Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon

Weitao Wang, Xinyu Crystal Wu, Praveen Tammana[†], Ang Chen, T. S. Eugene Ng
Rice University [†]Indian Institute of Technology Hyderabad

Abstract

Performance monitoring and diagnosis are essential for data centers. The emergence of programmable switches has led to the development of a slew of monitoring systems, but most of them do not explicitly target posterior diagnosis. On one hand, "query-driven" monitoring systems must be pre-configured with a static query, but it is difficult to achieve high coverage because the right query for posterior diagnosis may not be known in advance. On the other hand, "blanket" monitoring systems have high coverage as they always collect telemetry data from all switches, but they collect excessive data. SpiderMon is a system that co-designs monitoring and posterior diagnosis in a closed loop to achieve low overhead and high coverage simultaneously, by leveraging "wait-for" relations to guide its operations. We evaluate SpiderMon in both Tofino hardware and BMv2 software switches and show that SpiderMon diagnoses performance problems accurately and quickly with low overhead.

1 Introduction

An efficient network monitoring and diagnosis system are essential to meeting the performance requirements of modern applications. Since production clouds have stringent SLAs, even a small network performance degradation may lead to significant application slowdown [13, 30]. Many network performance problems, such as high end-to-end latency, low throughput, and packet drops [38], can be attributed to traffic contention of some kind [4], although across scenarios, the root causes for the contention are diverse (e.g., bursty UDP traffic, ECMP load imbalance, and routing loops).

The emergence of programmable switches has led to a slew of monitoring systems being developed [12, 16, 32, 33, 39, 44, 48], but most of them do not explicitly target posterior diagnosis. For instance, "query-driven" monitoring systems [16, 32] need to be pre-configured with a static query. Since root causes for performance degradation could vary, and there may be a wide variety of reasons for performance problems, it is challenging to select the right query in advance. In principle, one could adaptively change the monitoring query based on the observed symptom; but in practice, many transient problems happen at fine timescales and their sporadic nature requires always-on monitoring. On the other hand, "blanket" monitoring systems always monitor and collect telemetry data from the switches to achieve high coverage [10, 14, 22, 26, 27]. However, this would result in excessive data that may not be needed by the diagnosis in the first place.

Therefore, having a monitoring and diagnosis system that achieves either low overhead or high coverage is not hard, but achieving both simultaneously is challenging. The key question we explore is whether it is possible to design a streamlined system that performs efficient monitoring but achieves high coverage, achieving the "best of both worlds". We present SpiderMon, a system where the monitoring and diagnosis operations are explicitly designed to work with each other in a closed loop. It enables a suitable tradeoff between accuracy and overhead when debugging network-wide performance problems. To achieve efficient and accurate monitoring, SpiderMon leverages a concept called "wait-for" [46] relations. Since many performance problems stem from in-network contention, "wait-for" relations target such behaviors in the telemetry collection in a precise manner. Moreover, such information is also exactly what is needed in diagnosis. For instance, a victim flow with high latency may have "waited for" many interfering events across multiple hops. By capturing and analyzing such relations, SpiderMon can achieve an effective diagnosis, with precise, targeted, but also high-coverage operations.

Since the symptom of "wait-for" events is usually high latency, SpiderMon uses timing information to trigger reactive telemetry collection. Precisely, SpiderMon detects performance problems when it encounters flows with excessively high queuing delay. After a problem is detected, SpiderMon uses the wait-for relations to track and collect other relevant information in the data plane across the network. For diagnosis, SpiderMon also identifies the root causes of the performance problem by summarizing the most significant wait-for relations from the collected telemetry data. It does so by jointly analyzing wait-for patterns together with other types of network knowledge (e.g., topology) and telemetry data (e.g., flow-level results). In this way, SpiderMon collects telemetry data only when the diagnosis process needs to analyze a problem, and it performs targeted collection based on what the diagnosis process would require.

To realize this idea, SpiderMon addresses three technical challenges. The first challenge is to detect performance degradation without interfering with actual packet processing. SpiderMon leverages programmable switches to record telemetry data about network traffic. It piggybacks telemetry data in packet headers and checks for performance anomalies. The second challenge is to precisely collect the relevant telemetry information across the network. Relying on wait-for relations, SpiderMon notifies relevant switches and activates telemetry data collection from these locations. Finally, SpiderMon identifies the root causes of the performance problem using the telemetry information and the knowledge of the network configuration. The wait-for relation again is critical for identifying abnormal network behaviors, and for matching those behaviors to the signatures of root causes.

Contributions. Overall, SpiderMon is a *closed-loop* system for monitoring and diagnosing performance problems in the network. We have implemented a prototype of SpiderMon, and our results show that SpiderMon can diagnose performance problems accurately and quickly with low overhead.

2 Motivation

SpiderMon focuses on network performance problems that arise due to contention, which are challenging for at least three reasons. First, network contention may occur due to many root causes, so its diagnosis requires a general mechanism. Second, the root cause can be unpredictable both spatially and temporally, requiring agile solutions that can capture transient problems. A third practical challenge is that the solution must have a sufficiently low overhead on the network. SpiderMon does not target problems that happen because of silent packet drops, packet corruptions, control plane misconfigurations, slow servers, or other causes unrelated to network contention, although it can be used in combination with other techniques for these scenarios.

2.1 Root Causes Are Diverse

To illustrate the diversity of root causes of network performance problems, consider some examples in a 3-layer Clos network as shown in Figure 1.

Micro-bursts. Recent studies [10, 22, 45] found micro-bursts—i.e. momentary surges in traffic volume—to be a common root cause for sporadic excessive delays and packet losses. Detecting and diagnosing a micro-burst requires switch queuing delays to be monitored and the main contributor to queuing delays to be identified before the micro-burst disappears.

Multiple flow contentions. A victim flow encounters multiple contentions at different switches—flow 1 (e.g., a bursty UDP flow) and flow 2 (e.g., a high-priority flow) contend with the victim flow at switch 0 and switch 6, respectively (Figure 1(a)). The end-to-end latency for the victim flow becomes very high. For detection, we need to monitor per-flow latency; for diagnosis, information about all contending flows is needed to identify the root causes.

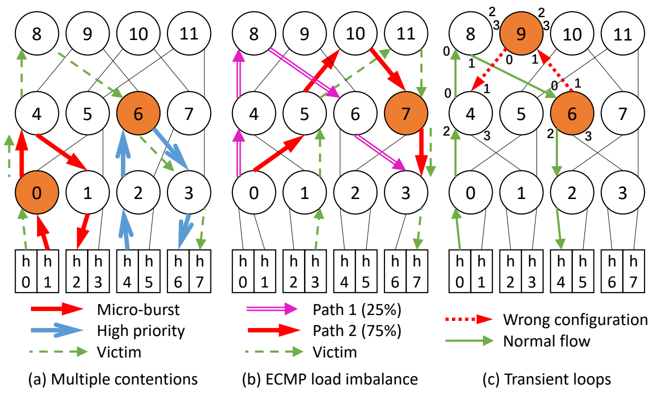


Figure 1: Several performance degradation problems

ECMP load imbalance. Due to the skewed nature of flow distributions or imperfect hash mechanisms, ECMP load imbalance is a common problem in data centers [3]. Consider the network in Figure 1(b), where all links are 40Gbps. Switch 0 assigns 25% of the total traffic (32Gbps) to path 1 and 75% to path 2. The victim flow contends with the flows on path 2, which leads to high congestion at switch 7. This could be avoided if switch 0 assigns the traffic for the two paths equally. The root cause for this problem is the imbalanced assignment at switch 0, but the performance degradation occurs at switch 7, which is 3 hops away from switch 0. Once high latency is detected at switch 7, the previous hops' information of the flows involved in the congestion is required for debugging.

Transient/persistent loops. During network updates, the configurations of different switches may not be synchronized. Some switches may fail to execute the reconfiguration commands silently. Under those circumstances, a forwarding loop may form [28]. An example is shown in Figure 1(c), where switches 6 and 9 are wrongly configured, which causes some flows to be stuck in a loop, leading to congestion and packet drops. The incompatible switch configurations should be blamed for the loop in the network. However, to identify the switches that need to be reconfigured, information from all the switches along the loop, namely, switches 6, 9, 4, and 8, needs to be collected for analysis.

2.2 Root Causes Are Unpredictable

There are three key features that make network performance problems challenging to detect or diagnose.

Sporadic. Performance degradation is usually sporadic, occurring occasionally at different places and at an unpredictable time [1]. Any flow may be affected, so detection algorithms need to monitor every flow all the time.

Network-wide. The root causes may be network-wide, e.g., contention at different hops. The interfering flows may even have normal performance [38], despite the fact that they cause performance degradation to other flows. Thus root cause diagnosis requires network-wide monitoring.

Transient. Traffic contentions sometimes are transient and disappear quickly [21]. For instance, transient loops may only form for a short time during network updates, but the performance problem introduced by packet drops may need a much longer time to fully recover. This feature requires the debugging system to maintain fine-grained information about recent events, in case the problems disappear quickly but happen in the network frequently.

2.3 Existing Solutions Fall Short

Existing solutions all fall short in monitoring and diagnosing network performance problems due to the above challenges.

Host-based solutions. Solutions like Trumpet [31] and Dapper [14] rely on end hosts to store telemetry data for diagnosis. But they all use inference algorithms to reconstruct what may have happened in the network from the collected data, which may not be accurate. Instead, SpiderMon collects data from the switches to achieve a better in-network view for diagnosis.

In-network solutions. Some existing solutions also collect telemetry data from the switches. **(i) Blanket telemetry systems** like NetSight [17] and PINT [8] collect information network-wide indiscriminately, even on network nodes unrelated to the problem. Those systems usually have high overheads, and much of the collected data is unnecessary for diagnosis. **(ii) Query-based systems** deploy queries into switches for data collection, such as Sonata [16], Marple [32], FlowRadar [26], and NetSeer [47]. They require that the operators know the nature and location of the problems, but problems could arise from sporadic congestion at random locations. Although in principle, queries can be changed based on the monitoring results, this happens at coarse timescales and cannot capture transient problems. SpiderMon can cover problems that cannot be succinctly defined using static queries and only capture events relevant to the problems.

3 SpiderMon Design

SpiderMon monitors and diagnoses performance problems caused by in-network contention in three steps: 1) SpiderMon encodes every packet's accumulated latency in header fields, and triggers telemetry collection once excessive latency is detected (§3.1); 2) the switch that detects high latency initiates "spider" packets and rapidly delivers them to relevant switches using the wait-for relations; relevant switches receiving spider packets report their telemetry data (§3.2); 3) the root cause analyzer constructs wait-for relations from the evidence for root cause analysis (§3.3).

