# Distributed Network Telemetry with Resource Efficiency and Full Accuracy

Haifeng Sun, Qun Huang, *Member, IEEE,* Patrick P. C. Lee, *Senior Member, IEEE,* Wei Bai, *Member, IEEE,* Feng Zhu, and Yungang Bao, *Member, IEEE,*

**Abstract**—Network telemetry is essential for administrators to monitor massive data traffic in a network-wide manner. Existing telemetry solutions often face the dilemma between *resource efficiency* (i.e., low CPU, memory, and bandwidth overhead) and *full accuracy* (i.e., error-free and holistic measurement). We break this dilemma via a network-wide architectural design OmniMon, which simultaneously achieves resource efficiency and full accuracy in flow-level telemetry for large-scale data centers. OmniMon carefully coordinates the collaboration among different types of entities in the whole network to execute telemetry operations, such that the resource constraints of each entity are satisfied without compromising full accuracy. It further addresses *consistency* in network-wide epoch synchronization and *accountability* in error-free packet loss inference. We prototype OmniMon in DPDK and P4. Testbed experiments on commodity servers and Tofino switches demonstrate the effectiveness of OmniMon over state-of-the-art solutions.

**Index Terms**—Network measurement, Distributed systems, Telemetry language.

## 1 Introduction

Modern data centers host numerous critical applications, thereby imposing high demands on network management. Administrators need efficient *network telemetry* solutions to monitor massive traffic and understand its network-wide behaviors. Ideally, we aim for both *resource efficiency* and *full accuracy* when developing network telemetry. For resource efficiency, network entities (e.g., end-hosts and switches) face heterogeneous resource constraints in computation, memory, and bandwidth, so telemetry systems should limit their performance overhead and never exhaust any specific resources. In particular, the performance overhead of network telemetry must be much lower than that of normal packet processing in routing, NAT, and firewall. Note that achieving resource efficiency does not mean that the resource usage remains constant as the network scale grows; instead, the goal of resource efficiency here is to keep the resource overhead of network telemetry much lower than that of other network operations. For full accuracy, telemetry systems should have in-network, always-on visibility that covers all flows and all entities with error-free flow-level statistics.

Conventional wisdom views resource efficiency and full accuracy as a design trade-off. At one extreme, per-flow monitoring (e.g., Trumpet [1] and Cisco Tetration [2]) aims for fine-grained, error-free measurement but incurs possibly unbounded resource usage; at another extreme, coarse-grained monitoring (e.g., SNMP [3], sFlow [4], and NetFlow [5]) provides best-effort measurement without accuracy guarantees. Between the extremes, many telemetry systems adopt approximation algorithms [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16] that achieve high resource efficiency with provably bounded errors, or event matching techniques [17], [18], [19], [20], [21], [22] that only focus on the traffic of interest.

We pose a fundamental question: *Can we break the dilemma between resource efficiency and full accuracy in network telemetry?* We observe that there exist delicate design trade-offs between resource availability and in-network visibility among different network entities, including end-hosts and switches in the data plane as well as a centralized controller in the control plane (§2.1). By carefully re-architecting the collaboration among network entities subject to their resource constraints, we can design a new telemetry architecture that achieves both resource efficiency and full accuracy. Note that recent telemetry proposals also take a collaborative approach. However, they address different design goals (e.g., expressiveness [19], [23] or in-network visibility [24]) rather than the resource-accuracy trade-off. Their architectures cannot readily achieve full accuracy due to the tight switch resources (§2.2). We fill this void through a complementary collaborative architecture that treats both resource efficiency and full accuracy as “first-class citizens”.

We present OmniMon, a novel network-wide architecture that carefully coordinates telemetry operations among all entities (i.e., end-hosts, switches, and the controller), with a primary goal of simultaneously achieving both resource efficiency and full accuracy in flow-level network telemetry for large-scale data centers. Specifically, OmniMon re-architects network telemetry in a *split-and-merge* fashion. By *split*, OmniMon decomposes network telemetry into partial operations and schedules these partial operations among different entities. This design fully utilizes the available resources of the entire network and bypasses the resource

- *H. Sun and Q. Huang are with the Department of Computer and Science, Peking University, Beijing 100871, China. E-mail: sunhaifeng@stu.pku.edu.cn, huangqun@pku.edu.cn.*
- *P.P.C. Lee is with the Department of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong, China. E-mail: pclee@cse.cuhk.edu.hk.*
- *W. Bai is with Microsoft ResearchLab, Redmond, WA 98052 USA. E-mail: baiwei0427@gmail.com.*
- *F. Zhu and Y. Bao are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: {zhufeng, baoyg}@ict.ac.cn.*

constraints of individual entities. By *merge*, OmniMon coordinates all entities to collaboratively execute the partial operations. It combines the strengths of both end-hosts and switches (in the data plane) for error-free per-flow tracking across the entire network; meanwhile, its controller (in the control plane) conducts network-wide resource management and collective analysis of all end-hosts and switches. Note that OmniMon is incrementally deployable on a subset of end-hosts and switches across the network.

The network-wide architectural design of OmniMon needs to address the reliability concerns in network telemetry. In particular, OmniMon incorporates two specific reliability guarantees, namely *consistency* and *accountability*, to preserve both resource efficiency and full accuracy in practical deployment. For the consistency concern, due to the absence of a global clock, when the values of one flow on different entities differ, we cannot determine whether it is the result of packet loss or the same packet is measured in different windows across different entities. Thus, we propose a network-wide epoch synchronization mechanism with a *hybrid* consistency model, such that (i) all entities reside at the same epoch at most times and their epoch boundaries only deviate by a small time difference (up to 60 $\mu$s; see §9) and (ii) every packet is synchronized at the same epoch during its transmission even under cross-epoch network delays. For the accountability concern, it is challenging to ensure full accuracy because packet loss is inevitable and switches cannot maintain per-flow counters. Therefore, we formulate a system of linear equations for per-switch, per-flow packet loss inference, and ensure that a unique solution exists in the common case by taking into account data center network characteristics.

We prototype OmniMon in DPDK [25] and P4 [26]. We also provide a query engine to enhance the expressiveness of OmniMon. Our testbed experiments on commodity servers and Barefoot Tofino switches [27], as well as large-scale simulation, justify the effectiveness of OmniMon over 11 state-of-the-art telemetry designs. For example, OmniMon reduces the memory usage by 33-96% and the number of actions by 66-90%, while achieving zero errors, compared to sketch-based solutions (which have non-zero errors) [13], [15], [16], [28] in P4 switch deployment.

The source code of our OmniMon prototype is now available at: https://github.com/N2-Sys/OmniMon.

## 2 Background and Motivation

We first pose the resource and accuracy requirements in network telemetry and identify the use cases (§2.1). We then discuss the root causes of having the resource-accuracy trade-off in existing telemetry systems (§2.2).

### 2.1 Resource and Accuracy Requirements

We focus on flow-level network telemetry in a data center network, in which we identify a flow by a *flowkey* formed by a combination of packet fields (e.g., 5-tuples). We measure flow-level statistics, in the form of *values*, across fixed-length time intervals called *epochs*. We represent an epoch as an integer, by dividing the current timestamp by the epoch length. In particular, we address two requirements in network telemetry: *resource efficiency* and *full accuracy*.

**Resource efficiency.** We target a data-center-scale network telemetry framework that includes three types of entities: multiple *end-hosts*, multiple *switches*, and a centralized *controller*. Each type of entities imposes different resource constraints. Thus, the framework should satisfy the resource constraints of the corresponding entities to maintain the overall packet forwarding performance. Also, it should only leverage the existing functionalities of commodity hardware. We elaborate each type of entities as follows.

*(i) End-host:* End-hosts process network traffic in the network edge in software (e.g., kernel space [29] and user space [25]). They provide ample memory space for holistic flow tracking, and high flexibility for realizing various telemetry approaches. However, as end-hosts reside in the edge, they have poor in-network visibility for network-wide measurement. Also, software packet processing on a commodity host is computationally expensive (e.g., requiring multiple CPU cores to achieve a rate of 40 Gbps [30]), yet end-hosts cannot dedicate all CPU resources only for network telemetry as the CPU resources are also shared by other co-locating applications.

*(ii) Switch:* Commodity switch ASICs achieve high forwarding throughput (e.g., several Tbps [27], [31]) and ultra low processing latency (e.g., sub-microseconds). However, they have scarce on-chip memory space (e.g., tens of MB to keep states [32], [33]), and provide limited programmability due to overheating concerns and high manufacturing costs.

*(iii) Controller:* The controller is a *logical* entity that centrally coordinates all end-hosts and switches to obtain a global network view. It can comprise multiple servers, so as to combine their CPU and memory resources for sophisticated analytics and provide fault tolerance. However, it has limited network bandwidth to receive a substantial amount of traffic from all end-hosts and switches.

**Full accuracy.** We target network telemetry with *full accuracy*, which implies both *completeness* and *correctness*. By completeness, we mean to track all traffic (and hence all flows) in all entities and epochs without missing any information; by correctness, we mean to track the flowkey and complete values for each flow without errors. Full accuracy is critical for many telemetry tasks. We show three use cases that greatly benefit from full accuracy. Note that such use cases are also supported in existing telemetry systems (e.g., [23], [34]), yet any measurement error can degrade their efficiency and usability in practical deployment.

*(i) Performance evaluation:* Administrators often propose new techniques to improve network performance, and verify their correctness and performance gain in a real deployment. One example is to evaluate the benefits of flowlet-based load balancing [35] over conventional flow-based load balancing. Administrators can make refinements using complete *per-flow* trajectories, which unveil both per-link utilization and top flows in each link. Full accuracy in per-flow trajectories guarantees that all monitored performance impacts are caused by the load balancing algorithm rather than measurement variations. This allows administrators to focus on evaluating and tuning the proposed techniques without concerning measurement errors, thereby preventing flawed algorithmic refinement.

*(ii) Anomaly detection:* Administrators identify network anomalies (e.g., misbehaviors or attacks) for reliability and

security purposes. Such anomalies are hard to predict before they actually occur. Thus, administrators require *per-epoch* statistics of *all traffic* to respond to any detected anomalies Full accuracy in per-epoch statistics eliminates false alarms or undetected anomalies due to any measurement error, and hence prevents any anomaly from compromising the network robustness. Here, we focus on collecting accurate flow statistics, while providing precise anomaly definitions and configurations is beyond our scope.

*(iii) Network diagnosis:* Performance degradations (e.g., sudden packet loss) are common for various reasons, such as misconfigurations and hardware failures. Administrators require *per-switch, per-flow* statistics to localize and repair problematic switches and lossy flows. Full accuracy in per-switch, per-flow statistics implies that any performance issue can be correctly pinpointed in a timely manner without any interference from inaccurate measurement.

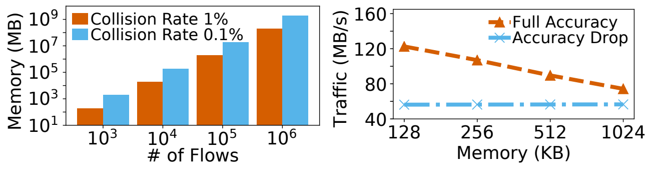
## 2.2 Motivation

Existing telemetry systems (§10) pose resource efficiency and full accuracy as a design trade-off: they either provision enough resources (i.e., CPU, memory, and bandwidth) to achieve full accuracy, or sacrifice full accuracy to satisfy the resource constraints of individual entities. We argue that the trade-off exists due to two root causes.

**Root cause 1: Tight coupling between flow tracking and resource management.** Existing approaches realize telemetry operations *monolithically*, in which an end-host or a switch extracts the values of each observed flow and provisions resources to track the values (referenced by flowkeys). To resolve any resource conflict among flows, the amount of provisioned resources must be sufficiently large, yet it also heavily depends on the required level of accuracy and the resource constraints of individual entities.

Consider an example that we use a hash table of counters (e.g., as in Trumpet [1]) to track the flowkeys and values of all flows. To resolve hash collisions, we can leverage chaining or Cuckoo hashing [36] to organize counters for colliding flows. Such a design can be feasibly realized in software, but not in switch ASICs as the complex memory management of the hash table is hard to implement due to limited switch programmability. If we solely increase the hash table size and rely on uniform hash functions to resolve hash collisions in switches, the number of counters is huge. Fig. 1(a) shows the memory usage (for 4-byte counters) versus the number of flows for different hash collision rates (in general, the collision rate is $1 - \frac{m!}{(m-n)!m^n}$ for $m$ counters and $n$ flows). Even for 1,000 flows, the memory usage reaches 190 MB for a collision rate of 1%, which is far beyond the switch capacity.

Existing switch-based telemetry systems sacrifice accuracy to fit all operations into switch ASICs [37], [38]. One solution leverages approximation algorithms (e.g., sampling [6], [7], top-$k$ counting [8], [9], [10], and sketches [12], [13], [14], [15], [28]) to fit compact structures in switch memory. However, full accuracy is inevitably violated due to limited provisioned resources. Another solution offloads traffic to the controller to mitigate switch-side resource usage [17], [18], [20], [21], but it only monitors the traffic of interest to avoid overwhelming the link capacity of the controller.



(a) Root cause 1 (b) Root cause 2

Fig. 1. Root causes of resource-accuracy trade-off: (a) Memory usage using simple hashing with full accuracy; (b) Switch resource overheads in Sonata with controller-switch collaboration.

**Root cause 2: Limited network-wide collaboration.** Several telemetry designs adopt some forms of collaboration among entities, yet they target different design goals and remain bounded by the resource-accuracy trade-off. For example, some systems deploy a controller to assemble measurement results from end-hosts or switches [1], [13], [14], [15] and/or tune resources among switches [6], [12], [28], [37]. Some hybrid architectures [19], [23] allow the collaboration among switches and the controller to empower efficient and expressive telemetry query engines, while others [24], [39], [40] allow the collaboration among end-hosts and switches to combine programmability in end-hosts and visibility in switches. However, existing collaborative approaches do not coordinate the resources for full accuracy among *all* entities (i.e., end-hosts, switches, and the controller). The limited resources of some entities can make full accuracy infeasible.

We use Sonata [23], a collaborative telemetry architecture, to motivate our claims. Sonata targets expressiveness and scalability. It offloads dataflow operators from the controller to switches to exploit fast packet processing in switches. However, it cannot mitigate the memory and bandwidth usage of switch-side operators without sacrificing accuracy. We take the open-source Sonata prototype and evaluate its resource consumption using the 11 applications reported in [23]. We vary per-application memory from 128 KB to 1,024 KB, and measure the total switch traffic to the controller on Barefoot Tofino switches [27]. We compare two cases: (i) "Accuracy drop", the default setup in Sonata's prototype that does not deal with hash collisions; and (ii) "Full accuracy", in which we achieve full accuracy by mapping each flow to $d$ (set to 2 here) counters and evicting a flow to the controller if it has hash collisions in all the $d$ counters (c.f. §3.1 of [23]). We configure $10^5$ active flows at each epoch. Fig. 1(b) shows that around 56 MB/s of traffic is triggered in all cases with accuracy degradation. For full accuracy, the amount of triggered traffic substantially increases due to hash collisions (e.g., 2.2× the accuracy degradation case for 128 KB of per-application memory). Note that handling hash collisions also incurs excessive usage of other types of switch resources (§9).

## 3 OmniMon Design

We state our design goals and assumptions (§3.1), and show how OmniMon achieves resource efficiency and full accuracy (§3.2). We pose the consistency and accountability issues in our design (§3.3).

### 3.1 Design Overview

OmniMon is a network-wide telemetry architecture spanning different entities (i.e., end-hosts, switches, and the

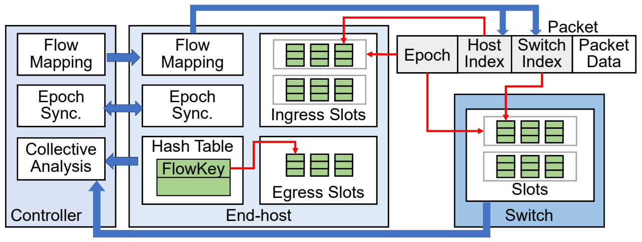


Fig. 2. OmniMon architecture.

controller) in a data center network. It aims for both resource efficiency and full accuracy from an architectural perspective, by re-architecting the collaboration among all entities subject to their resource constraints. It is compatible with existing collaborative telemetry designs. In addition, it also provides a query engine for expressive telemetry (§6).

