OmniWindow: A General and Efficient Window Mechanism Framework for Network Telemetry

Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang*
Peking University

ABSTRACT

Recent network telemetry solutions typically target programmable switches to achieve high performance and in-network visibility. They partition the packet stream into windows and then apply various stream processing techniques to summarize flow-level statistics. However, existing studies focus on the measurement within each window. Window management is still a missing piece due to the resource limitation of programmable switches. In this paper, we propose OmniWindow, a general and efficient window mechanism framework. OmniWindow splits the original window into fine-grained sub-windows such that the sub-windows can be merged into various window types. To deal with the resource restriction, OmniWindow carefully designs its data plane memory layout and proposes a window synchronization method. It also employs a collaborative architecture that can collect and reset stateful data in sub-windows within a limited time. We prototype OmniWindow on Tofino. We incorporate OmniWindow into a SOTA query-driven telemetry system and eight sketch-based telemetry algorithms. Our experiments demonstrate that OmniWindow enables these telemetry solutions to achieve higher accuracy than conventional window mechanism.

CCS CONCEPTS

• **Networks** → **Network measurement**.

KEYWORDS

Window mechanism; Network telemetry

ACM Reference Format:
Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. 2023. OmniWindow: A General and Efficient Window Mechanism Framework for Network Telemetry. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604847

1 INTRODUCTION

Recent network telemetry solutions [74, 81] have followed two major trends. First, network telemetry solutions are built atop programmable switches to exploit the high-speed processing capacity and in-network insights. Second, they abstract network traffic as a packet stream and apply various stream processing techniques, including streaming operators (e.g., map, reduce, distinct) [24, 52] and streaming algorithms (e.g., sketch) [66, 68, 72]. In such stream-based network telemetry, windowing is a core component. In particular, the packet stream is unbounded and hence it is unrealistic to process it entirely. A window mechanism is responsible to split the packet stream into *windows* of finite size. Network telemetry solutions then apply streaming computations in each window to yield measurement results.

Existing telemetry solutions have very limited support for window mechanisms. Almost all the telemetry solutions use fixed-size tumbling window, while lacking support for other types of window mechanisms such as sliding window or variable window sizes. This void is due to the resource and programmability constraints of existing programmable switches [6]. First, existing switches employ a pipeline architecture that only allows single-pass processing for each packet, while complicated window mechanisms typically need to process a packet multiple times (e.g., repeatedly process one packet in multiple windows to which this packet belongs). If we build sliding window on the top of tumbling window, the data-plane sliding window requires multiple windows in one pipeline to process packets separately, whose resource overhead is unacceptable. Second, network telemetry usually stores intermediate data in in-memory states. However, the memory in switches is scarce due to concerns about heat consumption and chip footprints. This forbids large window size that needs more memory to hold telemetry states. Finally, when a window terminates, it requires to perform collect-and-reset (C&R) operations: (1) collect the states in the terminated window and (2) reset the states to process future traffic. Due to the lack of instructions to travel the entire states in switch ASICs, prior network telemetry solutions have to rely on switch OSes to perform the C&R operations. They suffer from poor performance and high collection errors due to the slow switch OS [12, 13, 19, 56].

In this paper, we present OmniWindow, a framework for network telemetry to support general and efficient window mechanisms. By generality, we mean that (1) OmniWindow supports different types of window mechanisms and variable window size, and (2) can be integrated with different types of network telemetry solutions, e.g., query-driven network telemetry [14, 24, 38, 52, 80] and sketch-based telemetry algorithms [27, 28, 46, 50, 66, 68, 72]. By efficiency, we mean that OmniWindow fulfills the constraints imposed by programmable switches. In addition, OmniWindow boosts the performance of C&R operations.

The key idea of OmniWindow is to split windows into fine-grained *sub-windows*. Then it allocates resources and performs

*Qun Huang is the corresponding author.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604847

measurement operations in the granularity of sub-windows. Sub-windows can be merged into windows of variable size or arbitrary window types (e.g., sliding window, tumbling window). However, sub-windows exacerbate the resource usage in programmable switches. First, sub-windows inevitably increase the frequency of window termination, which imposes a more strict time budget for C&R operations. Second, sub-windows require more frequent synchronization among switches to determine which sub-window a packet belongs to. Finally, each sub-window maintains its own states, which consumes more computational resources if we do not optimize per-window overheads.

To this end, OmniWindow proposes several techniques as follows.

- First, OmniWindow designs a fast and precise C&R approach. It proposes a flow-level abstraction namely application-derived flow record to eliminate the potential collection error. To reduce the C&R latency, OmniWindow directly connects the controller and the switch ASIC for bypassing slow switch OSes. Specifically, the controller injects special packets into the switch. The switch recirculates these special packets to mimic the enumeration operations. If the controller is equipped with RDMA, OmniWindow supports an RDMA-based optimization to further reduce the CPU overhead of the controller.
- Second, OmniWindow follows Lamport Timestamp [39] to design a lightweight consistency model for sub-window synchronization. It ensures that every packet is monitored at the same sub-window during its transmission even under network delays. This model embeds sub-window information in normal packets and propagates the information along the packet path. Thus, it only requires simple operations in switches and no additional messages are needed.
- Finally, OmniWindow performs careful state management to reduce hardware resource overhead. It employs a compact memory layout in which normal measurement operations and sub-window C&R operations can be performed simultaneously. In addition, the compact layout also reduces the number of used Stateful ALUs.

We prototype OmniWindow in DPDK [36] and P4. We integrate OmniWindow with a query-driven network telemetry Sonata [24] and eight sketch-based telemetry algorithms. Our testbed experiments demonstrate that with the aid of OmniWindow, network telemetry solutions can detect 14.3% more anomalies than traditional window mechanisms.

The source code of our OmniWindow prototype is now available at: **https://github.com/N2-Sys/OmniWindow**. This work does not raise any ethical issues.

2 PROBLEM

Network telemetry and window mechanism. In this paper, we focus on flow-level network telemetry based on programmable switches because they achieve high performance and in-network visibility. Recent network telemetry solutions (e.g., query-driven telemetry systems [24, 52], sketch-based telemetry algorithms [66, 68]) work in a *streaming* fashion. Specifically, telemetry solutions model network packets as an unbounded sequence. They calculate various flow-level statistics over the packet stream to summarize information for network management. Window mechanisms are the foundation of such stream-based network telemetry. To deal with the unbounded network traffic, telemetry solutions need to split the packet stream into *windows*, each of which contains finite packets such that we can apply telemetry operations. For example, query-driven telemetry provides stream operators to customize telemetry operations for each window. Sketch-based algorithms use compact data structures to summarize traffic statistics of all packets within one window, while incurring only bounded errors.

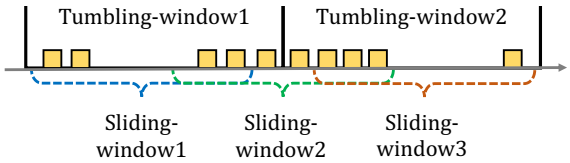


Figure 1: Limitation of fixed-size tumbling window.

Window management. A window mechanism is responsible to manage the resources to ensure the correctness and efficiency of network telemetry. In particular, network telemetry typically performs *stateful* operations: they hold *states* to record flow-level information, while each packet incurs an update to the states. Therefore, even though a window mechanism itself does not keep any state, it must provide sufficient resources to run the stateful telemetry applications. In addition, when a window ends, the window mechanism needs to perform collect-and-reset (C&R) operations: (1) collect the states in the terminated window and (2) reset the data structure to process the traffic of future windows.

