# Debugging Transient Faults in Data Center Networks using Synchronized Network-wide Packet Histories

*Pravein Govindan Kannan*[1]* *Nishant Budhdev*[2]† *Raj Joshi*[2]† *Mun Choon Chan*[2]
[1]*IBM Research - India* [2]*National University of Singapore*

## Abstract

Data center network faults are hard to debug due to their scale and complexity. With the prevalence of hard-to-reproduce transient faults, root-cause analysis of network faults is extremely difficult due to unavailability of historical data, and inability to correlate the distributed data across the network. Often, it is not possible to find the root cause using only switch-local information. To find the root cause of such transient faults, we need: 1) *Visibility*: fine-grained, packet-level and network-wide observability, 2) *Retrospection*: ability to get historical information before the fault occurs, and 3) *Correlation*: ability to correlate the information across the network.

In this work, we present the design and implementation of *S*yNDB, a tool with the aforementioned capabilities to enable root cause analysis of network faults. We implement and evaluate *S*yNDB with realistic topologies using large scale simulation and programmable switches. Our evaluations show that *S*yNDB can capture and correlate packet records over sufficiently large time windows (∼4 ms), thus facilitating the root cause analysis of various network faults.

## 1 Introduction

Large cloud providers need to quickly resolve network faults to meet their high SLA (service level agreement) requirements [35, 66]. However, a data center is a distributed system that is prone to bugs caused by non-deterministic timing of distributed events [43]. Therefore, debugging network failures occurring in modern data centers is extremely challenging due to the scale and complexity of interactions in a dynamic environment. Network faults in modern data center networks are often transient and non-reproducible. A recent study [66] reports that some network faults could not be reproduced even with techniques such as EverFlow [67] or Pingmesh [30]. Further, for a given network fault, the root cause can come in many forms. For example, a packet drop due to a table miss can happen either due to a parity error [66] or due to temporal inconsistency during a network update [36, 54]. Due to the complex nature of network faults, the key bottleneck in quickly resolving them lies in finding the root cause. For example, in AliBaba's production network, 90% of the total time required to resolve a network fault is spent in finding the root cause [66]. Another study from Facebook's network [47] notes that 29% of network failures go without establishing the root cause.

In this paper, we focus on the problem of finding the root cause of transient and hard-to-reproduce network faults with many possible root causes. To better understand the difficulty in finding the root cause, let's consider an example scenario of a microburst. When a microburst occurs at a switch port, there is a uniform distribution of packets from different sending hosts responsible for the microburst (more details in §7.2.1). This indicates a fan-in traffic pattern with no single offending flow. Now, there are two possible root causes for such a fan-in to occur. First, the sending hosts themselves are sending the data in a synchronized fashion. In this case, the issue needs to be resolved at the sending hosts by using techniques such as application-level jitter [53]. The other possible cause could be that the sending hosts are already introducing application-level jitter, but still result in a microburst due to non-deterministic interaction with other network traffic (see §7.2.1 for details).

One way to differentiate the root causes is to look at the packets involved in the microburst before it happens. If the packet arrival times at the first-hop switches connected directly to the sending hosts are synchronized, then the root cause is synchronized traffic. Otherwise, the microburst is due to non-deterministic interactions of flows in the network and even more details are required to identify the root cause. Therefore, in order to find the root causes of complex network faults, we believe that a network monitoring system needs to have the following capabilities:

- ***Visibility:*** Observability in space, the ability to observe network-wide metrics at a packet-level resolution (e.g. packet arrivals and departures at all ports for all switches). Aggregate states such as flow level statistics will not be able to provide sufficient visibility to the underlying sequence of events.
- ***Retrospection:*** Observability in time, the ability to be "always on" and look back on past network-wide states *before* the fault has occurred. When the problem is detected, the events leading to the problem would have already occurred and information related to these past events is lost unless additional effort is made to preserve the history. Such a capability is especially necessary to deal with faults that are transient, hard to reproduce and caused by non-deterministic interactions.
- ***Correlation:*** The ability to correlate network-wide events at small timescales. This is required if faults occur due to the interaction of traffic flows across multiple switches. Without this capability, it would not be possible to correlate events from different parts of the network.

Going back to the aforementioned microburst scenario,

*Work done at National University of Singapore
†Equal Contribution

*retrospection* allows us to look at network metrics before the microburst at the different first-hop switches with *visibility* at the granularity of packet arrivals. *Correlation* allows us to compare these arrival times across the different first-hop switches. If the root cause is not synchronous source traffic, we can again make use of these three capabilities to see what other non-deterministic interactions transpired in the network that led to the fan-in microburst. We elaborate more on this latter root cause in §7.2.1.

Several of the existing approaches [7, 32, 34, 37, 62, 63, 66] provide visibility to a good extent. But they either do not provide retrospection and correlation capabilities or provide them only partially (see Table 1). Since network faults occur unpredictably, providing retrospection typically requires the system to be "always on" in terms of collecting telemetry information. In this sense, NetSight [32], an "always on" version of postcard mode INT [7] comes closest to providing all the three properties with its packet-level visibility and retrospection capabilities. However, it does not provide a strong correlation property since it assumes that the postcards are received in order and out-of-order postcards can be corrected using topology information. Therefore, it can only correlate packets within a single flow at best. A strawman approach to achieving the three properties would be to augment a NetSight like approach with a precise dataplane time synchronization mechanism such as DPTP [38] so that it can now provide strong correlation with synchronized timestamps across the switches. However, such a solution does not scale to today's large data center networks because of the "always on" approach of NetSight in recording telemetry data. While efficient recording of telemetry data is achieved by trigger-based approaches such as INT-MX [7], PathDump [59], BurstRadar [37] and NetSeer [66], they compromise on retrospection since the packet history is not recorded, especially for switches and flows not involved in the trigger.

In this paper, we present, *S*yNDB, a packet-level, synchronized network-wide monitoring and debugging framework that provides all the 3 desired capabilities of *visibility*, *retrospection* and *correlation*. For *visibility*, *S*yNDB leverages programmable data-plane switches to capture packet-level telemetry information at nanosecond time resolution. A common issue with dataplane-based telemetry systems is that the metrics to be captured need to be specified at compile time [51]. To address this issue, *S*yNDB provides an interface to the network operator to specify and change the metrics at runtime *without* having to re-program the switch data-plane. For achieving retrospection, the key trade-off is that "always on" approaches are too expensive, while cheaper trigger-based approaches do not provide strong network-wide retrospection. We find a middle ground with *S*yNDB. Our key idea is to leverage the switch data-plane as a fast temporal storage to perform continuous recording of packet-level telemetry information (*packet records*) over a moving time window (*recording window*). When no network fault is detected, the recording window moves ahead and the older data beyond the record time-length is discarded. When a network fault is detected on any switch, the switch broadcasts a priority message to other switches in the network. On receiving this message, these switches send the packet records from the recent recording window to a monitoring server (collector) for permanent storage. At the monitoring server, the synchronized, network-wide packet-level data enables root cause analysis. In this way, *S*yNDB provides retrospection efficiently by exporting network-wide historical telemetry information only when a fault occurs. To correlate the telemetry information from different switches, *S*yNDB uses DPTP [38] to synchronize the switch data-planes. DPTP is a recently proposed time synchronization protocol for the network data-plane. *S*yNDB is thus able to provide visibility, retrospection and correlation capabilities all under the same framework.
In summary, we make the following contributions:

1. We present the design of *S*yNDB, the first network monitoring and debugging framework that provides all the three capabilities of visibility, retrospection and correlation for finding the root cause of transient and hard-to-reproduce network faults (§3).
2. For flexible visibility, we develop an abstract interface and a run-time support for the operator to configure and dynamically change the operating parameters of *S*yNDB such as fault detection conditions and the recorded metrics, without needing to re-program the data-plane (§5).
3. In order to achieve efficient retrospection capability, we design packet-level telemetry caching mechanism in the data-plane (§3.2). In doing so, we address the challenges of limited data-plane storage by developing compression techniques to minimize memory requirement while still retaining packet-level statistics (§3.2.1). We also develop techniques to further reduce the telemetry information exported for each fault trigger (§3.2.1).
4. We demonstrate the effectiveness of *S*yNDB by showing how it can be used to identify the root cause for transient and hard-to-reproduce network faults (§7). In particular, we demonstrate how *S*yNDB can identify different root causes for the same network fault using two different scenarios involving a microburst.

We have implemented *S*yNDB on Intel Tofino [8] switches using P4. The packet records at the collector are stored in a relational DBMS facilitating debugging of network faults using SQL queries (§4 and §6).

While *S*yNDB is designed to deal with transient and hard-to-reproduce network faults, it can be used as a tool to debug common network faults as well (Appendix B). In addition, *S*yNDB can be considered complimentary to existing frameworks such as INT [7] and NetSeer [66] by providing the capability to perform network-wide event correlation and retrospection.