3.1 Problem Monitoring

Goal: Detect excessive cumulative queuing delays. Rather than wait for the occurrence of harmful events (e.g., packet loss, TCP congestion window back-off), SpiderMon detects the performance problems based on a much earlier sign—abnormal cumulative queuing delays experienced by packets. It reacts quickly to performance degradation.

Design: 1) Use cumulative latency for detection. Instead of storing per-hop latency information in the header, SpiderMon uses cumulative latency to guarantee that the header length stays constant regardless of hop count. The cumulative latency L is updated at every hop based on the current queuing delay and the cumulative latency experienced by the packet so far, $L = L + queuing\_delay$. Every switch on the path checks whether the cumulative delay exceeds the latency threshold. To further reduce overhead, SpiderMon can compress the additional fields to less than 2 bytes by extracting the most significant bits (more in §C.2). **2) Assign different latency thresholds for different traffic types.** Given that the tolerable latency varies for different applications, SpiderMon allows network operators to customize the latency thresholds for different applications. **3) Detect problems and trigger telemetry in the switch data plane.** Unlike some monitoring systems using a central controller to monitor network problems [6, 31, 48], SpiderMon triggers fast reactions in the data plane. The communication delay within the data plane (tens of ns) is much lower than that between the data plane and the control plane (hundreds of μs). **4) Monitor every packet at every hop for target flows.** Compared to sampling-based detection [2, 34], SpiderMon achieves full coverage without losing any important information. Also, rather than detecting problems at the end hosts [9, 24], SpiderMon detects performance problems inside the network and reacts more quickly to the problem. **5) Be transparent to end-hosts.** The latency threshold and cumulative latency are added at the edge switches when packets enter the network and removed when packets leave the network. Hosts remain unchanged.

Consider Figure 1(a) as an example. The victim flow suffers from queuing delay at switches 0 and 6, but the cumulative latency exceeds the threshold only at switch 6. Thus the problem is detected at switch 6, and switch 6 triggers the telemetry collection procedure.

3.2 Telemetry Collection

Goal: Only collect evidence relevant to root cause analysis. SpiderMon maintains a small amount of telemetry information as evidence on the switches to facilitate subsequent diagnosis; this information is not collected from the switches unless needed. First, to minimize the amount of telemetry data collected to the analyzer while maintaining the diagnosis accuracy, SpiderMon only targets switches relevant to the observed performance problem as detailed in §3.2.1. Second, SpiderMon collects the relevant telemetry data within a short time such that each switch only needs to keep a small amount of historical telemetry data as detailed in §3.2.2.

3.2.1 Relevant Switches Notification

#1: Only collect data after problem detection. Compared to other systems which collect data to a centralized collector all the time [6, 16, 32, 48], SpiderMon uses a default-off collection strategy to minimize overhead. After the problem is detected, a special 'spider" packet is generated to notify relevant switches and start the telemetry collection on those switches. A "spider" packet carries: 1) an event_ID, which concatenates the switch ID and the event index to uniquely identify the problem, and 2) the 5-tuple of the victim flow. Spider packets are generated by mirroring the packet that triggered the diagnosis and recirculating it for transmission, while the original packet transmits as normal. To prevent possible packet drops during the transmission, all "spider" packets are prioritized in the network for lossless transfer.

#2: Only collect data from relevant switches. Instead of collecting telemetry from all switches, SpiderMon identifies the switches that are relevant to the detected problem by tracking packet-level provenance; it only retrieves data from these switches to minimize overhead. Packet-level provenance is modeled as $G := (V, E)$ for a detected event and the corresponding causality relations. G is a directed acyclic graph, where each node v represents an event, and each directed edge $e = (v_1 \rightarrow v_2)$ represents that v_1 leads to the event v_2. For latency problems in a network, all wait-for contentions in the switch queues are considered events in the provenance data. Since events at the upstream switches affect the events at the downstream switch, such upstream events are also incorporated into the provenance model. In this way, we can construct a provenance graph for a performance problem. By analyzing the locations of events, SpiderMon can select switches relevant to the specific problem.

#3: Track the provenance graph in the data plane. Unlike the central controller that Trumpet uses to inform relevant nodes, SpiderMon performs this procedure entirely in the data plane to reduce the latency of notifying relevant switches. It only requires switches to maintain telemetry data for a shorter time for the recent interval without losing necessary data. To achieve this, SpiderMon repeats the following two steps on each switch that receives the "spider" packet: 1) sends a traceback "spider" packet along the historical path of the victim flow, where the path is obtained using a bloom filter, 2) sends branch-search "spider" packets to ports that sent traffic and contended with the victim flow, where the ports are identified by a per-port traffic meter. Switches drop spider packets with duplicate IDs to avoid unnecessary processing (§C.1).

Timeout bloom filter. SpiderMon uses a timeout bloom filter (TBF) to track the victim flow's historical path. Regular bloom filters allow the insertion and the membership test of a flow ID. However, they can only support insertions, and the false positive rates increase with the number of inserted flows. A rotating bloom filter, on the other hand, can instantiate one instance per epoch, so that older data can be safely discarded; however, this is very coarse-grained as it only supports per-epoch deletion. To address those problems, SpiderMon adds a timeout feature to remove unneeded data from the bloom filter; this method provides a "sliding window" of historical flow information. For a switch with N ports, each egress pipeline maintains a bloom filter group with M rows and N cells per row, and each column represents a bloom filter for the corresponding port. The TBF replaces the bit record with a short timestamp, which can be used to recognize the outdated records when querying the TBF. The details about maintaining and querying the TBF are shown in Algorithm 1, Figure 2(a) and Figure 2(b). The memory footprint of TBF can be reduced by shrinking the size of stored timestamps (§C.2).

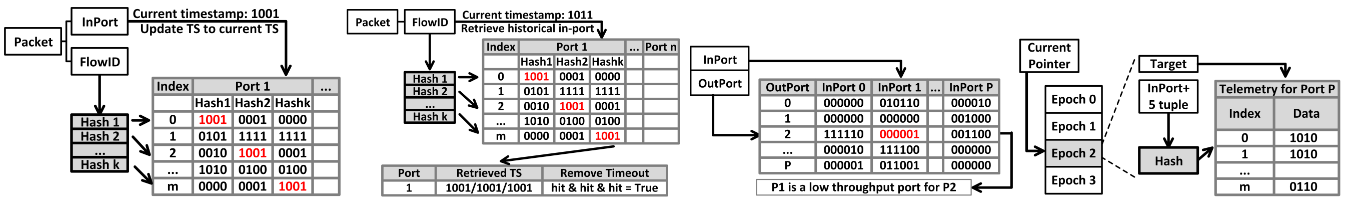
Algorithm 1: Timeout bloom filter data structure

```
Input: $B$: Timeout bloom filter, $inPort$: Incoming port index,
       $5-tuple$: 5-tuple, $curr\_TS$: Current timestamp, $epoch$:
       Timeout epoch
1  Function updateBF ($inPort$, $5-tuple$):
2      $hashValues = HASH(5-tuple)$
3      for $hashValue \in hashValues$ do
4          $B[hashValue][inPort] \leftarrow curr\_TS$
5      return
6  Function checkBF ($inPort$, $5-tuple$):
7      $hashValues \leftarrow HASH(5-tuple)$
8      for $hashValue$ in $hashValues$ do
9          $stamps \leftarrow B[hashValue][inPort]$
10         if $curr\_TS - stamp > epoch$ then
11             return False
12     return True
```

Most recent, per-port traffic meter. SpiderMon identifies the relevant ports that contribute to high latency. To distinguish an ingress port with low throughput, SpiderMon maintains a traffic meter for each ingress port's traffic volume in the most recent time. Normal traffic meters in the switch are reset to 0 periodically, leading to information loss. Therefore, SpiderMon divides the time window into several small windows and associates those meters' values to realize a sliding window of the traffic amount within the most recent time window (details in §B).

#4: Reduce the collected telemetry data by pruning the provenance graph. Some causality relations are more important than others. SpiderMon leverages this to reduce overhead without sacrificing diagnosis accuracy. Specifically, if the traffic volume from some ingress ports is significantly lower than others, it is excluded from the possible root causes; so switches that contribute minimally to the problems are ignored. SpiderMon provides a tunable threshold and only sends spiders to the ports with high traffic rates. The robustness of this threshold is shown in §4.3.

To illustrate the relevant switch notification procedure, we use Figure 3 as an example of a multiple contention scenario. The high latency is detected at switch 0. Then the traceback "spider" is sent to the reverse path of the victim flow, namely, switches 1, 2, and 3. At the same time, the branch-search "spider" is sent to switches 4 and 6, with switch 5 being ignored due to the small traffic volume. If the traffic from switch 4 came from two other switches has sufficient volume, the branch-search "spider" packets will also be sent to those ports.



Index	Port 1			...
	Hash1	Hash2	Hashk	
0	1001	0001	0000	
1	0101	1111	1111	
2	0010	1001	0001	
...	1010	0100	0100	
m	0000	0001	1001	

Index	Port 1			...	Port n
	Hash1	Hash2	Hashk		
0	1001	0001	0000		
1	0101	1111	1111		
2	0010	1001	0001		
...	1010	0100	0100		
m	0000	0001	1001		

Port	Retrieved TS	Remove Timeout
1	1001/1001/1001	hit & hit & hit = True

OutPort	InPort 0	InPort 1	...	InPort P
0	000000	010110		000010
1	000000	000000		001000
2	111110	000001		001100
...	000010	111100		000000
P	000001	011001		000000

Telemetry for Port P	
Index	Data
0	1010
1	1010
...	
m	0110

(a) Update timeout bloom filter (b) Test timeout bloom filter (c) Per-port Traffic Meter (d) Telemetry data structure

Figure 2: SpiderMon data structures

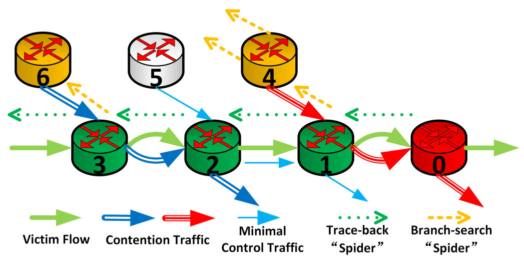


Figure 3: "Spider" packets propagation

3.2.2 Telemetry Data Collection

#1: Collect per-epoch per-flow information. Per-packet telemetry incurs a very high overhead and usually is unnecessarily fine-grained for diagnosis. SpiderMon records the history with a per-epoch flow-level log, which is stored in the switches' egress pipeline and each egress port has its own log. Dividing into epochs this way allows SpiderMon to observe changes among epochs. Each switch keeps a fixed number of epochs on the data plane and keeps the most recent ones in a circular buffer. When reporting the telemetry data, information of all epochs will be sent to the analyzer.

