-host modules and collects diagnostic information from them. The coordinator interface facilitates consistent distributed analysis for high-speed networks via a *distributed atomic snapshot* algorithm.

Existing approaches for distributed snapshots either use a centralized sequencer to order all writes to the system (*e.g.*, transaction managers [51–53], log sequencers [54–56]) simplifying global snapshots, or employ algorithms with weak consistency guarantees (*e.g.*, causal consistency [57]). However, neither is acceptable for Confluo; the former is infeasible for high speed networks, while the latter provides weaker consistency semantics than Confluo end-host stack.

Confluo does not attempt to resolve complex distributed consistency issues, but instead strives for an efficient distributed *atomic* snapshot algorithm. We note that append-only semantics in Confluo greatly simplify snapshot for individual Atomic MultiLogs[3]. While naively reading readTails at individual Atomic MultiLogs across multiple end-hosts

[3] Atomic snapshot of any Atomic MultiLog is trivially obtained by reading its globalReadTail.

Algorithm 1 Distributed Atomic Snapshot

Obtains the snapshot vector (Atomic MultiLog readTails).

At Coordinator:

1: `snapshotVector` $\leftarrow \emptyset$
2: Broadcast `FreezeReadTail` requests to all Atomic MultiLogs
3: **for each** `mLog` in `multiLogSet` **do**
4: Receive `readTail` from `mLog` & add to `snapshotVector`
5: Broadcast `UnfreezeReadTail` requests to all Atomic MultiLogs
6: **for each** Atomic MultiLogmLog **do**
7: Wait for `ACK` from `mLog`
8: **return** `snapshotVector`

At Each Atomic MultiLog:

On receiving `FreezeReadTail` **request**

1: Atomically read and freeze `readTail` using `CompareAndSwap`
2: Send `readTail` value to Coordinator

On receiving `UnfreezeReadTail` **request**

1: Atomically unfreeze `readTail` using `CompareAndSwap`
2: Send `ACK` to Coordinator

may not produce an atomic snapshot (Figure 7(a)), it does hint towards a possible solution.

In particular, atomic distributed snapshot in Confluo reduces to the widely studied problem of obtaining a snapshot of n atomic registers in shared memory architectures [58–60]. These approaches, however, rely on multiple iterations of register reads with large theoretical bounds on iteration counts. While feasible in shared memory architectures where reads are cheap, they are impractical for distributed settings since reads over the network are expensive.

Confluo's atomic distributed snapshot algorithm exploits the observation that any snapshot can be rendered atomic by *delaying* completion of certain writes that would otherwise break atomicity for the snapshot. For instance, in Figure 7(a), if we ensure that packet writes p_1 and p_5 do not complete until after the globalReadTail read on Atomic MultiLog #4 (dashed line in Figure 7(b)), the original snapshot becomes atomic since p_1 and p_5 now appear to be written after p_3 and p_7, in line with the actual order of events.

Algorithm 1 outlines the steps involved in obtaining an atomic snapshot. The coordinator interface first sends out `FreezeReadTail` requests to all Atomic MultiLogs in parallel. The Atomic MultiLogs then freeze and return the value of their readTail atomically via `CompareAndSwap`. This temporarily prevents packet writes across the Atomic MultiLogs from completing since they are unable to update the corresponding readTails, but does not affect Confluo queries. Once the coordinator receives all the readTails, it issues `UnfreezeReadTail` requests to all the Atomic MultiLogs, causing them to unfreeze their readTail via `CompareAndSwap`. They then send an acknowledgement to the coordinator interface, allowing pending writes to complete at once. Since the first `UnfreezeReadTail` message is sent out only after the last Atomic MultiLog readTail has been read, all writes that would conflict with the snapshot are delayed until after the snapshot has been obtained.

The coordinator interface executes the snapshot algorithm across arbitrary collections of end-hosts based on the provided control program, and generates a snapshot vector. Note that while the readTails remain frozen, write operations can still update HeaderLog, IndexLogs, FilterLogs and AggregateLogs, but wait for the readTail to unfreeze (up to one network round-trip time) in order to make their effects visible. As such, write throughput in Confluo is minimally impacted, but write latencies can increase for short durations. Moreover, since Confluo supports annotating packets with NIC timestamps to determine ordering (§2.3) before potentially delaying packet writes, Confluo's atomic snapshot algorithm does not affect the accuracy of diagnostic queries.