Coordinating the collaboration among different types of entities is non-trivial due to their heterogeneity in resource availability and in-network visibility (§2.1). To this end, we address three classical issues adapted from building practical distributed systems: (i) How do we assign telemetry operations across different entities (i.e., end-hosts, switches, and the controller)? (ii) How should different entities coordinate among themselves? (iii) How should we achieve reliable coordination in the face of unreliable events?

**Assumptions.** We make several design assumptions.

- We focus on a data center network governed by a single administrative domain, which is a common setting in data centers. Administrators have access to all entities to apply configurations. The controller has full knowledge of the packet processing policies (e.g., routing tables, access control lists, multicast groups, and path changes due to link failures) in each entity.
- We only consider the packets that actually enter the network (i.e., packets dropped in end-hosts are ignored). This can be done by realizing measurement in proper positions of the network stack in each end-host (e.g., QDISC in the Linux kernel). Measuring and dissecting an end-host's network stack is out of our scope.
- The packet format is extensible. During transmissions, packets can embed new information in various unused fields (e.g., reserved bits in VXLAN) in existing packet headers [35], [41]. Recent advances in user-space networking stacks [25] and programmable switch ASICs [27] also allow administrators to readily define customized packet fields. Note that the embedded information should be of small size to limit transmission overhead.

## 3.2 Split-and-Merge Telemetry

Fig. 2 presents the OmniMon architecture. Recall that OmniMon follows a split-and-merge approach for network telemetry (§1). Specifically, it performs flow tracking and resource management via four partial operations, namely *flowkey tracking*, *value updates*, *flow mapping*, and *collective analysis*. It coordinates these partial operations among end-hosts, switches, and the controller, subject to the heterogeneous resource requirements of different entities.

In the following, we describe each partial operation and show how both resource efficiency and full accuracy are simultaneously achieved. We first assume that there is a global clock that precisely synchronizes all entities and there is no packet loss. We later relax these assumptions in §3.3.

**Flowkey tracking.** OmniMon holds a copy of the flowkey of each active flow in the network. When a new flow starts, OmniMon records its flowkey in a hash table in the source end-host. It removes the flowkey when the flow terminates, determined by FIN/RST packets in TCP flows or a long idle time (e.g., 30 minutes [42]).

**Value updates.** OmniMon tracks flow values in both end-hosts (with ample memory space) and switches (with in-network visibility). Each end-host or switch maintains a set of *slots* with multiple counters each. These slots are partitioned into groups for different epochs. Thus, OmniMon can perform value updates in one group of slots for the current epoch, and simultaneously collect the flow values in another group of the previous epoch to perform collective analysis. As end-hosts and switches have different resource requirements, they perform value updates differently.

With sufficient memory space, an end-host dynamically allocates one *dedicated* slot for each flow per epoch. When a flow terminates, its slots will be recycled. An end-host holds two types of slots, namely *ingress* and *egress* slots. It stores the values of an incoming flow destined for itself in an ingress slot, and also stores the values of an outgoing flow originated from itself in an egress slot.

On the other hand, since switches have limited memory space, we allow a slot to be shared by multiple flows. Each switch maps a flow to a subset of slots and updates the counters in all those slots by the flow values. Thus, a slot holds aggregate values from multiple flows. We later recover the per-flow values via collective analysis (see below).

**Flow mapping.** OmniMon coordinates all end-hosts and switches to properly map flows to slots for flowkey tracking and value updates. Its flow mapping design builds on both in-band packet embedding and out-band messages.

*(i) Flow mapping in end-hosts.* OmniMon maps a flow to its slots in both source and destination end-hosts via an index number called the *host index*. When the source end-host applies flowkey tracking for a new flow, it first computes the host index, which denotes the entry position of the flowkey in the hash table. Then it associates the host index with the egress slot of the flow, and embeds the host index and the epoch into each outgoing packet of the flow. The destination end-host extracts the host index and epoch from each received packet, and associates them with the ingress slot. The host index eliminates another hash lookup, which is the major overhead for end-hosts (Exp#1 in §9). Note that the host index is not unique across all the hosts. The end-host of OmniMon can distinguish the flows with the same host index by their flowkeys.

*(ii) Flow mapping in switches.* OmniMon uses an index number called the *switch index* for each flow to determine how the flow maps to multiple slots in a switch. The switch index is collaboratively selected by the controller and source end-hosts. Specifically, the controller divides all the switch indexes into several disjoint subsets equally according to the number of end-hosts. It then pre-assigns one subset to each end-host in the initialization phase. A source end-host chooses one switch index from its assigned subset for a new flow (detailed in §5.2) and embeds the switch

index (together with the host index and the epoch) in each outgoing packet of the flow. When a switch receives a packet, it retrieves the switch index and epoch from the packet and updates the corresponding mapped slots. Such a collaborative approach not only exploits the visibility of the controller to mitigate the hash collisions in switch slots, but also allows the source end-host to quickly choose switch slots without querying the controller.

**Collective analysis.** For each epoch, the controller collects the hash tables (flowkeys) and the slots (values) from all end-hosts and switches. It performs collective analysis to associate each flow with the values recorded in its source and destination end-hosts and each switch the flow traverses, to recover per-flow values from each end-host and switch. Since OmniMon keeps all flowkeys in their source end-hosts, the controller can directly obtain per-flow values (via the host index) from the dedicated slots in the source and destination end-hosts. Per-flow values in switches can also be recovered from the shared slots based on the flowkeys and values in end-hosts. Specifically, the controller knows all flows mapped to a shared slot (via the switch index). If there is no packet loss, the value of the slot is the aggregation of the flows mapped to the slot (e.g., for summable values, the value in the slot equals the sum of per-flow values); otherwise, with packet losses, OmniMon recovers per-flow values via packet loss inference (§5). Note that OmniMon can collect the telemetry data in a timely manner [43] before the switch reuses the slots of one epoch. We forward limited delayed traffic that belongs to the epoch under collection to the controller, incurring negligible bandwidth usage.

**Summary.** OmniMon achieves both resource efficiency and full accuracy. For full accuracy, the controller exactly recovers per-flow values in all end-hosts and switches on a per-epoch basis (i.e., both correctness and completeness are addressed). OmniMon addresses resource efficiency via three aspects: (i) for memory, it maintains flowkeys and dedicated per-flow values in end-hosts only, while keeping only flow values in the shared slots in switches; (ii) for computation, it performs flow mapping operations collaboratively in the controller and all end-hosts without involving switches, while incurring limited overhead (Exp#1 in §9 and Exp#10 in [44]); (iii) for bandwidth, the controller collects hash tables and slots without mirroring all traffic.

Note that such resource usage is insignificant compared to other network functionalities. For end-hosts, per-packet hashing is affordable as shown in prior work [1], [14]; in contrast, network stacks in OS kernels or virtualization frameworks perform much more complicated per-packet processing. For switches, the partial operations of OmniMon incur much less resource usage than standalone telemetry solutions, as shown in our evaluation (§9).

### 3.3 Consistency and Accountability

OmniMon faces two unreliability concerns in its network-wide architectural design: (i) lack of a global clock and (ii) packet losses. We extend OmniMon with *consistency* and *accountability* guarantees, so as to preserve both resource efficiency and full accuracy in the face of unreliability events.

- *Consistency:* Without a global clock, OmniMon provides two consistency guarantees to ensure full accuracy. First, all entities reside at the same epoch at most times. This provides correctness guarantees for per-epoch applications (e.g., anomaly detection (§2.1)). Second, each packet is always monitored at the same epoch during its transmission (even under cross-epoch network delays). Without this guarantee, when there is a difference between the flow values on different entities, we cannot distinguish whether the packet is lost or measured at other epochs. That is to say, this guarantee permits collective analysis to correctly associate per-flow values across entities in a consistent network-wide view of traffic. See §4 for details.
- *Accountability:* If packet losses happen (e.g., due to network congestion or switch failures), OmniMon needs to correctly infer per-switch, per-flow packet losses. It is critical to network diagnosis (§2.1), and maintains full accuracy in the collective analysis. See §5 for details.

## 4 CONSISTENCY

OmniMon achieves consistency via network-wide epoch synchronization, with the goal of synchronizing all entities at the same epoch and monitoring each packet at the same epoch in its transmission. We state the challenge of realizing consistency in OmniMon (§4.1), and propose a new consistency model that closely matches our objective yet can be feasibly realized (§4.2).

### 4.1 Challenge

While consistency has been well studied in distributed computing research, providing consistency guarantees for network telemetry with both resource efficiency and full accuracy remains unexplored. Existing consistency protocols mainly target either *strong consistency* or *weak consistency*. We argue why each of them is difficult to achieve in our context via the examples in Fig. 3, in which two entities (end-hosts or switches) $e_1$ and $e_2$ exchange packets.

**Strong consistency.** Strong consistency aligns epoch boundaries and ensures that every packet is observed by *all* entities at the same epoch. For example, in Fig. 3(a), both $e_1$ and $e_2$ see Packet A at Epoch 1 and Packet B at Epoch 2. However, strong consistency protocols (e.g., Paxos [45]) are complex to implement in switches and trigger multiple rounds of out-band messages for synchronization.

**Weak consistency.** Weak consistency synchronizes per-packet epochs in a *best-effort* manner, so the epochs among entities may diverge and a packet may be monitored at different epochs. In weak consistency, each entity renews its epoch via a local clock and embeds the current epoch in outgoing packets (as in Lamport Clock [46] and Distributed Snapshots [47], [48]). When an entity receives a packet, it updates its current epoch with the packet's embedded epoch if the latter is newer. Otherwise, the entity updates the packet's embedded epoch with its epoch before forwarding.

Such in-band synchronization incurs limited costs. However, entities may observe a packet at different epochs as the embedded epoch can be modified along the packet transmission path. Also, a heavily clock-delayed entity never updates its epoch until it receives a packet with a newer epoch, causing *unbounded* deviations among epoch boundaries. For example, in Fig. 3(b), both $e_1$ and $e_2$ synchronize

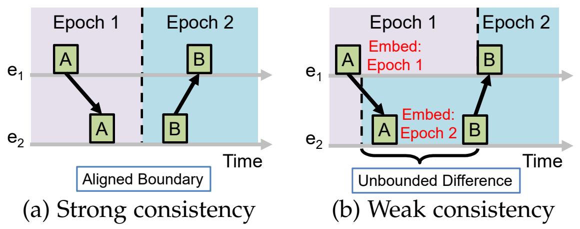


Fig. 3. Existing consistency models.

their epochs via in-band packets. However, $e_1$'s local clock is delayed and remains at Epoch 1 until it receives Packet B from $e_2$. Furthermore, $e_1$ embeds Epoch 1 in Packet A, while $e_2$ embeds Epoch 2. Thus, Packet A may be inconsistently treated in different epochs during its transmission.

## 4.2 Hybrid Consistency

OmniMon realizes a *hybrid* consistency model, which ensures strong consistency at most times and is downgraded to weak consistency for only a small bounded time period (e.g., tens of $\mu$s). Our idea is that OmniMon not only embeds epoch information in in-band packets (as in weak consistency), but also leverages the controller's coordination for network-wide epoch synchronization.

**Design overview.** OmniMon decides the epoch for each packet *once* during its transmission. Each end-host maintains a local clock and decides its current epoch. When a source end-host sends a packet, it embeds its current epoch into the packet. All switches along the path and the destination end-host monitor the packet in the corresponding slots based on the embedded epoch. The destination end-host further updates its current epoch if the embedded epoch is newer. Note that switches do not maintain local epochs to keep their processing pipelines simple.

When an end-host updates its epoch via its local clock, it sends the new epoch to the controller, which relays the new epoch to all other end-hosts via out-band messages. Thus, all end-hosts can keep up-to-date epochs even though their local clocks are delayed or they do not receive any packets with newer embedded epochs. With this controller-assisted approach, it takes at most two hops to propagate a new epoch from one end-host to another. Thus, the time difference between the epoch updates of any two end-hosts is roughly twice the path delay between the controller and the end-hosts, which is around tens to hundreds of microseconds in practice (e.g., less than 60 $\mu$s for 1,024 end-hosts in our evaluation) and is much smaller than the epoch length (e.g., tens or hundreds of milliseconds [1], [13]).

The controller maintains its local epochs. It updates and relays the epoch only when receiving an out-band message with a newer epoch. We can show that each end-host triggers only $O(1)$ out-band messages at each epoch (see the proof in [44]). Also, the variations of in-band traffic have limited impact on the consistency model (Exp#3 in §9).

Fig. 4 depicts the hybrid consistency model. At most times, all end-hosts (e.g., $h_1$ and $h_2$ in the figure) reside at the same epoch. However, end-hosts may still reside at different epochs at times (e.g., when $h_2$ receives Packet C), in which case OmniMon only ensures weak consistency. Nevertheless, the duration of weak consistency is limited by the controller-assisted synchronization. Also, OmniMon

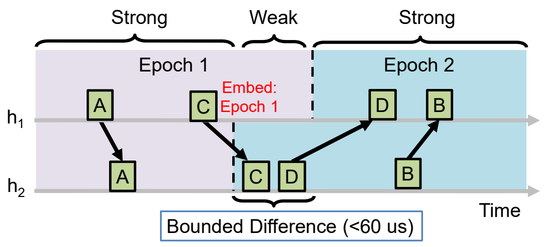


Fig. 4. Hybrid consistency in OmniMon.

**Algorithm 1** Epoch updates in hybrid consistency

```
End-host: local epoch $e_h$
1: function ON_NEW_EPOCH_LOCAL_CLOCK(epoch $e$)
2:     if $e > e_h$ then
3:         $e_h = e$, and send $e_h$ to the controller
4: function ON_NEW_EPOCH_FROM_EMBEDDED_PACKET(epoch $e$)
5:     if $e > e_h$ then
6:         $e_h = e$
7: function ON_NEW_EPOCH_FROM_CONTROLLER(epoch $e$)
8:     if $e > e_h$ then
9:         $e_h = e$
Controller: local epoch $e_c$
10: function ON_NEW_EPOCH(epoch $e$)
11:     if $e > e_c$ then
12:         $e_c = e$
13:         for all end-host $h$ do
14:             Send $e_c$ to $h$
```

ensures that every packet is monitored at the same epoch along the path, even when its source and destination end-hosts are not yet synchronized. For example, Packet C is sent by $h_1$ at Epoch 1 but received at Epoch 2 by $h_2$. However, both $h_1$ and $h_2$ monitor Packet C at Epoch 1 (its embedded epoch). In technical report [44], we formally prove the properties of hybrid consistency.

**Algorithm.** Algorithm 1 shows the epoch update operations among the controller and all end-hosts for realizing hybrid consistency. An end-host triggers an update of its epoch by its local clock (lines 1-3), a newer embedded epoch of a received packet (lines 4-6), or a newer epoch from an out-band message from the controller (lines 7-9). If the end-host triggers its epoch update by its local clock, it also sends the new epoch to the controller. For the controller, if it receives a newer epoch from an end-host, it updates its local epoch and sends it to all other end-hosts (lines 10-14).

**Remark.** Algorithm 1 is in essence a variant of Lamport Clock (LC) [46], but differs from LC in two aspects. First, LC solely relies on packet embedding for epoch updates, while Algorithm 1 relies on both packet embedding and the controller's out-band messages to speed up epoch updates. Second, LC involves all entities to embed new epochs for value updates, while Algorithm 1 determines the epochs via the controller and all end-hosts, but not switches. Thus, LC only provides weak consistency, while Algorithm 1 provides hybrid consistency.