SpiderMon collects 36 bytes of data per flow, including the flow's 5-tuple, sequence number range, total traffic volume, total packet count, total queuing depth, the priority of the flow, and the incoming port. The network operators can add extra flow-level information in the telemetry data structure for diagnosing other network problems. The total amount of telemetry data varies with the flow arrival rate. To update, SpiderMon first identifies the right telemetry table based on the outgoing port, then hashes the flow ID to assign a slot in the telemetry data structure for that flow. By doing a bit-wise XOR between the packet's 5-tuple and the 5-tuple in the slot, we can determine whether this packet belongs to the existing flow by checking whether the result is 0. If so, this packet will be used to update this entry; otherwise, it will be considered as a new flow and replace the old one. The old entry will be packed and sent to the control plane for storage.

SpiderMon must maintain telemetry data for a minimum duration to ensure that the needed evidence for diagnosis is available, and this duration can be estimated as follows. Denote the threshold for detecting an unacceptable cumulative delay as T and the maximum round-trip propagation delay across the network as RTT. The time it takes to propagate spider packets from the initiator to relevant switches—recall that spider packets have high transmission priority and do not wait for normal traffic—is half RTT in the worst case. Since the problem is detected after accumulated delay exceeds T, the time duration a switch must maintain telemetry data to diagnose this problem is, therefore, $T + \frac{RTT}{2}$. The common RTT and T in the data center network is 0.5-2 ms and 10-15 ms respectively [15], so it would be more than enough for SpiderMon to preserve history for 20 ms.

#2: Provide synchronization among switches using flows' sequence number. The host-based solution cannot replay accurately, one of the reasons is the various network delay for packets, namely, the order of packets is not preserved at switches. SpiderMon has a similar problem when choosing the most relevant epoch on different switches for analysis. The correct epoch for the switch that triggered the problem is no doubt the most recent epoch, but for other switches on the historical path, the delay from the queuing and propagation may have caused the most relevant epoch to become a historical epoch. To solve this, SpiderMon keeps track of the [min_seq, max_seq] for each flow, and uses the victim flow's sequence numbers to find the correct epoch with the maximum overlap with this sequence number interval for the relevant switches.

#3: Trigger telemetry packet generation in the data plane. Unlike NetSight that uses mirroring for collection, SpiderMon uses the packet generator to report the per-epoch per-flow log to the root causes analyzer. The packet generator can be directly triggered in the data plane to minimize latency. Compared to retrieving the data via the switch control plane as in several previous works [27], SpiderMon is much more agile because it bypasses the low bandwidth and high latency connection between the data plane and the control plane.

The telemetry packet header contains 1) an event ID for identifying the performance problem; 2) a switch ID; 3) a partition index of the telemetry data; 4) a part of the telemetry data. The telemetry packets are generated by the packet generator on a programmable switch. The generated packets only have Ethernet and IPv4 headers without the payload for bandwidth savings. The IPv4 destination address of telemetry packets is set to the root cause analyzer so that the network will forward the packets to the analyzer. There is a maximum amount of telemetry data that can be inserted into a single packet, which is around 200 bytes due to the limitation of the PHV fields for the programmable switches. So the packet generator will generate a fixed number of telemetry packets according to the size of the telemetry tables.

Algorithm 2: Replay the queue condition

```
Input: $T$: the epoch period; $N$: flow packets count, $s$: time for the
       last packet
Output: $time\_list$: time list for the packets
1 for $t \in N$ do
2     $t \leftarrow s + \frac{T}{N}$
3     $time\_list \leftarrow time\_list + t$
4 return $time\_list$
```

#4: Only collect the telemetry data from relevant ports to reduce overhead. When a switch receives a spider packet from a certain port, usually only the telemetry data for that port will be reported to the analyzer, which reduces the amount of data collected.

3.3 Root Cause Analysis

SpiderMon develops a diagnosis strategy that is generalizable to diverse root causes with high precision and recall.

Efficiently localizing network problems and accurately identifying the root causes can be difficult, especially when the network conditions are dynamic and complex. Firstly, a good diagnosis algorithm needs to understand flow interactions and find the corresponding flows that occupied the queue. Secondly, once the problem has been localized, the diagnostic algorithm needs to further identify each problematic scenario with one or more root causes, such as micro-bursts or transient loops. However, most existing diagnostic algorithms do not have a clear boundary between those two steps. The identifications of the root causes are based on the matching of the problem patterns and observations, leading to slow diagnosis time and reduced diagnosis accuracy.

SpiderMon addresses these challenges with a two-step diagnostic algorithm: 1) efficiently analyze the queuing information at both flow level and aggregate level to recall all the problematic flows using wait-for graphs (WFG), as discussed in §3.3.1; 2) apply signature matching between the problematic flows and the root cause type, as described in §3.3.2.

3.3.1 Find the Possible Root Causes

To find all possible root causes with a high recall rate, SpiderMon uses WFG at both flow-level and aggregate-level to identify the abnormal behaviors from the telemetry data.

Wait-for relation. *If a packet from flow A enters a queue where the packets from flow B already exist in the queue, then flow A waits for flow B at this queue.*

Flow-level wait-for graph (WFG). *Each vertex represents a flow, and a directed edge from vertex A to vertex B represents that flow A waits for flow B.*

Wait-for weight. *Each directed edge's weight is calculated as follows: for a packet p_k from flow A, if x_k packets from flow B exist in the queue when p_k enters, then flow B blocks flow A with weight x_k. For all n packets from flow A during a certain period, the average weight $\frac{1}{n} \cdot \sum_{k \in [1,n]} x_k$ is the wait-for weight for the directed edge from vertex A to vertex B.*

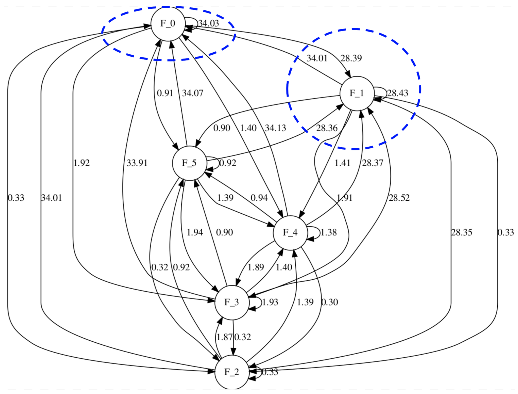


Figure 4: Identify the main contributors in WFG

Algorithm 3: Wait-for Graph Construction

```
Input: $Seq$: A sequence of packet, $level$: flow or port
Output: $G$: Wait-for graph for the given sequence
1  for $i \in [0, Seq.length]$ do
2      if $level$=$flow$ then
3          $Seq[i].vertex \leftarrow Seq[i].flow$
4      else if $level$=$port$ then
5          $Seq[i].vertex \leftarrow Seq[i].port$
6      if $Seq[i].vertex \notin G$ then
7          $G.AddVertex(Seq[i].vertex)$
8  for $i \in [0, Seq.length]$ do
9      for $j \in [0, pkt.qdepth]$ do
10         $edge \leftarrow (Seq[i].vertex \Rightarrow Seq[i-j].vertex)$
11         $G.AddEdgeWeight(edge, 1)$
12 return $G$
```

Aggregated wait-for graph. *SpiderMon also aggregates the flow according to the source IP, incoming port, or other keys to construct aggregated-level WFGs to find root causes other than flows' misbehavior. One typical example used in SpiderMon is the port-level WFG.*

After receiving all the telemetry data from the switches, SpiderMon uses the gap-based sampling strategy [25] to replay the queuing condition on the switch (Algorithm 2). The actual sequence of the packets is not important since we only need the generated wait-for graph to be similar.

To find the main contributors for the queuing, we rely on the wait-for graphs to show the provenance relations between contending flows. For each queue, SpiderMon will construct flow-level WFGs and port-level WFGs as in Algorithm 3, which will be used to determine the main contributors. Basically, to identify the main contributors of the queue is to divide the flows in the queue into victims (suffer from queuing) and main contributors (contribute to queuing) and maximize the wait-for relations between those two groups. SpiderMon is able to show that this division can be easily derived by the following Theorem 1, and identify the main contributors as in Algorithm 4. We prove Theorem 1 in Appendix §A.

Degree of the vertex. *Sum of all incoming edge weights subtracts the outgoing edge weights.*

Algorithm 4: FindContributor

```
Input: $G$: Wait-for graph for the given sequence
Output: $ctrs$: A set of main contributors
for $X \in G$ do
    $D(X) = \sum_{e \in \{<i,j>|j=A\}}^{e} w_e - \sum_{e \in \{<i,j>|i=A\}}^{e} w_e$
    if $D(X)>0$ then
        $ctrs \leftarrow ctrs + X$
return $ctrs$
```

Theorem 1. *The wait-for relation between two groups, divided by one cut, is maximum, if and only if one group only contains positive degree vertices while the other contains only negative degree vertices.*

Figure 4 is an example scenario of micro-burst with flows 0 and 1 as the burst flows, and both of them have been identified by the algorithm as the main contributors.

3.3.2 Precisely Identify Root Causes

To precisely identify the reason behind the main contributors determined in the first step, SpiderMon relies on signature matching to recognize different root causes. We give four signatures for four common root causes in Algorithm 5, using both telemetry and network configuration information. The signatures can be extended if more root causes are added. For better illustration, we consider the scenarios in Figure 1 and show the signatures in Figure 5. A detailed signature definition can be found at §G.

Micro-bursts. SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flow has many wait-for edges with large weights pointing to itself due to a large amount of traffic during the problematic time.

Different priorities. For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b).

ECMP load imbalance. For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is largely imbalanced as in Figure 5(c).

Transient/persistent loops. For the latency problem caused by transient or persistent loops as shown in Figure 1(c), SpiderMon searches the port-level contributors along the contributor traffic's path. If the same port is observed twice during the search procedure, all those ports have a high possibility to form a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop.