5 Evaluation

Confluo prototype is implemented in $\sim$ 20K lines of C++. In this section, we evaluate Confluo to demonstrate:

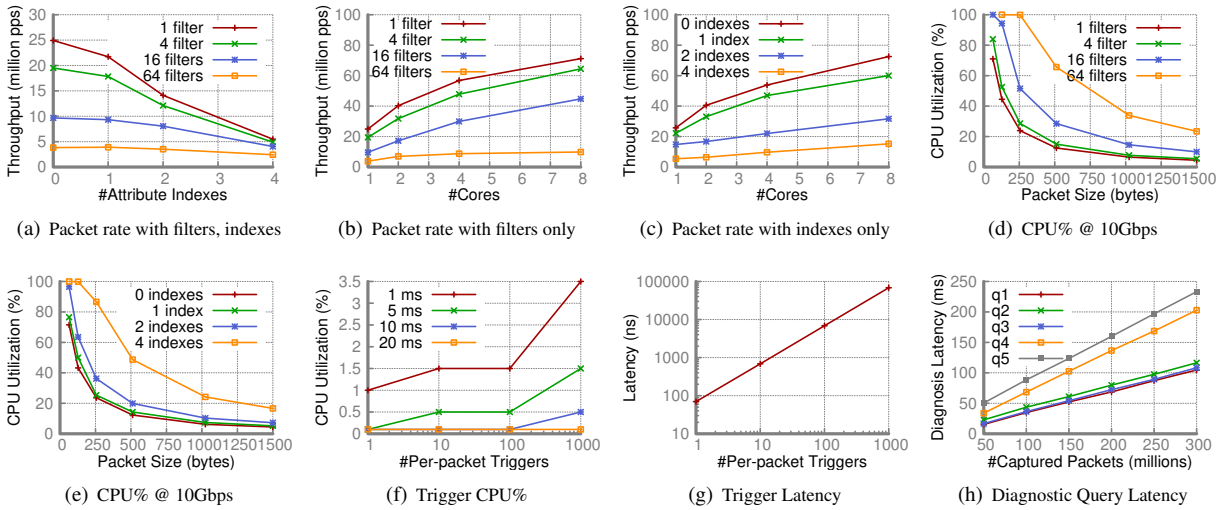
- Confluo can capture packet headers at line rate (even for 10Gbps and higher bandwidth links) while evaluating thousands of triggers and tens of filters with minimal CPU utilization (§5.1);
- Confluo can exploit rich telemetry data embedded in packet headers to enable a large class of network monitoring and diagnosis applications (§5.2).

5.1 Confluo Performance

We now evaluate Confluo performance on servers with 2 $\times$ 12-core 2.30GHz Xeon CPUs and 252GB RAM, connected via 10Gbps links. We used DPDK's `pktgen` tool [28] to generate network traffic composed of TCP packets with 54 byte headers, IPs drawn from a /24 prefix and ports drawn from 10 common application port values. Our experiments used up to 5 attribute indexes, corresponding to the connection 4-tuple (source/destination IPs and ports) and the packet timestamp. We perform all our evaluations with Confluo running in the user space to avoid the performance bottlenecks out of Confluo implementation (*e.g.*, hypervisor overheads).

Packet Capture. Figure 8(a) shows Confluo peak packet capture rate as the number of attribute indexes and predefined filters are increased on a single core. Without any filters or indexes, the Atomic MultiLog is able to sustain $\sim$ 25 million packets/s per core, with throughput degrading gracefully as more filters or indexes are added. The degradation is close to linear with the number of indexes, since each additional index incurs fixed indexing overhead for every packet. The degradation is sub-linear for filters, since additional filters incur negligible overheads for packets that do not match them. Interestingly, as we show in [24], monitoring and diagnosing even complex network issues only requires a few filters (often bounded by the number of active flows on a server) and 1-2 indexes in Confluo.