**Distributed controller.** As the network size grows, OmniMon can deploy multiple instances of the controller, each of which manages a finite number of end-hosts. We extend our epoch update mechanism to enable OmniMon to synchronize the epochs among the controller instances. Specifically, in addition to the controller instances that connect to end-hosts (referred to as the *leaf instances*), we also configure a controller instance, called the *root instance*, that communicates with all leaf instances. Once any leaf instance enters a new epoch (on receiving a message from one of its connected end-hosts), it notifies the root instance, which then broadcasts this new epoch to other leaf instances. Each leaf instance also sends the epoch update to its connected end-hosts. This two-layer mechanism takes at most four hops (i.e., two between a leaf instance and an end-host, and another two between a leaf instance and the root instance) to propagate a new epoch. Thus, the time difference between the epoch updates of any two end-hosts remains small and bounded. Our evaluation (Exp#4) shows that a controller instance can manage 1,024 end-hosts, so our two-layer epoch update mechanism can support the large-scale data centers.

## 5 Accountability

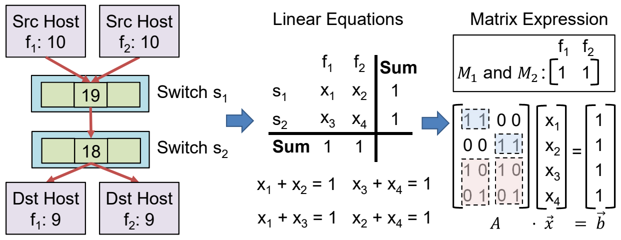
OmniMon achieves accountability by inferring the exact traffic losses on a per-switch, per-flow basis, such that it can recover per-flow values from the *shared slots* in each switch. Our loss inference is related to classical network tomography [49], [50], yet the latter often focuses on only end-host information due to limited in-network visibility and aims to minimize accuracy loss. In contrast, our inference model is network-wide (covering all end-hosts and switches) and aims for full accuracy. Here, we target *additive* flow values, and describe how to convert non-additive flow statistics to additive values in our technical report [44].

### 5.1 Problem

**Motivating example.** Without loss of generality, we focus on packet counts. We formulate the packet loss inference problem as solving a system of linear equations (or *linear system* in short), where the variables represent individual per-switch, per-flow packet losses. However, the linear system may have multiple feasible solutions (i.e., we cannot uniquely infer the exact packet losses). We consider a toy example (Fig. 5) that motivates the problem. Suppose that there are two lossy flows $f_1$ and $f_2$, both of which traverse switches $s_1$ and $s_2$ and share the same slots in both switches. Based on the flow values in the source and destination end-hosts, we know that each flow loses exactly one packet. Also, by checking the aggregate flow value in each switch, we know that each switch drops exactly one packet. However, we cannot tell which flow's packet is dropped in each switch: either $s_1$ drops $f_1$'s packet and $s_2$ drops $f_2$'s packet, or $s_1$ drops $f_2$'s packet and $s_2$ drops $f_1$'s packet.

The above example shows an extreme case that even only two lossy flows can make exact packet loss inference infeasible. Nevertheless, by including the characteristics of data center networks, we can design a *collaborative flow mapping* algorithm (§5.2) that maps flows to different switch slots. The algorithm simplifies the linear system to avoid the extreme case in Fig. 5. It also returns a unique solution of per-switch, per-flow packet losses for the common case.

**Formulation.** Let $n$ be the number of lossy flows $(f_1, \cdots, f_n)$ and $m$ be the number of switches $(s_1, \cdots, s_m)$. Let $k_t$ be the number of slots of switch $s_t$ to which the lossy flows are mapped $(1 \le t \le m)$. For example, Fig. 5 has $n = 2$, $m = 2$, $k_1 = 1$, and $k_2 = 1$. We formulate the network-wide packet loss inference problem as a linear system as follows.



| | $f_1$ | $f_2$ | Sum |
|---|---|---|---|
| $s_1$ | $x_1$ | $x_2$ | 1 |
| $s_2$ | $x_3$ | $x_4$ | 1 |
| Sum | 1 | 1 | |

$x_1 + x_2 = 1$ $x_3 + x_4 = 1$

$x_1 + x_3 = 1$ $x_2 + x_4 = 1$

$$M_1 \text{ and } M_2: \begin{array}{cc} f_1 & f_2 \\ \hline 1 & 1 \end{array}$$

$$\underset{A}{\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}} \cdot \underset{\vec{x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}} = \underset{\vec{b}}{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}}$$

Fig. 5. Motivating example of accountability.

- *Variables:* Let $x_i$ $(1 \le i \le mn)$ be a variable that quantifies the per-switch, per-flow packet loss (in number of lost packets), such that $x_1$ to $x_n$ are respectively the packet losses of $f_1$ to $f_n$ in $s_1$, $x_{n+1}$ to $x_{2n}$ are respectively the packet losses of $f_1$ to $f_n$ in $s_2$, and so on. For example, in Fig. 5, we define four variables: $x_1$ and $x_2$ are respectively the packet losses of $f_1$ and $f_2$ in $s_1$, while $x_3$ and $x_4$ are respectively the packet losses of $f_1$ and $f_2$ in $s_2$.
- *Equations:* We formulate two sets of equations. First, each slot induces an equation for the lossy flows that are mapped to the slot itself (i.e., $\sum_{t=1}^{m} k_t$ equations in total). For example, in Fig. 5, the slot in $s_1$ induces $x_1 + x_2 = 1$, while the slot in $s_2$ induces $x_3 + x_4 = 1$. Second, each flow induces an equation for its total packet loss based on the difference between the flow values in the source and destination end-hosts (i.e., $n$ equations in total). For example, in Fig. 5, $f_1$ induces $x_1 + x_3 = 1$, while $f_2$ induces $x_2 + x_4 = 1$.

Let $M_t$ $(1 \le t \le m)$ be a $k_t \times n$ 0-1 matrix that specifies the mappings of the $n$ lossy flows to the $k_t$ shared slots in switch $s_t$. $M_t(i, j)$ is one if flow $f_j$ $(1 \le j \le n)$ traverses $s_t$ and is mapped to slot $i$ $(1 \le i \le k_t)$ in $s_t$; or zero otherwise. Let $I$ be the identity matrix of size $n$, in which each row indicates the packet loss variables of one lossy flow along $m$ switches. We express the linear system as $A \cdot \vec{x} = \vec{b}$ (e.g., see Fig. 5), where $A$ is a $(\sum_{t=1}^{m} k_t + n) \times mn$ matrix given by:

$$A = \begin{bmatrix} M_1 & 0 & \cdots & 0 \\ 0 & M_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & M_m \\ I & I & \cdots & I \end{bmatrix},$$

$\vec{x}$ is a column vector of size $mn$ with all $x_i$'s, and $\vec{b}$ is a column vector of size $\sum_{t=1}^{m} k_t + n$, composed of $\sum_{t=1}^{m} k_t$ packet losses of all $\sum_{t=1}^{m} k_t$ slots in $m$ switches as well as $n$ total packet losses of all $n$ flows. Our goal is to solve for $\vec{x}$ given $A$ and $\vec{b}$.

**Rank analysis.** The linear system has a unique solution if and only if the rank of $A$ equals the number of variables $mn$. The rank of $A$ depends on $M_t$ and $I$. Clearly, $I$ has rank $n$. Suppose that we construct a new matrix $M$ of size $(\sum_{t=1}^{m} k_t) \times n$, that comprises the rows of each $M_t$ $(1 \le t \le m)$. Let $r$ be the rank of $M$. We can prove that the tight upper bound for the rank of $A$ is $r(m-1)+n$ (see our technical report [44]). Thus, we can pose the sufficient and necessary condition for accountability (i.e., the exact packet losses can be inferred):

$$\textit{Accountability condition:} \quad r(m-1) + n = mn.$$

If $m = 1$, the accountability condition always holds; otherwise if $m > 1$, it holds only when $r = n$. Note that each row of $M$ represents a flow mapping to a slot in a switch. This creates a connection between accountability and the flow mappings to the slots in different switches specified by $M$, thereby motivating us to design a flow mapping algorithm to return an $M$ that fulfills the accountability condition.

**Challenge.** Unfortunately, the accountability condition generally fails to hold. For example, in Fig. 5, the matrix $M$ formed by $M_1$ and $M_2$, has a rank $r = 1$, yet there are $n = 2$ flows. In general, the rank $r \leq \min\{\sum_{t=1}^{m} k_t, n\}$. If the number of mapped slots ($\sum_{t=1}^{m} k_t$) is less than that of flows ($n$), it is inevitable that $r < n$ for any flow mapping.

## 5.2 Collaborative Flow Mapping

**Observations.** We show how we leverage the characteristics of data center networks to make the accountability condition hold for the common case. We make two observations, which have been confirmed by recent studies on production data centers [18], [51], [52], [53], [54], [55].

*(i) Traffic locality.* Modern data centers exhibit strong traffic locality, in which flow trajectories are limited within a rack or a cluster [52], [53]. For instance, about 85% of intra-data-center traffic at Facebook stays within a cluster (c.f. Table 3 in [53]). Thus, a lossy flow often traverses only a few switches, and many variables in $\vec{x}$ can be set to zero to make the rank requirement of $A$ less stringent (i.e., less than $mn$).

*(ii) Loss sparsity.* Packet drops are sparsely distributed across a data center network, given that recent congestion control mechanisms [51] and failure mitigation mechanisms [18], [54], [55] greatly suppress packet losses. Such loss sparsity implies that packet losses likely occur in exactly one switch within a local region (e.g., a small group of racks or clusters). In other words, for all local lossy flows that traverse solely within a region, their packet losses are attributed to a single switch. Thus, we can infer the variables in $\vec{x}$ for such local flows (recall that we can solve for the unique solution with $m = 1$ for such local flows based on the accountability condition). This further relaxes the rank requirement of $A$.

**Design.** The above two observations greatly simplify the linear system by reducing the number of variables in $\vec{x}$. For the remaining variables in $\vec{x}$, we aim to construct via flow mapping a new matrix $M$ that has a maximum rank $r$, so that the simplified linear system can return a unique solution with a high likelihood. However, this problem remains non-trivial, since lossy flows are unpredictable and we cannot tell in advance how lossy flows are mapped to different slots.

OmniMon addresses this issue by *mapping a flow to $d > 1$ slots in each switch*, inspired by Bloom filters [56]. Such a design has two benefits. First, the number of slots mapped by lossy flows increases and has an expected value equal to $nd$ [44]. Second, the number of possible flow mappings increases to $O(K^d)$, where $K$ is the total number of available slots in a switch, so that the likelihood that two lossy flows are mapped to the same $d$ slots decreases. Thus, the rank of $A$ increases, as the number of mapped slots (i.e., rows of $A$) increases and the number of overlapped flow mappings (i.e., linearly dependent columns) decreases. For example, the extreme case in Fig. 5 can be avoided.

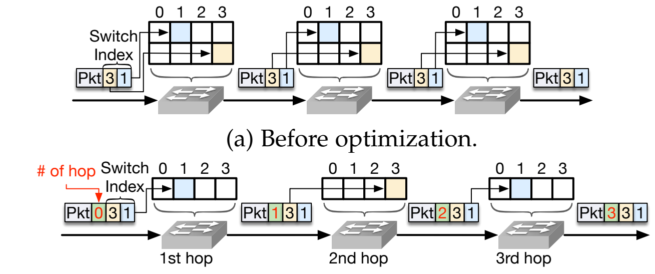


Fig. 6. Network-wide mapping optimization.

OmniMon takes a two-phase collaborative approach to ensure that any new flow is mapped to a different set of $d$ slots. In the first phase, the controller generates all $\binom{K}{d}$ possible flow mappings and partitions the flow mappings for all end-hosts during their startups, such that each end-host receives a distinct list of flow mappings. In the second phase, a source end-host selects an unassigned flow mapping when it sees a new flow, and recycles any assigned flow mapping upon flow termination, during flowkey tracking (§3.2). Note that if there are too many active flows, the source end-host may run out of unassigned flow mappings. In this case, we choose the least recently used flow mapping.

**Summary.** OmniMon simplifies the linear system through the characteristics of data center networks to infer the exact packet losses. We summarize the steps below.

- *Step 1:* Identify any lossy flow whose difference of flow values in source and destination end-hosts is non-zero.
- *Step 2:* Construct $M_t$ (and hence $A$) and $\vec{b}$.
- *Step 3:* Refine the variables in $\vec{x}$. With traffic locality, if a flow does not traverse a switch, we set the corresponding variable as zero. With loss sparsity, we examine every local region (by racks or clusters). If a region has exactly one switch with packet losses, we identify all local flows in this region and solve for the unique solution for the corresponding variables of these local flows.
- *Step 4:* Solve $A \cdot \vec{x} = \vec{b}$ for the remaining variables.

## 5.3 Network-wide Optimization

Mapping a flow to $d$ slots at a switch incurs non-trivial resource overheads in some switch devices. For example, in Tofino switches, a packet can only access a single location of a slot array. In this case, we have to maintain $d$ duplicated slot arrays, which consume a large amount of memory and stateful ALUs.

**Solution.** OmniMon reduces the memory overheads in the data plane by utilizing knowledge about the number of switch hops during packet transmission. Specifically, we still map a flow to $d$ slots, but update only one slot at each switch. Which slot is updated is determined by the number of switch hops through which the packet passes. We embed a *hop* field in the extended packet header of OmniMon to keep track of how many switches the packet travels. By taking the remainder of $d$ based on the *hop* count, the switch updates the slot indicated by the $hop\%d^{th}$ switch index. Fig. 6 compares the case of $d = 2$ without and with our hop-based mapping optimization, where the end-host embeds two switch indexes 1 and 3 into the packet. Before optimization, each switch needs to allocate two register

```
1  Q1 = PacketStream
2    .filter(pkt => pkt.ip.proto == TCP)
3    .map(pkt => (ip.dstIP, ip.pktlen))
4    .reduce(keys=(dstIP), func=sum, vname="bytes")
5
6  Q2 = PacketStream
7    .filter(pkt => pkt.ip.proto == TCP)
8    .distinct((pkt.ip.dstIP, pkt.ip.srcIP, pkt.tcp.
       srcPort))
9    .map((dstIP, srcIP, srcPort) => (dstIP,cnt), cnt
       =1)
10   .reduce(keys=(dstIP), func=sum, vname="flows")
11
12 sa = Q1.join(Q2, Q1.dstIP == Q2.dstIP)
13   .map((dstIP, bytes, flows) => (dstIP, val), val=
       bytes/flows)
14   .filter((dstIP, val) => (val > Threshold))
```

Query 1. Detect Slowloris Anomaly.

arrays (`arr1` and `arr2`) with the same size and update `arr1`[1] and `arr2`[3] (Fig. 6(a)), respectively. Fig. 6(b) shows that each switch only deploys one register array `arr` after optimization. The switch updates `arr`[1] on odd hops and `arr`[3] on even hops.

## 6 OMNIMON QUERY ENGINE

OmniMon provides a query engine for network administrators to ease development efforts. It contains a declarative language to describe high-level telemetry applications (§6.1) and a compiler that translates the high-level applications into OmniMon data plane and control plane (§6.2).

### 6.1 Language

**Data stream abstraction.** We follow the data stream abstraction in recent query-driven telemetry systems [19], [23], [57]. Specifically, OmniMon abstracts network traffic as a stream of key-value tuples. The administrators can use data-stream operators (see next) to process these items. A query task is defined as a directed acyclic graph (DAG) in which a vertex is an operator. An edge indicates that the downstream operator consumes the output tuples of the upstream operator.