Generality requirement. Ideally, a window mechanism should be general to support various telemetry purposes. First, a window can be of arbitrary size (**G1**). Second, a window can move forward by any distance (**G2**). For example, in *sliding window* mechanism, the window moves by a small step each time and windows overlap with each other.

Unfortunately, existing telemetry solutions based on commodity programmable switches only support fixed-size *tumbling window*. The tumbling window mechanism refers to windows that have no overlaps with each other. Figure 1 demonstrates that fixed-size tumbling window is not sufficient to provide fine-grained insights for network management. Here, we consider a telemetry application of heavy hitter detection. Assume a flow that has a burst near the boundary of two tumbling windows. It is possible that neither window reports this flow as a heavy hitter because the flow does not reach the heavy hitter threshold in either window. In contrast, if we use the sliding window, the flow must be reported as windows move forward. Note that such bursts concentrated around window boundaries are common in various anomaly detection telemetry applications. This accuracy improvement is significant to various measurement scenarios (e.g., performance evaluation, anomaly detection, network diagnose), improving their efficiency and usability in practical deployment [28, 29]. After identifying this flow, we may also want to examine more traffic in a longer period. For example, administrators are typically interested in the whole lifetime of each identified suspicious flow. Since these flows have different duration, the examined window size varies.

Resource constraints. The root cause that only time-based tumbling window is supported is that realizing general window mechanisms incurs nontrivial overheads. In particular, network devices that run telemetry applications impose various constraints on resources and programmability. Here, commodity programmable switches are based on reconfigurable match-action table (RMT) architecture [6], which consists of several *stages* arranged in a pipeline. We summarize the constraints of RMT switches:

- **C1**: C&R operations need to travel the entire memory region that holds window states. However, existing ASICs have no instructions for memory traversal. Thus, current telemetry solutions rely on a switch OS to issue C&R operations. Unfortunately, the switch OS incurs long latency due to its slow PCIe and RPC communication [13, 17–19, 56]. If C&R operations are not completed timely, the measurement accuracy will be degraded [12, 13].
- **C2**: Network-wide telemetry needs to synchronize windows among switches. Existing programmable switches use Precision Time Protocol (PTP) [1]. However, PTP is significantly affected by network workloads, and hence the accuracy varies from hundreds of nanoseconds to hundreds of microseconds [21, 42, 70].
- **C3**: The memory and computing resources to perform stateful operations are scarce. In [69], the on-chip memory (i.e., tens of megabytes) is partitioned between stages; and the number of *Stateful ALUs* (SALUs) per stage is less than eight. This directly limits the window size (G1).
- **C4**: The RMT pipeline employs a single-pass packet processing mode. And, the memory access operation of SALU in current ASICs is limited, i.e., each packet can only access a single location of each on-chip memory block. The aforementioned two points are in conflict with the sliding window (G2), where a packet needs to be monitored in contiguous and overlapping windows. This requires multiple copies of window resources in one pipeline to realize the sliding window, which is unacceptable.

3 OMNIWINDOW OVERVIEW

3.1 Architecture and Challenges

We propose OmniWindow, a general and efficient window mechanism framework for network telemetry based on programmable switches. As shown in Figure 2, OmniWindow employs a collaborative architecture that consists of a data plane and a control plane. In the data plane, OmniWindow splits *complete windows* into more fine-grained *sub-windows*. It then allocates resources and monitors traffic in the granularity of sub-windows. In the control plane, the controller collects and merges sub-windows to form complete windows as needed.

The sub-window approach fulfills the generality requirements in §2. First, when merging sub-windows, the controller tunes the number of sub-windows to achieve different sizes of the merged windows **(G1 achieved)**. Second, two merged windows can contain overlapping sub-windows, which realizes sliding window mechanism or more complicated window types **(G2 achieved)**.

Challenges. However, the sub-window granularity exacerbates the resource constraints of programmable switches in §2. For C1, sub-windows increase the frequency of window collection. The time budget for C&R operations is more strict. For C2, sub-windows incur more frequent synchronization among switches to determine the current sub-window. For C3, each sub-window maintains its own in-memory states. If we do not perform any optimization, the overall memory usage may exceed the switch capacity. For C4, since a pipeline stage has limited resources, the increasing sub-windows need to be deployed in more stages. This leads to more SALUs to locate state addresses.

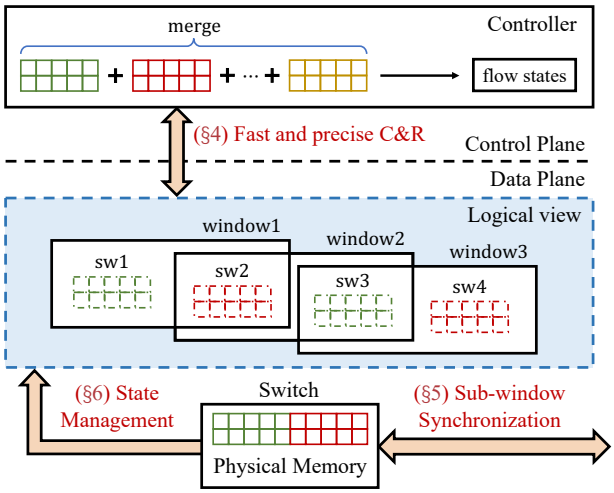


Figure 2: OmniWindow overview.

3.2 Solutions

OmniWindow proposes several techniques to address the challenges as follows.

Fast and precise C&R (§4). OmniWindow efficiently collects the telemetry data of each sub-window to merge the complete window by coordinating the control plane and data plane. We first design application-derived flow record (AFR) as a flow-level abstraction for telemetry data. AFRs can eliminate the controller overhead of querying states and the potential errors of merging sub-windows (e.g., information loss, counter conflicts). Then we propose a collaborative approach to speed up C&R operations. We inject collection packets from the controller to switch ASICs. The switch ASICs recirculate the injected packets to enumerate the states and generate AFRs. These injectes packets are reused to reset the states in the data plane, referred to *in-switch reset*. Since the controller and switch ASICs are connected directly, this approach eliminates the heavy overheads of switch OS **(C1 addressed)**. In addition, if the controller is equipped with RDMA, OmniWindow can further reduce the C&R time and the computational overheads of the controller.

Lightweight window synchronization (§5). OmniWindow follows the idea of Lamport Timestamp [39] to design a lightweight consistency model for sub-window synchronization. For each packet, OmniWindow determines its residing sub-window once by its first-hop switch. We leverage the capability of programmable switches to embed the sub-window information in the packet and propagate it to other switches. This ensures that each packet is monitored in the same window by all switches **(C2 addressed)**. This consistency model is lightweight in two aspects: (1) it involves simple

in-switch operations, and (2) incurs no extra messages. The consistency guarantee is significant to network telemetry. Taking packet loss detection as an example, assume that the values of one flow vary across switches. Without this guarantee, we cannot determine whether this is due to packet loss or whether the same packet is being measured in distinct windows on different switches.

State management (§6). We observe that only the current sub-window is under measurement at any time. To reduce memory usage, OmniWindow only maintains two memory regions for states: one is used by the current sub-window to monitor incoming traffic, while the other stores the information of previous sub-window and is being collected by the controller. Since the time of our fast and preceisr C&R is less than the sub-window size, the two regions are enough to empower the concurrent execution of multiple sub-windows **(C3 addressed)**. OmniWindow also designs a layout that concatenates the two regions such that one SALU is sufficient to locate values in all sub-windows **(C4 addressed)**.