Algorithm 5: Root Causes Diagnostic Algorithm

```
Input: $f\_WFG$: flow-level WFG, $p\_WFG$: port-level WFG, $T$:
       Telemetry information, $K$: Network topology and
       configuration
/* Diagnose flow-level problems                              */
for $sw \in$ Switches on victim's path do
    $f\_CTR_{sw} \leftarrow FindContributor(f\_WFG_{sw})$
    for $f \in f\_CTR_{sw}$ do
        // Is micro-burst?
        check flow $f$ throughput
        // Is priority problem?
        check flow $f$ priority
        // Is routed by ECMP rules?
        check aggregated throughput for ECMP switches
/* Diagnose port-level problems                              */
for $sw \in$ Switches on victim flow's path do
    $p \leftarrow$ victim flow's outgoing port
    $CheckPort(p, \{\})$
/* Recursive function for port-level                         */
Function CheckPort($p$, $p\_set$):
    // Does routing contain loop?
    check whether there is a loop
    // Search dominant port contributors
    $p\_CTR_{sw} \leftarrow FindContributor(p\_WFG_{sw})$
    for $p' \in p\_CTR_{sw}$ do
        // Check the related port
        $src\_p \leftarrow$ the port connect to port $p'$
        $CheckPort(src\_p, p\_set + p)$
```

4 Evaluation

Next, we evaluate SpiderMon along several dimensions: diagnosis effectiveness, overheads, and robustness.

Setup. Our hardware testbed deploys SpiderMon to a Barefoot Tofino switch, written in 1147 lines of P4-Tofino code, to evaluate the switch-level performance. The switch is logically partitioned to emulate a topology with multiple logical switches; logical links are emulated by port-to-port connections using direct attach cables. The switch is also physically connected to eight servers through 25 Gbps links. The switch has $32\times$ 100Gbps ports, and each can be configured as four 25Gbps ports with a breakout cable; each server has two six-core 3.4GHz CPUs, 128GB RAM, and one 25Gbps NIC. In addition, we have set up a simulation environment that uses the `BMv2` software switches in the NS3 simulator with 945 lines of P4 code running on CloudLab servers, evaluating the network-level performance. Each server has an eight-core 2.0GHz CPU and 32GB RAM. A K=4 standard fat-tree topology with 20 switches and 32 hosts is simulated with 1 Gbps link bandwidth. We also implement the root causes analyzer with 843 lines of Python code.

Workloads. We simulate empirical workloads from production networks for our evaluation. The flow size distribution is taken from three different traces: web search [5], cache [35], and Hadoop [35]. The arrival time of different flows is based on a Poisson process and the flow arrival rate is varied to obtain different load utilizations in the network. The source and destination for each flow are chosen uniformly at random.

All flows are TCP.

Baseline systems. We compare SpiderMon against five baseline solutions. 1) **Trumpet** [31]: a trigger-based reactive host system. When it detects a problem requiring network-wide information on one host, the controller will collect data from related servers upon a trigger. This incurs a latency of at least an RTT. 2) **NetSight** [17]: an in-network system that proactively collects 'postcards' for each packet from the switches. 3) **Marple** [32]: a query-based in-network system, which is deployed to all switches using monitoring queries that a) detect high latency, b) query packet counts, and c) perform 'EWMA over latencies'. 4) **Pathdump** [37] and **SwitchPointer** [38]: two proactive, network+host solutions. Pathdump tracks paths and performs diagnosis on end-hosts, and SwitchPointer further tracks packet epochs in the network.

4.1 Diagnosis Effectiveness

We evaluate the diagnostic effectiveness of SpiderMon using multiple scenarios.

1. Micro-bursts are created by injecting 5 short-lived (10-100 μs) UDP flows from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). The throughput of micro-burst flows is set to 90%$\times$line-rate. **Diagnosis:** Fig. 5(a) shows the combined wait-for graph at two switch ports generated by SpiderMon, which shows that the two micro-burst flows E and H dominate the queues and are the only two main contributors with positive degrees. The other 3 UDP flows are not included in the WFG since they end before the victim flow starts or start later than the 2 contending UDP flows.

2. Priority contentions inject 5 high-priority TCP flows with priority queuing from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). **Diagnosis:** As Figure 5(b) shows, flow C and D are the main contributors to the congestion with higher priority and larger degrees. Other priority flows have no interference with the victim flow so the WFG excludes them.

3. ECMP imbalance scenarios randomly pick a switch (except core switches) and split traffic to two uplink ports with 4:1 imbalanced load. The ECMP group imbalanced lasted for hundreds of microseconds. **Diagnosis:** When we find the main contributors to the queuing, SpiderMon will check whether they are routed by ECMP policy. In Figure 5(c), both main contributors (flow C and D) are routed by ECMP rules on switch 0, so SpiderMon uses the telemetry information for switch 0 and computes the number of flows and traffic amount sent to each ECMP port. If the number of flows or traffic amount within that epoch is largely imbalanced, then there is an issue with the ECMP rules or hash functions.

4. Loops create a 4-hop routing loop with 2 aggregation switches and 2 core switches as in Figure 1(c). The routing loop only affects a small group of flows and the problem only lasts for 100 μs. **Diagnosis:** Port-level WFGs identify a loop as the root cause: the victim flow is reported on switch 8 port 1 so that the WFG leads us to the main contributor, port 0. Since SW8-P0 receives traffic from SW4-P0, we further construct a WFG for SW4-P0 and determine another main contributor. With this recursive searching procedure, SpiderMon finds that the port-level contributors form a loop and the traffic belongs to the same group of flows.

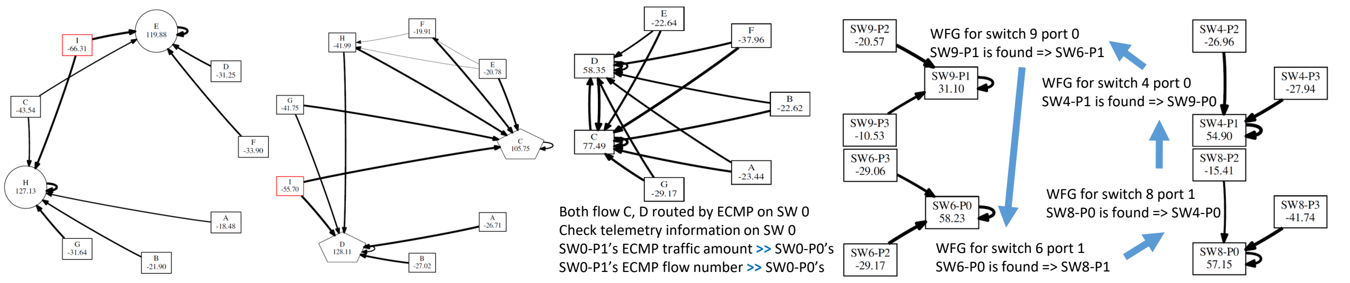
5. Complex problem diagnosis. Next, we test a diagnostic scenario with multiple problems. In Figure 6, the victim flow contends with a micro-burst flow at switch 1, a high priority flow at switch 7, and high-volume traffic caused by ECMP imbalance at switch 5. First, SpiderMon constructs the WFG with the collected information for the problem and identifies 5 flows (flow C, E, F, J, and L) with positive degrees. Next, SpiderMon checks the property of each such flow and identifies flow C as a micro-burst flow without any congestion control, while flow J is a flow with higher priority than any other flows crossing those switches. Then it checks the amount of the transmitted traffic in the same epoch and identifies flows E and F to be related to an ECMP imbalance. However, flow L is removed from the root causes; it is a normal TCP flow since its degree is small and there is no further evidence from the telemetry information to show that this flow is problematic.

6. Sporadic & transient problem diagnosis. We also evaluate multiple diagnostic situations with sporadic and transient problems. The traffic workloads are generated from random sources and destinations, and the problems could happen at different locations in the network randomly with short-lived root causes. Take the micro-burst experiment as an example. A high throughput UDP flow is introduced between a random source and destination at a random time, lasting for 100 μs. The experimental results shown in Section 4.2 are generated with sporadic problems for each scenario.

4.2 Comparison with Baseline Systems

Precision and recall. We first show the precision and recall rate for different solutions, by tuning the parameters of each system so that it can achieve the best performance for each scenario. Those include the maximum tolerable link load imbalance ratio, link utilization, per-flow throughput, and so on. Details about each scenario's parameters are in §F. Here we show the results for web trace only, the results for cache and Hadoop traces are included in §E.2. For the web trace, 30% of the flows are 1–30MB, so that multiple large flows can be concurrently active from/to one switch port.

As shown in Figure 7, Trumpet cannot achieve both high recall and accuracy at the same time for the transient congestion since it can only infer the in-network condition based on the calculated link utilization and end-to-end delay. Due to the different network delays and packet loss, the evidence for the transient problems may be inaccurate and unreliable on the host. Trumpet also fails to diagnose the ECMP imbalance problem because it does not have path information for every flow to identify the traffic split at the ECMP switches. Trumpet also fails to diagnose the loop problem because packets involved in loops do not reach the hosts, leaving no evidence for Trumpet to find out the root cause.


Both flow C, D routed by ECMP on SW 0
Check telemetry information on SW 0
SW0-P1's ECMP traffic amount >> SW0-P0's
SW0-P1's ECMP flow number >> SW0-P0's
WFG for switch 9 port 0
SW9-P1 is found => SW6-P1
WFG for switch 4 port 0
SW4-P1 is found => SW9-P0
WFG for switch 8 port 1
SW8-P0 is found => SW4-P0
WFG for switch 6 port 1
SW6-P0 is found => SW8-P1

(a) Micro-burst Contention (b) Priority Contention (c) ECMP Load Imbalance (d) Loop

Figure 5: Example wait-for graphs of several root causes. Each box (TCP flow/port), circle (UDP flow), and pentagon (High priority flows) represent one flow or port, and the port name is described according to Figure 1(c). Bolder edges represent heavier wait-for relations, edges with small weights are tailored. The number under the flow/port name shows the node degree, and positive degrees will be identified as main contributors.