Table 1: Comparison of *S*yNDB with existing solutions

| Solution | Visibility | Retrospection | Correlation |
|---|---|---|---|
| Per-packet Postcards [32] | Packet-based | Yes, Always On | Partial |
| INT [7] | Packet-based | No, network trigger | Partial |
| SwitchPointer [60] | Packet-based (Flow-level locality) | Yes, host trigger | Partial |
| Sketch Frameworks [46, 62] | Flow-based | Past Aggregated Counts, N/A | No |
| BurstRadar [37] | Packet-based | No, fixed network trigger | No |
| Speedlight [63] | Switch Metrics | No, Scheduled | Causal Consistency($\mu s$) |
| NetSeer [66] | Flow-based | No, fixed network triggers | No |
| *S*yNDB | Packet-based | Recent History, Programmable network triggers | $<100ns$ (DPTP [38]) |

## 2 Related Work

Network monitoring literature is wide and extensive, but none of the existing works provide all the three capabilities of visibility, retrospection and correlation required to find the root causes of transient and non-reproducible network faults. Here we mention some of the most relevant works.

**Network Based.** Query-based streaming telemetry systems like Marple [51], Sonata [31], etc as well as sketch-based frameworks [34, 46, 62] can provide network-wide visibility and retrospection but only with aggregated metrics and without any network-wide correlation capability. *S*yNDB is complementary to these streaming telemetry systems in that it additionally correlates information collected across the network and supports complex fault triggers based on input from the streaming telemetry system. Systems such as BurstRadar [37], Mozart [45], and INT-MX [7] that perform trigger-based data collection cannot provide retrospection. Mozart [45] involves coordination between network devices to start collection of telemetry data while coordination in *S*yNDB is to export already collected network-wide telemetry data. Through switch-local timestamps, INT-MX can only provide partial correlation and no network-wide correlation. Both PathDump [59] and SwitchPointer [60] leverage end-host storage to collect packet-level telemetry information providing retrospection. However, the fault triggers for both of them are only host-based and SwitchPointer provides only partial correlation with its millisecond-level epochs. Speedlight [63] uses synchronized network snapshots to provide microsecond-level casual consistency which is insufficient for correlation required in the example scenario in §1. Further, since it requires advance scheduling of snapshots every few milliseconds, it cannot provide retrospection. NetSeer [66] captures the flows that were affected by certain events like packet drops, path changes, and congestion, and mainly helps in fault localization - where the fault occurred and affected which flow (5-tuple). It only provides flow-level visibility and no retrospection or correlation. NetSight [32] which is equivalent of an "always on" version of postcard mode INT [7] (INT-XD) provides network-wide packet-level visibility and retrospection. However, it does not provide strong correlation capability as it generally assumes in order arrival of postcards. Even if strong correlation can be achieved with mechanisms such as DPTP [38], NetSight's "always on" recording is very expensive and does not scale to multi-petabit data center networks [58]. As we show in §7.3, NetSight can require 5TB or more storage per switch for every hour. The main bottleneck is the limited network bandwidth available for exporting the recorded data from the switches and the slow write speeds of data storage devices where the data would eventually end up. Other than scalability, this approach is also wasteful since network faults do not occur all the time [48] and, more than 60% of data center bugs occur due to the untimely delivery of a single message [43].

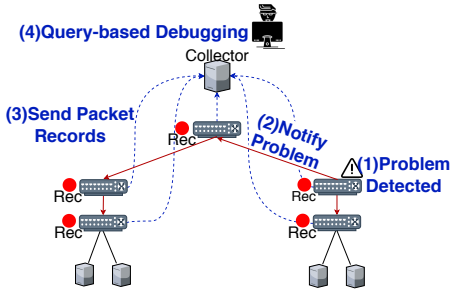



Figure 1: *S*yNDB Overview : Switches continuously maintain packet records, but send them to collector for debugging only upon detecting a problem

**End-Host Based.** Trumpet [50] performs end-host based monitoring of packets based on specific triggers to perform coordinated monitoring of network events. DETER [44] performs TCP replay for diagnosis of fine-grained TCP events. It constructs switch queues using simulation of the recorded TCP Packets in the end-hosts. Although it can help in debugging obscure TCP performance issues, it cannot diagnose problems inside the network at small time scales. Confluo [40] provides an end-host stack to diagnose network-wide events. However, it cannot provide packet level event correlation inside the network. SIMON [26] applies network tomography technique to reconstruct the switch queuing behaviour based on NIC packet timestamps. *S*yNDB is complementary to these techniques as it provides visibility, retrospection and correlation capabilities inside the network.

Table 1 summarizes recent related works based on their capabilities of visibility, retrospection and correlation.

## 3 Design

*S*yNDB provides fine-grained (packet-level, nanosecond resolution) and network-wide telemetry information, in a synchronized manner to ensure *visibility*, *correlation* and *retrospection*. *Visibility* and *correlation* are ensured by collecting packet-level telemetry and leveraging data-plane time synchronization. For Retrospection, the key idea is to leverage

the switch data-plane as a fast temporal storage for recording packet telemetry information over a moving time window. One of the advantages of recording telemetry information in the switch data-plane is that *S*yNDB can record information about *all* packets within a time window at line rate.

When no network fault is detected, the recording window moves ahead and older telemetry data beyond the window size is discarded. Hence, the recording window always maintains the recent history. When a trigger condition (e.g. packet loss or high latency) is observed at any switch (Step 1), high priority trigger packets are broadcast to all switches (Step 2), as shown in Figure 1. Upon reception of trigger packets, a switch collects all the packet records from the recording window and forwards them to the collector to be stored in a database (Step 3). Once data collection is completed, the operator can debug the fault using packet records collected both before and after the fault (Step 4). Debugging can be performed by operators using SQL queries. Details of the various components of *S*yNDB are explained in the following sections, namely *Visibility* (§3.1), *Retrospection* (§3.2) and *Correlation* (§3.3).

## 3.1 Visibility

**Packet Records**. To generate packet-level telemetry data, we record information for each packet that enters a switch. We call this information a *p*-record. Each *p*-record contains 3 basic fields: $[pID, pTime_{in}, pTime_{out}]$. *pID* is the packet ID which is comprised of a combination of the hash value of the packet headers (5-tuple flow key) and TCP/UDP checksum. The hash value helps in associating packets from the same flow, whereas the checksum helps in uniquely tracking each packet within the flow. Although hash collisions are possible, we can resolve them using topology and timing information. $pTime_{in}$ captures the time when the packet enters the switch. $pTime_{out}$ is the time when the packet leaves the switch. Additional fields are appended by the network operator to a *p*-record to capture statistics, such as queue depth, link utilization, forwarding table version, port counters, etc. An operator can specify such additional fields via *S*yNDB configuration (§5). To identify a *p*-record with a particular flow, the flow hash to 5-tuple flow key mapping could be temporarily stored in NICs (or hosts) and retrieved on demand.

## 3.2 Retrospection

After a *p*-record is generated in the data-plane for each packet, we store them in a ring buffer array in the switch data-plane. This ring buffer array maintains only the recent *p*-records and we call this the *history* buffer.

**Trigger Initiation**. While *S*yNDB collects *p*-records for each packet in the data-plane, it requires a trigger to initiate data collection. These triggers can be events such as congestion at a link, packet drops or packet reordering. The trigger conditions are monitored in the data-plane. Once a trigger is hit, the *p*-records can be transmitted to the collector.

To initiate network-wide *p*-record collection, we create a trigger packet to be broadcast (with priority) to other switches through the data-plane. In *SyNDB*, when a trigger condition is hit, a new trigger packet is created. The trigger packet is an Ethernet frame with a trigger header consisting of: 1) Trigger ID, 2) Trigger Type: unique type to classify trigger, and 3) Trigger Time: time when the trigger was hit.

The switches receiving the trigger packets further broadcast it to their neighboring switches and so on. Due to redundancies in trigger packet broadcast (multiple paths in data center topology), unless the network is partitioned, trigger packets reach the entire network. On receiving a trigger packet in the data-plane, the switch stops storing *p*-records in the *history* buffer and instead uses a fixed buffer (we call it *future* buffer) for subsequent storage of *p*-records. If a switch had previously received a trigger packet with the same ID, then it is dropped. The *history* buffer and *future* buffer contains *p*-records of packets before and after the trigger condition respectively.

Conceptually, the size of the *history* buffer that is needed to be stored for debugging purpose depends on the round-trip-time (RTT). In a data center context, recent measurements show that VM-to-VM RTTs vary between 5$\mu$s to 100$\mu$s [24]. For a packet rate of 1 Bpps, 1 million *p*-record entries could maintain at least 1 millisecond duration of history if the switch pipeline is fully utilized. This translates to packets corresponding to at least 10's of RTTs available for debugging. In practice, the packet rate is usually much lower than 1 Bpps and this translates to a much longer time window. We present our evaluation on the time window in §7.1 and discuss the sizing of *future* buffer to enable continuous recording in §7.3.

***p*-record Collection**. Upon receiving a trigger packet with a new trigger ID in the data-plane, collection of *p*-records is performed. The control-plane initiates the data-plane packet generator which generates collection packets to read the *p*-records from the history buffer. A collection packet can read only one *p*-record each time it traverses through the switch [16], before being forwarded to the collector via a mirror-port. Consequently, we recirculate the collection packet multiple times in the data-pane to coalesce multiple *p*-records into a single packet. This reduces the large serialization overhead, if each packet contained exactly one *p*-record.