4 FAST COLLECTION AND RESET

OmniWindow coordinates the control plane and data plane to collect sub-windows and merge them into complete windows. We first propose application-derived flow records (AFRs) that summarize the telemetry data of each sub-window to avoid the potential merging errors in §4.1. Then, we elaborate on the fast C&R approach, which efficiently collects AFRs (§4.2) and resets data-plane states (§4.3), bypassing the heavy switch OSes.

4.1 AFR Abstraction for Sub-Windows

Motivation. OmniWindow collects terminated sub-windows and merges them to form entire windows. There are two straightforward approaches. The first directly merges the measurement results of sub-windows. However, this approach fails to work for many telemetry applications, especially anomaly detection [32, 77]. We illustrate this issue using heavy hitter detection. In this case, we aim to identify flows whose packet counts exceed a threshold of 100. If a flow has 60 packets in one sub-window and 80 packets in the next, both sub-windows do not report this flow as a heavy hitter although its total packet count is above the threshold.

The second method collects the states of sub-windows and merges them into a state spanning multiple sub-windows. The controller raises queries to the merged state. However, given that many telemetry solutions are approximate, this approach amplifies the errors. For example, sketch-based telemetry algorithms allow each counter to be shared by multiple flows. Merging counters from multiple sub-windows leads to more flow conflicts in the counter. Note that existing approaches on merging sketch instances [2, 44] are limited to specific types of sketches and scenarios, lacking a general solution to address the error amplification brought by merging.

Application-derived Flow Record (AFR). To this end, OmniWindow proposes a flow-level abstraction to summarize sub-window states for further merging. In a nutshell, OmniWindow queries the desired statistics of each possible flow in a sub-window. The queried result forms an *application-derived flow record* (AFR) that consists of the flowkey and several flow attributes. These AFRs are sent to the controller that reconstructs the entire windows.

Algorithm 1 Flowkey tracking in the data plane

Input : receiving packet p with flowkey fk and metadata md
Data-plane States : `bloomfilter`, `fk_counter` (Initial value is 0)
Data-plane States : `fk_buffer` (flowkey array, size=M)
1: **MAINTAIN_FLOWKEY**
2: **if** fk **not in** `bloomfilter` **then**
3: Insert fk into `bloomfilter`
4: $md.index$ = `fk_counter++`
5: **if** $md.index >= M$ **then**
6: Clone p and send its copy to the controller
7: **else**
8: `fk_buffer`[$md.index$] = fk

Feasibility analysis. Note that the types of flowkeys and attributes vary across telemetry applications. For example, the AFR of heavy-hitter detection is {key: five-tuple, attr: packet count}, while the AFR of flow size estimation is {key: five-tuple, attr: bytes}. OmniWindow requires telemetry applications to explicitly specify the flowkey definition (e.g., five-tuple or IP address). It also requires that the data plane program of a telemetry application supports flow query. Fortunately, most telemetry applications allow such data plane flow query. For example, sketch-based telemetry solutions hash flowkeys to locate the corresponding counters. The AFRs are generated by simple operations supported by the data plane such as taking the minimum [15, 67, 68, 72]. Similarly, query-driven telemetry systems [24, 52, 80] typically employ hash-based structures to store flow information. OmniWindow constructs AFRs by mapping flowkeys in the hash-based structures. For those that do not support data plane query, we discuss them in §8.

Advantages of AFRs. Our abstraction of AFR brings two aspects of benefits. First, AFR is general because it provides flow-level information that can be seamlessly exploited by all flow-level telemetry applications. Second, AFR achieves high accuracy. Compared to directly merging measurement results that could lose partial information (e.g., non-heavy hitters in a sub-window), AFR offers more flows to the controller. Compared to merging sub-window states, AFR avoids additional counter conflicts. Finally, we eliminate the overhead of querying states in the controller.

4.2 AFR Operation

Lifetime of AFRs. We now elaborate on how to manipulate AFRs to realize fast and precise state collection. The complete lifetime of AFRs consists of the following six steps.

(1) During measurement runtime, OmniWindow keeps track of the flowkeys in each sub-window to generate AFRs.
(2) When a sub-window terminates, the switch invokes a query for each tracked flowkey to generate an AFR and then send it to the controller.
(3) The controller collects and stores these AFRs via the approach of PCIe-bypass collection.
(4) The controller merges the AFRs after all the sub-windows required for a window end.
(5) The controller queries the merged AFRs to complete the telemetry application.
(6) The controller releases the AFRs out of the window size.

Flowkey tracking for AFRs. A data plane switch needs to track active flowkeys of each sub-window such that it can query the

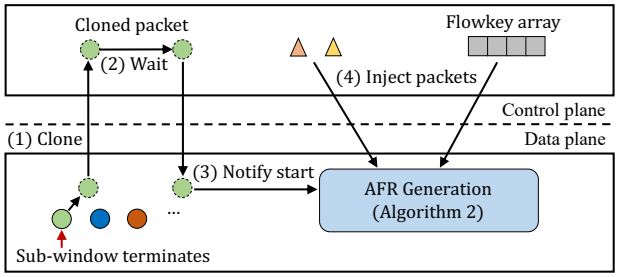


Figure 3: Workflow of AFR generation.

sub-window states to generate AFR. However, many telemetry solutions do not completely track flowkeys themselves. For example, UnivMon [50], Elastic Sketch [72], SeqSketch [28] only store heavy keys in the switch, while Sonata [24] and Count-Min Sketch [15] maintain no flowkeys.

OmniWindow proposes a small array to store the missing flowkeys in the data plane. We keep this array small to limit its resource usage. For the remaining flowkeys that cannot be stored in the array, OmniWindow sends them to the controller. To mitigate bandwidth usage, OmniWindow employs a Bloom Filter that eliminates duplicated flowkeys. Algorithm 1 presents the flowkey tracking operations. For each incoming packet, OmniWindow queries its fk in `bloomfilter` to check whether the key appears before (line 2). For a new flowkey, OmniWindow updates `bloomfilter` and appends fk into `fk_buffer` if the array is not full (lines 3,4,7,8). Otherwise, the cloned flowkey is sent to the controller (lines 5,6).

AFR generation. OmniWindow employs the controller and data plane switches to generate AFRs collaboratively. The basic idea is to inject packets from the controller such that the switches recirculate these injected packets to enumerate flowkeys and generate AFRs accordingly. Figure 3 illustrates the detailed workflow. Every time a sub-window terminates, the switch clones the packet that triggers the termination and sends the copy to the controller. After receiving this cloned packet, the controller waits for a certain time and then sends this packet back to the switch to start the AFR generation. In this way, the switch can deal with out-of-order packets before it starts AFR generation (see §5). Then, the controller injects a few (less than 20) special collection packets into the switch to enumerate flowkeys stored in the data plane. OmniWindow *recirculates* these packets to realize the enumeration. For the flowkeys stored in the controller, the controller also encapsulates these flowkeys into packets and sends them to the switch. The switch extracts the flowkeys, performs queries to generate AFRs, and then sends them back to the controller.