**Operators.** As summarized by BeauCoup [58], most network queries follow the same pattern: (Step 1) filter out useless packets and map remaining packets into groups; (Step 2) count packets in each group; and (Step 3) perform analysis on the counting results to determine whether to report an event. To realize this pattern, our language provides the following operators:

- `filter`: It examines each input tuple and preserves those satisfying certain conditions. This operator is often applied in Step 1 and Step 3.
- `distinct`: It groups input tuples based on user-defined key fields and preserves the first tuple for each group. The `distinct` operator is mostly used in Step 1 to drop duplicated tuples.
- `map`: It transforms each input tuple into an output tuple by inserting, deleting, and modifying fields in the tuple. The `map` operator is typically employed in Step 1 (preprocessing) and Step 3 (generating final output).
- `reduce`: It categorizes tuples according to user-defined key fields and sums up the value field for each category. The `reduce` operator is responsible for the counting operation of Step 2.

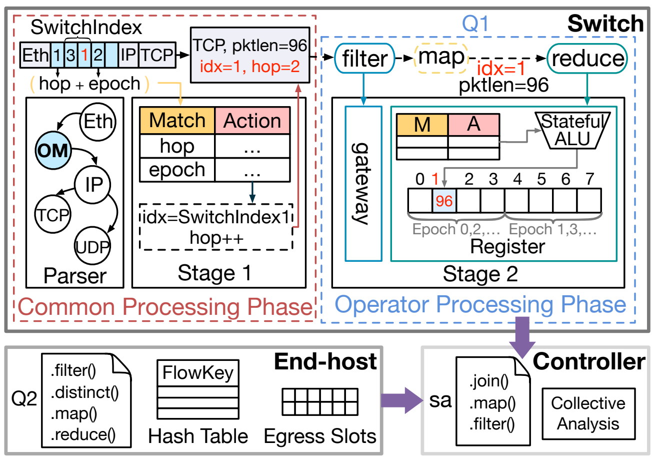


Fig. 7. Slowloris anomaly detection in OmniMon.

- `join`: It merges the input stream with tuples of another stream containing the same tuple key. The `join` operator is used by Step 3 to report events.

**Query example.** We demonstrate an example with a query namely slowloris anomaly detection [59] (Query 1). This query detects IP addresses that maintain multiple TCP flows but with low traffic volume per flow. Firstly, the query uses two sub-queries to count the total number of TCP flows maintained by each host (lines 1-4) and the total traffic received by each host (lines 6-10), respectively. Specifically, `Q1` first applies a `filter` operator over the entire packet stream to select TCP packets (lines 1,2). `Q1` then uses the `map` and `reduce` operators to accumulate the total size of all TCP flows received by each destination host (lines 3,4). At the same time, `Q2` preserves only one packet of each TCP flow via the `filter` and `distinct` operators at first (lines 6-8). Next, `Q2` determines the number of TCP flows on each host by counting the number of remaining packets destinated to the same destination using the `map` and `reduce` (lines 9,10). This query then merges the tuples with the same destination address in the results of `Q1` and `Q2` via the `join` operator (line 12). Finally, it applies the `map` and `filter` operators to detect hosts whose average bytes per flow exceeds `Threshold` (lines 13,14).

### 6.2 Compiler

The compiler translates the query operators into programs running atop OmniMon components, i.e., flow-key tracking at end-hosts, value updating at end-hosts and switches, and collective analysis at the controller. Recall that a query application consists of multiple operator sequences (e.g., three sequences in Query 1). Our compiler assigns an operator sequence in one type of entity: (i) if a sequence contains a `join` operator, OmniMon executes the sequence in the controller because the `join` operator is too heavy to be realized in the data plane; (ii) sequences with `distinct` operator is executed at end-hosts because we need precise flow-key tracking to eliminate duplicate tuples; (iii) other sequences are placed at switches to exploit the high-speed processing of switching pipeline. Note that some prior works (e.g., Sonata [23]) place queries at the granularity of operators to offload as many operators as possible. However, this incurs excessive communication overheads if two adjacent operators are placed in different entities (see Exp#10 in [44]).

Thus, we choose sequence-based placement to make a trade-off between performance and communication efficiency.

**Switch programs.** We focus on the translated programs at switches because switches typically impose more strict programming and resource models (see §2.1). As shown in Fig. 7, a switch program contains two phases. The first phase performs common processing such as packet parsing, updating `hop` field, and locating the physical address. The physical address is located by the `epoch`, `hop`, `switch_index` embedded in the packet header and the number of slots for each epoch. Assuming we deploy four epochs in the switch and the number of slots for one epoch is 512, we have $\texttt{physical\_address} = (\texttt{epoch}\&3) * 512 + (\texttt{hop}\%2 == 0?\texttt{switch\_index0} : \texttt{switch\_index1})$. This incurs high stage resource usage in the pipeline-based programmable switch. Therefore, we use a match-action table (MAT) which matches the `hop` and `epoch` fields and performs the physical address calculation in one action. This approach only requires one stage, leaving enough resources for value updates of different queries.

The second phase then performs the actual operators with in-switch computational resources, which are elaborated as follows. Here, we do not apply existing P4 program optimizations [60], [61] to our compiler. The reason is that these optimizations focus on complex and redundant operations, whereas OmniMon only performs simple operations (i.e. value updating) at switches but leaves complex operations (e.g., flowkey tracking, flow mapping) at end-hosts.

- `filter`: The compiler utilizes gateway tables or CMPs in the Stateful ALU to perform conditional judgment in the `filter` operator. These two hardware resources do not require the extra stage and on-chip memory for table entry, compared to the realization of the MAT.
- `map`: The compiler adopts different strategies according to the way the `map` operator updates the tuple. For adding a new field, the compiler defines the metadata of the added one (e.g., `cnt` and `val` in Query 1) in the data plane program in advance. For deleting useless fields, OmniMon views it as a logical function because it has no impact on subsequent processing. For modifying fields, the compiler translates the addition and subtraction of integers and bit operations into switch action primitives.
- `reduce`: OmniMon compiles the `reduce` operator into a register array which maintains the counting results and a stateful ALU which executes the counting operations. Since the physical address location is complete in the common processing phase, the stateful ALU only needs to access the physical address and sums the flow value.

**Discussion.** Note that our compiler is significantly different from that of existing query-driven telemetry [19], [23], [57], although our telemetry language is similar to them. In particular, OmniMon adopts sequence-based offloading, which translates the operator sequence into the value updating operation at switches. Sonata [23] abstracts the switch and end-host into a big streaming processor, whose compiler focuses on making partition plan for operator placement. Marple [19] treats the switch memory as the cache of the key-value store on an end-host. Thus, its compiler is responsible for the implementation of data-plane cache.

**Multiple queries.** The query engine of OmniMon can compile multiple queries into one switch program. In particular, OmniMon uses different on-chip registers to store statistics from different queries, whose switch indexes are all the same. Here, the same positions of different registers can be logically viewed as a big slot.

**Example.** Fig. 7 shows the compiled slowloris detection (Query 1). `Q1` is offloaded to the switch. In the common processing phase, the *Parser* extracts the OmniMon extend header and other fields. Then, the MAT at Stage 1 matches the `hop` and `epoch` field, to calculate the physical address and update the `hop` field. In Phase 2, the `filter` operator of `Q1` (line 2) is compiled into a gateway table. The `reduce` operator (line 4) utilizes one stateful ALU (invoked by the MAT on Stage 2) to update two sets of slots (sharing one register array) according to the `idx` metadata field. Here, we omit the compilation of the `map` operator (line 3), because removing irrelevant fields is a logical function at switches. Moreover, the other operator sequences `Q2` and `sa` are executed at end-hosts and controller, respectively.

## 7 Discussion

We discuss some practical deployment issues of OmniMon.

**Scalability.** As the network size increases, OmniMon needs more resources (e.g., memory in switches, CPU power in end-hosts and the controller, etc.) to achieve full accuracy. However, its resource usage required for full accuracy grows at a much lower rate than the growth of the network scale. For end-hosts, since flows are independent, OmniMon can process flows in different CPU cores when a single CPU core cannot process all flows. It remains much more resource-efficient compared to the processing overhead of network stacks, as its hash computations and value updates incur low overhead in commodity CPUs [1]. For switches, as $K$ slots provide $\binom{K}{d}$ flow mappings, a small increment of $K$ can handle much more flows when $d > 1$. OmniMon remains accurate even when the actual number of flows exceeds $\binom{K}{d}$ (Exp#5 in §9). For the controller, it is a logical entity that can support multiple physical servers (§2.1). Also, it only relays limited out-band traffic even for a large number of end-hosts (§4.2), and can accelerate the linear system solving with multiple CPU cores [62], [63].

**Parameterization.** OmniMon allows simple parameter configurations. For end-hosts, we can adjust resources on demand. For switches, we reuse the slots for different epochs, where the maximum number of epochs being tracked and the number of slots per epoch need to be configured.

- The number of tracked epochs is derived based on the epoch length and the tolerable network delay, both of which are configurable. Here, we adjust the number of epochs maintained at end-hosts according to switch setup, i.e., each end-host tracks one more epoch to tolerate the network delay. For example, if we use four epochs of length 100 ms each, we can track the packets with a network delay of up to 400 ms; any packet that is delayed by more than 400 ms is not monitored. Note that since the network delay in the data center is very short, OmniMon can cover common delays by setting the maximum number of tracked epochs and the epoch length carefully.

- The number of slots per epoch depends on the maximum number of flows being tracked (§5.2). Recall that $K$ slots imply $\binom{K}{d}$ possible flow mappings if we map each flow to $d$ slots. Given the maximum number of flows $n$, we select $K$ and $d$ such that $\binom{K}{d} \geq n$. We now choose $d = 2$ (a larger $d$ implies more computational overhead in switches).

**Generality.** OmniMon supports various combinations of packet fields as flowkey definitions, as well as different types of flow statistics (e.g., the 344 statistics in [64], [65], [66], [67]). Since OmniMon works at the granularity of epochs, it cannot perform fine-grained collective analysis for packet timestamps, which requires clock synchronization among end-hosts. However, common synchronization protocols are significantly affected by the workload, whose accuracy varies from hundreds of nanoseconds to hundreds of microseconds [68], [69]. Instead, OmniMon now provides coarse-grained timestamp measurement. For example, it estimates the flow completion time in number of epochs by tracking the first and last epochs where a flow occurs.

**Incremental deployment.** OmniMon now assumes full deployment across end-hosts and switches. We can incrementally deploy OmniMon on a subset of end-hosts and switches of interest (e.g., core switches) for partial measurement (i.e., only the traffic traversing the deployed devices is measured). Characterizing the trade-off between incremental deployment and full accuracy is our future work.

**Failure handling.** OmniMon needs to handle the failures for different types of entities during measurement. For end-hosts, OmniMon discards the switch slots of any failed end-host, as the flow mappings of those slots are assigned and known by the controller (§5.2). Recovering end-host information is out of our scope, yet it can be done with OS-level fault-tolerance mechanisms [70], [71]. For switches, OmniMon localizes switch failures via loss inference (Exp#11 in technical report [44]). For the controller, its fault tolerance can be achieved via multiple servers (§2.1).

## 8 Implementation

We implement a prototype of OmniMon. We present the pseudo-code of all its operations in technical report [44].

**End-host.** We leverage DPDK [25] to remove kernel-space overhead. We use `mTCP` [72], a user-space network stack, to communicate between end-hosts and the controller. We implement different operations (e.g., packet processing, result reporting, and epoch synchronization) in distinct threads, and encapsulate them as a library that can be plugged into any platforms (e.g., OVS-DPDK [73] and PktGen [74]).

**Switch.** We implement switch operations in P4 [26] and place the embedded fields between the Ethernet and IP headers. We support both *per-switch* and *per-port* deployments; in per-port deployment, each port can be viewed as a logical switch. We allocate a region of registers for all slots, such that a packet can locate its slots in a region based on its embedded switch index, epoch, and switch port (in per-port deployment). For efficient slot location, we precompute the partial results of some dependent operations (e.g., modulo, left-shift, and add) and store them in match-action tables. For each packet, we perform a table lookup for partial results and compute the final results in ALUs.

**Controller.** We connect the controller to each end-host and switch with `mTCP` [72] and `ZeroMQ` [75], respectively. The controller parallelizes its operations via multi-threading. It solves the linear system (§5) with the `eigen` library [76].

## 9 Evaluation

We conduct testbed experiments that compare OmniMon with 11 state-of-the-art telemetry designs in various aspects. We summarize our findings for OmniMon:

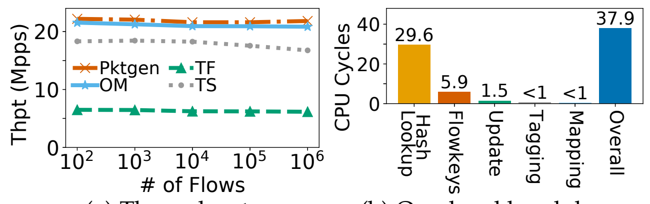
- It achieves higher throughput than Trumpet [1], an end-host-based telemetry system (Exp#1).
- It consumes less switch resource usage than four sketch solutions (which have non-zero errors) [13], [15], [16], [28], while achieving zero errors (Exp#2).
- It completes controller operations in limited time even with $10^6$ flows (Exp#3).
- It incurs less switch resource usage and smaller epoch boundary deviations than Speedlight [48], a synchronization solution for network telemetry (Exp#4).
- It infers 99.7% of packet losses in an extreme case (Exp#5).
- It consumes much less resource usage, while achieving better scalability, than SwitchPointer [24], even when the number of flows increases to $10^6$ (Exp#6).
- It reduces switch memory overhead by around 50% by network-wide mapping optimization (Exp#7).
- It realizes 14 telemetry applications with similar lines of code to Marple [19] and Sonata [23] (Exp#8).

In our technical report [44], we evaluate OmniMon via simulation in large-scale deployment. It achieves perfect diagnosis in four types of packet losses and outperforms another diagnosis system 007 [34] (Exp#9). It supports 11 anomaly detection applications, in which it incurs less overhead than Marple [19] and Sonata [23], while achieving zero errors (Exp#10). It supports performance evaluation for load balancing algorithms, and locates per-link top flows more accurately than HashPipe [9] (Exp#11). Finally, we provide a breakdown of switch resource usage for Exp#2, and per-application resource-accuracy trade-off for Exp#10.

### 9.1 Setup

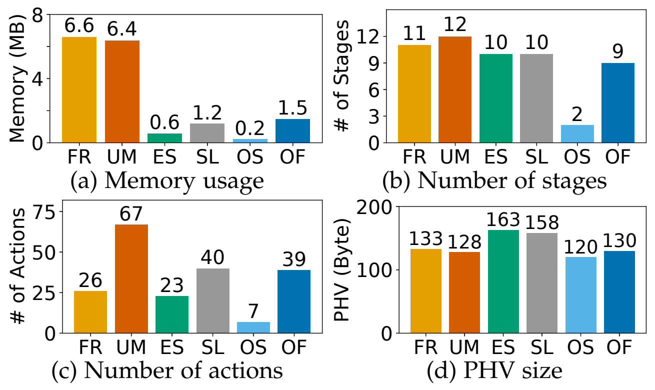
**Testbed.** We deploy our OmniMon prototype in eight servers and three Barefoot Tofino switches [27]. Each server has two 12-core 2.2 GHz CPUs, 32 GB RAM, and a 40 Gbps NIC, while each switch has 32 100 Gb ports. We deploy an end-host in each server and co-locate the controller with one end-host server. We deploy two switches as edge switches connected by four servers each and the remaining switch as the core switch connected by both edge switches.

**Workloads.** We pick a one-hour trace in CAIDA 2018 [77]. To simulate the locality of data center traffic, we map the IP addresses in the trace to our end-hosts and generate a workload in which 85% of flows are edge-local, as reported in [53]. We use different epoch lengths, such that the number of active flows per end-host ranges from $10^3$ to $10^6$ (much more than reported in [51], [53]). We use PktGen [74] to generate trace workloads in end-hosts. To remove disk I/O overhead, each end-host loads traces (with modified IP addresses) into the memory buffer of PktGen, and emits traffic as fast as possible to stress-test our prototype.



(a) Throughput (b) Overhead breakdown

Fig. 8. (Exp#1) End-host overhead.