Once the number of *p*-records in a packet has reached a threshold (configured by switch control-plane), the collection packet is forwarded to the collector. A collection cycle ends when all the *p*-records stored in the data-plane have been transmitted or sufficient time has elapsed since the trigger. The collection cycle repeats upon a new trigger hit. It is important to note that regular traffic forwarding is not disrupted during the trigger and collection process. For cases when an additional trigger is hit during collection process of the previous trigger, a new trigger packet is generated and collection period is extended (discussion in §7.3). Techniques such as

bulk DMA read could also be employed to collect *p*-records. However, such techniques require additional packetization in the control-plane to forward the whole set of *p*-records to a controller. The pseudo-code for recording, trigger and collection is shown in Appendix A.

**Aggregating Triggers**. *S*yNDB supports operator to specify collection to be performed when a set of trigger conditions occur within the historical time window. To support this, upon receiving a trigger packet, *S*yNDB maps the trigger type to a bit-index in a temporal trigger bit-array. It also sends the trigger packet to the control-plane where a timer for each trigger type is maintained, and the bit corresponding to the trigger type is cleared upon expiration. Hence, the temporal trigger bit-array represents the lists of triggers that occurred in the network for the past historical time window. Based on this trigger bit-array value, collection could be configured to be performed. The triggers are customizable by the network programmer, and is presented later (§5).

#### 3.2.1 Reducing Collection Overhead

As *S*yNDB collects data only on event triggers, the amount of data collected is expected to be much smaller compared to continuous monitoring. To further reduce the data collected for each trigger, *S*yNDB implements two mechanisms, a compression scheme on the p-record and a scope reduction on the network level.

**p-Record Compression:** *S*yNDB performs the following *p*-record compression while ensuring packet ordering:
*1. Compress pID* : Consecutive packets from the same flow do not need their *pID* stored. Instead, a counter corresponding to the last *pID* is incremented.
*2. Compress* $pTime_{in}$ : Similarly, incoming packets within a time window (e.g. 64 ns) do not need the $pTime_{in}$ stored individually. Instead, a counter for the number of packets received in the past time window is incremented. The packets within the time window are assumed to have uniform inter-arrival times. The same approach is applied to $pTime_{out}$. In the best case, a single (*pID*, $pTime_{in}$, $pTime_{out}$) tuple plus the corresponding (n-bit) packet counters are sufficient to represent ($2^n$) packets in the same time window.

**Reduction on the Network Level:** On detecting a fault, *S*yNDB performs collection of *p*-records from all switches in the network. This may burden the collector with unnecessary data if the network is huge and the root cause is localized. We mitigate this by a simple technique. Each switch maintains a list of links from which it received the packets for the historical time window. Upon fault trigger, instead of broadcasting trigger packets, they are selectively multicast to only the links from where the packets were received in the recent recording window. The intuition behind this is that we are interested in where the current set of packets came from. This solution provides the ability to trace every packet which appeared in the trigger switch to its source, while reducing the number of switches involved in the collection.

### 3.3 Correlation

**Time-synchronization**. *S*yNDB uses global timing information to correlate packet records from multiple switches to help construct an accurate network-wide ordering of events. Hence, the data-plane clocks (used for $pTime_{in}$ and $pTime_{out}$) across switches are synchronized to a fine granularity to avoid timing inconsistencies. To rightly correlate established events in distributed systems using "happened before" relation, causal consistency [41] is essential. In *SyNDB*, since a trigger event is the captured reference point, causal consistency is the right model to correlate events happened before the trigger event. We derive the necessary condition to ensure causal consistency below.

Let's consider two directly connected switches X and Y, with internal clocks $C_X$ and $C_Y$ respectively. We denote the synchronization error $|C_X - C_Y|$ between the internal clocks by $T_{err}$. Packet A is transmitted from switch X to switch Y. Packet A leaves switch X at $TimeOut_X$, and enters switch Y at $TimeIn_Y$, after a propagation delay $D$. $TimeOut_X$ corresponds to the time packet A enters the egress pipeline in switch X after queuing. This is the latest available time in the data-plane for a packet [16]. Similarly, $TimeIn_Y$ corresponds to the time packet A enters the ingress pipeline at switch Y. Consequently, propagation delay now becomes the sum of egress pipeline delay, packet deparsing delay, MAC processing delay, and wire delay.

$$D = EgressDelay + DeparserDelay + MACDelay + WireDelay \quad (1)$$

To ensure *causal consistency* between packet records, we should see packet A leave switch X before reaching switch Y. In short, $TimeOut_X$ should be less than $TimeIn_Y$. This will be true if the synchronization error between the internal clocks is less than the propagation delay.

$$T_{err} < D \quad (2)$$

If the condition stated in this equation can be met, we can ensure consistency between any set of packets transmitted between two adjacent switches.

Previous works on network time synchronization have shown that the $T_{err}$ between neighbouring switches is in the order of tens of ns [38, 42]. Additionally, real world data shows that $D$ between two adjacent switches ranges between 360 ns to 1900 ns under varying traffic conditions [38]. Thus it is possible to achieve causal consistency between adjacent switches, using current time synchronization techniques. The same principle can also be extended between switches separated by several hops in the network. In such cases, we observe that the increase in propagation delay is higher than the increase in $T_{err}$, thus ensuring consistency between switches across multiple hops.

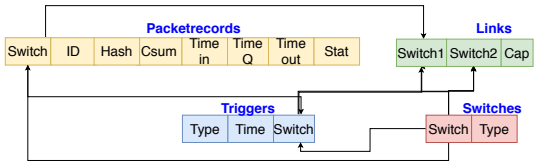


Figure 2: *S*yNDB Debugger Database Schema

## 4 *S*yNDB Debugger

The collector composes of multiple servers which store and analyze the *p*-records. The *p*-records are stored in a relational database (RDBMS) which allows the *p*-records to be queried using SQL. The collector also stores information regarding the trigger events, network topology, and position of switches within the topology. Before storing *p*-records in database, *S*yNDB performs hash collision removal using topology and timing information. The hash collision removal is performed using a simple heuristic based on the ground-truth of the queuing time of a *p*-record and the identity of the switch. Duplicate hashes found are then re-assigned with other *p*-record ids. The *p*-records are organized using tables in a relational database as shown in Figure 2.

**1. Packetrecords**: This table stores basic and custom fields within each *p*-record. Each *p*-record stores: 1) Switch ID, 2) Packet ID, 3) Packet Hash, 4) TCP/UDP Checksum 5) Time In, 6) Time Queued, 7) Time Out and, 8) Operator-specified statistics. Note that Packet ID is just a combination of packet hash and the checksum. They are stored separately to facilitate flow-level queries as well as packet-level queries.

**2. Triggers**: This table stores information regarding each trigger event. Each trigger event stores: 1) Trigger Type, 2) Trigger Time, and 3) Trigger Origin Switch. This enables *S*yNDB to classify network faults based on the trigger type.

**3. Links**: This table stores the topology of the data center, as specified by the network operator. We do not infer the topology from the packetrecords table because it is possible for some links to have zero utilization. Each link stores the endpoints and the link capacity.

**4. Switches**: This table stores the position of a switch in the topology, e.g. ToR, Aggregation, Core, etc.

To determine the root cause of a network fault, we use SQL queries on the above tables. For example, in the case of an incast, culprit packets and their routes can be obtained by combining information from packetrecords, triggers, and links tables. The output of these queries can also be used to replay or build dashboards using tools [6, 12, 17], which are beyond the scope of this work.

We list some example queries below (scenarios in §7.2):

1) List the events in the trigger switch using:

```
Select * FROM packetrecords JOIN triggers
         ON packetrecords.switch = triggers.switch;
```

2) List the packets in the trigger switch and the routes taken:

```
Select * FROM (packetrecords as P) WHERE id
         IN (select id from packetrecords JOIN triggers
         ON p.switch = triggers.switch
         AND p.time_in<triggers.time) ORDERBY time_in ASC;
```

```
precord {
  fields {
    field_list_1;
    field_list_2;
    ...
  }
  default_field : field_list_{x};
  history       : {y};
  future        : {z};
  time_window : {t ms};
}
trigger {
  conditions {
    c1 = condition_1;
    c2 = condition_2;
    ...
  }
  collection {
    c1 [&|] c2' [&|] c3' ...
  }
}
```
Figure 3: *S*yNDB Configuration Syntax

## 5 *S*yNDB Configuration

*S*yNDB provides an interface for defining *p*-records and triggers for programmable switches. The programmer configures the following parameters in *S*yNDB: 1) the network statistics (fields) to be collected in *p*-records, 2) the number of *p*-record entries to be collected, and 3) the trigger (fault) conditions to initiate a collection of *p*-records. The fields specified in the configuration could be: 1) switch-provided metadata (queue depth, ingress port, egress port), 2) packet header data (flow-id), and 3) data that is computed and stored in user metadata by the programmer (link_utilization, counters, EWMA). The *S*yNDB configuration is compiled, then translated to P4 and finally embedded with the original switch P4 program. Figure 3 shows the interfaces to define *p*-records and trigger conditions.