Algorithm 2 presents how the switch handles such special collection packets. The switch maintains a `counter` to control the in-switch enumeration. For each in-coming packet p, if p is a collection packet, the switch increments the `counter` as the index to read the flowkey in `fk_buffer` (lines 2,3,7). Then, the switch queries the flow attributes via the flowkey and appends the newly generated AFR into p (lines 8,9). Next, the switch clones p and sends its copy to the controller (line 10). At the same time, the original p is recirculated back to the ingress pipeline to move on the AFR generation (line 11). The enumeration ends when the `counter` reaches the

Algorithm 2 AFR Generation for flowkeys in the data plane

Input : packet p with its metadata md
Data-plane States : `counter`: Initial value is 0
Data-plane States : `fk_buffer` contains flowkeys of last sub-window
1: **AFR_Generation**
2: **if** $p.flag$ == COLLECTION **then** ▷ special collection packet
3: $md.index$ = `counter++`
4: **if** $md.index$ >= the number of flowkeys in `fk_buffer` **then**
5: $p.flag$ = RESET
6: Recirculate p to perform *in-switch reset* (see §4.3)
7: $md.flowkey$ = `fk_buffer`[$md.index$]
8: Perform query operation by $md.flowkey$ to generate an AFR
9: Append the newly generated AFR into p
10: Clone p and Send the cloned packet to the controller
11: Recirculate p to move on AFR generation

number of flowkeys in `fk_buffer` (line 4). Finally, the switch converts these special packets into *clear packets* and then recirculates them back to the first stage for in-switch reset (lines 5,6).

The recirculation has limited impacts on normal traffic for two reasons. First, a commodity switch ASIC has an independent recirculation hard-wired connection. Second, the number of recirculated packets is limited. Our evaluation (Exp#5 and Exp#7) shows that 16 packets are sufficient to obtain our desired performance of collection and reset.

Merging AFRs. The controller is responsible to form final flow-level results from the receiving AFRs. Note that a flowkey appears in multiple sub-windows. Thus, the controller aims to aggregate the flow statistics of a flowkey from multiple sub-windows. The aggregation depends on the property of flow statistics. Recent studies summarize that flow statistics follow four patterns [79]. OmniWindow employs different strategies for them.

- Frequency statistics: OmniWindow sums up the statistics from multiple sub-windows.
- Existence statistics: they indicate whether a flowkey appears. Thus, OmniWindow checks the flowkey in each sub-window to determine the existence.
- Max/min statistics: OmniWindow takes the maximum or minimum value among all sub-windows.
- Distinction statistics: they count the number of distinct values of a flowkey. Therefore, for each flowkey, OmniWindow first merges its distinct values of all sub-windows, and then counts the merged results.

OmniWindow stores the merged results in a key-value table in which each flowkey is associated with its merged statistics. Then OmniWindow performs queries in the flow-level key-value table. Taking heavy hitter detection as an example, OmniWindow traverses the key-value table to compare the merged result (i.e., the number of packets) of each flow with the threshold. Then, OmniWindow reports the detected heavy hitters to the user.

4.3 In-switch reset

OmniWindow reuses the collection packets injected by the controller as *clear packets* to reset the states of a terminated sub-window after AFR generation. Similar to AFR generation, OmniWindow recirculates the clear packets to enumerate the states in each register. OmniWindow controls the enumeration using another counter, namely `reset_counter`. When this counter reaches the number of

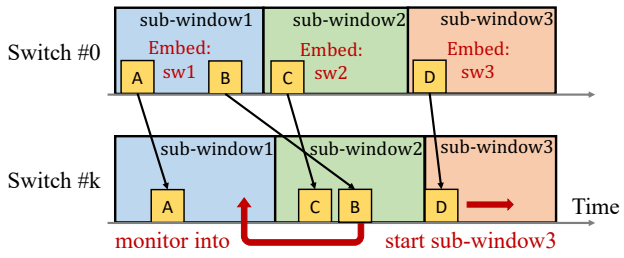


Figure 4: Lightweight consistency model.

states in the instance of terminated sub-window, the enumeration ends and all the clear packets are dropped. In each iteration, each clear packet resets the state at the position indexed by the number of terminated sub-window and `reset_counter`. As the AFR generation, the in-switch reset mechanism of OmniWindow also bypasses switch OSes and achieves low latency.

5 SUB-WINDOW SYNCHRONIZATION

OmniWindow proposes a lightweight consistency model to ensure that each packet is monitored in the same sub-window by all switches. This eliminates the measurement errors introduced by window splitting. In addition, OmniWindow provides rich signals to trigger sub-window termination.

Synchronization problem. Without a global clock, switches in a network work asynchronously, i.e., they are triggered by uncoordinated termination signals and hence reside in different sub-windows. Thus, two switches could monitor a packet in different windows. In this case, it is hard to exploit the measurement results for network management. For example, an administrator plans to compare the packet counts in switches to infer packet loss. If two switches monitor a packet in different windows, their packet counts differ. In this case, the administrator cannot determine whether the packet is actually lost.

Consistency model. OmniWindow deals with this issue by Lamport Timestamp [39]. For each packet, its sub-window is determined *only once* by the first switch in its path. This sub-window index is embedded in the packet and propagated along the path. Each remaining switch monitors the packet into its embedded sub-window, even though the embedded sub-window differs from the sub-window of the switch. In addition, if the embedded sub-window is newer, the switch updates its sub-window. In this way, OmniWindow guarantees that (i) each packet is always monitored at the same sub-window during its transmission (even under long network delays); (ii) window-moving signals are quickly propagated in the network.

Out-of-order packets. Our consistency model needs to deal with out-of-order packets. Here, we identify out-of-order packets by comparing their embedded sub-window number with the current sub-window of each switch. This eliminates additional storage of packet sequence numbers in the switch. Consider a packet that is delayed in a switch. When this packet arrives at the next switch, we must ensure that the embedded sub-window of the packet remains available in the new switch. To this end, OmniWindow preserves each sub-window for a certain time after the sub-window end,

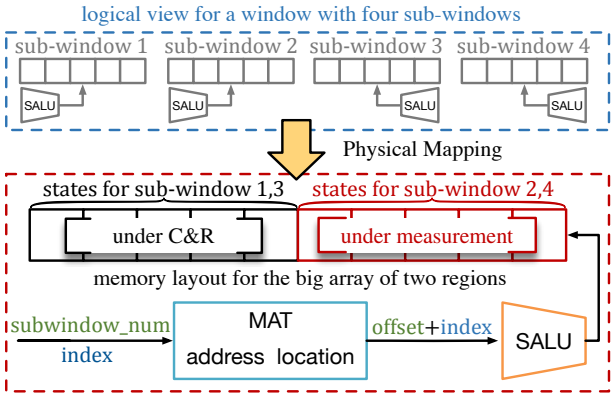


Figure 5: Shared memory regions with SALU optimization.

such that out-of-order packets can still be monitored into the sub-window. The preserving time depends on the maximum latency of the network. In a data center network whose latency is quite small, the number of preserving sub-windows is limited. For latency spikes in the network (i.e., packets that far exceed common network latency), we forward the copies of such packets to the controller. The controller can process these packets as needed. Since the packets affected by latency spikes are rare in the data center, the bandwidth overhead is negligible.

Example. Figure 4 illustrates how the consistency model of OmniWindow works. The first hop switch (Switch#0) monitors packet B in sub-window1 and embeds the `subwindow_num` into B. Although Switch#k is already in sub-window2 when B arrives, Switch#k extracts the embedded `subwindow_num` and monitors B in sub-window1. In addition, packet D triggers the window-moving of Switch#k from sub-window2 to sub-window3 instead of using the local clock of Switch#k.

Window termination signal. In OmniWindow, a sub-window terminates when it encounters a special signal. Currently, OmniWindow supports a rich set of signals that terminate sub-windows.