(a) Memory usage (b) Number of stages (c) Number of actions (d) PHV size

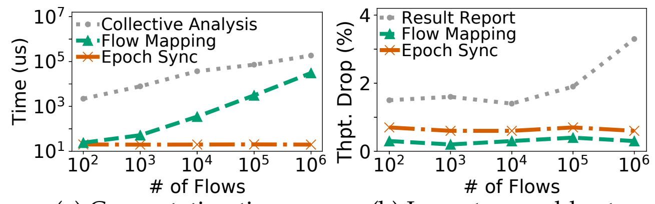
Fig. 9. (Exp#2) Switch resource usage.

**Methodology.** OmniMon counts nine types of per-flow statistics: packet counts, byte counts, SYN counts, FIN/RST counts, SYN-ACK counts, ACK counts, 40-byte packet counts, number of packets with special strings (e.g., "zorro"), and the latest epoch of a flow. Each switch deploys four groups of slots. Each group has 3,072 slots for one epoch, and is recycled when an epoch completes. We map each flow to $d = 2$ slots (§7). Thus, there are $\binom{3072}{2} \approx 4.7 \times 10^6$ flow mappings per epoch, which sufficiently accommodate all active flows (note that a flow appears in both source and destination end-hosts, so there are at most $4 \times 10^6$ active flows in the network assuming that each of the eight end-hosts has at most $10^6$ active flows). Each plot shows the average results over 100 runs, and omits error bars as they are negligible.

## 9.2 Results

**(Exp#1) End-host overhead.** We first evaluate the end-host overhead by comparing OmniMon with Trumpet [1], a zero-error end-host-based telemetry system. Note that Trumpet is designed for only end-hosts, while OmniMon also provides switch-side information. The open-source Trumpet prototype (called "Trumpet-Full (TF)") realizes 4,096 triggers for different network event matchings. We also implement a simplified variant (called "Trumpet-Simple (TS)") that uses a single trigger to track the same nine flow statistics as OmniMon for fair comparisons. Fig. 8(a) shows the throughput of OmniMon (OM), the two Trumpet variants, and the packet generation using PktGen on a single CPU core. TF achieves only 6.5 Mpps as it performs matching on 4,096 triggers. TS is faster than TF, but is slower than OmniMon due to event matching overhead. OmniMon only has a slight throughput drop of 5.4% (for $10^6$ flows) compared to PktGen. Fig. 8(b) shows a breakdown of the number of CPU cycles for per-packet processing in OmniMon. The throughput drop mainly comes from per-packet hash lookups.

**(Exp#2) Switch resource usage.** We compare OmniMon in switch resource usage with four sketch solutions:



(a) Computation time (b) Impact on end-hosts

Fig. 10. (Exp#3) Controller overhead.

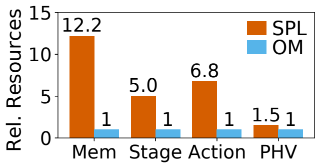
FlowRadar (FR) [13], UnivMon (UM) [28], ElasticSketch (ES) [16], and SketchLearn (SL) [15]. All sketch solutions perform approximate measurement using a compact data structure to fit in limited switch memory. We re-build them to measure per-flow packet counts based on their published designs in P4 in our switches. For fair comparisons, we build a simplified OmniMon (OS) that measures only per-flow packet counts. We compare all solutions with our full OmniMon version (OF) that tracks nine flow statistics.

Fig. 9 depicts the switch resource usage, in memory usage, numbers of stages and actions (which measure computational resources), and packet header vector (PHV) size (which measures the message size passed across stages). Both FR and UM maintain extra data structures (e.g., bloom filters and heaps) for flowkey recovery, and hence incur high memory usage (Fig. 9(a)). Also, sketch solutions perform multiple hash computations that trigger many stages and actions (Fig.s 9(b) and 9(c)). ES and SL update different parts of counters and incur high PHV sizes (Fig. 9(d)). For OmniMon, OS is resource-efficient since its collaborative design imposes only value updates in switches; even for OF, which measures more statistics, its switch resource usage is comparable to the sketch solutions. Note that OmniMon achieves full accuracy, but sketch solutions cannot.

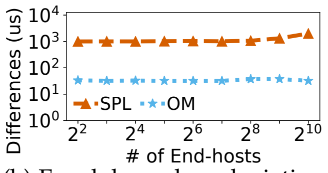
**(Exp#3) Controller overhead.** We evaluate the controller overhead for flow mapping, epoch synchronization, and collective analysis in our testbed. Recall that we run these operations in distinct threads (§8). Here, we configure each thread to run in a single CPU core. Fig. 10(a) shows that epoch synchronization can be completed within 20 $\mu$s, which is critical for epoch boundary deviations. Flow mapping takes more time, but it is invoked only at the startup. The most expensive operation is collective analysis, whose completion time is 180 ms for $10^6$ flows. We find that its major overhead comes from solving the linear system. Nevertheless, we can speed up the computation with more CPU cores [62], [63]. Fig. 10(b) shows the impact of these operations on end-hosts by comparing the end-host throughput with and without these operations. All operations show limited throughput drop as we isolate them in distinct threads from packet processing (§8).

**(Exp#4) Consistency.** We compare OmniMon with the open-source Speedlight (SPL) prototype [48], (an in-network synchronization solution) in epoch synchronization. We adapt the SPL prototype, written in P4 BMv2 [78], to our Tofino switches. Fig. 11(a) shows the switch resource usage, normalized to that of the full OmniMon version. SPL realizes the Chandy-Lamport algorithm [47] and incurs high overhead, while OmniMon is more resource-efficient as its epoch synchronization is done by end-hosts and the controller.

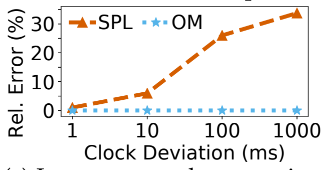
We examine the time deviations in per-epoch boundaries



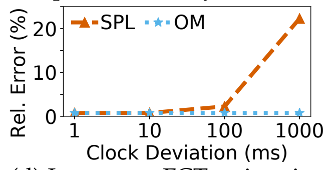
(a) Resource consumption



(b) Epoch boundary deviations

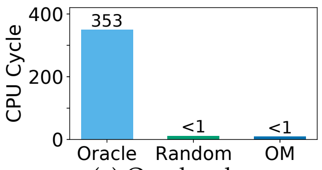


(c) Impact on packet counting

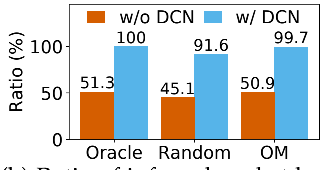


(d) Impact on FCT estimation

Fig. 11. (Exp#4) Consistency.

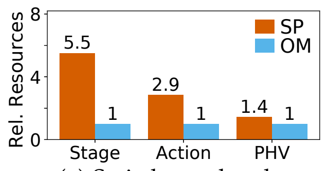


(a) Overhead

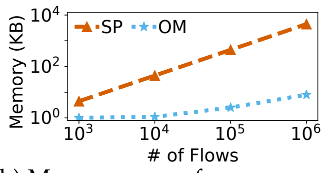


(b) Ratio of inferred packet loss

Fig. 12. (Exp#5) Accountability.

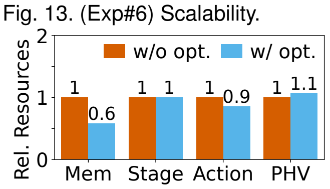


(a) Switch overhead

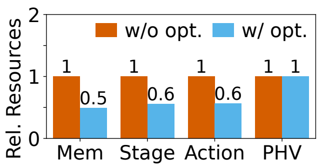


(b) Memory usage for zero errors

Fig. 13. (Exp#6) Scalability.



(a) Simplified OmniMon (OS)



(b) Full OmniMon (OF)

Fig. 14. (Exp#7) Switch resource usage of network-wide optimization.

when the local clocks are loosely synchronized. We deploy multiple (up to 1,024) end-host instances in each server. We tune the local clocks of end-hosts to differ by 1 ms. Fig. 11(b) shows that SPL suffers from a large deviation (nearly to the clock difference 1 ms) since it relies on only in-band packets for synchronization. If the traffic does not span all end-hosts, some end-hosts remain in the old epoch. OmniMon limits the deviation to 60 $\mu$s even for 1,024 end-hosts, as it leverages the controller to bound the deviation and uses `mTCP` to bypass kernel space (§8).

We further evaluate the accuracy of epoch synchronization in a straggler case. We configure one straggler end-host whose local clock differs from those of others by a range of 1 ms to 1 s. Fig. 11(c) measures the average relative error of per-flow packet counts in this straggler end-host. For SPL, the error increases to 26% and 33% when the difference is 100 ms and 1 s, respectively, as the epoch boundary deviation maps many packets to a wrong epoch. OmniMon retains zero errors as it includes every packet in the same epoch across all end-hosts and switches. Fig. 11(d) measures the average relative error of flow completion time. We focus on long-lived flows that exceed 1 s (which often imply important network events [79]). Although OmniMon measures coarse-grained timestamps in epochs (§7), its error is only 0.7%, as it bounds the epoch boundary deviation. For SPL, the error is above 22% when the clock deviation is 1 s.

**(Exp#5) Accountability.** Our default setup provisions sufficient slots to allocate each flow a unique flow mapping. Here, we consider an extreme case, in which we use 512 slots per switch (i.e., $\binom{512}{2} \approx$ 130K flow mappings). We set the epoch length to two minutes, so that an epoch now has 380K concurrent flows on average. We also set a packet drop rate of 0.5%, which randomly drops 0.5% packets. We compare OmniMon with two variants: (i) "Oracle", which knows all lossy flows in advance and maps the lossy flows for minimum slot overlaps to have the maximum matrix rank; and (ii) "Random", which randomly selects a slot for each flow. Fig. 12(a) shows the number of CPU cycles for each invocation of the three schemes. OmniMon and Random consume less than one CPU cycle, while Oracle requires over 300 cycles per packet for maximum-rank mappings.

We also evaluate the fractions of per-switch, per-flow packet loss variables that can be solved via the four steps in §5.2. For each of the three schemes, we build the linear system using Steps 1 and 2, and consider two variants: (i) "w/o DCN", in which we skip Step 3 and solve directly the linear system; and (ii) "w/ DCN", in which we execute Step 3 that simplifies the linear system with data center network characteristics. Fig. 12(b) shows that without Step 3, all schemes infer only around 50% of variables, as the number of lossy flows (i.e., $n$) exceeds the total number of available slots (i.e., 512 per switch). With the simplification of Step 3, Oracle infers all variables, while OmniMon infers 99.7% of variables. Here, OmniMon cannot achieve 100% of inference as we consider an extreme scenario, yet full accuracy is achievable in practical scenarios (Exp#11 in [44]). Note that the packet loss inference only requires a few seconds, since our linear system only considers lossy flows, which restricts the scale of the inference problem.

**(Exp#6) Scalability.** We compare OmniMon with the open-source SwitchPointer prototype [24] (SP) in scalability. We adapt the SP prototype, written in P4 BMv2, to our Tofino switches. Fig. 13(a) shows the switch resource usage normalized to that of OmniMon. SP focuses on in-network visibility, while the resource-accuracy trade-off is not its focus. It achieves zero errors with perfect hashing, but incurs higher resource overhead for the complicated calculations. Note that perfect hashing needs to acquire all possible flowkeys a priori to design perfect hash functions. This is feasible for IP addresses, but not for 5-tuples (with up to $2^{104}$ flows). Fig. 13(b) compares the memory usage needed for zero-error guarantees. The memory usage of SP grows linearly with the number of flowkeys as each slot is associated with one flow, while that of OmniMon increases at a much slower rate, as $K$ slots can accommodate $\binom{K}{d}$ flows (§5.2).

**(Exp#7) Network-wide optimization.** Fig. 15 compares per-switch resource usage of OmniMon with and without the optimization of network-wide mapping. As in Exp#2, we present two OmniMon versions: OS that counts per-flow number of packets (Fig. 15(a)) and OF that monitors nine flow statistics (Fig. 15(b)). We normalize the optimized resource usage to that without optimization. For memory usage, both OS and OF achieved significant optimization results, saving nearly half of the memory overhead. For OF,

| Query | Lines of Code | | |
|---|---|---|---|
| | Sonata | Marple | OmniMon |
| New TCP Conns. [59] | 6 | 8 | 6 |
| SSH Brute Force [80] | 7 | 12 | 7 |
| Superspreader [11] | 6 | 11 | 6 |
| Port Scan [81] | 6 | 11 | 6 |
| DDoS [11] | 9 | 14 | 9 |
| TCP SYN Flood [59] | 17 | 17 | 17 |
| TCP Incomplete Flows [59] | 12 | 16 | 12 |
| Slowloris Attack [59] | 13 | 18 | 13 |
| Zorro Attack [82] | 13 | x | 13 |
| Packet Counts [19] | 2 | 4 | 2 |
| Flowlet size histogram [19] | 8 | 12 | 8 |
| Cardinality [14] | 3 | 10 | 3 |
| Lossy connections [83] | x | 8 | 9 |
| TCP incast [84] | x | 9 | 4 |

TABLE 1
(Exp#8) LoC of Queries.

which keeps track of more flow statistics, cutting on-chip memory in half results in a 60% reduction in the number of stages the query occupies and the number of actions needed to update register memory. Here, in addition to recording flow statistics, OmniMon needs to pick `SwitchIndex` by hop, update the number of hops, and perform basic forwarding, which all require some resources. Therefore, OS does not achieve such a significant optimization effect on the number of stages and actions.

**(Exp#8) Query engine.** This experiment demonstrates that our query engine can ease the development efforts. We compare OmniMon with Sonata and Marple. Table 9.2 presents the lines of code to implement 14 telemetry queries. OmniMon supports all the queries with the similar LoC (see Exp#10 in [44] for the resource usage). Marple fails to perform zorro attack because it cannot process packet payload. Sonata cannot identify data plane queue length and packet loss, so it cannot implement lossy connections and TCP incast.

## 10 Related Work

**Resource-accuracy trade-off.** Existing network telemetry proposals often make the resource-accuracy trade-off. End-host-based telemetry systems [1], [85] perform complete per-flow tracking, but have limited network-wide visibility and incur excessive memory usage. Switch-based approaches either relax accuracy [37] or restrict to specific queries [38]. Approximation algorithms [86], [87], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [28] trade accuracy for resource efficiency. Some systems mirror traffic to the controller, in a best-effort manner [88] or on pre-defined traffic patterns [89], [17], [18], [19], [20], [21], [90], but inevitably miss the traffic that is not mirrored.

Recent hybrid architectures combine different entities for network telemetry [19], [23], [24], [39], [40]. To mitigate resource burdens, they rely on sampling [40] and/or event matching [19], [23], [24], [40] to focus on partial traffic, and hence cannot achieve full accuracy. Also, they do not address consistency and accountability in entity coordination.

**Consistency.** HUYGENS [91] proposes software clock synchronization, but its machine-learning design is complicated in switch deployment. Swing State [92] ensures consistent data-plane states via state migration, but poses heavy state update costs. Synchronized Network Snapshot [48] realizes causal consistency (a form of weak consistency) in P4 switches. OmniMon achieves hybrid consistency and is more lightweight (Exp#4 in §9).

**Accountability.** Loss inference is well studied in network telemetry. Active probing [54], [55] sends probe packets to measure path losses, but only covers a subset of paths. Passive monitoring analyzes network traffic, but relies on domain knowledge [18] and/or employs approximation techniques [34], [49], [83], [93] to estimate loss rates. OmniMon exploits the complete statistics in end-hosts as the ground truth to form a linear system for exact packet loss inference.

## 11 Conclusion

OmniMon is a network telemetry architecture for large-scale data centers. Its design principle is to re-architect the network-wide collaboration and carefully coordinate telemetry operations among all entities. We show how OmniMon achieves both resource efficiency and full accuracy, with consistency and accountability guarantees. Experiments show the effectiveness of OmniMon over 11 state-of-the-art telemetry designs; more results are available in technical report.