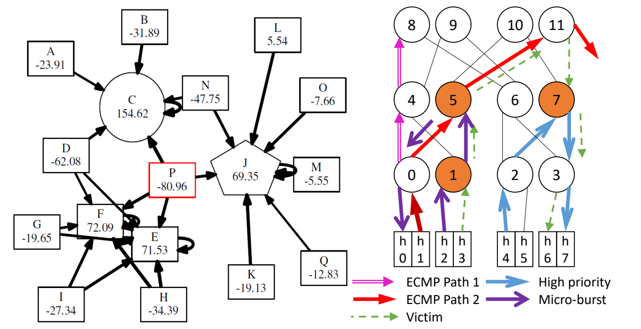
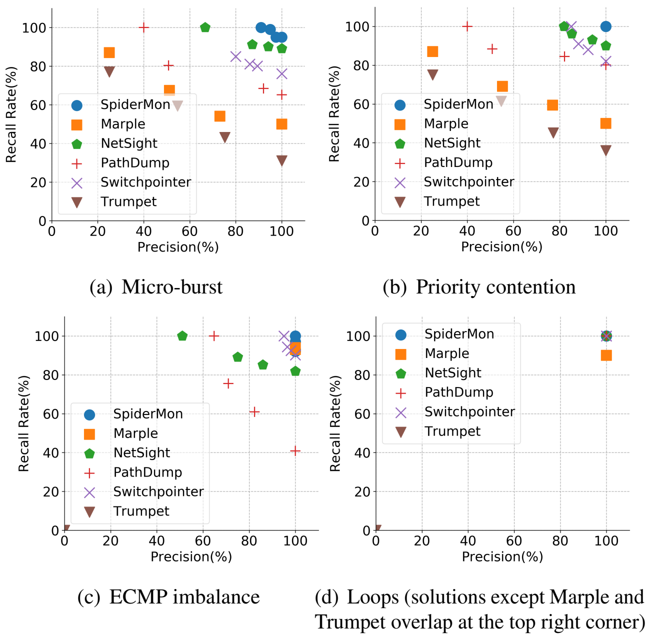


Figure 6: The WFG for victim flow P, with a micro-burst, a priority-related contention, and an ECMP imbalance at different hops.

Marple falls short in diagnosing transient contention like micro-bursts. This is because Marple enables queries only when needed, so it collects data reactively, which incurs an additional latency. The per-hop queuing information is only collected when the accumulated queuing latency exceeds the threshold. This control loop delay leads to information loss for transient problems—when the system begins collecting data from a switch near the destination, the transient bursty flow at a previous hop may have already ended. Only Marple and Trumpet are reactive systems in our evaluation.

PathDump and SwitchPointer both achieve relatively good performance. PathDump carries path information along with the packets, and SwitchPointer upgrades PathDump with switch data that records the flows that travel the same switch in the same epoch, which outperforms PathDump. However, both of them failed to identify transient problems since they lack queuing information—they instead recompute link utilization using packets received at end hosts. If a large amount of packets are dropped in the network due to congestion loss or TTL expiration, it would be very hard to reconstruct the transient network condition. Another interesting fact is that both solutions add extra in-network mechanisms (path tracking [37]) to detect the routing loop, so they both achieve great performance in detecting and diagnosing loops.

NetSight achieves the second-best performance since it



(a) Micro-burst (b) Priority contention

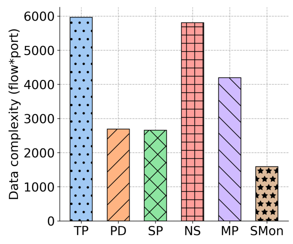
(c) ECMP imbalance (d) Loops (solutions except Marple and Trumpet overlap at the top right corner)

Figure 7: Diagnostic effectiveness for different solutions

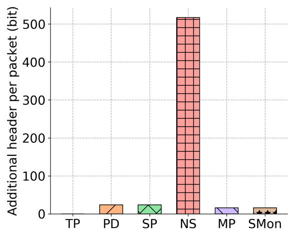
collects per-packet postcards. One drawback is that to keep overhead down, NetSight omits important data like packet priority or precise timestamps. Instead, it uses topology information to place the postcards in order. However, information that describes how flows interact cannot be obtained, which is essential for diagnosing transient problems.

SpiderMon is able to achieve nearly 100% recall and precision for all tested scenarios. The reason is that SpiderMon collects accurate packet-level information within a time interval. For micro-burst and priority flow contention, each flow's throughput within the same epoch where congestion happens will be recorded and reported in the telemetry data; for the ECMP imbalance problem, the flow ID and output port will be recorded, so that the ECMP imbalance ratio can be calculated; for the loop problem, the loop can be easily detected in the procedure of WFG construction.

To summarize, host-based solutions (Trumpet, PathDump

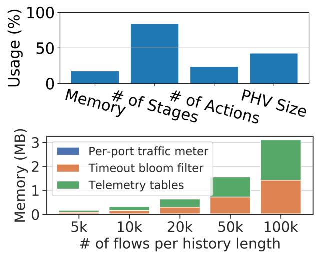


(a) Diagnostic data complexity

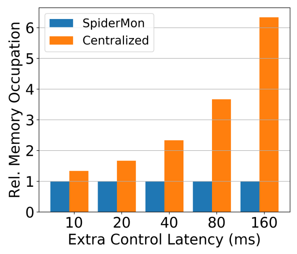


(b) Additional bandwidth overhead

Figure 8: Diagnostic data complexity for different systems; the additional per-packet header shows the bandwidth overheads for Trumpet (TP), PathDump (PD), SwitchPointer (SP), NetSight (NS), Marple (MP), and SpiderMon (SMon).



(a) The resource usage on Tofino switch is low. Per-port traffic meter is too small to be visible in the figure.



(b) Relative memory usage under different controller latency with SpiderMon as the baseline.

Figure 9: Switch memory occupation

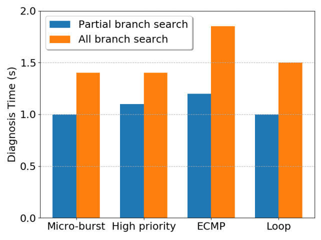
and SwitchPointer) all lack accurate in-network information, like accurate queuing information and the packet loss for traffic other than TCP (they can only observe packet loss at the sender with the help of TCP's congestion control). As for the proactive in-network approach in NetSight, it scarifies the telemetry data granularity to keep overhead low. Only the packet header, switch ID, output port, and a version number are included. It uses topology information to assemble out-of-order postcards since the fine-grained timestamps and queuing information are not included in the postcards. The reactive in-network Marple system can potentially collect the information at very fine granularity but it can only start this reactive network-wide query after a half-RTT delay after the problem has been detected. The experiments over Cache and Hadoop traces have qualitatively similar results with the web search trace; more details can be found in §E.2.

Diagnostic overhead. To evaluate the diagnosis complexity and resource usage of different solutions, we measure the amount of collected data and the extra bandwidth requirements. We measure the diagnosis complexity using the amount of telemetry data stored and used in the diagnostic procedure, using (flow×port) as the unit to denote the complexity of flow information collected at switch ports. Since the host-based solutions collect information from the end hosts, and they reconstruct the utilization of different links [37], we multiply the average path length with the flow×host as the overall complexity. Both switches and hosts have limited storage spaces and may restrict the scalability of the solutions. Under the same scenario for diagnosing micro-bursts, we show the amount of telemetry data for different systems in Figure 8(a). Reducing the diagnosis complexity not only relieves the burden to process the collected information for the central controller but also saves the storage space to store the diagnostic data for future usages.

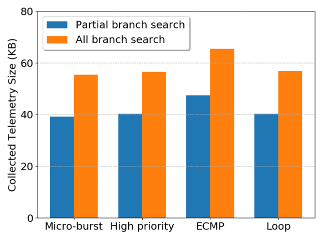
Trumpet processes packets and match triggers in real time during the monitoring phase, so no packet is stored. But in the reactive data gather-report phase, data from multiple hosts will be reported. In order to construct every link utilization, the throughput of all flows will be reported and stored for further analysis. Pathdump and SwitchPointer need to store per-packet history, since the problem may be detected after analysis. But both systems rely on the path information to find out the flows that travel the same link with the victim flow so that the data complexity can be reduced by filtering out irrelevant flows. Marple stores the query results from every switch to reproduce the scenarios, so such data will be transmitted as well as stored on the hosts. But Marple starts the collection after problem detection and stops after the problem disappears, collecting less but potentially incomplete data. NetSight stores all packet postcards and processes them in real time. All flows from all the switch ports are collected and stored, leading to a similar data complexity as Trumpet. SpiderMon only collects data after a problem is detected and only from relevant switches. Thus, the overhead for collecting telemetry data is much lower than the other systems.

Monitoring bandwidth overhead. Next, we measure the amount of extra bandwidth usage during monitoring. Trumpet never collects in-network data; it only uses the network to communicate with other servers, so it has a low overhead. PathDump and SwitchPointer both use two VLAN tags of 24 bits for path and switch epoch information. NetSight always collects per-packet postcards to the host for analysis, and the per-packet additional bandwidth occupation is 15 bytes/packet × average hop count because NetSight will generate a postcard for the packet at every hop. Marple introduces a 16-bit header to carry the per-packet end-to-end latency, and during the monitoring phase, it will group the packets with their per-hop queuing latency and sent them to the controller. SpiderMon adds a 16-bit monitor header to every packet when it enters the network, and removes it before forwarding the packet to the end-host as mentioned in §3.1.

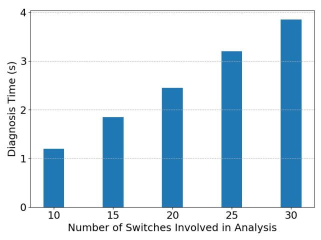
Switch resource overhead. Figure 9(a) shows the switch resource usage of SpiderMon, which fits comfortably in a Tofino pipeline. It also shows how SpiderMon scales with the number of flows seen during a collection period. Modern data centers have millions of concurrent flows per switch, but since SpiderMon only keeps tens of milliseconds of history, the number of flows per epoch is much smaller. Switch memory size increases steadily over time [29], so SpiderMon can scale to even more flows with more recent hardware.



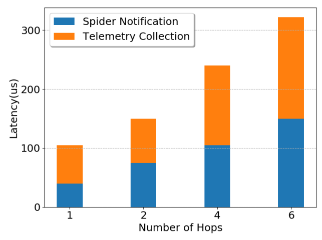
(a) Diagnosis time for root cause



(b) Collected telemetry data size

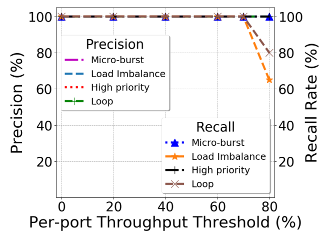


(c) Diagnosis time with different number of switches

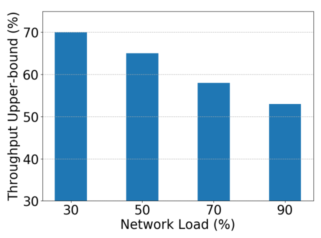


(d) The latencies for "spider" packets and telemetry

Figure 10: Branch-search metrics for SpiderMon



(a) Precision & recall rate for the root causes with 30% load



(b) Upper-bound of throughput threshold

Figure 11: Throughput metrics for SpiderMon

To show the benefit of informing related switches in the data plane in a distributed manner, we compare SpiderMon with a centralized reactive strawman system, which uses a centralized node to receive the detected problems, identifies the related switches, and retrieves data from them. We vary the additional latency that this centralized controller introduces. Figure 9(b) shows that this solution requires more memory to store a larger amount of historical data to avoid the loss of relevant evidence for diagnosis. In comparison, SpiderMon only needs to preserve the history within the maximum queuing latency + half RTT (§3.2.2).