- **Timeout signals:** OmniWindow periodically generates timeout signals that yield new windows. Each timeout-based window is a time interval of fixed length.
- **Counter signals:** OmniWindow maintains additional counters for a telemetry application. Each counter is incremented by specific conditions (e.g., a counter for TCP packets). When a counter reaches a threshold, the telemetry application switches into a new window.
- **Session signals:** OmniWindow monitors the elapsed time since last packet. If there is no active traffic for a certain time, a session signal is generated to indicate the end of a window. Such session-based windows have different time lengths.
- **User-defined signals:** OmniWindow allows applications to embed window boundaries in packets. For example, distributed machine learning systems can consider a training iteration as a window and users may be interested in the traffic statistics across iterations. In this case, applications embed the (monotonically increasing) iteration number in packets. OmniWindow extracts them to determine the window to which a packet belongs.

6 STATE MANAGEMENT

OmniWindow proposes a compact memory layout with SALU optimization to address the additional hardware resource overhead brought by window splitting.

Shared memory regions. OmniWindow performs state management to eliminate additional resource usage due to the increasing number of sub-windows. As shown in Figure 5, OmniWindow maintains two memory regions and allocates them to different sub-windows. For example, we assign the first region to sub-window 1 and sub-window 3, while the other is used by sub-window 2 and sub-window 4. Note that only one sub-window is active at any time. Thus, when one region is under measurement by the active sub-window, OmniWindow can perform C&R operations on the other region. With our fast C&R design, the C&R operations can be completed within 2 ms, which is much smaller than typical sub-window sizes (e.g., 50 ms and 100 ms). Thus, two regions are sufficient to monitor continuously arrival packets.

Because the traffic in a sub-window is less than in an original window, we argue that the total memory usage in the regions of two sub-windows is limited. For a window containing W sub-windows, assuming that the traffic arrives at a uniform rate, the number of states (i.e., memory resource) that each sub-window needs to maintain is only $\frac{1}{W}$ of the origin window. Therefore, the memory overhead of deploying two memory regions is acceptable, which is $\frac{2}{W}$ of the original memory.

SALU optimization. However, shared memory regions incur high consumption of SALUs, since each memory region (i.e., on-chip register) requires a dedicated SALU. On the other hand, many telemetry solutions already consume multiple SALUs per stage. For example, most sketch instances [27, 46, 50, 62, 72] need to use many hash functions. If we deploy two regions in the data plane, the number of hash units and SALUs for the sketch instances directly doubles. This is unacceptable for scarce SALU resources (C3).

Therefore, our enhancement to shared memory regions focuses on alleviating the shortage of SALUs. As shown in Figure 5, OmniWindow employs a flattened memory layout. Specifically, OmniWindow concatenates the two memory regions into one big array. OmniWindow also installs the starting position (`offset`) of each region in a MAT. Thus, to access a specific value of the state in the big array, this MAT sums the `offset` decided by `subwindow_num` and the state `index` to locate the physical address. Then, the SALU performs the stateful operation at the calculated location. In this way, no matter how many regions we deploy, no additional SALUs are introduced.

7 RDMA-BASED OPTIMIZATION

As mentioned in §3, sub-windows raise more strict performance requirements on C&R operations, i.e., a C&R operation must be completed within the duration of one sub-window. Although we present a C&R design that bypasses switch OSes, OmniWindow supports leveraging RDMA to further reduce the time for sub-window collection. Note that our RDMA-based optimization can support more fine-grained sub-windows (e.g., 20 ms) to further reduce resource overhead for each sub-window.

Key idea. Our main idea follows recent studies that demonstrate that it is feasible to generate RDMA messages in programmable

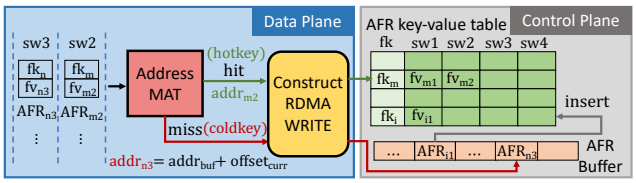


Figure 6: OmniWindow RMDA-based collection.

switches [34, 35, 41]. Thus, OmniWindow initializes an RDMA context and registers a memory region in the controller. When a switch needs to collect sub-window data, it encapsulates AFR messages into RDMA messages. The messages are directly written into the memory region in the controller without involving the controller CPU. This eliminates controller-side packet processing overheads, which are nontrivial based on recent reports [7, 45].

Hardness. One difficulty of our RDMA-based collection is that switches must be aware of the position to write in the controller. Existing methods let switches calculate global hash functions as the RDMA WRITE destinations [41], which needs extra communications to resolve hash conflicts. This is unacceptable for OmniWindow, compromising our low-latency requirements.

RDMA-based AFR collection. OmniWindow proposes a new approach in which the controller determines the memory addresses for RDMA-based AFRs and then notifies the switches. As shown in Figure 6, OmniWindow distinguishes hot keys and cold keys. Recall that the controller eventually merges AFRs in a key-value table. For hot keys, the controller notifies the base addresses of the key-value table. The switch directly writes the flow attributes to the table using their base addresses and sub-window indexes. In this way, the AFRs of different sub-windows are grouped based on keys. For cold keys, the controller allocates a buffer that temporarily stores AFRs. The controller continuously fetches AFRs from the array and inserts them into the key-value table as §4.2.

To realize this idea, each data plane switch maintains a match-action table (MAT) that caches memory addresses for hot keys, namely *address MAT*. To generate an AFR, the switch matches the flowkey in the address MAT. If the matching is successful, the switch constructs an RDMA message using the addresses in the address MAT. Otherwise, the RDMA message appends the AFR to the buffer. Since the buffer grows sequentially, the switch can calculate the memory address itself.

The controller monitors the hotness of each key. It sends notifications to the address MAT in the data plane. This notification includes the insertion of hot keys and the deletion of cold keys.

RDMA-based AFR aggregation. OmniWindow uses an RDMA atomic operation Fetch-and-Add to offload the operation of summing AFRs to the RNIC, which further reduces the CPU overhead of the controller. Although this optimization cannot be applied to all the merge operations, a considerable number of applications (i.e., frequency and distinction statistics) use the sum operation to merge AFRs. Thus, OmniWindow still benefits from this optimization to achieve CPU efficiency. Specifically, consider an AFR that knows its location in the key-value table via address MAT, OmniWindow directly sums the flow attribute with the merged result of prior sub-windows to update the controller memory using RDMA Fetch-and-Add.

8 IMPLEMENTATION

Switch. We implement the data plane component of OmniWindow in $P4_{16}$ [5] atop Intel Tofino ASIC [69]. OmniWindow places a customized header between the Ethernet and IP headers, in which the fields include the number of subwindow, collection/reset flag, and injected flowkey. The switch also inserts the generated AFR into this header. The RDMA request constructor crafts WRITE/Fetch-and-Add requests using the RoCEv2 [60] protocol, which maintains a register to count the packet sequence number.

Controller. The OmniWindow controller is responsible for four functionalities. First, the controller injects flowkeys and receives generated AFRs via DPDK [36] to remove kernel-space overhead. Second, the controller stores the AFRs in the DPDK hash table `rte_hash` for high performance, which adopts many optimizations (e.g., cache-line alignment, huge page, `crc` instructions from Intel SSE4.2 [58], and etc.). Third, the controller merges the AFRs using `AVX-512` [57] (the latest SIMD instructions), which can perform the same operation (e.g., sum, max, min, and comparison) on multiple AFRs using a single instruction. This instruction-level parallelism mitigates processing latency. Finally, the controller maintains the RDMA context for RDMA-based optimization. It notifies RDMA memory address via gRPC [23].