## Acknowledgments

We sincerely thank all the anonymous reviewers for their constructive and helpful comments on the paper. The work was supported in part by National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007).

## References

[1] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and Precise Triggers in Data Centers," in *Proc. of ACM SIGCOMM*, 2016.
[2] Cisco Tetration, https://www.cisco.com/c/en/us/products/data-center-analytics/tetration-analytics.
[3] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," 1990.
[4] sFlow, http://www.sflow.org/.
[5] NetFlow, https://www.ietf.org/rfc/rfc3954.txt.
[6] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "cSAMP: A System for Network-Wide Flow Monitoring," in *Proc. of USENIX NSDI*, 2008.
[7] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the Case for a Minimalist Approach for Network Flow Monitoring," in *Proc. of ACM IMC*, 2010.
[8] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant Time Updates in Hierarchical Heavy Hitters," in *Proc. of ACM SIGCOMM*, 2017.
[9] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proc. of ACM SOSR*, 2017.
[10] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-Wide Heavy Hitter Detection with Commodity Switches," in *Proc. of ACM SOSR*, 2018.
[11] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *Proc. of USENIX NSDI*, 2013.
[12] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch Resource Allocation for Software-defined Measurement," in *Proc. of ACM CoNEXT*, 2015.
[13] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," in *Proc. of USENIX NSDI*, 2016.

[14] 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 *Proc. of ACM SIGCOMM*, 2017.

[15] Q. Huang, P. P. C. Lee, and Y. Bao, "SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference," in *Proc. of ACM SIGCOMM*, 2018.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," in *Proc. of ACM SIGCOMM*, 2018.

[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 *Proc. of USENIX NSDI*, 2014.

[18] 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 *Proc. of ACM SIGCOMM*, 2015.

[19] 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 *Proc. of ACM SIGCOMM*, 2017.

[20] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative Traffic Mirroring on a Budget," in *Proc. of USENIX NSDI*, 2018.

[21] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, "dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces," in *Proc. of USENIX NSDI*, 2019.

[22] Khandelwal, Anurag and Agarwal, Rachit and Stoica, Ion, "Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks," in *Proc. of USENIX NSDI*, 2019.

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

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

[25] Data Plane Development Kit, https://dpdk.org.

[26] P4 Language, https://p4.org.

[27] Tofino, https://www.barefootnetworks.com/products/brief-tofino/.

[28] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *Proc. of ACM SIGCOMM*, 2016.

[29] Linux Netfilter, https://www.netfilter.org.

[30] M. Dobrescu, N. Egi, K. Argyraki, B.-g. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks : Exploiting Parallelism To Scale Software Routers," in *Proc. of ACM SOSP*, 2009.

[31] Broadcom Tomahawk Ethernet Switch Series, https://www.broadcom.cn/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[32] 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 *Proc. of ACM SIGCOMM*, 2017.

[33] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic External Memory for Switch Data Planes," in *Proc. of ACM HotNets*, 2018.

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

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

[36] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 121–133, 2001.

[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic Resource Allocation for Software-defined Measurement," in *Proc. of ACM SIGCOMM*, 2014.

[38] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker, "Compiling Path Queries," in *Proc. of USENIX NSDI*, 2016.

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

[40] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang, "MOZART: Temporal Coordination of Measurement," in *Proc. of ACM SOSR*, 2016.

[41] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *Proc. of NSDI*, 2014.

[42] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance," in *Proc. of ACM SIGCOMM*, 2013.

[43] H. Sun, J. Li, J. He, J. Gui, and Q. Huang, "Omniwindow: A general and efficient window mechanism framework for network telemetry," in *Proc. of ACM SIGCOMM*, 2023.

[44] O. T. Report, https://github.com/N2-Sys/OmniMon/blob/master/doc/techreport.pdf.

[45] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column)*, pp. 51–58, 2001.

[46] ——, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[47] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[48] N. Yaseen, J. Sonchack, and V. Liu, "Synchronized Network Snapshots," in *Proc. of ACM SIGCOMM*, 2018.

[49] D. Ghita, K. Argyraki, and P. Thiran, "Network Tomography on Correlated Links," in *Proceedings of ACM IMC*, 2010.

[50] Bowden, Rhys Alistair and Roughan, Matthew and Bean, Nigel, "Network Link Tomography and Compressive Sensing," in *Proc. of SIGMETRICS*, 2011.

[51] M. Alizadeh, A. Greenberg, D. A. Maltz, and J. Padhye, "Data Center TCP (DCTCP)," in *Proc. of ACM SIGCOMM*, 2010.

[52] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. of ACM IMC*, 2010.

[53] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network' s (Datacenter) Network," *Proc. of ACM SIGCOMM*, 2015.

[54] 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," *Proc. of ACM SIGCOMM*, 2015.

[55] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "NetBouncer: Active Device and Link Failure Localization in Data Center Networks," in *Proc. of USENIX NSDI*, 2019.

[56] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of ACM*, vol. 13, no. 7, 1970.

[57] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu, "Newton: Intent-driven network traffic monitoring," in *Proc. of ACM CoNEXT*, 2020.

[58] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. of ACM SIGCOMM*, 2020.

[59] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative Network Monitoring with NetQRE," in *Proc. of ACM SIGCOMM*, 2017.

[60] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *Proc. of USENIX NSDI*, 2021.

[61] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. of USENIX NSDI*, 2022.

[62] MUMPS, http://mumps.enseeiht.fr.

[63] SuperLU, http://crd-legacy.lbl.gov/~xiaoye/SuperLU.

[64] A. W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," in *Proc. of SIGMETRICS*, 2005.

[65] N. Williams, S. Zander, and G. Armitage, "A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 5, pp. 5–16, 2006.

[66] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, "Internet Traffic Classification Demystified : Myths , Caveats , and the Best Practices," in *Proc. of CoNEXT*, 2008.

[67] Y.-s. Lim, H.-c. Kim, J. Jeong, C.-k. Kim, T. T. Kwon, and Y. Choi, "Internet Traffic Classification Demystified : On the Sources of the Discriminative Power," in *Proc. of CoNEXT*, 2010.

[68] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosunblum, and A. Vahdat, "Exploiting a natural network effect for scalable, fine-grained clock synchronization," in *Proc. of USENIX NSDI*, 2018.

[69] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, "Globally synchronized time via datacenter networks," in *Proc. of ACM SIGCOMM*, 2016.
[70] T. C. Bressoud and F. B. Schneider, "Hypervisor-based Fault Tolerance," in *Proc. of SOSP*, 1995.
[71] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," in *Proc. of OSDI*, 2002.
[72] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *Proc. of USENIX NSDI*, 2014.
[73] OpenvSwitch, http://openvswitch.org.
[74] PktGen, https://pktgen-dpdk.readthedocs.io.
[75] ZeroMQ, http://zeromq.org.
[76] Eigen, http://eigen.tuxfamily.org.
[77] Caida Anonymized Internet Traces 2018 Dataset, http://www.caida.org/data/passive/passive_dataset.xml.
[78] P4 Behavioral Model v2, https://github.com/p4lang/behavioral-model.
[79] A. Chen, Y. Jin, J. Cao, and L. E. Li, "Tracking Long Duration Flows in Network Traffic," in *Proc. of INFOCOM*, 2010.
[80] M. Javed and V. Paxson, "Detecting stealthy, distributed ssh bruteforcing," in *Proc. of ACM SIGSAC*, 2013.
[81] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proc. of IEEE Symposium on Security and Privacy*, 2004.
[82] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: Analysing the rise of IoT compromises," in *Proc. of USENIX WOOT*, 2015.
[83] Y. Li, R. Miao, C. Kim, and M. Yu, "LossRadar: Fast Detection of Lost Packets in Data Center Networks," in *Proc. of ACM CoNEXT*, 2016.
[84] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proc. of ACM WREN*, 2009.
[85] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu, "A Comparison of Performance and Accuracy of Measurement Algorithms in Software," in *Proc. of ACM SOSR*, 2018.
[86] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, J. Xue, T. Zhao, Z. Jia, and Y. Yang, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," in *Proc. of IEEE ICNP*, 2021.
[87] A. Agarwal, Z. Liu, and S. Seshan, "HeteroSketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks," in *Proc. of USENIX NSDI*, 2022.
[88] 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 *Proc. of ACM SIGCOMM*, 2014.
[89] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. E. Ng, "Closed-loop network performance monitoring and diagnosis with SpiderMon," in *Proc. of USENIX NSDI*, 2022.
[90] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: Towards Programmable Network Measurement," in *Proc. of ACM SIGCOMM*, 2007.
[91] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization," in *Proc. of USENIX NSDI*, 2018.
[92] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *Proc. of ACM SOSR*, 2017.
[93] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *Proc. of USENIX NSDI*, 2017.
[94] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-Resolution Measurement of Data Center Microbursts," in *Proc. of ACM IMC*, 2017.

## APPENDIX

Appendices are supporting materials that have not been peer-reviewed.

### Appendix A: Proofs for §4

We formally prove the properties of hybrid consistency in five steps.

**Step 1.** We start with a lemma regarding the epoch of each packet in Lemma 1.

**Lemma 1.** *Every packet is included at the same epoch during its transmission.*

*Proof.* Recall that the epoch of a packet is decided only once in the source end-host and embedded in the packet. The embedded epoch will not be modified during the packet transmission. Every entity along the packet transmission path will follow this decision and monitor the packet at the epoch that is being embedded. The lemma follows. □

**Step 2.** We next examine the specific weak consistency model that OmniMon achieves. We first define the concept of *partial ordering*. In our context, we say that two packets $A$ and $B$ exhibit a partial order "$A$ before $B$" if and only if either of the following two cases happens: (i) $A$ and $B$ are sent by the same end-host and $A$ is sent before $B$, or (ii) $A$ and $B$ are sent by different end-hosts, but $B$ is sent after its source end-host receives $A$. Let $Epoch(P)$ denote the epoch at which some packet $P$ resides. Lemma 2 proves that OmniMon always achieves a specific weak consistency model based on the *Clock Condition* [46].

**Lemma 2.** *If two packets $A$ and $B$ exhibit a partial order "$A$ before $B$", then their embedded epochs must satisfy $Epoch(A) \leq Epoch(B)$ in all entities.*

*Proof.* We examine the two cases under the definition of partial ordering. In Case (i), $A$ and $B$ are sent by the same end-host and $A$ is sent before $B$. In this case, based on the local clock in the source end-host, we must have $Epoch(A) \leq Epoch(B)$. In Case (ii), $A$ and $B$ are sent by different end-hosts, but $B$ is sent after its source end-host receives $A$. Suppose that $B$'s source end-host receives $A$ at its local epoch (Epoch $b$), where $b \leq Epoch(B)$. If $Epoch(A) \leq b$, then clearly $Epoch(A) \leq Epoch(B)$; otherwise, $B$'s source end-host will update its local epoch to $Epoch(A)$, so $Epoch(B) \geq b = Epoch(A)$. □

**Step 3.** Lemma 3 provides an upper bound of the time difference of an epoch among all end-hosts.

**Lemma 3.** *Let $\Delta$ be the maximum delay between an end-host and the controller. The time difference of an epoch among all end-hosts is at most $2\Delta$.*

*Proof.* We examine the time difference between the first end-host and the last end-host that enter the same epoch. For any Epoch $x$, suppose that the first end-host enters Epoch $x$ at time $t_s$. Let $t_l$, $t_c$, and $t_p$ be the times at which the last end-host receives the epoch notification from its local clock, the controller, and the first embedded packet, respectively. Then the time difference is

$$\min\{t_l, t_c, t_p\} - t_s \leq t_c - t_s.$$

Since the first end-host sends a message to the controller immediately at $t_s$, $t_c - t_s$ is at most the round-trip delay between an end-host and the controller, i.e., $2\Delta$. The lemma follows. □

**Step 4.** With the bounded time difference of an epoch, Lemma 4 shows that all end-hosts reside at the same epoch (i.e., strong consistency) at most times.

**Lemma 4.** *Let $\Delta$ be the maximum delay between an end-host and the controller. Let $\ell$ be the length of an epoch. If $\ell > 2\Delta$, OmniMon achieves strong consistency for a duration of no less than $\ell - 2\Delta$ at each epoch.*

*Proof.* For any Epoch $x$, let $t_1$ and $t_2$ be the times at which the first and last end-hosts that enter Epoch $x$, respectively. Based on Lemma 3, we have $t_2 - t_1 \leq 2\Delta$. Also, let $h$ be the first end-host that enters Epoch $x+1$. Let $t_3$ and $t_4$ be the times at which $h$ enters Epoch $x$ and Epoch $x+1$, respectively. By definition, we have $t_4 - t_3 = \ell$ and $t_1 \leq t_3 \leq t_2$. Now, we compute the following.

$$t_4 - t_2 = \ell + t_3 - t_2 \geq \ell + t_1 - t_2 \geq \ell - 2\Delta.$$

Note that $t_4 - t_2$ represents the duration during which all end-hosts are synchronized at Epoch $x$. The lemma follows. □

**Remark.** If $\ell$ is sufficiently larger than $\Delta$, OmniMon achieves strong consistency at most times.

**Step 5.** Finally, Lemma 5 bounds the number of out-band messages triggered per epoch in OmniMon.

**Lemma 5.** *Suppose that OmniMon has $u$ end-hosts. For each epoch, the controller and each end-host process $O(u)$ and $O(1)$ out-band messages, respectively, for epoch synchronization.*

*Proof.* At each epoch, the controller receives at most $u$ out-band messages, each from one end-host. However, the controller only relays the first received out-band message to all $u$ end-hosts. Thus, the number of processed (including received and relayed) out-band messages in the controller is at most $2u$.

Also, at each epoch, an end-host sends at most one out-band message to the controller (only when its epoch update is triggered by the local clock). Also, it receives one message from the controller based on the above discussion. Thus, the overhead is $O(1)$. □

**Remark.** We consider an extreme scenario with 10K end-hosts and an epoch length of 100 ms. Suppose that an out-band message is of size 64 bytes (including Ethernet, IP and TCP headers). This leads to at most $2 \times 10K \times (1000/100) * 64$ =12.8 MB/s bandwidth, which is quite low in a data center network with 10K end-hosts.

## Appendix B: Proofs for §5

We present proofs for the following two properties: (i) The rank of $A$ has a tight upper bound $r(m-1)+n$, and (ii) the number of mapped slots has an expected value equal to $nd$.

**Rank of $A$.** We prove the tight upper bound of the rank of matrix $A$. Recall that we construct a matrix $M$ that comprises the rows of each $M_t$ ($1 \leq t \leq m$), and the rank of $M$ is $r$. We show that $r(m-1)+n$ is the tight upper bound of the rank of $A$.

We first construct a large matrix

$$A^* = \begin{bmatrix} M & 0 & \cdots & 0 \\ 0 & M & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & M \\ I & I & \cdots & I \end{bmatrix}.$$

Obviously, $A$ is a sub-matrix of $A^*$. Thus, the rank of $A$ is tightly bounded by the rank of $A^*$. The remaining problem is to show that the rank of $A^*$ is $r(m-1)+n$.

Our proof builds on the definition of rank: the rank of a matrix is equal to the maximum number of linearly independent rows or linearly independent columns in the matrix. Here, we focus on the linearly independent rows (LIRs). Our idea is to first construct $r(m-1)+n$ LIRs for $A^*$, and then show that any $r(m-1)+n+1$ rows in $A^*$ must be linearly dependent.