A *p*-record defines a list of field_lists. Each field_list contains one or more (metadata) fields [13] from the Packet Header Vector (PHV) [16] supported by the switch architecture and defined in the user's P4 program. A "default_field" list is specified by the programmer which is the active field_list to be included in each *p*-record. The current *active* field_list can be changed during runtime. The "history" refers to the total size of the *history* buffer, while "future" refers to the size of the *future* buffer. The "time_window" is the target historical window (in milliseconds), and this is used to maintain the trigger and broadcast window. The user declares a list of trigger conditions, which are predicates operating on header/metadata fields. For example, meta.link_utilization > 90. Finally, based on the triggers declared, the collection can be configured to be performed using individual triggers or a combination (AND(&), OR(|) of multiple triggers defined). For example, let $c$ be the local trigger condition and $c'$ be a trigger condition happening elsewhere in the network. A representation like $c1\&c2'$ would trigger a coordinated collection by a switch A only if condition $c1$ occurs at A, and $c2$ has occurred in another switch in the network. Defining triggers conditions and collection could be based on several

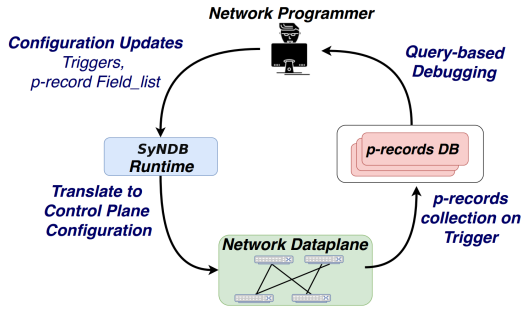



Figure 4: *S*yNDB from Programmer's perspective

network metrics in the network like packet drop, high packet queuing, loops, etc. Additionally, it could be based on well documented symptoms and alarms observed by network operators [28, 43, 47].

***SyNDB*-Runtime**. In practice, there is a need to make changes to the *p*-record structure and trigger configurations while *S*yNDB is running without the need to recompile and load a new P4 program. *SyNDB*-Runtime facilitates changing the configuration in the following ways: 1) Adding a new field_list or editing the active field_list, and 2) Adding/removing trigger conditions. Note that these changes are restricted to the available PHV contents in the data-plane as there is no modification to the parser of the underlying P4 program.

When the *S*yNDB configuration is compiled, the compiler enumerates all the PHV contents (packet headers, switch and user-defined metadata) of the P4 program. It then creates template tables with actions for each PHV container to be stored in the *p*-record. This facilitates the runtime to dynamically add/remove the fields to be recorded in each *p*-record. The fields could be TCP sequence number, TCP flags which are part of packet headers, or ingress_port, queue_depth, etc. which are part of the switch meta-data. The field to be added cannot be a metric (e.g. EWMA) that is not defined or a packet header that is not parsed by the already compiled P4 program. Since PHV contents are limited, enumerating and storing them in actions do not significantly increase data-plane resource consumption. The maximum bytes in a *p*-record and the number of *p*-record entries (*recording window*) is fixed at compile-time based on the available hardware resources (stateful ALUs and SRAM). To facilitate addition/removal of trigger conditions at runtime, *S*yNDB configuration compiler uses similar enumeration technique and generates range-based match-action tables. Since collection is performed based on the trigger bit-array value, this value is added/modified based on the collection condition changes. Additionally, *SyNDB*-Runtime updates the collector each time the *S*yNDB configuration is changed, to ensure that *p*-records are stored correctly.

Figure 4 summarizes the *S*yNDB workflow. A network programmer configures the statistics to be recorded and the fault triggers. The configuration can be continuously tweaked to suit the statistics that the programmer wants to keep an eye on using *S*yNDB-Runtime.

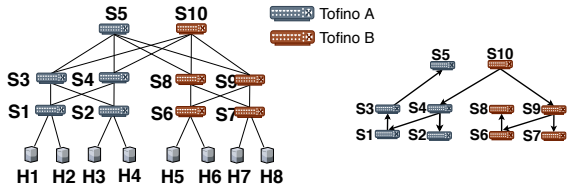
# 6 Implementation

***S*yNDB Dataplane.** We have implemented *S*yNDB on Intel Tofino [8] switches using P4 (∼1900 LoC). We use DPTP[1] for time synchronization between switches. We use DPTP since it is implemented on PSA [16] and provides a global timestamp in the data-plane. We store the baseline contents of *p*-record in both ingress and egress pipeline. Ingress pipeline maintains the write_index of the *history* ring buffer array upon a packet arrival and stores the *pID* and $pTime_{in}$. Egress pipeline stores $pTime_{out}$ and custom field_list to be captured. *pID* is a combination of 16-bit flow hash, and 16-bit TCP/UDP checksum. $pTime_{in}$ is a 32-bit global timestamp (at nanosecond granularity) of the packet when it enters ingress pipeline of the switch. On the other hand, $pTime_{out}$ is a 24-bit field which captures time when the packet enters egress pipeline. A 24-bit value allows to calculate upto 16 ms of queuing. The basic uncompressed *p*-record is 11-bytes in size. To implement compression, we maintain separate 8-bit counter array associated with *pID*, $pTime_{in}$ and $pTime_{out}$. Trigger conditions are implemented using TCAM tables which create a trigger packet upon match. A trigger packet is created by cloning the current packet and inserting a new header type for trigger packet after stripping the payload and other headers. By using a TCAM match for the trigger table, different aggregate conditions of individual triggers can be supported using wildcards. We implement *S*yNDB control-plane, which performs : 1) Time-keeping of temporal trigger bit-array, 2) Updating multicast port-group, 3) Set up packet generators for *p*-record collection.

***S*yNDB Runtime.** We have implemented the compiler for *S*yNDB configuration using Rust (∼4000 LoC). It takes as input the configuration and the switch P4 program, and generates a P4 code that implements *p*-record storage, trigger conditions and collection logic. *p*-record storage and trigger conditions are executed using stateful ALUs. Additionally, the runtime environment (implemented in Python) accepts commands to modify configuration such as: 1) changing the active field_list, 2) adding new field_list, and 3) adding/removing trigger conditions. These configurations translate to control-plane configuration updates of the composed switch P4 program. The runtime supports addition of new fields to *p*-records, by varying the recording parameters of the set of pre-enumerated PHV contents from the control-plane. A similar approach is also used for modifying trigger conditions. The maximum set of trigger conditions and contents in *p*-records are specified during the generation of the P4 program.

Finally, we implement the collector using `n2disk` utility (with *PF_RING* [15]) to store collection packets as PCAP files in the local disk. Additionally, we implement a Python program to parse the PCAP files, decompress and store individual *p*-records in a MySQL database. The collector also

[1] https://github.com/praveingk/DPTP




a. Testbed Topology

b. Network-level DAG used for DPTP synchronization

Figure 5: Evaluation Topology Testbed

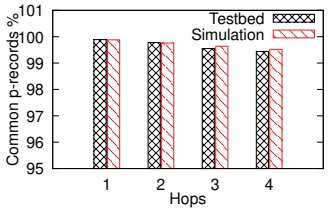



Figure 6: Comparison of testbed and simulator for common *p*-records

takes as input the *S*yNDB configuration for initializing the database. Each time the active field_list is modified through the *S*yNDB runtime, the update is passed on to the collector.

## 7 Evaluation

We evaluate *S*yNDB using a hardware testbed as well as large-scale simulations. Our hardware testbed consists of 4 physical servers, and 2 Intel Tofino Wedge100BF-32X switches [18]. The servers and the switches are virtualized to create a fat-tree topology [19] (see Figure 5(a)). Switches S1-S5 are virtualized on the first physical switch ("Tofino A") using 10G loopback links while switches S6-S10 are virtualized on the other physical switch ("Tofino B"). Each virtual switch is configured to have 10K and 5K *p*-records in the history and future buffer respectively. Each *p*-record entry is 16 bytes in size. Figure 7 shows the *S*yNDB configuration that we use in the testbed-based evaluation. Additionally, we synchronize each virtual switch's data-plane to S10 using DPTP (see Figure 5(b)).

For large-scale simulations, we have built a packet-level simulator[2] in C++ (∼6K LoC) that implements low-level packet transmission and forwarding behaviors for hosts, links and switches. To validate our simulator, we compare its results with those from the testbed for the following experiment. For the topology in Figure 5(a), we send 10 Mpps CBR traffic along the path S1-S4-S10-S9-S7 with each switch storing 10K *p*-record entries. Switch S7 generates a trigger after receiving 10,000 packets which is then broadcast to other switches to initiate *p*-record collection. Based on the *p*-records available at the collector, in Figure 6, we plot the percentage of common *p*-records seen by other switches compared to those seen by S7. We observe that the testbed and simulation results match each other closely – due to hop delays experienced by the trigger packet, the percentage of common *p*-records reduces slightly with increasing number of hops from S7.