4.3 Diagnostic Robustness

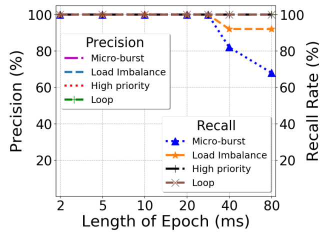
We finally evaluate the diagnostic robustness of SpiderMon using different metrics related to branch-search coverage, epoch length, and cumulative latency. Within a range of adjustments, SpiderMon can diagnose the performance problems with ideal precision and recall. Network operators are allowed to adjust the parameters of SpiderMon according to their requirements.

Overall methodology. SpiderMon empirically adjusts the parameters under different network loads. Given a particular network traffic load, operators could systematically test the precision and recall rates of SpiderMon with different metric choices. Suitable choices should strike a good balance between the recall rate and the size of collected telemetry data for throughput metrics, switch memory consumption for epoch metrics, and the sensitivity of problem detection for latency metrics. The optimal parameters vary under different network loads. We provide the results of parameter adjustments using our experimental settings in the following, while network operators could follow the same methodology to obtain their preferred parameters.

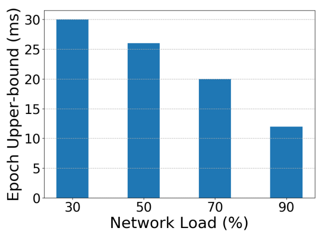
Branch-search threshold. SpiderMon provides different options for spider packet propagation in terms of its reach (e.g., all or some branches). Figure 10(a) and Figure 10(b) provide comparisons with different options on both the diagnosis time of root cause analysis and the size of collected telemetry data. Note that the number of relevant switches in SpiderMon is generally much smaller than the total network size since SpiderMon uses the wait-for relation and provenance model to precisely target only those relevant switches that contribute to the observed performance problem. Therefore, even with all-branches spider packets propagation (search all ports with > 0 throughputs), SpiderMon is efficient compared to more rudimentary diagnosis strategies that must comb through all data from all switches. Even for relatively widespread performance problems involving up to 30 relevant switches, it takes under 4 seconds to run the root cause diagnosis algorithm (Algorithm 5) on a 4.3GHz CPU, as shown in Figure 10(c). In addition, we evaluate the latency for spider packets propagation and the subsequent retrieval of the telemetry data, using 50 Gbps link bandwidth and 20μs link delay. From the results shown in Figure 10(d), we can see that a few microseconds are enough to perform the entire retrieval operation with arbitrary fat-tree topologies, no matter the choices of branch-search options. This is because SpiderMon's mechanisms run in the data plane. As a result, network operators can send "spider" packets without setting the branch-search threshold if the overhead can be tolerated based on their requirements.

We further evaluate the precision and recall rates under different branch-search coverage with different network loads. Figure 11(a) shows the results under 30% network load, indicating that the precision can always achieve 100% while the recall rates decrease if the threshold is too high. To trade-off the branch-search overhead and the recall rates, we suggest using 70% as the threshold in this case since it strikes a good balance. Following the same strategy, we summarize the upper bound of branch-search thresholds for operators to adjust under different network loads, as shown in Figure 11(b).

Epoch length. SpiderMon can change the length of the telemetry epoch to save memory but trade-off telemetry granularity. Network operators can adjust the telemetry epoch according to their requirements. Under different network loads, we provide the upper bound of the epoch length. For example, Figure 12(a) shows the results with the network load at 30%. We evaluate the precision and recall rates under different epoch lengths. The precision is always 100%, while

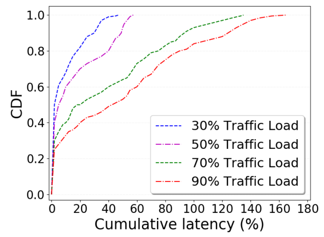


(a) Precision & recall rate for the root causes with 30% load

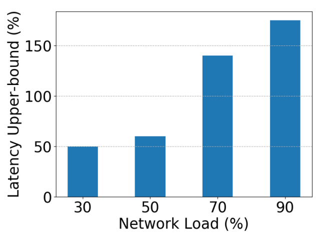


(b) Upper-bound of epoch length

Figure 12: Epoch metrics for SpiderMon



(a) Cumulative latency under different network loads



(b) Upper-bound of cumulative latency threshold

Figure 13: Latency metrics for SpiderMon

the recall rate decreases in some scenarios when the length of epoch exceeds 30 ms. We further measure the precision and recall rates under different network loads, and identify the upper-bounds of epoch length, as shown in Figure 12(b). The upper-bound epoch length used for telemetry collection decreases with increasing network load.

Cumulative latency threshold. SpiderMon provides a tunable cumulative latency threshold for problem detection, allowing network operators to customize problem trigger frequency for different applications. Figure 13(a) shows the CDF of different cumulative latency under different network loads in the absence of problems, where the cumulative latency is normalized by the maximum queuing latency of a single switch. Under different loads, the choice of cumulative latency threshold varies according to the trade-off between overhead and recall rate. The higher the sensitivity of the network to problem detection, the more switches are visited, and thus higher overhead. We further evaluate the recall rates of SpiderMon under different loads and summarize the upper bound of cumulative latency thresholds for reaching 100% recall in all scenarios in Figure 13(b).

5 Related Work

Switch-based telemetry. Telemetry systems such as Sonata [16], Marple [32], FlowRadar [26], *Flow [36], NetSeer [47] and Dapper [14] leverage programmable switches for fine-grained data collection. However, query-driven systems [16, 32] cannot dynamically change the targeted events at small timescales, and blanket monitoring systems [17,36] incur high collection overhead. SpiderMon aims to achieve lightweight yet accurate telemetry information collection. Two recent works, NetSeer [47] and PINT [8], share our high-level goal of reducing telemetry overhead. NetSeer detects per-flow performance events for compression, and PINT aggregates telemetry information across hops or flows to save bandwidth. Compared to these works, SpiderMon co-designs monitoring and posterior diagnosis based on wait-for relations for closed-loop diagnosis.

Diagnosis systems. SwitchPointer [38] and PathDump [37] collect both in-network and host data for diagnosis. Trumpet [31] monitors every packet at hosts and reports triggered events. SNAP [43] diagnoses network problems using logs (e.g., TCP statistics, socket calls) collected at hosts. However, these systems rely on a central controller and perform software-based monitoring. NetMedic [23], 007 [6], NetPoirot [7] use statistical methods and/or machine learning to identify root causes. Network provenance [42] tracks how packets flow through a network and apply formal reasoning to identify root causes. Deter [25] can process and replay a TCP trace to diagnose performance degradation. Compared to these works, SpiderMon leverages the telemetry information from programmable switches, and it uses wait-for relations to reason about performance contention in-network. Our recent workshop paper sketches a similar roadmap [41], but it does not contain a concrete design, implementation, or evaluation.

Monitoring. Another line of recent work focuses on designing compact data structures [11, 18, 19, 27, 44] with tradeoffs between accuracy and resource footprints. OmniMon [19] divides flow-level monitoring across different network entities to satisfy resource constraints. BeauCoup [11] supports multiple distinct counting queries simultaneously while requiring a small number of memory accesses. These data structures complement SpiderMon by reducing switch resource usage.

6 Conclusion

SpiderMon is a system that achieves high coverage and low overhead in monitoring and diagnosing network performance problems. It monitors every flow in the data plane and triggers diagnostic events upon problem detection. It precisely collects diagnostic information in an as-needed fashion. We prototype SpiderMon on Tofino hardware and BMv2 software switches and show that it can leverage wait-for relations to accurately pinpoint root causes for complex problems. SpiderMon also has low overheads for telemetry collection, switch resources, and network bandwidths.

Acknowledgment

We thank our shepherd Theophilus A. Benson and the anonymous reviewers for their valuable feedback. This research is sponsored by the NSF under CNS-1718980, CNS-1801884, and CNS-1815525.

References

[1] Network Congestion Management: Considerations and Techniques. https://www.sandvine.com/hubfs/downloads/archive/whitepaper-network-congestion-management.pdf.

[2] sFlow. http://www.sflow.org/.

[3] Solving the mystery of link imbalance: A metastable failure state at scale. https://engineering.fb.com/production-engineering/solving-the-mystery-of-link-imbalance-a-metastable-failure-state-at-scale/.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[6] 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.

[7] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM*, 2016.

[8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, 2020.

[9] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SOSR*, 2016.

[10] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich. Catching the microburst culprits with Snappy. In *SelfDN*, 2018.

[11] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, 2020.

[12] J. Cho, H. Chang, S. Mukherjee, T. Lakshman, and J. Van der Merwe. Typhoon: An SDN enhanced real-time big data streaming framework. In *ACM CoNEXT*, 2017.

[13] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*, 2018.

[14] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SOSR*, 2017.

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

[16] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In *ACM HotNets*, 2016.

[17] 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.

[18] 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.

[19] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy. In *ACM SIGCOMM*, 2020.

[20] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4-> NetFPGA workflow for line-rate packet processing. In *ACM FPGA*, 2019.

[21] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally. Network congestion avoidance through speculative reservation. In *IEEE HPCA*, 2012.

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

[23] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, 2010.

[24] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX NSDI*, 2019.

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

[26] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, 2016.

[27] 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.

[28] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *ACM PODC*, 2015.

[29] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, 2017.

[30] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.

[31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[32] 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.

[33] Y. Ran, X. Wu, P. Li, C. Xu, Y. Luo, and L.-M. Wang. EQuery: Enable event-driven declarative queries in programmable network measurement. In *IEEE NOMS*, 2018.

[34] 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.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[36] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *USENIX ATC*, 2018.

[37] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with PathDump. In *USENIX OSDI*, 2016.

[38] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX NSDI*, 2018.

[39] Y. Tang, Y. Wu, G. Cheng, and Z. Xu. Intelligence enabled SDN fault localization via programmable inband network telemetry. In *IEEE HPSR*, 2019.

[40] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *ACM SOSR*, 2017.

[41] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng. Grasp the root causes in the data plane: Diagnosing latency problems with SpiderMon. In *ACM SOSR*, 2020.