Merging intermediate data without AFRs. In some telemetry solutions, the data plane is only responsible for updating states while flow statistics must be computed in the controller. For example, FlowRadar [46] needs to decode in the controller and NZE [28] recovers the flow attributes by solving an optimization problem. For these solutions, we cannot generate AFRs in the data plane. Instead, OmniWindow migrates the entire states to the controller for constructing and merging AFRs. Similar to AFR generation, OmniWindow recirculates and clones special packets to send the states in the register iteratively.

Reliability of AFRs. Since the priority of cloned packets is the lowest, OmniWindow needs to handle the AFR losses (e.g., due to network congestion). Specifically, for each terminated sub-window, the switch notifies the controller of the number of flowkeys in its flowkey array via the trigger packet. The switch also encodes a unique sequence ID for each flowkey. After AFR generation, the controller examines whether all flowkeys are received. If not, the controller asks the switch to retransmit the missing flowkey.

9 EVALUATION

9.1 Setup

Testbed. We deploy our OmniWindow prototype in five servers and two Intel Tofino switches [69]. The servers are equipped with two 12-core 3 GHz CPUs and 256 GB RAM. Each server is connected to both of the switches via a dual-port 100 Gbps NVIDIA ConnectX5 Ethernet-RNIC [9]. This topology enables the packets to pass through one or two switches, depending on the experimental requirements. We run the OmniWindow controller on one of the servers. Each of the other four servers is equipped with one NVIDIA GeForce RTX 3090 GPU.

Workloads and window parameters. We use a real-world packet trace from CAIDA [8] to simulate the data center traffic. We use PktGen [55] to inject the trace from the four servers into the switch for stress testing. We monitor the traffic using time-based tumbling window and sliding window. The `window_size` is 500 ms for both types and `slide_size` is 100 ms for sliding window. We split each window into five sub-windows (100 ms). Each window contains around 213 K∼440 K active flows, while each sub-window contains around 64 K∼96 K flows. Since the traffic in the trace arrives at a non-uniform rate, we allocate $\frac{1}{4}$ memory resources of the original window for each sub-window instead of $\frac{1}{5}$.

#	Telemetry Application Description
	Used in query-driven telemetry (Exp#1).
Q_1	Detect hosts which open too many new TCP connections [76]
Q_2	Detect hosts under SSH brute force attack [31]
Q_3	Detect hosts under port scanning [33]
Q_4	Detect hosts under DDoS attack [75]
Q_5	Detect hosts under SYN-flood attack [76]
Q_6	Detect hosts with too many incomplete TCP connections [76]
Q_7	Detect hosts under Slowloris attack [76]
	Used in sketch-based algorithm (Exp#2).
Q_8	Detect super-spreaders [75]
Q_9	Detect heavy-hitters [15]
Q_{10}	Monitor the number of flows [26]
Q_{11}	Monitor the total packet size of each flow [52]

Table 1: Telemetry applications in evaluation.

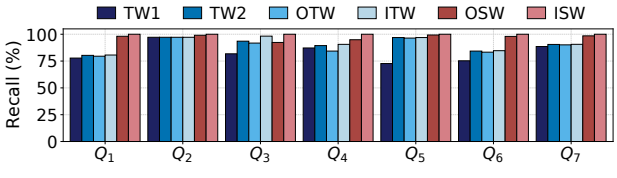
9.2 Telemetry with OmniWindow

Methodology. We evaluate 11 telemetry applications as listed in Table 1. These applications are representative, which are widely evaluated in existing telemetry works [24, 28, 72, 80]. We measure the tumbling window and sliding window realized by OmniWindow. We denote them by OTW and OSW, respectively. We also evaluate two tumbling window implementations in existing telemetry solutions: TW1 performs C&R operations of the old window and starts the measurement of the new window on the same memory region immediately; TW2 deploys two memory regions to perform the measurement in one region and C&R operations in the other. We use the ideal tumbling window (ITW) and ideal sliding window (ISW) as the ground truth, which are computed offline using error-free data structures.

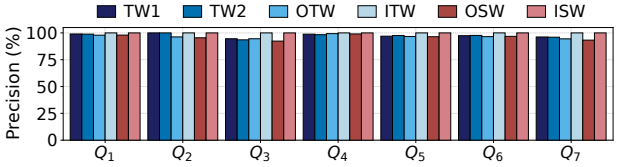
(Exp#1) Query-driven telemetry. We integrate OmniWindow with a state-of-the-art query-driven telemetry Sonata [24] for seven anomaly detection applications (Q_1-Q_7). Here, existing query-driven telemetry systems [24, 29, 38, 80] are all based on the tumbling window mechanism. Figure 7 compares their accuracy under different window settings.

Benefit of sliding window (*ITW v.s. ISW*): ITW has the same precision as ISW, but its recall is only 92.2% of ISW on average. Here, both ITW and ISW are measured using error-free structures, which implies that tumbling window is easy to miss anomalies at window boundaries.

Impact of C&R time on accuracy (*TW1 v.s. TW2*): Because TW1 measures traffic of the new window and performs C&R operations of the old window at the same memory region, some traffic during the C&R operations is not measured correctly, resulting in its recall being 2%(Q_7) to 24%(Q_7) lower than TW2. Here, the recall of Q_2 is not affected because there is very little SSH traffic in this trace. This tells us that the completion time of C&R has a significant impact on accuracy.



(a) Recall.



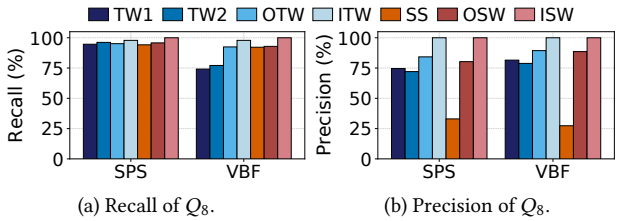
(b) Precision.

Figure 7: (Exp#1) Query-driven telemetry.

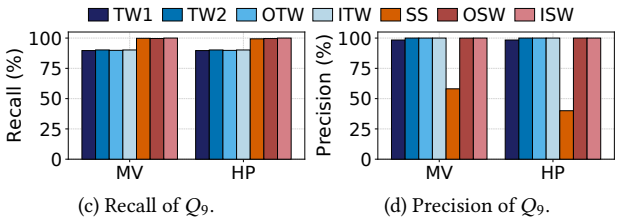
Accuracy of OmniWindow: For tumbling window (*OTW v.s. TW2 v.s. ITW*), the accuracy of OTW is very close to that of ITW, which is only 2% (recall) and 3% (precision) lower than ITW. In addition, OTW achieves almost the same accuracy as TW2, but incurs only $\frac{1}{4}$ memory resources. For sliding window (*OSW v.s. ISW*), the accuracy difference between OSW and ISW is also very small. The recall and precision of OSW are 2.8% and 4.1% lower than ISW on average. The error is caused by the fact that the stateful operators of Sonata do not handle hash conflicts, which cannot be avoided by OmniWindow. This indicates that OmniWindow can achieve high accuracy close to the ideal window mechanisms.

(Exp#2) Sketch-based algorithm. Figure 8 compares the accuracy of sketch-based algorithms under different window settings. We apply OmniWindow to eight different sketch-based algorithms to monitor Q_8-Q_{11}. Specifically, we use SpreadSketch (SPS) [68] and Vector Bloom Filter (VBF) [48] for Q_8, MV Sketch (MV) [67] and HashPipe (HP) [62] for Q_9, Count-Min Sketch (CM) [15] and SuMax Sketch (SM) [78] for Q_{10}, and Linear Counting (LC) [71] and HyperLogLog Counting (HLL) [25] for Q_{11}. In addition to the above six window settings, we also compare OmniWindow with Sliding Sketch (SS) [22], a general sketch framework that supports sliding window for common sketch algorithms. We implement the basic design of Sliding Sketch, which extends each bucket of Count-Min Sketch and SuMax Sketch into two buckets. One bucket stores the information of the latest tumbling window, and the other stores the telemetry data of the previous tumbling window.