The packet capture performance indicates that, even when average packet size is 128B or larger, Confluo can sustain



(a) Packet rate with filters, indexes (b) Packet rate with filters only (c) Packet rate with indexes only (d) CPU% @ 10Gbps

(e) CPU% @ 10Gbps (f) Trigger CPU% (g) Trigger Latency (h) Diagnostic Query Latency

Figure 8: (a) Confluo's peak packet capture throughput (measured in packets per second or pps) for 64B packets degrades gracefully on increasing the number of attribute indexes and the number of pre-defined filters; (b, c) the peak throughput scales well with the number of cores, even as the number of pre-defined filters and indexes are increased. (d, e) At line rate of 10Gbps, Confluo can handle average packet size as small as 128B with 16 filters and 2 indexes on a single core. (f, g) Confluo can evaluate 1000s of trigger queries with less than 4% CPU utilization at 1ms intervals, and with latency less than 70μs. (h) Diagnostic query latency in Confluo increases linearly with number of captured packets in Confluo, and varies across different queries due to differing intermediate result cardinalities and complexity for combining them. The filters in the figures use the following templates (varying value of A, B, IP, and port for various filters): `(q1)` packets from VM `A` to VM `B`; `(q2)` packets to VM `A`; `(q3)` packets from VM `A` on destination port `P`; `(q4)` packets between (IP_1, P_1) and (IP_2, P_2); and `(q5)` packets to or from VM `A`.

line rate for 10Gbps link using a single core! Real-world workloads [61] show that average packet size in datacenter networks is much larger. Confluo is able to ingest such workloads on a single core with each of 64 filters, 1000 triggers, and 5 indexes, updated for each packet. Figure 8(b) and 8(c) show packet capture scaling with number of cores. We note that, while packet capture scales well, it is not perfectly linear; this is due to stalling of globalReadTail updates for Confluo writers that attempt to update the Atomic MultiLog out-of-order (§3.2). However, the impact of stalling is mitigated to a great extent due to the use of lock-free primitives, and the use of a globalReadTail instead of separate readTails for each log in Atomic MultiLog.

CPU Utilization at 10Gbps. Figure 8(d) and 8(e) show CPU utilization for Confluo updating data structures, varying with the packet size for different number of filters and indexes. Observe that CPU utilization is higher for smaller packet sizes, since smaller packet sizes at line rate correspond to higher packet rates. For smaller packet sizes along with 4 indexes and 64 filters, CPU becomes a bottleneck; however, CPU utilization drops dramatically with fewer filters or indexes. Confluo can scale up its packet capture rate with more CPU cores, as discussed before.

Evaluating Triggers. Recall from §3.2 that Confluo evaluates triggers over pre-defined aggregates, making trigger evaluation extremely cheap. Figure 8(f) shows that even when Confluo evaluates 1000 triggers at 1ms time intervals, the CPU utilization remains $< 4\%$ of a single core. This is because a single trigger evaluation incurs roughly 100ns latency, with latency increasing to 70μs for 1000 triggers[4].

Diagnosis Latency. We evaluate Confluo's diagnostic query performance using five queries (`q1` to `q5` outlined in Figure 8). Since these queries combine results from different Confluo IndexLogs, query latency depends on intermediate result cardinalities. Consequently, the query latency increases linearly with the number of captured packets, since cardinalities of intermediate results also grow linearly with the latter. As such, Confluo is able to perform complex diagnostic queries on-the-fly with sub-second latencies on 100s of millions of packets (Figure 8(h)).

Atomic Snapshots. To evaluate the overhead of atomic snapshots in Confluo, we measure percentage decrease in packet capture rate while periodically performing snapshots across $1-8$ end-hosts (to emulate diagnostic queries). We found the impact of atomic snapshots on write rate to be insignificant — while performing snapshots every 1ms, packet rate at each end-host drops by $< 2\%$, even as number of end-hosts in the snapshot is increased from 1 to 8. This result might be non-intuitive; the reason is that Confluo only

[4] A 70μs latency over 1ms period may result in as high as 7% CPU utilization; we believe the discrepancy is because of the reporting frequency for CPU utilization metrics from the OS.

blocks updates to the globalReadTail during the snapshot operation — bulk of the writes including those to HeaderLog, IndexLogs and FilterLogs can still proceed, with entire set of pending globalReadTail updates going through at once when the snapshot operation completes.