[2] https://github.com/rajkiranjoshi/syndb-sim

```
field_list SyNDB_verification {
    meta.packet_sequence;
}

field_list SyNDB_scenario {
    meta.ingress_port;
    meta.link_utilization;
    meta.drop_counter;
}
precord {
    fields {
        SyNDB_verification;
        SyNDB_scenario;
    }
    default_field : SyNDB_scenario;
    history       : 10000;
    future        : 10000;
    time_window   : 1;
}
```

```
trigger {
    conditions {
        a = meta.time_queue > 10000;
        b = meta.fwding_table_miss == 1;
        c = meta.fwding_update == 1;
        d = meta.pkt_sequence == 5000;
    }
    collection {
        a;
        b & c';
        d;
    }
}
```

Figure 7: *S*yNDB Configuration for Evaluation

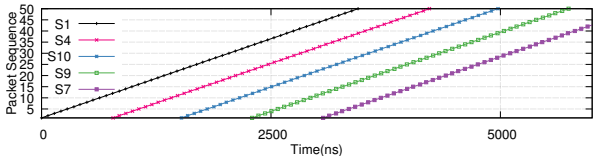



Figure 8: Sequence of packet arrival for 10Mpps CBR traffic

The evaluation is divided into three parts. In §7.1, we evaluate *S*yNDB for consistency in *p*-records and large scale operation. In §7.2, we present different network debugging scenarios with one in full details. Finally, we evaluate the overhead of *S*yNDB in §7.3.

### 7.1 Design Validation

In this section, we evaluate the *S*yNDB for its ability to provide consistent *p*-records at packet-level granularity and to provide retrospection and correlation at scale.

#### 7.1.1 Consistency of *p*-records

To ensure that the *p*-records captured by *S*yNDB are consistent, the time synchronization error should be less than the propagation delay between adjacent switches (equation (2)). In our hardware testbed (Figure 5a), we measure DPTP synchronization error as well as the propagation delay between adjacent switches. We observe that the worst case synchronization error is less than 50 ns while the propagation delay varies between 400-450 ns. Thus, synchronization error is much lower than the propagation delay between two switches and hence captured *p*-records should be consistent with the ground truth. To validate this further, we send a CBR traffic of 10 Mpps (limited due to 10G host links) along the path S1-S4-S10-S9-S7, with each packet annotated with a sequence number along. The switches record the sequence number of each packet in the corresponding *p*-record. After receiving 5000 packets, switch S1 generates a trigger packet (trigger *d* in Figure 7) which when received, each switch sends the *p*-records to the collector. In Figure 8, we plot the packet sequence number against time for a sequence of 50 packets. We observe that every packet is recorded in the next switch

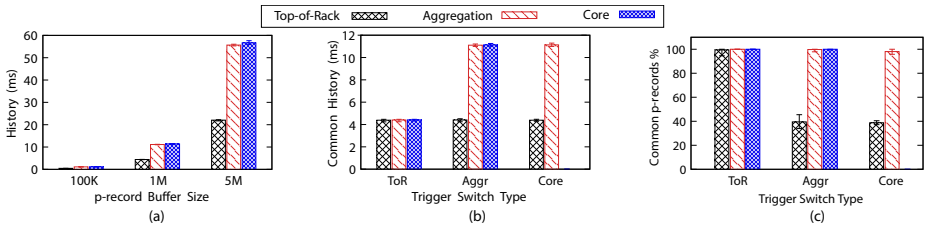



Figure 9: Simulation results for history captured at the trigger switch, correlation history and percentage of common p-records for different trigger switch types

only *after* it has left the previous switch. Furthermore, the timestamps increase linearly. This behavior confirms that the *p*-records captured by *S*yNDB across all the switches in the network are consistent and at expected intervals.

### 7.1.2 Retrospection and Correlation at Scale.

We perform large scale simulations to evaluate how much retrospection and correlation *S*yNDB provides in realistic traffic scenarios.

**Simulation Setup.** All simulation runs were done on a k=24 fat-tree topology (720 switches, 3456 hosts) with 100G links. The network has a total bisection bandwidth of 172.8 Tbps. For generating packets from the hosts, we use distributions of packet size and inter-packet gap as measured by Benson et al. [20] from a data center hosting web applications. We configure the traffic pattern such that 75% of total traffic is intra-rack as observed in cloud datacenters [20]. On top of these basic traffic characteristics, we add additional incast traffic such that 30% of the host links experience 100% utilization for 10% of total simulation time. The inter-packet gap distribution from Benson et al. is originally for 1 Gbps switch links. We scale it to adjust the load on 100 Gbps host links such that the average utilization (over 100 ms interval) is about 34%, with the busiest 5% of links experiencing about 42% utilization. These utilization characteristics are similar to those reported by Facebook [55]. All switches are configured with a *p*-record history buffer of 1M entries. We also configure a hop delay of 1 $\mu$s per switch such that the maximum RTT across the network (inter-pod) is $\sim$11.5 $\mu$s [21]. Each simulation run simulates 100 ms of network run time and delivers about 5.2B packets. Within each run, we generate 50 triggers on randomly chosen switches across the three switch types – top-of-rack (ToR), aggregation (Aggr) and Core. The following results are based on the aggregate data from 10 independent simulation runs.

We use two metrics to compare *p*-record buffers between the triggering switch and the upstream switches from which it receives packets: (i) Common *p*-records (Figure 9(c)): the percentage of common *p*-records between the trigger switch and the upstream switches. (ii) Common History (Figure 9(b)): the time difference between the latest and oldest common *p*-record. While the first metric quantifies the correlation using the similarity of *p*-records between the switches, the latter reflects the ability of *S*yNDB to perform retrospection.

For triggers originating at the ToR switch, the maximum common history that can be captured at other upstream switches is limited by the history at the ToR switch ($\sim$4 ms). Note that $\sim$4 ms history is worth $\sim$350 RTTs since maximum RTT in our setup is $\sim$11.5 $\mu$s. For triggers originating at the Aggr/Core switches, history of $\sim$11 ms is recorded. This is expected since the ToR switches experience higher packet rates (due to 75% intra-rack traffic) and hence provide smaller history relative to Aggr and ToR switches.

As for percentage of common *p*-record, we can capture $\sim$100% of common *p*-records in the upstream switches in many cases. The exceptions are for cases where the trigger switch is the Aggr/Core. The percentage of common *p*-records with other ToR switches is $\sim$40% since the *p*-records in upstream ToR switches are quickly overwritten by newer intra-rack packets. Note that in a fat-tree topology a packet passes through exactly one Core switch. Hence, if the trigger switch is a Core switch, there is no upstream Core switch.

Figure 9(a) shows the time window history that can be recorded for different *p*-record buffer sizes.

**Takeaway:** The simulation results show that the amount of history that can be captured depends on the incoming packet rate and the buffer size. For the traffic load and distribution used in the evaluation, *S*yNDB is able to consistently capture common *p*-record across different upstream switches. The time history available for retrospection varies from 4ms to 11ms using a buffer size of 1M *p*-record.

## 7.2 Network Debugging Scenario

In this section, we show how *S*yNDB can be used to debug one of the most common transient network faults, namely microburst [37, 47]. The evaluation uses the same configuration setup as defined in Figure 7 on the hardware testbed in Figure 5. Each *p*-record is configured to contain the custom "field_list: SyNDB_scenario". It contains three metrics : 1) Ingress Port 2) Link Utilization and 3) Drop Counter.

Ingress port of a packet is provided by the switch meta-data. Link utilization is calculated over a window of 10 $\mu$s in the data-plane using a low-pass-filter. Drop counter is the number of packets which missed the forwarding table. Additionally, we configure *S*yNDB to perform collection of *p*-records based on three triggers : (1) High Queuing Delay (trigger *a*), (2) Table Lookup Miss (trigger *b*) and (3) Network Configuration Update (trigger *c*). Data collection is initiated when a switch

receives trigger *a* or a switch receives both triggers *b* and *c*. In each of the following case studies, we generate data and control traffic to emulate the corresponding network faults. The host data traffic is generated using MoonGen [23].

### 7.2.1 Microbursts

Microburst is a common problem in data centers where congestion is caused by a short burst of packets lasting for at most a few hundred microseconds [56, 65]. Traffic bursts occur due to various reasons like application traffic patterns (e.g. DFS, MapReduce), TCP behavior and also NIC-offloads (segmentation, receive) [39]. The complex interactions and traffic patterns make microbursts debugging extremely complicated. It is necessary to find the root cause to determine how the issue should be resolved.

In this experiment, we demonstrate how two microburst events that are detected by the same trigger can be attributed to different root causes using *S*yNDB. In one scenario, the microburst is due to incast of synchronized application traffic. In the other scenario, the microburst is caused by the interaction of uncorrelated flows with different source-destinations.

**Synchronized Application Traffic.** We consider the commonly known fan-in traffic pattern of data center networks exhibited by applications such as MapReduce and Distributed File System (DFS). This is an incast traffic pattern where many sources transmit to a small number of destinations within a short time window. These short bursts of traffic increase the queuing delay at microsecond time-scales. The challenge in identifying the root cause of such microburst is that many sources contribute to the total traffic and the burst occurs only for a very short time.

We setup the experiment with hosts H1 to H6 sending data to H8 as shown in Figure 10. Each host sends a burst of 10 1500-byte packets at an average rate of 1 Gbps to H8 via ToR switch S7. All links have capacity of 10 Gbps. In the experiment, the sources started in an asynchronized fashion, but over time transmissions from different hosts can synchronize their transmissions causing sudden spikes in queuing delays on switch S7, triggering the trigger *a*. Such synchronization of periodic messages over time has been known to occur in routing message updates [25].