[42] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing performance problems with temporal provenance. In *USENIX NSDI*, 2019.

[43] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.

[44] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

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

[46] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: Generic off-CPU analysis to identify critical waiting events. In *USENIX OSDI*, 2018.

[47] 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 *ACM SIGCOMM*, 2020.

[48] 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.

A Proof for Contributors Identification Algorithm

Definition 6. *Degree of vertex. In a WFG, the degree of vertex A is the sum of all the adjacent edges' weights w_e:*

$$D(A) = \sum_{\{e=<i,j>|i=A\|j=A\}}^{e} \alpha_e \cdot w_e \tag{1}$$

where α_e is 1 when A is the sink of edge e and -1 when vertex A is the source.

Lemma 1. *For a WFG, the sum of all the vertex's degree is 0:*

$$\sum_{X\in V}^{X} D(X) = 0 \tag{2}$$

Proof: the WFG is a directed graph where every edge is pointing from a vertex to another vertex in the graph, so each edge will add weight w to the sink vertex and weight $-w$ to the source vertex.

Definition 7. *Flux of cut. For a cut in a WFG, the vertex will be divided into two sets, S1 and S2. Given all edges in the WFG has a positive weight according to the definition, we denote the flux of this cut as:*

$$Flux(cut) = \left| \sum_{i\in S1, j\notin S1}^{e=<i,j>} w_e + \sum_{i\notin S1, j\in S1}^{e=<i,j>} -w_e \right| \tag{3}$$

where e represents the edge from vertex i to vertex j

Though the sum of all vertex's degree is 0, we can always find a cut whose flux is maximum, representing the provenance relation between vertexes from those two groups is the strongest. The set with a positive degree considers as the main contributor to the queue, while the other set contains victims of the queue, like normal flows or small flows. To find this cut efficiently, we have shown the hints by the following lemmas and theorems.

Lemma 2. *The flux of one cut is just the absolute value of the sum of all vertices' degrees in either set.*

Proof: The absolute value of the sum of all vertices' degrees in one set (ASD) can be written as:

$$\begin{aligned}
\text{ASD} &= \left| \sum_{i\in S1|j\in S1}^{e=<i,j>} \alpha_e \cdot w_e \right| \\
&= \left| \sum_{i\in S1\&j\in S1}^{e=<i,j>} \alpha_e w_e + \sum_{i\in S1\&j\notin S1}^{e=<i,j>} \alpha_e w_e + \sum_{i\notin S1\&j\in S1}^{e=<i,j>} \alpha_e w_e \right| \\
&= \left| 0 + \sum_{i\in S1\&j\notin S1}^{e=<i,j>} -w_e + \sum_{i\notin S1\&j\in S1}^{e=<i,j>} w_e \right| = Flux(cut)
\end{aligned} \tag{4}$$

Theorem 1*. *The WFG cut with maximum flux will divide the vertices with positive degrees into one set and negative degrees into the other set.*

Given the sum of all vertices' degrees are 0, for any cut: $\sum_{X\in S1} D(X) = -\sum_{Y\in S2} D(Y)$, namely, the absolute sum of degree for two sets are the same. Thus, for the cut that divide all vertices with positive degrees into one set, by contradiction, we can easily prove this is the cut with maximum flux.

The flux represents the wait-for relation between two groups from a cut of the wait-for graph, and the degree represents the value of incoming edges weights subtracting outgoing edges weights so that Theorem 1 is proofed.

B Fine-grained Sliding Window

During the telemetry collection process, SpiderMon maintains bloom filter and per-port per-epoch data structures to trace back all the relevant switches. However, part of these structures (e.g. traffic meter) needs to be reset to 0 at the beginning of an epoch due to the limited resources of the switch data plane. Therefore, there will be some information loss at the beginning of an epoch, leading to the diagnosis algorithm being inaccurate. SpiderMon employs a fine-grained sliding window on the data plane to achieve high accuracy for the used data structures.

The sliding window strategy slices each epoch into multiple pieces, and it proceeds in two actions: an update action and a decrease action. To explain simply, we take the traffic meter in the per-port data structure as an example. Assume one epoch T is divided into n small time slots. There will be n sub-traffic meters and each of them aims at a single time slot. When a switch receives a new packet during the update phase, the switch will update the corresponding sub-traffic meter based on the current time slot, as well as the total traffic meter. For decrease action, when the oldest sub-traffic meter no longer exists in the sliding window, the value of the corresponding sub-traffic will be subtracted from the total traffic meter and that sub-traffic meter will be reset to 0. Network operators are able to tune the fine-grained sliding window according to their demands. Basically, the more time slots an epoch is divided into, the higher the accuracy that the system can achieve. On the other hand, the overhead of telemetry data structures can be reduced with fewer time slots.

C Resource Usage Optimization

C.1 Avoid Duplicate Detection

In the scenario of the performance problem, there are lots of packets from the victim flow suffering from high latency problems, but not all of them will generate a diagnostic event independently. SpiderMon sets a limitation on the interval between two diagnostic events generated by the same flow, meaning that during one congestion, only the first packet suffering from high accumulated latency will trigger the diagnostic event. To avoid receiving multiple audit requests for the same diagnosis event, the switches will drop the duplicate "spider" packets with the same event ID as well.

C.2 Data Field Compression

For the applications like SpiderMon built on top of the programmable switches, keeping track of some data fields in the packet header or on the switch memory is always required. Compressing those data fields in order to reduce the extra header size or switch memory occupation is critical to the application performance. SpiderMon provides a method to compress the size of the data by extracting the most significant bits. This idea can be widely applied to many recorded data in such systems, and here are two typical examples that use this strategy:

The timeout bloomfilter in SpiderMon requires storing a large number of timestamps for each slot in the bloom filter, which is very resource consuming and inefficient. The timestamp is usually stored with 48 bits on the switch and SpiderMon uses the timestamp to perform the timeout operation. Given that the only operation on the timestamp is the subtraction of two timestamps and compare the difference with the timeout period, we can easily observe that the only significant bits in the timestamp are the bits around the period. Take the timeout period as 1 ms as an example, the most significant bits in the timestamp are the 10th, 11th, and 12th bits from the right, representing 0.512 ms, 1.024 ms, and 2.048 ms respectively. By extracting these three bits from the original timestamps and comparing the difference with bit array 010, we can get an approximation of the exact value that is calculated with the original timestamp. Adding more bits on the left (*e.g.* 13th and 14th) can prevent us from the danger of overflow while adding more bits on the right (*e.g.* 9th and 8th) can help us obtain a more precise result of the subtraction. With this method, SpiderMon only needs to store 6 bits for each timestamp and reduce the memory usage of the timeout bloomfilter by 87.5%.

Another example is the queuing information carried by the packets in SpiderMon, which is used to detect the performance problem by comparing the accumulated delay with the maximum delay threshold. For a certain application, the maximum delay threshold may be 1 ms. Then when we calculate the accumulated delay, the most significant bits are 8th, 9th, and 10th bits from the right, representing 0.128 ms, 0.256 ms, and 0.512 ms respectively. If any bit on the left of the 10th bit is not 0, SpiderMon will trigger the problem immediately, since it exceeds the threshold with this single-hop delay. In this way, SpiderMon only needs to add an extra header with 4 bits to carry each delay field instead of 19 bits, shrinking the overhead from the extra header by 78.95%. Note that in evaluation, we use 8 bits for each data field to provide better accuracy.

D Implementation

We have implemented SpiderMon on a Barefoot Tofino switch with 1147 lines of P4-Tofino code and also a BMv2 version for NS3 and MiniNet environments with 945 lines of P4 code. We also implement the root causes analyzer on the end-host with 843 lines of Python code.

Figure 14 depicts different components in a switch and the workflow for different packet types. The event record is used for checking duplicate "spider" packets, and the telemetry counter for guiding telemetry packet generation. Those two data structures are placed in the ingress because they need to make decisions on whether to mirror packets in the traffic management unit. The per-port meter and timeout bloom filter provide provenance data to guide the propagation of the "spider" packets, and the telemetry data structure stores historical flow information for diagnosis. Those two data structures, along with the problem detection component, are placed in the egress pipeline because they may require queuing information, which is only available in the egress pipeline. Note that the per-port telemetry information is stored separately on the switch, but not necessarily one table per stage. One stage in SpiderMon can store multiple egress ports' telemetry information.

To implement SpiderMon, the egress pipeline is required to detect the problems, store telemetry information, and provide temporary provenance hints for "spider" packet propagation. For switch architectures like SimpleSumeSwitch [20] (NetFPGA), P4FPGA [40], and SmartNICs, SpiderMon can also be implemented by taking the next switch's pipeline as the "egress pipeline" of former switches to detect congestion and collect telemetry information. This design requires more communication among switches, so both the latency for diagnosing the problem and the link bandwidth used by SpiderMon would also increase.

As for the hardware switch resource, modern switches have increasing memory sizes [29], and more ports usually represent more on-chip memory, which, we shall demonstrate in §4, is more than sufficient to support SpiderMon.

E Additional Experiment Results

E.1 Header Bandwidth Usage

Packet Size (B)	1480	1000	500	100
SpiderMon (Gbps)	23.51	23.5	22.84	20.51
Baseline (Gbps)	23.65	23.5	22.84	21.87

Table 1: SpiderMon's maximum throughput is quite close to the baseline switch with only forwarding rule.

As the monitor header added by SpiderMon is removed before forwarding the packet to the end-host, the corresponding overhead of the additional header is very trivial. We use iPerf to show the maximum throughput of traffic with different average packet sizes on the Tofino switch equipped with SpiderMon in Table 1 and compare it with a baseline switch program with only basic forwarding rules. As expected, SpiderMon 's end-to-end throughput is nearly identical to the baseline, meaning that the bandwidth overhead of the monitoring phase could be neglected.