We note that a diagnostic query that spans multiple servers would incur the end-host query execution latency shown in Figure 8(h), as well as the atomic snapshot latency. Since the snapshot algorithm queries different Atomic MultiLogs across different end-hosts in parallel, the snapshot is obtained in roughly 1 network round-trip (about $\sim 180\mu$s in our setup), with slightly higher latencies across larger number of end-hosts due to skew in queuing and scheduling delays (about 1.2ms for 128 end-hosts). Since network-wide diagnosis tasks often only involve a very small fraction of a data center's end-hosts, Confluo can employ switch metadata to isolate the end-hosts it needs to query, similar to [9].

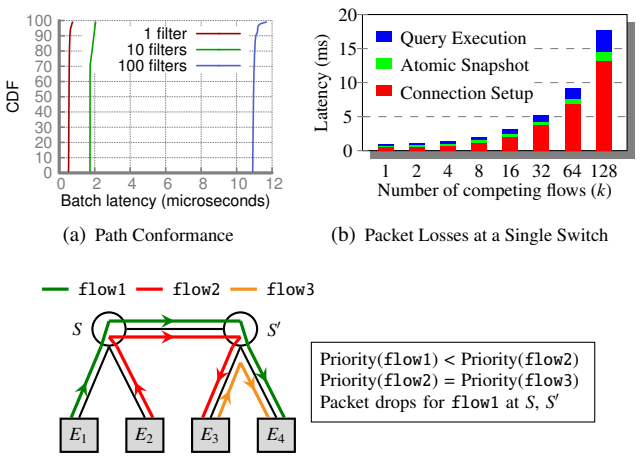
5.2 Confluo Applications

We now use Confluo to detect and debug a variety of network issues in modern data center networks. Our setup (comprising 96 virtual machines and Pica8 P-3297 switches), deployment and workloads are exactly the same as those in [8, 9], but with the end-host stack replaced with Confluo. Consequently, our setup inherits (1) in-network mechanisms that embed switch ID and timestamp at each switch traversed by a packet in its header, and (2) switch pointers to end hosts where the telemetry data for packets processed by the switch are stored. While we present only a subset of Confluo applications here for brevity, we discuss more applications in [24].

Path Conformance. We demonstrate Confluo's ability to quickly monitor and debug path conformance violations by randomly routing a subset of the packets *within a flow* via a particular switch S. Each end-host is configured with a single filter that matches packets that passes through switch S. A companion trigger to the filter raises an alert if the count of packets satisfying the filter is non-zero. Confluo monitor evaluates the trigger at 1 ms intervals, and alerts the presence of path non-conformant packets within milliseconds of its incidence at the end-host.

Figure 9(a) shows the latency in Confluo with varying number of path conformance checks (filters). We note that while a single conformance check incurs average batch latency of 1μs, 100 checks incur 11μs latency; this indicates sub-linear increase in latency with the number of checks. As such, Confluo is able to perform *per-packet* path conformance checks with minimal overheads.

Packet Losses at a Single Switch. In this application, we consider monitoring and diagnosis for generalized versions of the scenarios from Figure 2 (left), where k flows compete at a common output port at switch S and one or more of these flows experience packet losses. Confluo's approach is outlined in Figure 2 (right). Confluo exploits network telemetry



(a) Path Conformance (b) Packet Losses at a Single Switch

(c) Packet Losses at Multiple Switches

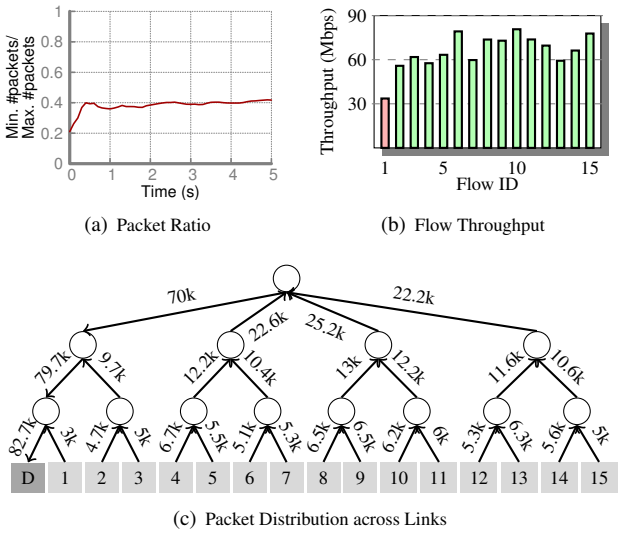
Figure 9: (a) Confluo can perform 100 path conformance checks and ingest packet headers in batches of size 32 in about 11μs per-batch ($\sim$ 350ns per-packet); (b) Diagnosis latency for packet losses due to traffic congestion; most of the time is spent in connection setup. Confluo takes $<$ 18ms for querying 128 hosts. (c) Setup used for monitoring and diagnosing packet losses at multiple switches.

data in packet headers (switch IDs and timestamps) to identify contending TCP flows and their destination end-points.