With *S*yNDB, to determine if the issue of microburst is caused by synchronized fan-in traffic, a query of the queuing delay at S7 together with the packet arrival information at the ToR switches before the microburst detection can be performed at the collector as shown below:

```
SELECT switch, ingress_port, time_in  FROM packetrecords
    WHERE id IN (SELECT id FROM packetrecords AS A
    JOIN triggers as T  ON (A.time_in < T.time
    AND A.switch = T.switch)) AND switch
    IN (SELECT switch FROM switches WHERE type = "tor");
SELECT time_queue FROM packetrecords where switch=7;
```

Listing 1: Query to list the packet arrival times at ToR switch ports and queuing delay at S7

The answer to the query is shown in Figure 10. The top

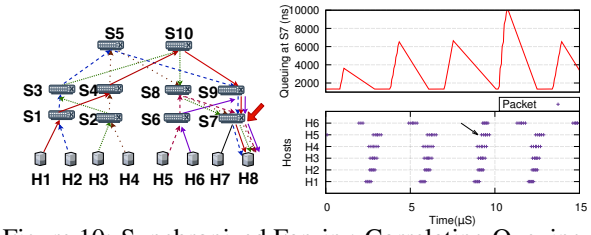



Figure 10: Synchronized Fan-in : Correlating Queuing at S7 and Packets arrival sequence at ToR Switches

right plot in the figure illustrates the queue buildup over time. When we correlate the packet arrivals from different hosts before the bursts occurred, we see that the packets that make up the bursts are transmitted by hosts H1 to H6 synchronously and reach S7 at about the same time. The root cause of this microburst from H1 to H6 can thus be determined as host-based synchronized traffic.

**Non-Synchronized Application Traffic**. Synchronized incast is just one possible cause for microburst. As discussed by Shan et al. [57], there are many other scenarios for microbursts. In this experiment, we generate microburst events through the interaction of traffic from multiple hosts that are not synchronized at host. However the individual flows, due to different queuing behaviour across hops (due to cross-traffic), arrive synchronously at the bottleneck link. In this scenario, hosts H1 to H6 send bursts of 10 packets at an average rate of 1Gbps to H8. A randomized delay of upto 5$\mu$s is added before sending a burst to minimize traffic synchronization. In addition, another flow sends a burst of 10 packets every 1ms of packets from H9 to H6 (through S1-S4-S5-S8-S6) at an average rate of 2Gbps. Note that this flow (H9 - H6) runs asynchronously and does not travel through the bottleneck switch (S7) where the microburst occurred. Nevertheless, we observe microbursts on the link from S7 to H8.

A query of the queuing delay at S7 together with the packet arrival information is shown in Figure 11. The shaded portion in the bottom right plot shows the duration in which the flow from H9 to H6 can occur. The information provided by *S*yNDB shows that the microburst is likely due to a combination of factors, namely (1) the synchronization of the bursts among the pair of flows from H1 & H2, H3 & H4 and H5 & H6; (2) the burst from H9 to H6 arriving just before the bursts from H1 & H2 in S1 and the bursts from H3 & H4 in S5. This causes a queue buildup resulting in packets from H1 to H6 arriving at S7 at about the same time. The root-cause is thus due to interaction of network queuing effect caused by cross traffic.

**Additional Use Cases:** Table 2 presents a list of additional use-case scenarios for *S*yNDB. We have experimentally evaluated (on the hardware testbed) the use-cases for debugging network faults related to network configuration updates and transient load imbalance whereby the use of multiple triggers is demonstrated. The details are provided in Appendix B.

**Takeaway:** The scenarios we presented show that in or-

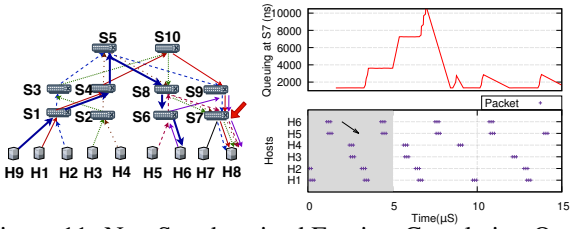



Figure 11: Non Synchronized Fan-in : Correlating Queuing at S7 and Packets arrival sequence at ToR Switches

Table 2: Use-cases of *SyNDB*

| Fault and Description |
|---|
| **Routing Bugs**. Bugs in the routing protocols, for example, synchronization between LDP IGP protocols [10] could be due to timing issues such as race conditions [49]. Since *Sy*NDB provides causal consistency, it helps in correlating different protocol packets and to narrow down the root-cause. |
| **App Timing Bugs**. *Sy*NDB can be used to debug timing bugs in distributed systems (Hadoop [1], ZooKeeper [2]) where more than 60% of the bugs are due to a single packet [43]. In these timing bugs, a dead-lock is caused by a missed or delayed message. *Sy*NDB can help to identify and track message lost or delayed by raising trigger conditions when it observes reordering or drops of certain packets. |
| **Traffic Pattern Analysis**. *Sy*NDB collection could be triggered at regular intervals to study and profile traffic patterns [64] and to optimize cloud applications. In this case, *p*-records could contain the flow-id (5 tuple) to understand the interactions on a flow-level granularity. |
| **Routing Loops** Routing Loops can be detected by observing duplicate *p*-record ids at the switches. |
| **Network Configuration Updates**. Refer Appendix B |
| **Transient Load imbalance**. Refer Appendix B |

der to identify the root cause of complex network faults, it is often necessary to have the *visibility* into packet statistics, the ability to look at past events (*retrospection*) and the timing information to correlate observations across switches (*correlation*). While NetSight [32] and INT-MX [7] can detect routing loops and bugs, NetSight has higher collection overhead due to its "always on" nature (§7.3). Also, while NetSight cannot perform *correlation* across the network, INT-MX cannot perform *retrospection* to identify whether the root-cause of such issues is due to configuration, race condition, etc. While Marple [51], BurstRadar [37] can detect microbursts, they do not provide *correlation* and packet-level *visibility* to inspect the root-cause of microbursts due to timing related issues like synchronized traffic. While it is possible to analyze traffic patterns in a coarse manner using systems like Speedlight [63] with tpprof [64], *Sy*NDB can be used to understand microsecond-level changes in traffic.

### 7.2.2 Partial Deployability

*Sy*NDB-enabled switches can be deployed incrementally with each new switch providing additional visibility into the network. To maximize effectiveness, deployment can start from ToR switches where most congestion events occur [65]. For DPTP synchronization, links can be added between adjacent ToR switches, which is not a complex undertaking [61]. With just *SyNDB*-enabled ToR switches, issues like microbursts(§7.2.1), application timing bugs can be debugged fully with just the ToR switches' *p*-records, while issues like routing loops, bugs, load imbalance and configuration updates can be partially debugged if the ToR switch is involved in the fault. In such cases, to infer the core network's states, network tomography techniques [26] can be employed.

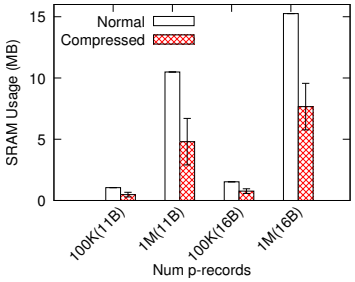



Figure 12: SRAM consumption for different packet rates and *p*-record size

## 7.3 *SyNDB* Overhead

**SRAM Overhead.** We estimate the total amount of SRAM consumption used by the *history* buffer based on a compressed *p*-record size of: 11 bytes (baseline compressed *p*-record), 16 bytes (evaluation configuration in Figure 7 + baseline compressed *p*-record) in Figure 12. We plot the SRAM consumption for different profiles in Figure 7. For example, "100K(11B)" represents 100K precords with 11-byte baseline *p*-record.*Sy*NDB consumes an average of ∼5 MB while consuming ∼10 MB of SRAM to record 1 Million uncompressed baseline *p*-records respectively. For 16-byte *p*-records, we observe the SRAM overhead to be about ∼7 MB on average.

Compression saves 50% of SRAM memory on average, and can save upto 80% depending on the traffic pattern. The SRAM consumption can be easily accommodated by latest switching ASICs [3, 4, 11] which contain SRAM greater than 100 MB. Recent studies [65] have observed high utilization only across a few switch ports during congestion events. Thus the pipeline utilization is usually much lower than its capacity. To support lower packet rates like <500 Mpps, *Sy*NDB uses about 2 MB of SRAM. The programmer can trade-off between the total capture duration and the memory budget.

**Collection Overhead**. We measure the overhead incurred at the switch to collect the *p*-records. To perform collection, the switch control-plane sets up packet generator in the data-plane packets to inject collection packets at 100 Mpps. The collection packets typically coalesce *p*-records (64 *p*-records per packet) by recirculation. Collection of 10000 compressed *p*-records requires 104 collection packets on average. We observe that it takes a total time of 245$\mu$s to evict the *p*-records. Also, it takes only 45$\mu$s to collect these packets in the data-plane using packet generator and recirculation, with the majority of the time being signalling from the control-plane to start the packet generator. We believe this timing overhead would be reduced drastically in upcoming architectures [9] which support triggering packet generation from data-plane events.