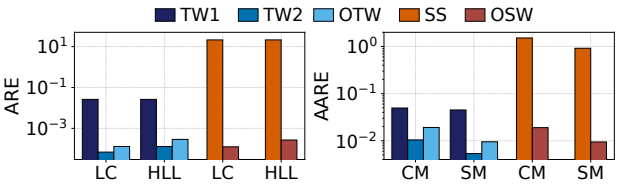
Sketch configuration and metrics. In the experiments, we allocate 8 MB of memory for each original window. In particular, SpreadSketch, MV Sketch, HashPipe, Count-Min Sketch, and SuMax Sketch are all two-dimensional arrays. The depth of them is set to 4, and their width is calculated according to the depth and the memory usage of each bucket. We set the same depth but half width for the Count-Min Sketch and SuMax Sketch in Sliding Sketch to ensure the same memory resource occupation. For Vector Bloom Filter, we have five arrays, each of which contains 4096 bitmaps. For HyperLogLog, each bucket is one byte long. We compute the average relative error (ARE) for Q_{10} and average ARE (AARE) for Q_{11}. Notice that for Q_{10} and Q_{11}, the results of the tumbling window and the sliding window are not comparable, because the sliding window monitors more windows.



(a) Recall of Q_8. (b) Precision of Q_8.



(c) Recall of Q_9. (d) Precision of Q_9.



(e) ARE of Q_{10}. (f) AARE of Q_{11}.

Figure 8: (Exp#2) Sketch-based algorithms.

Accuracy of OmniWindow tumbling window (*OTW v.s. TW2 v.s. ITW*): The errors of OTW and TW2 are at the same level. For example, the AAREs of Count-Min Sketch with OTW and TW2 are 1.9% and 1.0%, respectively. In theory, the error of OTW should be slightly higher than that of TW2 because the memory resources of OTW are only $\frac{1}{4}$ of the original window. However, the precision of OTW for Q_8 is higher than that of TW2, since its hash conflicts in each sub-window are fewer.

Accuracy of OmniWindow sliding window(*OSW v.s. SS v.s. ISW*): The accuracy of OSW is much higher than that of Sliding Sketch. In particular, the precision of OSW for Q_8 - Q_9 is about 50% higher than SS on average. For Q_{11}, the AARE of OSW is two orders of magnitude smaller than Sliding Sketch. The reason is that the estimated results of Sliding Sketch actually contain information of 1 $\frac{k+2}{k}$ sliding windows, where k is the number of hash functions [22]. Since Count-Min Sketch and SuMax Sketch only have overestimate error, combining Count-Min Sketch and SuMax Sketch with Sliding Sketch inevitably causes overestimation. OSW tracks the information inside a window, so it gives a much more accurate estimation than Sliding Sketch. The difference of accuracy for Q_8 - Q_9 between OSW and ISW is small (e.g., the precision of MV reaches 99.9%).

(Exp#3) Case study: Usage in distributed machine learning. This experiment uses distributed machine learning application as a case to demonstrate how OmniWindow uses user-defined signals to monitor traffic within the application. As mentioned in §5, the application embeds the number of current training iteration into packets. OmniWindow extracts them to measure the time of each iteration, i.e., the switch records the timestamps of the first and the last packet in the same iteration.

In this case, we set up a distributed machine learning application on a cluster of four hosts. We use the parameter server architecture,

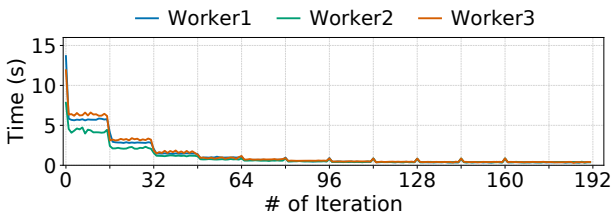
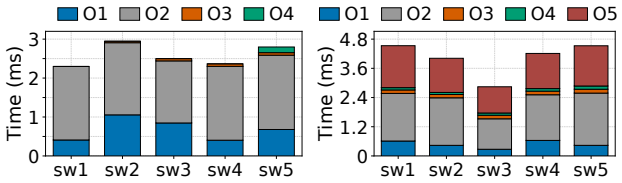


Figure 9: (Exp#3) Case study: Usage in distributed machine learning.



(a) Tumbling window. (b) Sliding window.

Figure 10: (Exp#4) Controller time usage breakdown.

one host works as a server node and the others serve as *worker* nodes. We use VGG19 [61] model and CIFAR-10 [37] dataset. As shown in Figure 9, OmniWindow properly measures the training time of each worker for each iteration. We use a dynamic compression ratio to compress gradients, which starts from 2 and doubles every 16 iterations until it reaches 2048. Since the initial transmitted gradients are large in volume, the training overhead (mainly composed of gradient transmission) of each worker varies significantly. Although this case can be monitored at the end-host, it demonstrates that OmniWindow can measure in-application traffic. This provides more fine-grained analysis for some high-performance applications or other in-network computing applications.

9.3 MicroBenchmark of OmniWindow

(Exp#4) Controller time usage breakdown. This experiment uses Q_1 as an example to show the time overhead breakdown of the OmniWindow controller in a complete window (five sub-windows). We present five operations of the controller: (O1) collect AFRs of each sub-window; (O2) insert AFRs into the key-value table; (O3) merge the AFRs of each flow; (O4) process the merged result after a complete window; and (O5) update the merged result by removing the oldest sub-window (only for sliding window).

For tumbling window (Figure 10(a)), the average processing time for all sub-windows of the controller is 2.58 ms. Even for the longest sub-window (sw5), it only takes no more than 3 ms. The heaviest operation is the insertion of AFR into the key-value table (O2), which takes 1.8 ms on average. The significant difference of O1 (404 μs~1055 μs) is because the flowkey array caches all the flowkeys in some sub-windows (sw1 and sw4), which requires no injection of flowkeys from the control plane. For sliding window (Figure 10(b)), the average processing time of the controller is 4 ms. Although the average time is 1.42 ms more than that of the tumbling window, it is still two orders of magnitude lower than the duration of the sub-window. The major time increase comes from O4 and O5, which incurs 100 μs and 1.46 ms on average, respectively. For O4, the sliding window needs to process the merged result after each sub-window, while the tumbling window only processes the merged result once after the complete window. For O5, the sliding window needs to remove the oldest sub-window, including updating the merged value and deleting the flows that only appear in the oldest sub-window from the key-value table.

(Exp#5) Switch resource breakdown. Table 2 presents the switch hardware resource usage of each OmniWindow design feature. Here, we use Q_1 as an example due to the page limit. Note that stage and VLIW can be shared by different features, which explains why their total values are lower than the summing resources of each feature directly. We normalize our resource usage to the total resources used by switch.p4 and Q_1 without OmniWindow. OmniWindow incurs high resource usage in SALU, VLIW, and gateway. For SALU (44.4%), OmniWindow needs to access signals, various counters, and flowkey tracking structures. For VLIW (40.7%), OmniWindow incurs many physical address calculations. For gateway (44.9%), OmniWindow differentiates normal and special traffic. Fortunately, the combined resource usage of Q_1 and switch.p4 is less than half of all available hardware resources, which still remains enough resources to support more telemetry solutions. Although some features of OmniWindow (e.g., consistency model, address location) introduce additional resource overhead, this is acceptable since sub-windows effectively reduce the resources required for the complete window, leaving enough resources for other features.