Confluo is able to detect the presence of packet loss due to TCP retransmissions in under 1ms (trigger periodicity), and the coordinator interface receives the alert within $\sim 250\mu$s. Figure 9(b) shows the diagnosis latency at the coordinator as the number of competing flows (k) at switch S increases. With more flows, Confluo has to contact more end-hosts to collect diagnostic information. Even while collecting diagnostic information across 128 end-hosts, the time taken to obtain the atomic snapshot and performing the diagnostic query at the coordinator are only 1.2 ms and 3 ms respectively. Most of the diagnosis time is spent in establishing connections to the relevant end-hosts, although this can be mitigated via connection pooling. Even so, Confluo is able to diagnose the issue across 128 hosts in under 18 ms.

Packet Losses at Multiple Switches. We now consider a scenario where a flow experiences packet losses at multiple switches, as outlined Figure 9(c). Again, we detect packet losses using TCP retransmissions, and employ telemetry data embedded in packet headers (switch IDs) to aid diagnosis. Using ideas discussed in [8, 9], we issue diagnostic queries to determine the flow information (IDs, traffic volume and priorities) that contended with `flow1` at switches S and S'. By comparing the traffic volume and priorities of contending flows, Confluo concludes that the losses for `flow1` are due to contention with higher priority `flow2` and `flow3` at switches S and S'. Confluo takes roughly 1.8ms for the end-to-end diagnosis: 1.15ms for connection setup, 180μs for the snapshot algorithm and 350μs for performing the actual query.

TCP Outcast. In TCP outcast problem [62], two sets of



(a) Packet Ratio (b) Flow Throughput

(c) Packet Distribution across Links

Figure 10: Diagnosing TCP Outcast. (a) Confluo measures the cumulative ratio of smallest and largest packet counts across all flows at 10ms intervals to diagnose outcast; smallest and largest packet counts correspond to flows with smallest and largest hop-counts respectively, with their ratio stabilizing to 0.4 in 1s after measurement starts. (b) Flow throughputs at $t = 1$s. (c) Using [7–10], Confluo can obtain packet distribution across links (numbers along links) in a 1s window during outcast. Circles represent switches, 1-15 represent flowIDs, and D represents destination end-host.

flows (one with small number of flows, and one with large number of flows) from two different input ports of a switch compete for the same output port; it has been shown [62] that in such a scenario, TCP can result in severe throughput degradation for the small set of flows. This occurs due to *port blackout* in switches that employ tail-drop queuing, wherein a batch of consecutive packets are dropped from an input port. In TCP Outcast, this disproportionately affects the small set of flows, leading to TCP timeouts.

In our experiment, we recreate a setup similar to [62], where 15 TCP flows with different sources and the same destination (shown as D in the figure) compete for a single output port at the final-hop switch. One flow traverses a 1-hop path, two of them traverse a 3-hop path, and the remaining 12 traverse a 5-hop path. All links in the setup have 1Gbps bandwidth. To monitor the TCP outcast problem, Confluo first adds triggers to detect packet losses (Table 2(b)). Once the trigger raises an alarm, the coordinator interface issues diagnostic queries at 10ms intervals to obtain packet count for each flow in that window, and compute cumulatively (1) ratio of smallest to largest packet counts across all flows, and (2) individual flow throughputs (Figure 10). Each diagnostic query incurs an average latency of 250μs.

Owing to port blackout, the flow with smallest hop-count observes the lowest throughput, while flows with larger hop-counts observe higher throughput (Figure 10(b)). By exploiting telemetry data embedded in packet headers, Confluo can also obtain the number of packets transmitted through each link in the network over a 1s window (Figure 10(c)).

6 Related Work

We already discussed related work in network monitoring and diagnosis in §2.1. In this section, we focus on related work in the context of Atomic MultiLog.