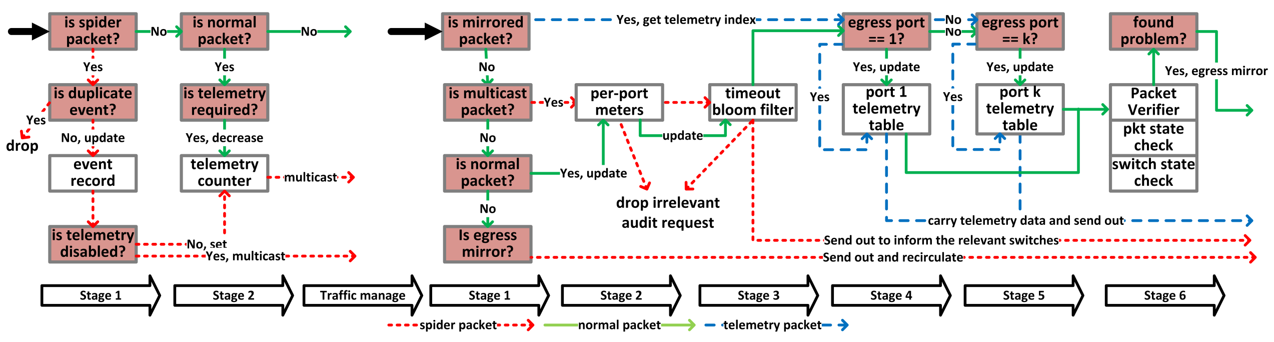


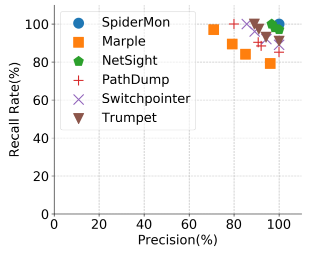
Figure 14: The placement of SpiderMon components on the switch stages

E.2 Cache & Hadoop Workloads

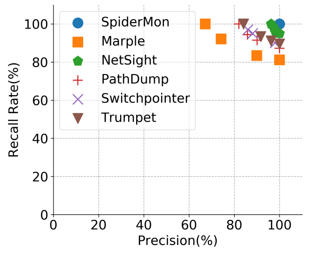
Besides the Web search trace, we also run the same experiments on the Cache trace and Hadoop Trace.

For the Cache trace, most of its flow sizes fall into 1KB to 100KB. Thus, to reach the same link utilization, we have to insert more number flows during the simulation. The results for Cache trace are similar to the Web search trace. The only difference is that all algorithms have improved performance. This is because the flow sizes are very small so that the root-cause traffic (e.g. micro-burst) flow can be easily distinguished from the normal flows; false positive and false negative are reduced.

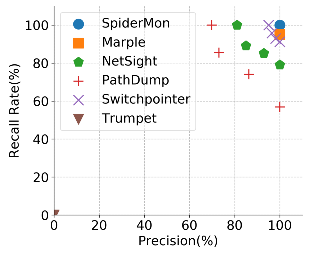
For the Hadoop trace, most of the flows have less than 10 KB flow size. Similar to the Cache trace, we also increase the number of flows to keep the same link utilization. The overall results for the Hadoop trace are also similar to the Cache trace.



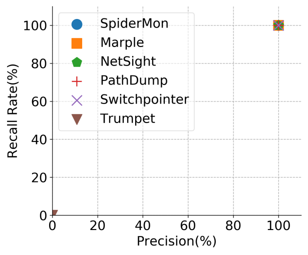
(a) Micro-burst



(b) Priority contention

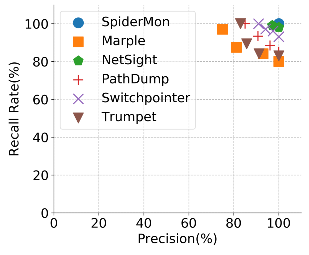


(c) ECMP imbalance

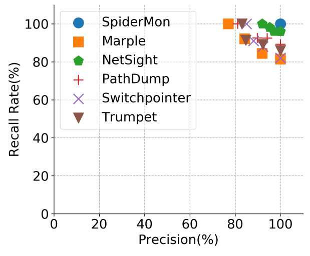


(d) Loops (solutions except Trumpet all overlap at the top right corner)

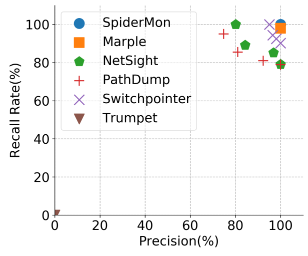
Figure 15: Diagnostic effectiveness with Cache trace



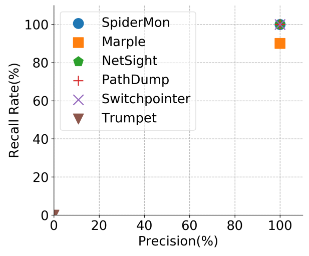
(a) Micro-burst



(b) Priority contention



(c) ECMP imbalance



(d) Loops (solutions except Marple and Trumpet all overlap at the top right corner)

Figure 16: Diagnostic effectiveness with Hadoop trace

F Tunable Parameters for Different Solutions

We vary the following parameters when using those systems to diagnose problems of the four scenarios. The goal is to find the parameter sets with the best precision and recall rate. We do nested iterations over different parameters by fixing some parameters and iterate the other parameters. The parameters are different across systems, and for the same system, the parameters vary according to the scenarios that we are trying to diagnose. The details are shown in Table 2 and Table 3.

G Constructing Signatures for Root Causes

SpiderMon uses both the collected telemetry information and the static network configuration information to recognize the root causes. The telemetry information is collected by SpiderMon, and the configuration information is simply provided

	Micro-burst-related Contention	Priority-related contention
Trumpet	Tolerable per-flow throughput, tolerable end-to-end latency difference, tolerable TCP packet loss	Tolerable per-flow throughput, tolerable end-to-end latency differences, tolerable TCP packet loss
PathDump	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
SwitchPointer	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
NetSight	Related time intervel length, tolerable link utilization	Related time intervel length, tolerable link utilization, postcard arrival sequences
Marple	Network-wide query lasting time, tolerable per-flow throughput	Network-wide query lasting time, tolerable per-flow throughput
SpiderMon	Maximum allowed flow throughput	/

Table 2: Parameters for micro-burst and priority

	ECMP load imbalance	Loop
Trumpet	/	/
PathDump	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
SwitchPointer	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
NetSight	Related time intervel length, tolerable link utilization, tolerable link utilization imbalance ratio	/
Marple	Network-wide query lasting time, tolerable link utilization imbalance ratio	Network-wide query lasting time
SpiderMon	Tolerable link utilization imbalance ratio	/

Table 3: Parameters for load imbalance and Loop

by the topology information and routing information, which is known by the operator in advance.

To add a new signature for a new root cause, network operators could simply use the above information to construct their own signatures. Here we provide some telemetry information and static configuration information used in the 4 example signatures in Table 4. This is not an exhaustive list and more information could be added when new signatures are introduced. To construct new signatures, we should know that any signature consists of two parts: 1) the root cause's pattern, like a flow with large throughput for the micro-burst root cause; 2) the relation between the problematic flow and the victim flow, namely, the problematic flow should be one of the main contributors to the victim flow's poor performance. Here we also provide 4 different signatures as examples.

Telemetry Info	Edge weight from flow i to flow j: E($flow_i$, $flow_j$)
	Main contributors for a queue: Contributors($Switch_i Port_j$)
	Flows traveling a switch port: Flows($Switch_i Port_j$)
	Priority: P(flow)
	Data volume: V(flow)
Config Info	Port mapping in Topology: Topo($Switch_i Port_j$)=$Switch_x Port_y$
	Flows belonging to an ECMP group: Flows(group)

Table 4: Selected telemetry information and static configuration information

Micro-bursts. SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flows have many wait-for edges with large weights pointing to themselves due to a large amount of traffic during the problematic time. For example, for the micro-burst problem, there must exist one micro-burst node *root* which satisfies:

The root cause flow has the same priority as the victim flow:

$$P(victim) = P(root) \tag{5}$$

The root cause flow has similar edge weight to itself as to other flows:

$$E(root, root) \approx E(victim, root) \tag{6}$$

The victim flow contends with the root cause flow:

$$\begin{aligned} &\exists m, n, \text{where} \\ &victim \in Flows(Switch_m Port_n) \\ &root \in Contributors(Switch_m Port_n) \end{aligned} \tag{7}$$

The larger the weights of $E(root, root)$ and $E(victim, root)$, the more confidence SpiderMon has on determining the micro-burst flow.

Different priorities. For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b). The high priority flow *root* should satisfy:

The root cause flow has higher priority than the victim flow:

$$P(victim) < P(root) \tag{8}$$

The root cause flow has smaller edge weight for the edge pointing to itself than the edge pointing to the victim:

$$E(root, root) < E(victim, root) \tag{9}$$

The victim flow contends with the high priority flow:

$$\begin{aligned} &\exists m, n, \text{where} \\ &victim \in Flows(Switch_m Port_n) \\ &root \in Contributors(Switch_m Port_n) \end{aligned} \tag{10}$$

ECMP load imbalance. For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is highly imbalanced as in Figure 5(c). Within the problematic ECMP group *ecmp* on Switch $Switch_x$, there must exist one or more flows *root*, which satisfies:

The ECMP traffic split on some switches is not balanced:

$$\begin{aligned}&Throughput(Switch_x Port_y) = \sum V(flow_i),\\ &\text{where } flow_i \in Flows(Switch_x Port_y)\end{aligned} \tag{11}$$

$$\begin{aligned}&\exists x, y, \forall i \neq y,\\ &Throughput(Switch_x Port_y)\\ &> Throughput(Switch_x Port_i)\end{aligned} \tag{12}$$

The root cause flow is one of the flows from the ECMP port that has larger throughput.

$$root \in Flows(ecmp) \cap Flows(Switch_x Port_y) \tag{13}$$

On another switch, the victim flow contends with the root cause flow:

$$\begin{aligned}&\exists m, n, \text{where}\\ &victim \in Flows(Switch_m Port_n)\\ &root \in Contributors(Switch_m Port_n)\end{aligned} \tag{14}$$

Transient/persistent loops. For the latency problem caused by transient or persistent loops as shown in Figure 1(c), SpiderMon searches the port-level contributors along the contributor traffic's path. If the same port is observed twice during the search procedure, all those ports are highly likely to have formed a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop. The formal signature for a flow *root* with a transient/persistent loop can be written as:

$$\text{Exist a port list:} [Switch_{m_0} Port_{n_0}, ..., Switch_{m_k} Port_{n_k}] \tag{15}$$

The port list forms a ring in the topology and the root cause flow routed in a loop on that ring:

$$\begin{aligned}&\forall i,\\ &Topo(Switch_{m_i} Port_{n_i}) == Switch_{m_{i+1}} Port_{n_{i+1}}\\ &root \in Flows(Switch_{m_i} Port_{n_i})\end{aligned} \tag{16}$$

The victim flow contends with the loop traffic on one of the switches on that ring:

$$\begin{aligned}&\exists j, where j \in [0, 1, ..., k]\\ &victim \in Flows(Switch_{m_j} Port_{n_j})\\ &root \in Contributors(Switch_{m_j} Port_{n_j})\end{aligned} \tag{17}$$