**Lemma 6.** *There exist $r(m-1)+n$ LIRs in $A^*$.*

*Proof.* We construct the LIRs as follows. Note that the rank of $M$ equals $r$. Thus, we have $r$ LIRs, each of which has size $n$. We can expand each of them with $(m-1)n$ zeros, such that the expanded row becomes one row of $A^*$. Specifically, for the $i$-th ($1 \leq i \leq m$) $M$ in $A^*$, we can add $n(i-1)$ zeros before each LIR, and $n(m-i)$ zeros after it. Thus, we obtain $rm$ expanded rows in total. We denote them by $R_1, R_2, \cdots, R_{rm}$, each of which is an expanded row of size $mn$, such that $R_{r(i-1)+1}, R_{r(i-1)+2}, \cdots, R_{r(i-1)+r}$ are expanded rows for the $i$-th $M$ ($1 \leq i \leq m$).

Now we select the first $r(m-1)$ expanded rows, i.e., $R_1, R_2, \cdots, R_{r(m-1)}$. We also select the last $n$ rows in $A^*$, which are denoted by $I_1, I_2, \cdots, I_n$. We show that this collection of $r(m-1)+n$ rows are linearly independent. Specifically, we consider the linear combination $L = \alpha_1 R_1 + \alpha_2 R_2 + \cdots + \alpha_{r(m-1)} R_{r(m-1)} + \beta_1 I_1 + \beta_2 I_2 + \cdots + \beta_n I_n$, which is a vector of length $mn$. We prove that $L$ is a vector with all zero elements only if all coefficients $\alpha_i$ ($1 \leq i \leq r(m-1)$) and $\beta_j$ ($1 \leq j \leq n$) are also zeros.

Our proof partitions $L$ into $m$ groups with $n$ elements each. We first analyze the last group. If $L$ is a zero vector, the $j$-th element ($1 \leq j \leq n$) in the last group is contributed only by $I_j$. Thus, $\beta_j$ must be zero. Then for the $i$-th group ($1 \leq i \leq m-1$), only $r$ rows $R_{r(i-1)+1}, R_{r(i-1)+2}, \cdots, R_{r(i-1)+r}$ contribute non-zero values for the $n$ elements in this group. Since $R_{r(i-1)+1}, R_{r(i-1)+2}, \cdots, R_{r(i-1)+r}$ are linearly independent, we can deduce that $\alpha_{r(i-1)+1}, \alpha_{r(i-1)+2}, \cdots, \alpha_{r(i-1)+r}$ must be all zeros to make the linear combination a zero vector. Thus, we prove that all coefficients are zeros. The lemma follows. □

Next, we show that any $r(m-1)+n+1$ rows in $A^*$ must be linearly dependent.

**Lemma 7.** *Any $r(m-1)+n+1$ rows in $A^*$ must be linearly dependent.*

*Proof.* Consider an arbitrary collection that is composed of $r(m-1)+n+1$ rows in $A$. If any $M$ contributes more than

$r$ rows, these rows must be linearly dependent because the rank of $M$ is $r$.

Thus, we focus on the case where each $M$ contributes at most $r$ LIRs. Without the loss of generality, suppose that the first $m-1$ matrices of $M$ contribute $r$ LIRs each, denoted by $R_1, R_2, \cdots, R_{r(m-1)}$ as the above proof. The $n+1$ rows come from the last $M$ and $I_1, I_2, \cdots, I_n$. Obviously, at least one row is from the last $M$. For each row (denoted by $P$) in the last $M$, we can construct a linear combination $\alpha_1 R_1 + \alpha_2 R_2 + \cdots + \alpha_{r(m-1)} R_{r(m-1)}$, such that $\alpha_1 R_1 + \alpha_2 R_2 + \cdots + \alpha_{r(m-1)} R_{r(m-1)} + P$ is a vector of size $mn$ and every group of $n$ elements is the same as each other. Note that each $I_i$ $(1 \leq i \leq n)$ also has $m$ groups of $n$ elements and all groups are the same. Such $n+1$ rows must be linearly dependent since there are at most $n$ linearly independent columns. Since the $n+1$ rows are either constructed by the linear combination of $\alpha_1 R_1 + \alpha_2 R_2 + \cdots + \alpha_{r(m-1)} R_{r(m-1)}$ + $P$ or directly from some $I_i$ $(1 \leq i \leq n)$, the $r(m-1)+n+1$ rows are linear dependent. □

The two lemmas lead to the following theorem.

**Theorem 1.** *If $M$ has a rank $r$, the rank of $A$ has a tight upper bound $r(m-1)+n$, where $n$ is the number of lossy flows and $m$ is the number of switches traversed by any lossy flow.*

**Number of mapped slots.** We analyze the number of slots $k$ mapped by $n$ lossy flows, given $K$ slots in total and $d$ distinct slots per flow. For each slot, the probability that it is mapped by a particular flow is

$$p_1 = \frac{\binom{K-1}{d-1}}{\binom{K}{d}} = \frac{d}{K}.$$

With $n$ lossy flows, the probability that it is touched by at least one flow is

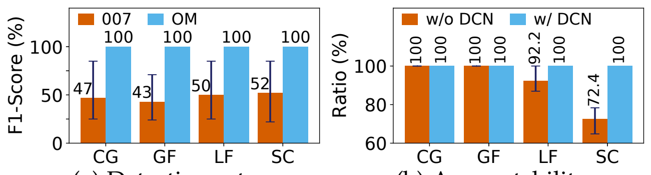
$$p_n = 1 - (1-p_1)^n \approx np_1 = \frac{nd}{K} \text{ with sufficiently large } K.$$

Since the $K$ slots are independent, the number of touched slots follows a Binomial distribution with expectation $K \cdot p_n = nd$ and variation $K \cdot p_n(1-p_n) = \frac{(nd)^2}{K}$.

## Appendix C: Additional Experiments

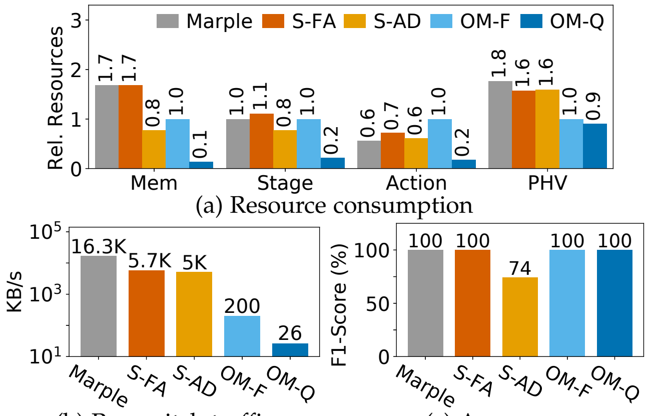
**Simulator.** We build a simulator to evaluate OmniMon in large-scale deployment. The simulator forms an 8-ary Fat-Tree topology with 128 end-hosts and 80 switches. It uses a single thread to realize the packet forwarding and OmniMon operations of all end-hosts, switches, and the controller across epochs. It can also inject packet losses to mimic lossy environments. We use the same CAIDA trace [77] to drive our simulations as in testbed experiments. In particular, we map the IP addresses in the trace to our end-hosts, such that the ratios of pod-local and aggregate-local flows are 15% and 70%, respectively. We also configure 8,192 slots in each switch (port) in per-switch (per-port) deployment.

**(Exp#9) Packet loss diagnosis.** We now use the simulator to show that OmniMon can detect four types of packet losses: (i) switch congestions (SG), (ii) gray failures (GF), (iii) switch crash (SC), and (iv) link failures (LF). We configure each type of packet losses as follows. We first randomly select



(a) Detection rate (b) Accountability

Fig. 15. (Exp#9) Packet loss diagnosis.



(a) Resource consumption

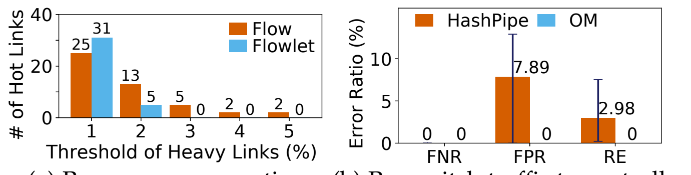
(b) Per-switch traffic (c) Accuracy

Fig. 16. (Exp#10) End-to-end performance comparison.

eight switches (resp. links) as problematic switches (resp. links), implying that the failure probabilities for switches and links are 10% and 3%, respectively, in our 8-ray fat tree. In a switch congestion, we randomly drop packets at a rate of 1%; in a gray failure, we randomly sample 1% of flows to drop; for switch crashes and link failures, we drop all traffic in a faulty switch/link after a failure occurs. We assume that the failures occur in the forwarding module, so that flow values in each faulty switch remain available. We repeat the experiment 100 times for each type of packet losses. Note that our scenarios are even more severe than those reported in recent studies [34], [53], [54], [94].

We compare OmniMon with 007 [34], a diagnosis system that infers packet losses on a per-path basis. Since the results of 007 significantly vary across the 100 runs, we plot the error bars showing its maximum and minimum. Figure 15(a) compares their F1 scores (i.e., the harmonic mean of precision and recall). OmniMon achieves perfect detection for all failures, while the accuracy of 007 is quite low. The reason is that 007 builds on a flow voting mechanism for finding faulty switches or links. When the failure probability is high, the presence of a large number of lossy flows can lead to many noisy votes. Figure 15(b) further shows how OmniMon achieves accountability by measuring the percentage of loss variables that can be solved based on the steps in §5.1. We consider two schemes: (i) skipping Step 3 and directly solving the original linear system ("w/o DCN"); (ii) taking into account the traffic characteristics of data center networks (DCNs) (i.e., executing Step 3) and solving the simplified linear system ("w/DCN"). Without the DCN characteristics, we can infer all loss variables except for switch failures (72%), since a switch failure produces much more lossy flows than the other three types of packet losses and makes the accountability condition fail. With the DCN characteristics, all loss variables can be solved, thereby achieving accountability.

**(Exp#10) End-to-end performance comparison.** We com-



(a) Resource consumption (b) Per-switch traffic to controller

Fig. 17. (Exp#11) Load balancing evaluation.

pare with two switch-controller architectures of Sonata [23] and Marple [19]. Our end-to-end performance comparison addresses 11 anomaly detection applications reported from Sonata [23]. The comparison includes three aspects: (i) per-switch resource usage; (ii) per-switch traffic sent to the controller; and (iii) per-application detection accuracy. We do not compare the end-host overhead, as Sonata and Marple do not involve end-hosts in their designs. As shown in Exp#1 and Exp#3, OmniMon has limited end-host overhead.

OmniMon tracks nine flow statistics to support the 11 applications. For Sonata, as in §2.2, we consider both "Sonata-FA (S-FA)" (with full accuracy) and "Sonata-AD (S-AD)" (with accuracy drop), in which every switch-side operator is configured with $2^{16}$ counters as in its open-source prototype. For Marple, we implement its key-value cache and evict old keys to handle hash collisions. We configure $2^{16}$ slots as in [19]. For OmniMon, we employ a full configuration that includes all nine statistics ("OM-F"). We also employ the Sonata-based query engine (§8) to generate a configuration for each application ("OM-Q"). We set $10^5$ flows at each epoch. Since Sonata, Marple, and OM-Q deploy each application individually, we average their results of the 11 applications (per-application results in [44]).

Fig. 16(a) compares per-switch resource usage, normalized to that of OM-F. Sonata-AD sacrifices accuracy and consumes the least memory, stages, and actions. Sonata-FA and Marple require more resources for flowkey tracking and cache eviction. OM-F uses less switch memory usage than Sonata-FA and Marple, while tracking all nine statistics, due to its collective analysis (§3.2). Its usage of the other three types of resources is comparable to the single application deployment in Sonata-FA and Marple, since it performs only (simple) value update operations in switches. With the query engine, the resource usage of OM-Q drops further. Fig. 16(b) compares the per-switch traffic sent to the control plane. Sonata-FA and Marple preserve the accuracy by evicting colliding flows to the controller, but trigger much traffic as hash collisions frequently occur with numerous flows. Sonata-AD reduces per-switch, per-application traffic to 5 MB/s, but incurs accuracy drop. In contrast, OM-F triggers 180 KB/s traffic without accuracy drop. OM-Q further reduces the traffic (26 KB/s) as it tracks only the statistics of interest for each application. Fig. 16(c) compares the detection accuracy quantified by the F1-score. The F1-score of Sonata-AD is only around 74%, as it does not resolve the hash collisions in the data plane for resource efficiency. Marple and Sonata-FA improve the accuracy, but incur high resource overhead in both switches and the controller. OmniMon achieves both resource efficiency and full accuracy.

**(Exp#11) Load balancing evaluation.** We study the performance evaluation effectiveness of OmniMon on the load balancing deployment in a data center network. We evaluate two load balancing algorithms: flow-based ECMP and flowlet-based CONGA [35]. For each algorithm, we feed our traces into the 8-ary Fat-Tree, and simulate the per-packet forwarding operations in each switch. OmniMon collects the actual per-flow bytes in each switch to construct a full view of the entire network.

Figure 17(a) counts the number of heavy links in both algorithms. Here, a link is considered to be heavy if the ratio of its traffic (in bytes) to the total traffic exceeds some threshold. We vary the threshold from 1% to 5% of the total traffic. The results show that CONGA introduces more heavy links than ECMP when the threshold is 1%. However, CONGA has fewer heavy links for larger thresholds (i.e., better load balancing). Figure 17(b) further identifies per-link heavy hitters that account for 10% of the link utilization. We measure the false negative rate (FNR), false positive rate (FPR), and per-flow relative error (RE) of OmniMon and HashPipe [9], a switch-based heavy-hitter detector. We plot the error bars showing its maximum and minimum. HashPipe incurs 8% of false positives and 3% of relative error since it only approximately records the top-$k$ flows. In contrast, OmniMon achieves zero errors. This provides guidelines for administrators to fine-tune the deployment of a new load balancing algorithm.

**Breakdown of resource consumptions.** Tables 2 and 4 list the complete breakdown of switch overhead and the amount of traffic transferred to the controller, respectively, of each solution in Exp#2 and Exp#6. For Sonata, Marple, and OmniMon, the tables show the per-application resource consumptions in detail. We provide not only the results of the per-switch setup as in §9, but also those of the per-port setup.

## Appendix D: Complete Entity Operations

Algorithms 2, 3, and 4 present the pseudo-code of the operations of end-hosts, switches, and the controller, respectively.

**End-hosts.** Algorithm 2 shows the operations of an end-host. Each end-host maintains: (i) a hash table (line 2), (ii) an array of egress slots (line 3), (iii) an array of ingress slots (line 4), (iv) a local epoch (line 5), (v) a set of switch indexes, which is pre-assigned by the controller and allocated to flows (line 6), and (vi) an array of counters for the usage of switch indexes (line 7).

When sending a packet (lines 8-19), an end-host first queries the flowkey in the hash table (line 9) to obtain both the host index and the switch index (line 10). Otherwise, if the hash table does not include the flowkey, the end-host chooses a new host index and a new switch index (line 12), and inserts the flowkey to the hash table (line 13) (the index selection (lines 37-42) has been elaborated in §3.2 and §5.2). It also increments the usage count of the switch index (line 14). With the host index, flow values in the egress slot will be updated (line 15). The end-host then embeds its current epoch, host index, and switch index into the packet (line 16) and emits the packet (line 17). If the flow terminates, it decrements the usage count (lines 18-19).