There has been a lot of work on the design of efficient, concurrent logs [39–42, 54–56, 63–65]. Since log-based systems have been around for several decades, it would be impractical to attempt an exhaustive comparison. However, at a high-level, we note that traditional log-based systems focus on simple atomic operations on a *single* log; in contrast, Confluo combines a *collection of logs* in the Atomic MultiLog to support atomic filters, aggregates and triggers over packet headers. By relaxing the atomicity guarantees for its individual logs and guaranteeing atomicity only for end-to-end MultiLog operations, Confluo achieves high concurrency for these collection of logs. Figure 1 compares the performance of Confluo against the state-of-the-art log-based system [54].

Database Management Systems (DBMS) [66–68] use secondary indexes to support filters and aggregates on records. Unfortunately, atomically updating tree-based index structures such as B-Trees [69, 70] and Tries [71–74] incur high write overheads due to complex tree traversals and locking overheads, resulting in low write throughput. On the other hand, hash-based indexes [75–77] sustain high throughput, but do not support ordered access to data items. Confluo borrows heavily from these approaches, but makes design tradeoffs to meet the high throughput and rich functionality requirements of network monitoring and diagnosis (§3.2).

7 Conclusion

Confluo is an end-host stack that can be integrated with existing network management tools to enable monitoring and diagnosis of network events. Confluo achieves this using Atomic MultiLog, a new data structure that exploits structure in network traffic to support highly concurrent read-write operations. Confluo executes 1000s of triggers and 10s of filters at line rate (for 10Gbps links) on a single core.

Acknowledgments

We would like to thank our shepherd, Cole Schlesinger, and anonymous NSDI reviewers for their insightful feedback. We are also grateful to Praveen Tammana for helping us in setting up experimental testbed, and for sharing packet traces from PathDump and SwitchPointer experiments. This research is supported in part by NSF CISE Expeditions Award CCF-1730628, NSF DGE-1106400, NSF CNS-1704742, and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] 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 *ACM SIGCOMM*, 2017.

[2] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "SketchVisor: Robust Network Measurement for Software Packet Processing," in *ACM SIGCOMM*, 2017.

[3] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *USENIX NSDI*, 2016.

[4] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *ACM SIGCOMM*, 2016.

[5] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX NSDI*, 2013.

[6] 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 *ACM SIGCOMM*, 2014.

[7] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks," in *USENIX SOSR*, 2015.

[8] P. Tammana, R. Agarwal, and M. Lee, "Simplifying Datacenter Network Debugging with PathDump," in *USENIX OSDI*, 2016.

[9] P. Tammana, R. Agarwal, and M. Lee, "Distributed Network Monitoring and Debugging with SwitchPointer," in *USENIX NSDI*, 2018.

[10] "In-band Network Telemetry (INT)." https://p4.org/assets/INT-current-spec.pdf.

[11] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *USENIX NSDI*, 2017.

[12] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang, "Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis," in *USENIX SOSR*, 2016.

[13] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling Path Queries," in *USENIX NSDI*, 2016.

[14] A. Gupta, R. Harrison, A. Pawar, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Streaming Network Telemetry," in *ACM SIGCOMM*, 2018.

[15] 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 *USENIX NSDI*, 2014.

[16] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in *ACM SIGCOMM*, 2014.

[17] 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 *ACM SIGCOMM*, 2015.

[18] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically Finding the Cause of Packet Drops," in *USENIX NSDI*, 2018.

[19] "OpenSOC." http://opensoc.github.io/.

[20] "Tigon." http://tigon.io.

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

[22] M. Sullivan, "Tribeca: A Stream Database Manager for Network Traffic Analysis," in *VLDB*, 1996.

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

[24] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed Monitoring and Diagnosis Stack for High Speed Networks." Technical Report, 2018.

[25] "Confluo GitHub Repository." https://github.com/ucbrise/confluo.

[26] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *USENIX OSDI*, 2016.

[27] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[28] "The Pktgen Application." https://pktgen.readthedocs.io/en/latest/.

[29] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching Network Security Analysis with Time Travel," in *ACM SIGCOMM*, 2008.

[30] "Deepfield Defender." http://deepfield.com/products/deepfield-defender/.

[31] "Kentik Detect." https://www.kentik.com.

[32] F. Fusco, M. P. Stoecklin, and M. Vlachos, "NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic," *VLDB*, 2010.