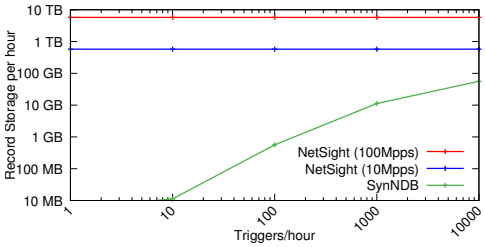


Figure 13: Storage overhead comparison with NetSight

The overall pipeline overhead incurred is about 100 Mpps and bandwidth consumption is limited to re-circulation port and the collection forwarding port (e.g. mirror port), thus not affecting regular data-plane traffic. In order to collect 1M compressed *p*-records (1ms history at 1 Bpps), it takes only about 323$\mu$s on an average. Out of this, it takes 123$\mu$s to recirculate 7800 packets to collect the compressed *p*-records, and 200$\mu$s to trigger the packet generation. *S*yNDB can resume recording (in *history* buffer) after half the *p*-records are collected in about 260$\mu$s. This means *S*yNDB can ideally support upto $\approx$ 6000 triggers/sec. Note that, *S*yNDB has a *future* buffer to store *p*-records once trigger condition is met. To support continuous recording of all future events, the minimum duration the *future* buffer needs to capture is 260$\mu$s.

With a 1ms *history* buffer, the ability to support 1000 triggers per second without any break in recording is sufficient to enable continuous monitoring. Hence, *S*yNDB can capture microbursts occurring every few milliseconds [65] as well as network incidents separated by hours [47].

We observe that the latency to receive, decompress and store the *p*-records in the collector takes few hundreds of milliseconds per switch on a single collector server. Complex queries with several join operations take several seconds or more. Query optimizations are beyond the scope of this work.

**Comparison with Other Debugging Tools.** Next, we compare the total storage overhead of *S*yNDB to that of NetSight [32]. NetSight creates a post-card by stripping the packet payload, and attaching switch ID and ingress port to the post-card. We compare the storage overhead incurred at the collector from a single switch for *S*yNDB compared to NetSight for a period of 1 hour. *S*yNDB performs collection only upon fault triggers while NetSight performs collection throughout the network operation. In Figure 13, we plot the overall storage incurred for an hour of network operation with increasing number of triggers/hr. We assume both NetSight and *S*yNDB store 16-byte post-cards/*p*-records per packet. Irrespective of the frequency of faults, NetSight collects about 500 GB and 5 TB of data per hour from a single switch at 10 Mpps and 100 Mpps packet rates, respectively. *S*yNDB on the other hand collects only 56 GB per hour for 10000 triggers/hr and 1 Bpps data-plane traffic. This means, when *S*yNDB monitors packets at the maximum rate (e.g. 1.6 Tbps), the total fraction of data exported for debugging is 0.01%.

**Switch Resource Overhead.** We evaluate the total hardware resource consumption of *SyNDB* (with configuration shown in Figure 7) compared to the baseline switch.p4 [14]. switch.p4[3] is a baseline P4 program that implements various common networking features applicable to a typical data center switch. As we implement *S*yNDB along with DPTP, we show the total resources consumed by all the components (switch.p4, DPTP and *SyNDB*) in Table 3. The majority of resources required for *S*yNDB arise from the need to store *p*-records in the data-plane. We observe that *S*yNDB consumes 33% of the stateful ALUs and 15% of the SRAM to store *p*-records and trigger conditions in the evaluation configuration. Thus, *S*yNDB can be implemented on top of switch.p4 in programmable switch ASICs available today.

Table 3: Hardware resource consumption of *S*yNDB compared to the baseline switch.p4

| Resource | switch.p4 | DPTP [5] | *S*yNDB | Combined |
|---|---|---|---|---|
| SRAM | 29.58% | 2.29% | 15.31% | 47.18% |
| Stateful ALU | 14.58% | 8.83% | 33.33% | 56.74% |
| VLIW Actions | 36.72% | 4.43% | 6.25% | 47.4% |
| TCAM | 32.29% | 0% | 1.04% | 33.33% |
| Hash Bits | 34.74% | 3.99% | 14.14% | 52.87% |
| Ternary Xbar | 43.18% | 0% | 0.63% | 43.81% |
| Exact Xbar | 29.36% | 2.34% | 12.5% | 44.2% |

## 8 Conclusion and Discussion

In this paper, we design and implement *SyNDB*, which to the best of our knowledge, the first system providing packet-level *visibility*, *retrospection* and *correlation* to tackle transient faults. *S*yNDB leverages data-plane time synchronization and data-plane storage (SRAM) to temporally store packet records which can be exported to aid in debugging upon network faults. It provides the unique ability of looking back at the trace of events before the occurrence of a network fault. Additionally, since it performs collection only upon occurrence of programmable event triggers, it exports only a small fraction of the data-plane traffic for targeted debugging. We study case-studies which uncover *S*yNDB's capabilities in finding the root cause of transient faults.

We believe that *S*yNDB's capability goes beyond debugging. A network device's configuration can be used along with network traces to create a "replay" of the network fault. This in turn, can be used to form regression test suites. Finally, it will also be interesting to develop tools that provide dashboards, query suggestions and an assistant (Similar to Dogga et al. [22]) to network operators using AI techniques to facilitate faster debugging.

**Acknowledgements.** We are grateful to our shepherd Ravi Netravali, the anonymous NSDI reviewers, Anirudh Sivaraman and Facebook research for their valuable feedback on the previous drafts of this paper. This work is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (MOE2019-T2-2-134) and the Facebook Networking Systems Award (2019).

[3] DC_BASIC_PROFILE in Intel P4 Studio 8.9.2

## References


[1] Apache Hadoop. http://hadoop.apache.org.

[2] Apache ZooKeeper. http://zookeeper.apache.org.

[3] Broadcom StrataXGS BCM56970 Tomahawk II. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[4] Broadcom StrataXGS BCM56980 Tomahawk III. https://www.broadcom.com/blog/at-a-glance-tomahawk-3-is-the-first-12-8-tb-s-chip-to-achieve-mass-production.

[5] DPTP Source Code. https://github.com/praveingk/DPTP.

[6] Graphana. https://grafana.com/.

[7] In-band Network Telemetry. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[8] Intel tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[9] Intel tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[10] LDP IGP Synchronization. https://tools.ietf.org/html/rfc5443.

[11] Mellanox Spectrum 2. https://www.mellanox.com/page/products_dyn?product_family=277&mtag=spectrum2_ic.

[12] NetworkX Library. https://networkx.github.io/.

[13] P4-16 Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[14] P4 Language Consortium. 2018. Baseline switch.p4. https://github.com/p4lang/switch/blob/master/p4src/switch.p4.

[15] PF_RING. https://www.ntop.org/products/packet-capture/pf_ring/.

[16] Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf.

[17] Pyplot – Matplotlib. https://matplotlib.org/api/pyplot_api.html.

[18] WEDGE 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[20] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[21] D.Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, 2018.

[22] P. Dogga, K. Narasimhan, A. Sivaraman, and R. Netravali. A System-Wide Debugging Assistant Powered by Natural Language Processing. In *SoCC*, 2019.

[23] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *IMC*, 2015.

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[25] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM transactions on networking*, 2(2):122–136, 1994.

[26] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *NSDI*, 2019.

[27] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*, 2017.

[28] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[29] A. Guidara, S. E. P. Hetnändez, L. M. X. R. Henríquez, H. H. Kacem, and A. H. Kacem. A Study of the Forwarding Blackhole phenomenon during Software-Defined Network Updates. In *Software Defined Systems (SDS)*, 2019.

[30] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *SIGCOMM*, 2018.

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

[33] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM '15*, 2015.

[34] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM*, 2018.

[35] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.

[36] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[37] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *APSys*, 2018.

[38] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *SOSR*, 2019.

[39] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*, 2013.

[40] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *NSDI*, 2019.

[41] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 1978.

[42] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *SIGCOMM*, 2016.

[43] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS*, 2016.

[44] Y. Li, R. Miao, M. Alizadeh, and M. Yu. DETER: Deterministic TCP replay for performance diagnosis. In *NSDI*, 2019.

[45] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang. MOZART: Temporal Coordination of Measurement. In *SOSR*, 2016.

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

[47] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A Large Scale Study of Data Center Network Reliability. In *IMC*, 2018.

[48] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A Large Scale Study of Data Center Network Reliability. In *IMC*, 2018.

[49] I. Minei and J. Lucek. *MPLS-Enabled Applications: Emerging Developments and New Technologies*. Wiley, 2008.

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

[51] 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*, 2017.

[52] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[53] J. Rothshild. High Performance At Massive Scale - Lessons Learned At Facebook, 2009.

[54] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Sneoren. Cloud Datacenter SDN Monitoring: Experiences and Challenges. In *IMC*, 2018.

[55] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of NSDI*, 2015.

[56] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Microburst in data centers: Observations, analysis, and mitigations. In *ICNP*, 2018.

[57] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Microburst in data centers: Observations, analysis, and mitigations. In *Proceedings of ICNP*, 2018.

[58] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

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

[60] P. Tammana, R. Agarwal, and M. Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *NSDI*, 2018.

[61] V.Liu et al. Subways: A case for redundant, inexpensive data center edge links. In *CoNEXT*, 2015.

[62] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM*, 2018.

[63] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *SIGCOMM*, 2018.

[64] N. Yaseen, J. Sonchack, and V. Liu. tpprof: A network traffic pattern profiler. In *NSDI*, 2020.

[65] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC*, 2017.