On the receipt of a packet (lines 20-27), the destination end-host extracts the source end-host (line 21), host index (line 22), and embedded epoch (line 23). It then updates

| Solutions | Tofino resources | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Memory (MB) | | | | | Stage | | | | | Action | | | | | PHV (Byte) | | | | |
| **Queries (Marple — Sonata-FA — Sonata-AD — Omnimon-QE-Switch — Omnimon-QE-Port)** | | | | | | | | | | | | | | | | | | | | |
| TCP new connection | 1.21 | 1.26 | 0.36 | 0.17 | 0.41 | 9 | 8 | 5 | 2 | 2 | 18 | 21 | 13 | 7 | 7 | 194 | 189 | 175 | 120 | 120 |
| SSH brute force | 2.92 | 2.47 | 1.85 | 0.17 | 0.69 | 12 | 11 | 7 | 2 | 2 | 24 | 30 | 25 | 7 | 7 | 244 | 201 | 218 | 116 | 116 |
| SuperSpreader | 2.33 | 2.41 | 0.67 | 0.02 | 0.02 | 10 | 11 | 5 | 1 | 1 | 21 | 26 | 14 | 2 | 2 | 235 | 198 | 162 | 112 | 112 |
| Port scan | 2.05 | 2.41 | 0.63 | 0.02 | 0.02 | 10 | 11 | 11 | 1 | 1 | 22 | 26 | 17 | 2 | 2 | 229 | 169 | 172 | 112 | 112 |
| DDoS | 2.33 | 2.41 | 1.23 | 0.02 | 0.02 | 10 | 11 | 8 | 1 | 1 | 20 | 26 | 17 | 2 | 2 | 233 | 201 | 205 | 112 | 112 |
| SYN flood | 3.49 | 2.51 | 1.39 | 0.45 | 1.94 | 9 | 12 | 6 | 3 | 3 | 31 | 35 | 37 | 15 | 15 | 272 | 223 | 258 | 120 | 120 |
| Completed flows | 2.36 | 2.49 | 0.84 | 0.31 | 0.76 | 10 | 12 | 11 | 2 | 2 | 26 | 34 | 45 | 11 | 11 | 235 | 223 | 255 | 120 | 120 |
| Slowloris attack | 2.62 | 2.49 | 1.97 | 0.31 | 2.43 | 11 | 11 | 7 | 2 | 4 | 24 | 34 | 37 | 11 | 11 | 277 | 232 | 247 | 120 | 120 |
| DNS tunneling | 1.27 | 2.6 | 0.8 | 0.17 | 1.25 | 6 | 12 | 5 | 2 | 2 | 19 | 34 | 22 | 7 | 7 | 201 | 231 | 205 | 120 | 120 |
| Zorro attack | 2.92 | 2.49 | 1.04 | 0.17 | 0.69 | 12 | 11 | 6 | 2 | 2 | 24 | 33 | 24 | 7 | 7 | 244 | 220 | 220 | 130 | 130 |
| Reflection DNS | 0.11 | 0.1 | 0.1 | 0.17 | 1.25 | 3 | 3 | 3 | 2 | 2 | 11 | 11 | 11 | 7 | 7 | 164 | 164 | 157 | 120 | 120 |
| Total | 23.61 | 23.64 | 10.88 | 1.98 | 9.48 | 102 | 35 | 74 | 20 | 22 | 240 | 133 | 262 | 78 | 78 | 2528 | 1981 | 2274 | 1302 | 1302 |
| **Sketch Solutions** | | | | | | | | | | | | | | | | | | | | |
| FlowRadar | 6.58 | | | | | 11 | | | | | 26 | | | | | 133 | | | | |
| UnivMon | 6.4 | | | | | 12 | | | | | 67 | | | | | 128 | | | | |
| ElasticSketch | 0.59 | | | | | 10 | | | | | 23 | | | | | 163 | | | | |
| SketchLearn | 1.16 | | | | | 10 | | | | | 40 | | | | | 158 | | | | |
| **OmniMon Solutions** | | | | | | | | | | | | | | | | | | | | |
| Only packet count (per switch) | 0.24 | | | | | 2 | | | | | 7 | | | | | 120 | | | | |
| Only packet count (per port) | 1.25 | | | | | 2 | | | | | 7 | | | | | 120 | | | | |
| Full statistics (per switch) | 1.49 | | | | | 9 | | | | | 39 | | | | | 130 | | | | |
| Full statistics (per port) | 6.25 | | | | | 10 | | | | | 39 | | | | | 130 | | | | |

TABLE 2
Complete switch resource overheads of solutions in Exp#2 and Exp#10.

| Solutions | Traffic to Controller | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Message (Msg/s) | | | | | Byte (KB/s) | | | | |
| **Queries (Marple — Sonata-FA — Sonata-AD — Omnimon-QE-Switch — Omnimon-QE-Port)** | | | | | | | | | | |
| TCP new connection | 171 | 17 | 16 | 10 | 10 | 10250.32 | 1.21 | 1.19 | 10 | 320 |
| SSH brute force | 10 | 0 | 0 | 10 | 10 | 25600.02 | 0 | 0 | 20 | 640 |
| SuperSpreader | 1555 | 3649 | 57 | 0 | 0 | 20578.89 | 3590.45 | 4.82 | 0 | 0 |
| Port scan | 816 | 1042 | 25 | 0 | 0 | 17971.58 | 1049.64 | 1.75 | 0 | 0 |
| DDoS | 1576 | 3528 | 94 | 0 | 0 | 20580.25 | 3843.88 | 150.45 | 0 | 0 |
| SYN flood | 536 | 23366 | 23366 | 10 | 10 | 30753.67 | 26172.16 | 26452.55 | 60 | 1920 |
| Completed flows | 72 | 1053 | 1053 | 10 | 10 | 20483.97 | 95.33 | 107.98 | 20 | 640 |
| Slowloris attack | 29153 | 29143 | 29143 | 10 | 10 | 1865.18 | 29025.58 | 29491.87 | 80 | 2560 |
| DNS tunneling | 315 | 474 | 474 | 10 | 10 | 10259.57 | 41.47 | 43.36 | 40 | 1280 |
| Zorro attack | 10 | 1 | 1 | 10 | 10 | 25600.05 | 0.03 | 0.08 | 20 | 640 |
| Reflection DNS | 315 | 305 | 305 | 10 | 10 | 19.57 | 23.26 | 23.26 | 40 | 1280 |
| Total | 34529 | 62398 | 54533 | 80 | 80 | 183963.07 | 63843.01 | 56277.31 | 290 | 9280 |
| **Sketch Solutions** | | | | | | | | | | |
| FlowRadar | 10 | | | | | 16000 | | | | |
| UnivMon | 10 | | | | | 42240 | | | | |
| ElasticSketch | 10 | | | | | 3840 | | | | |
| SketchLearn | 10 | | | | | 1320 | | | | |
| **OmniMon Solutions** | | | | | | | | | | |
| Only packet count (per switch) | 10 | | | | | 160 | | | | |
| Only packet count (per port) | 10 | | | | | 1280 | | | | |
| Full statistics (per switch) | 10 | | | | | 800 | | | | |
| Full statistics (per port) | 10 | | | | | 6400 | | | | |

TABLE 3
Complete triggered traffic of solutions in Exp#2 and Exp#10.

| Queries | Recall (%) | Precision (%) | F1-Score (%) |
|---|---|---|---|
| TCP new connection | 100 | 100 | 100 |
| SSH brute force | 100 | 100 | 100 |
| SuperSpreader | 45 | 75 | 56 |
| Port scan | 20 | 90 | 32 |
| DDoS | 54 | 92 | 68 |
| SYN flood | 99 | 40 | 57 |
| Completed flows | 100 | 68 | 81 |
| Slowloris attack | 33 | 82 | 47 |
| DNS tunneling | 100 | 100 | 100 |
| Zorro attack | - | - | - |
| Reflection DNS | 100 | 100 | 100 |
| Mean | 75 | 85 | 74 |

TABLE 4
Accuracy of Sonata-AD Queries in Exp#10.

the corresponding ingress slot (line 24). It updates its local epoch based on embedded packets (lines 25-27), local clock (lines 28-32), and controller messages (lines 33-36).

Whenever the local epoch is updated, an end-host not only performs synchronization, but also reports local results to the controller for collective analysis. It clears the hash table (i.e., removes terminated flows) and resets slot counters for the new epoch (lines 26, 30, and 35).

**Switches.** Algorithm 3 shows the switch operations. For each packet, a switch retrieves the switch index and epoch embedded in the packet (lines 2-3), as well as the set of slot indexes based on the switch index (line 4). It then updates the corresponding slots with the flow value in the packet (lines 5-6).

**Controller.** Algorithm 4 shows the controller operations. In

**Algorithm 2** End-host operations

```
 1: function INITIALIZATION
 2:     Hash table $H = \emptyset$                                        ▷ §3
 3:     Array of egress slots $ES[h]$ for end-host $h$                    ▷ §3
 4:     Array of ingress slots $IS[h][e]$ for end-host $h$ and epoch $e$  ▷ §3
 5:     Local epoch $e_h = 0$                                             ▷ §4
 6:     Set of switch indexes $I$, assigned by controller                 ▷ §5
 7:     Array of switch index usage counters $U[s]$ for switch index $s$
                                              ▷ Egress packet processing, §3
 8: function ON_SENDING_PACKET(packet $p$ with flowkey $k$)
 9:     if $k \in H$ then
10:         (switchIndex, hostIndex) $= H[k]$
11:     else
12:         (switchIndex, hostIndex) $=$ GETINDEXES($k$, $H$, $I$)
13:         Insert $H[k] =$ (switchIndex, hostIndex)
14:         Increment $U$[switchIndex] by 1
15:     Update egress slot $ES$[hostIndex] with $p$
16:     Embed $e_h$, switchIndex and hostIndex in $p$
17:     Emit $p$
18:     if Flow $k$ terminates then
19:         Decrement $U$[switchIndex] by 1
                                             ▷ Ingress packet processing, §3
20: function ON_RECEIVING_PACKET(packet $p$)
21:     Extract source end-host $h$ based on source_ip
22:     Extract epoch $e$ from $p$
23:     Extract hostIndex from $p$
24:     Update ingress slot $IS$[hostIndex][$e$] with $p$
25:     if $e > e_h$ then                           ▷ Epoch synchronization, §4
26:         Send $H$, $ES$, and $IS$ to controller, then reset
27:         $e_h = e$
                                                ▷ Epoch synchronization, §4
28: function ON_NEW_EPOCH_BY_LOCAL_CLOCK(new epoch $e$)
29:     if $e > e_h$ then
30:         Send $H$, $ES$, and $IS$ to controller, then reset
31:         $e_h = e$
32:         Notify $e$ to controller
                                                ▷ Epoch synchronization, §4
33: function ON_NEW_EPOCH_FROM_CONTROLLER(new epoch $e$)
34:     if $e > e_h$ then
35:         Send $H$, $ES$, and $IS$ to controller, then reset
36:         $e_h = e$
                                                ▷ Flow mapping, §3 and §5
37: function GETINDEXES(flowkey $k$, hash table $H$, switch index set $I$)
38:     hostIndex $= H$.getPosition($k$)
39:     if available switch index $i \in I$ then
40:         switchIndex $= i$
41:     else
42:         switchIndex $= \arg\min_{i \in I} U[i]$
43:     return (switchIndex, hostIndex)
```

**Algorithm 3** Switch operations

```
Data structures: Slots $S[e][i]$ for $i$-th slot at epoch $e$
 1: function ON_RECEIVING_PACKET(packet $p$)
 2:     Retrieve switch index switchIndex from $p$
 3:     Retrieve epoch $e$
 4:     Retrieve the set of slot indexes from switchIndex
 5:     for all $i \in$ set of slot indexes do
 6:         Update $S[e][i]$ with the flow value in $p$
```

**Algorithm 4** Controller operations

```
Data structures: Epoch $e_c$, Set of all switch indexes $I$
                                                                      ▷ §3
 1: function COLLECTIVE ANALYSIS(epoch $e$)
 2:     Collect hash table $H$, egress slots $IS$ and ingress slots $ES$ from
    all end-hosts
 3:     Collect switch slots $S$ from all switches
 4:     Set of lossy flows $L_f = \emptyset$
 5:     for all flow $f$ in every $H$ do
 6:         Obtain (hostIndex, switchIndex) of $f$ from $H$
 7:         if $IS$[hostIndex] == $ES$[hostIndex] then
 8:             Obtain true flow values from $ES$[switchIndex]
 9:             for all switch $s$ along $f$'s path do
10:                 Subtract flow values of $f$ from $S$[switchIndex] of $s$
11:         else
12:             $L_f = L_f \cup \{f\}$
13:     if $L_f \neq \emptyset$ then
14:         Perform packet loss inference (§5)
                                                                      ▷ §4
15: function ON_NEW_EPOCH(epoch $e$)
16:     if $e > e_c$ then
17:         $e_c = e$
18:         for all host $h$ do
19:             Send $e_c$ to $h$
                                                                      ▷ §5
20: function FLOW_MAPPING(end-host $h$)
21:     return $\{x \in I | hash(x) = h.id\}$
```

collective analysis (lines 1-14), the controller first collects results from all end-hosts and switches (lines 2-3). For each flow, the controller first examines its flow values in the source and destination end-hosts via the host index (line 7). If the values in the two end-hosts are equal, the controller decomposes the values from each switch slot along the packet transmission path with the flow values in end-hosts (lines 8-10). Otherwise, if the flow values differ in the source and destination end-hosts (line 13), this flow is a lossy one (line 14). The controller performs exact loss inference for all lossy flows (lines 13-14). The controller also performs network-wide epoch synchronization (lines 15-19) (§4) and collaborative flow mapping (lines 20-21) (§5). For collaborative flow mapping, the controller hashes each switch index (indicating one possible mapping to slots), and returns a set of switch indexes to the end-host, such that the hash value of each switch index equals the end-host ID.

## Appendix E: Generality of OmniMon

We provide more detailed elaboration on the generality of OmniMon.

**Flowkeys.** OmniMon supports various combinations of packet fields as flowkey definitions. Specifically, end-hosts can track the flowkeys that include all packet fields of interest (e.g., 5-tuple). During collective analysis, the controller can aggregate the flows for specific combinations of packet fields (e.g., source IP address). For example, end-hosts can track 5-tuple flowkeys, while the controller can group the flows by the source IP address for per-host measurement.

**Flow statistics.** OmniMon supports different types of flow statistics, include the 344 statistics in [64], [65], [66], [67]. In particular, a flow statistic typically performs a specific aggregated operator over all packets of a flow. Currently, OmniMon supports various basic online analytical processing (OLAP) operators (e.g., `COUNT`, `SUM`, `MAX`, `MIN`), which are enabled by end-hosts and programmable switches. More complex operators (e.g., `RANGE` and `AVG`) can be computed by the basic operators offline.

However, there are two concerns that may potentially limit the support of general flow statistics. First, OmniMon cannot perform collective analysis on packet timestamps. Currently, it addresses this limitation by measuring timestamps at the granularity of epochs, as elaborated in §7.

The second limitation comes from the accountability guarantee, in which the linear system requires additive flow statistics. OmniMon addresses this by converting non-additive statistics into additive ones, followed by applying

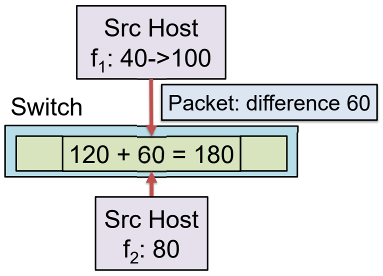


Fig. 18. Example of non-additive statistics.

the linear system to the converted additive statistics to infer missing values due to packet loss. Specifically, OmniMon still sums up the non-additive values in switches. To compute the correct sum in switches, it leverages the correct flow statistics in end-hosts: whenever the statistics are updated, a source end-host embeds the (zero-error) difference between the old and new values of the statistics in the packet header. When a switch receives the packet, it extracts the difference and adds the difference to the corresponding slots. This extension incurs no additional overhead in switches, since a switch performs the same operations as for additive values. However, it requires extra fields in packets. Given that the updates of non-additive statistics are relatively less frequent (e.g., `MAX` and `MIN`), we expect that the extra overhead is acceptable.

**Example.** We illustrate how OmniMon addresses non-additive values via an example, in which it measures the maximum packet size of each flow. As shown in Figure 18, we assume that there are two flows $f_1$ and $f_2$ share a slot in the switch. At the beginning, the maximum packet size of $f_1$ and $f_2$ are 40 and 80, respectively. The slot hence records their sum 120. Now $f_1$ has a new packet with size 100. The source end-host of $f_1$ first updates the maximum packet size of $f_1$, and then embeds the difference $100 - 40 = 60$ in the packet header. The switch adds up the difference so that the sum of maximum packet size becomes 180.