[33] P. Giura and N. Memon, "NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring," in *Springer-Verlag RAID*, 2010.

[34] J. Lee, S. Lee, J. Lee, Y. Yi, and K. Park, "FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency," in *USENIX ATC*, 2015.

[35] "The health insurance portability and accountability act." http://www.hhs.gov/ocr/privacy/.

[36] "Cisco Compliance Solution for HIPAA Security Rule Design and Implementation Guide." https://tinyurl.com/y94u8sqq.

[37] "Intel Data Plane Development Kit (DPDK)." http://dpdk.org.

[38] "Open vSwitch (OVS)." http://openvswitch.org.

[39] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling Concurrent Log-structured Data Stores," in *ACM EuroSys*, 2015.

[40] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM TOPLAS*, 1990.

[41] "A Fast Lock-Free Queue for C++." http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++.

[42] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *ACM SPAA*, 2001.

[43] P. E. Black, "perfect k-ary tree." https://www.nist.gov/dads/HTML/perfectKaryTree.html.

[44] "Disjunctive normal form." https://en.wikipedia.org/wiki/Disjunctive_normal_form.

[45] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling Queries on Compressed Data," in *USENIX NSDI*, 2015.

[46] "Configuring compression in Cassandra." https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_config_compress_t.html.

[47] "RocksDB Tuning Guide." https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[48] "Memtables in Cassandra." https://wiki.apache.org/cassandra/MemtableSSTable.

[49] "Configuring compaction in Cassandra." https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_configure_compaction_t.html.

[50] "SSTable and Log Structured Storage: LevelDB." https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb.

[51] "SQLServer: Distributed Transactions (Database Engine)." https://technet.microsoft.com/en-us/library/ms191440(v=sql.105).aspx.

[52] "Oracle: Distributed Transactions Concepts." https://docs.oracle.com/cd/B10501_01/server.920/a96521/ds_txns.htm.

[53] "Postgres: eXtensible Transaction Manager." https://wiki.postgresql.org/wiki/DTM.

[54] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "CORFU: A Shared Log Design for Flash Clusters," in *USENIX NSDI*, 2012.

[55] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures over a Shared Log," in *ACM SOSP*, 2013.

[56] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vCorfu: A Cloud-Scale Object Store on a Shared Log," in *USENIX NSDI*, 2017.

[57] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM TOCS*, 1985.

[58] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory," *JACM*, 1993.

[59] H. Attiya and O. Rachman, "Atomic Snapshots in O (N Log N) Operations," *SIAM Journal on Computing*, 1998.

[60] H. Attiya, M. Herlihy, and O. Rachman, "Atomic Snapshots Using Lattice Agreement," *Springer-Verlag Distributed Computing*, 1995.

[61] T. A. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," in *ACM SIGCOMM CCR*, 2009.

[62] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella, "The TCP Outcast Problem: Exposing Unfairness in Data Center Networks," in *USENIX NSDI*, 2012.

[63] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A Concurrent Key-Value Store with In-Place Updates," in *ACM SIGMOD*, 2018.

[64] "Lock-Free Programming." https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf.

[65] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box Concurrent Data Structures for NUMA Architectures," in *ACM ASPLOS*, 2017.

[66] "Oracle Database." https://www.oracle.com/index.html.

[67] "MySQL." https://www.mysql.com.

[68] "Microsoft SQL Server." https://www.microsoft.com/en-us/sql-server/sql-server-2016.

[69] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," in *ACM SIGMOD*, 1970.

[70] A. Braginsky and E. Petrank, "A Lock-free B+Tree," in *ACM SPAA*, 2012.

[71] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent Tries with Efficient Non-blocking Snapshots," in *ACM SIGPLAN PPoPP*, 2012.

[72] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: a fast, efficient data structure for string keys," *ACM TOIS*, 2002.

[73] N. Askitis and R. Sinha, "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings," in *ACSC*, 2007.

[74] D. R. Morrison, "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric," *JACM*, 1968.

[75] "MySQL: Comparison of B-Tree and Hash Indexes." https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html.

[76] "Oracle: About Hash Clusters." https://docs.oracle.com/cd/B28359_01/server.111/b28310/hash001.htm.

[77] "SQL Server: Hash Indexes." https://docs.microsoft.com/en-us/sql/database-engine/hash-indexes.