[66] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow Event Telemetry on Programmable Data Plane. In *SIGCOMM*, 2020.

[67] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*, 2015.

# A *SyNDB* Pseudocode

```
precordArray   : Register Buffer Array
writeIndex     : Current index to write
N              : Size of the ring buffer
POST_TRIG_SIZE : Size of buffer for post trigger
pwriteIndex    : Current index to write post trigger
triggerArray   : Temporal Trigger bit-array
triggerConditions : Bitmask configuration of
                    TriggerArray for collection
Time_Now       : Current Global Time
-----------------------------------------------------
 Packet Record Logic
if packet is normalPacket:
    if collectInProgress == False:
        Store Hash, Time_now, Time_queue,
            CustomStats in precordArray[writeIndex]
        writeIndex = (writeIndex + 1) % N
        add_to_port_group(ingress_port)
    else :
        if pwriteIndex < POST_TRIG_SIZE:
            Store Hash, Time_now, Time_queue,
            CustomStats in precordArray[pwriteIndex]
            pwriteIndex = (pwriteIndex + 1)
    if triggerHit is True:
         clone(packet)

if packet is clonedPacket:
    add_header(trigger)
    remove_header(ipv4/tcp/udp)
    trigger.time = Time_now
    trigger.id   = triggerId
    trigger.type = triggerType
    recirculate()
 Trigger Packet Logic
if packet is triggerPacket:
    if trigger.id != lastSeenId[trigger.source]:
        triggerArray |= 1 << (trigger.type - 1);
        lastseenId[trigger.source] = trigger.id;
    else:
        drop()
    if triggerArray in triggerConditions:
        collectInProgress = True
        Multicast(port_group)
 Collection Packet Logic
if packet is collectPacket:
    if collectPacket.entries < MAX_ENTRIES_PKT:
        p-record =  precordArray[readIndex]
        readIndex = (readIndex + 1) % N
        add_header(p-record)
        collectPacket.entries++
        recirculate()
    else:
        l2fwd_to_collector()
```

# B More Network Debugging scenarios

### B.0.1 Network Configuration Updates

Networks operate in a dynamic environment where operators frequently modify forwarding rules and link weights to perform tasks from fault management, traffic engineering, to planned maintenance [52]. However, dynamic network configurations are complex and error prone especially if they involve several devices. For example, updating the route for a flow(s) can lead to unexpected packet drops if the updates are not applied consistently or efficiently [36, 54]. In this case study, we use *S*yNDB to identify whether a transient error is due to a network update or localized hardware fault.

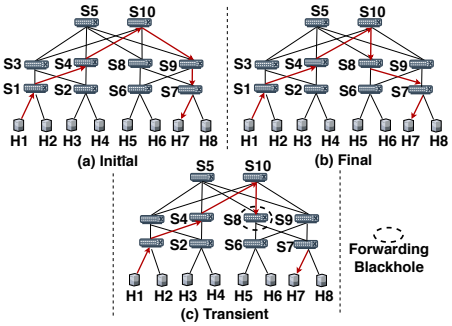



Figure 14: Network Update Scenario causing a Forwarding Blackhole at S8

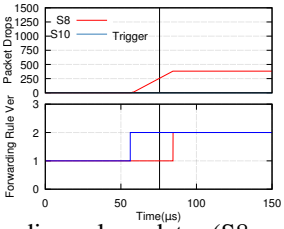



Figure 15: Forwarding rule updates (S8 and S10) leading to drops at S8

For the experiment, we add forward_rule_version to the *field_list SyNDB_scenario*. We assume that each forwarding rule indicates a version number and route based on destination MAC address as shown below.

```
table_add forward
    send_to_port ethernet_dstAddr <dstMac> =>
    output_port <num> entry_ver <num>
```

A transient forwarding blackhole occurs when an out-of-order execution of a network update gives rise to non-deterministic network behavior leading to temporary packet loss [29]. We emulate the transient blackhole using a setup shown in Figure 14. Figures 14(a) and (b) depict the initial and final state of the network after the updated route. The routing of a flow from H1 to H7 is updated by rerouting traffic from S10 to S8. However, transitioning from configuration (a) to (b) requires updates to both S10 and S8.

In this network update, a new rule to route the flow needs to be added to S8 first and then S10 needs to update the policy to route the flows from S9 to S8. If the update at S8 occurs later than the reroute at S10, a temporary forwarding blackhole will form, resulting in packet drops.

However, the packet drop at S8 due to table lookup miss could also be flagged as a parity error [66], when the context of the table miss is unknown. To check if a delayed network update is a possible cause, with *S*yNDB, we can query (Listing 2) the forwarding rule versions observed by each packets at S10 and S8 along with the number of drops observed in

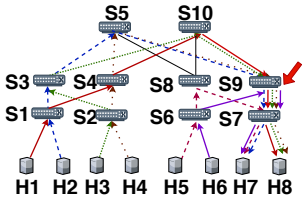



Figure 16: Congestion at S9 due to Link Load Balancing problem

Figure 15. From Figure 15, by correlating the rule version number and packet drops in time, it is clear that the packet dropped can be attributed to a transient inconsistency in rules between switches S8 and S10.

```
SELECT forwarding_rule_ver, drop_counter
    FROM packetrecords WHERE switch=8 OR switch=10;
```

Listing 2: Query for correlating network update with drops

Note that in this experiment, data collection is triggered based on an aggregating trigger defined over multiple switches. Switch S8 broadcasts the trigger *b* to other switches on detection of forwarding table miss and S10 broadcasts the trigger *c* on policy update. Trigger *b* or *c* by itself does not trigger data collection. When a switch receives both triggers (within a time window), then data collection is triggered. Such multi-switch trigger reduces both false-positives and collection overhead.

### B.0.2 Transient Load Balancing Issues

Modern data center topologies such as fat-tree provide redundant paths between a source-destination pair. ECMP [27, 33, 58] is a common load balancing policy for handling multi-path routes. However, it has a lot of inefficiencies in distributing the load evenly [27, 33]. As a result, it has been observed that a subset of core-links regularly experience congestion while there is spare capacity on other links [20].

In this scenario, we setup ECMP based load balancing. Each switch calculates the hash of the 5-tuple and redirects the flow via one out of the two links. We experiment with a variety of combinations of 5-tuple flows, and use a set of combinations which can lead to load imbalance in the network. In one such combination, S9-S7 is congested, even though spare capacity is available at S8-S7. We create multiple flows in the network originating from H1 to H6 with the destination as H7 and H8 (Figure 16). The traffic (containing faulty combination) is sent at short bursts, with an overall throughput of 1 Gbps per flow. The load imbalance happens when both the core switches (S5 and S10) direct too many flows to S9,

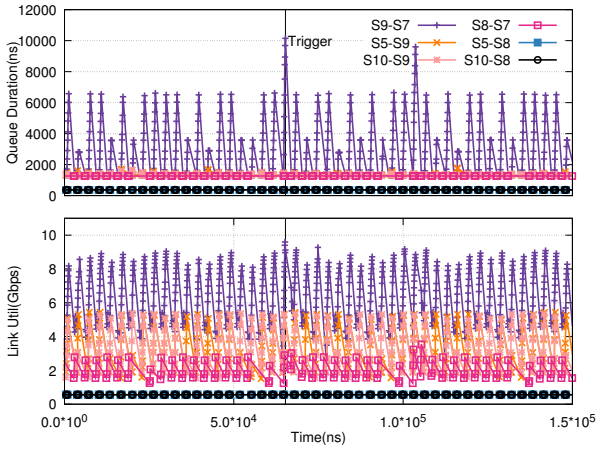



Figure 17: Link Utilization and Queuing delays observed at the core links points to load imbalance

resulting in congestion on the S9-S7 link.

With only the congestion indication, it is be difficult to determine the root cause. To determine if load imbalance is the root cause, one would have to observe the queuing duration and link utilization of various links at the same time. These network metrics are not available with both NetSight and INT. SpeedLight [63] can measure only coarse-grained link utilization (several $\mu$s). With *S*yNDB, we can plot the utilization of the links measured at the same time at packet-level granularity using the query shown in Listing 2.

```
SELECT switch1, switch2, link_utilization*8, time_queue
    FROM (SELECT switch1, switch2 FROM links
    WHERE (switch1 IN (select switch FROM switches
    WHERE type !="tor") AND switch2 IN (SELECT switch
    FROM switches WHERE type !="tor"))) AS L
    JOIN (SELECT * FROM packetrecords) AS A
    JOIN (SELECT * FROM packetrecords) AS B
    ON (A.hash = B.hash AND A.switch = L.switch1
    AND B.switch = L.switch2);
SELECT forwarding_rule_ver
    FROM packetrecords WHERE switch=10;
```

Listing 3: Query for link utilization and queue depths

The result is shown in Figure 17. We can observe that there is high link utilization at S9-S7 while link S8-S7 sees no significant utilization. Furthermore, the congestion trigger at the link S9-S7 is preceded by higher than normal link utilization in links S5-S9 and S10-S9. Thus, the load distribution from the core switches (S5 and S10) to S8 and S9 is heavily skewed, with most flows being routed via S9 during some time intervals. Based on this observation, one can infer that the root cause for the congestion at the link S9-S7 is the load imbalance cause by the load balancing scheme.