Feature	Switch Resources				
	Stage	SRAM	SALU	VLIW	Gateway
Signal	1	32 KB	1	3	2
Consistency model	1	0 KB	0	2	1
Address location	1	16 KB	0	2	0
Flowkey tracking	4	624 KB	4	7	7
AFR generation	1	0 KB	0	4	3
RDMA opt.	5	928 KB	2	20	13
In-switch reset	3	32 KB	1	5	5
Total	8	1632 KB	8	35	31
Normalized by that of (Q_1 + switch.p4)	75%	14.7%	44.4%	40.7%	44.9%

Table 2: (Exp#5) Switch resource breakdown of Q_1.

(Exp#6) Time of AFR generation and collection. We compare the time overhead of different methods for AFR generation (in the data plane) and collection (in the control plane). In this experiment, we use Q_{11} realized by Count-Min Sketch [15] as an example. We allocate 128 KB of memory for one state array in Count-Min Sketch and change its number of hash functions from 1 to 4. The number of flowkeys is 64 K. We measure the hybrid collection method of OmniWindow, denoted as OW. The data-plane flowkey array in OW caches 32 K flowkeys, thus the controller injects the remaining flowkeys for AFR generation. In addition, we provide two alternative methods for comparison: control-plane collection (CPC) injects 64 K flowkeys to generate and collect AFRs; data-plane collection (DPC) enumerates all the 64 K flowkeys for AFR collection by recirculating special packets. We apply our RDMA-based optimizations to these three methods, namely OW*, CPC*, and DPC*. We recirculate three special packets for OW and DPC, since DPDK cannot handle the PPS caused by more than three packets. For OW* and DPC*, we recirculate 16 packets, which can achieve desired performance and can be handled by RNIC. We also provide the result of the conventional switch OS method (OS).

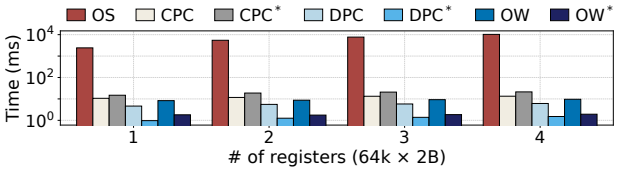
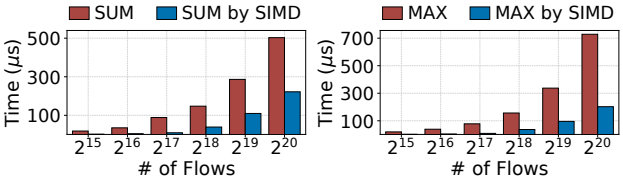


Figure 11: (Exp#6) Time of AFR generation & collection.



(a) Sum. (b) Max.

Figure 12: (Exp#7) Time of AFR aggregation.

Figure 11 shows the time used by these seven methods. For the OS-based method, it requires several seconds (2.4 s~10.3 s) to read and transmit the complete Count-Min Sketch to the controller, not including the query time. The other methods that bypass switch OSes only need a few milliseconds (1 ms~21.2 ms). Among them, the average time of CPC (12 ms) and CPC* (19 ms) is longer, in which the majority of time comes from the injection time of the controller. Additionally, CPC* uses more time than CPC. The reason is that it needs to look up the address in the key-value table for each flowkey and then inject it with the flowkey into the switch. On the contrary, DPC and DPC* only require 5.5 ms and 1.3 ms on average without injection overhead. However, they incur large memory overhead for the flowkey array and the address table of DPC*. OmniWindow achieves the trade-off between CPC and DPC. OW reduces the number of injected flowkeys by caching some flowkeys in a small amount of memory. OmniWindow takes 9 ms (OW) and 1.8 ms (OW*) on average, which achieves the same order of latency as the DPC.

(Exp#7) Time of AFR aggregation. We compare the time overhead of two common AFR aggregation operations (i.e., sum and max) with and without SIMD optimization. Here, we do not evaluate the merge operation for existence and distinction statistics, because these two cannot be optimized by SIMD. As shown in Figure 12, even without the SIMD optimization, the time to merge 1 M flows is less than 1 ms (502 μs for sum and 728 μs for max). The time overhead optimized by `avx512` is reduced by 75.6% (sum) and 81.2% (max) on average. This demonstrates that the time overhead of the merge operation is two to three orders of magnitude lower than the duration of a sub-window.

(Exp#8) Time of in-switch reset. Figure 13 compares the reset time between the conventional switch OS approach and OmniWindow. In the switch, we deploy four registers, each of which has 64 K two-byte entries. For our in-switch reset, OmniWindow simultaneously recirculates 4, 8, and 16 clear packets separately, namely OW-4, OW-8, OW-16. The reset time of the OS-based method increases linearly with the number of state registers to be reset. Because the OS-based method does not support concurrent resets of multiple registers. On the contrary, the number of registers does not affect the time needed for OmniWindow, since each clear packet can reset

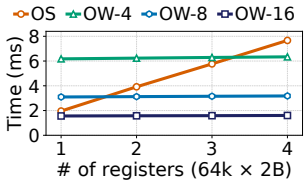


Figure 13: (Exp#8) Time of in-switch reset.

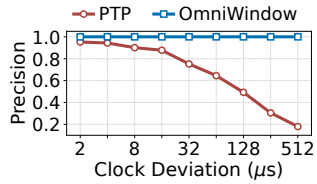
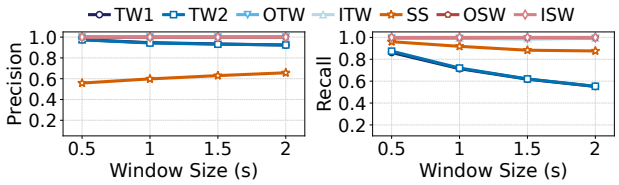


Figure 14: (Exp#9) Consistency.



(a) Precision of MV Sketch. (b) Recall of MV Sketch.

Figure 15: (Exp#10) Accuracy under different window sizes.

the same position of all the registers in one pipeline pass. Here, OmniWindow only requires less than 2 ms to clear any number of 128 KB registers, whose latency is much lower and more stable than the OS-based method.

(Exp#9) Consistency. In this experiment, we compare the consistency model of OmniWindow with PTP synchronization. In particular, we deploy two instances of LossRadar [47] in two adjacent switches to detect the flows incurring packet loss on the link. We tune the clock deviation between switch local clocks synchronized by PTP from 2 to 512 μs. As mentioned in §2, the clock deviation of PTP can vary from hundreds of nanoseconds to hundreds of microseconds, so this setting is reasonable. Figure 14 shows that the consistency model of OmniWindow always achieves a high precision (100%). This means that OmniWindow can fully guarantee consistency between switches. However, the precision of the local clock decreases as the clock deviation increases. When the deviation is 128 μs, the precision even decreases to 49.2%, because the local clock causes some packets to be monitored in different sub-windows. Note that such measurement error is amplified as the number of switches along the packet transmission path increases. This is because, in addition to clock deviation between switches, the accumulated packet transmission delay increases the risk of packets being measured into the wrong sub-windows.

(Exp#10) Accuracy under different window sizes. As shown in Figure 15, we use Q_8 (heavy-hitter detection) as an example to compare the accuracy of OmniWindow under different window sizes. We apply OmniWindow to MV Sketch [66], allocating four independent hash functions and 8 MB of memory for the sketch instance. We still set the sub-window size to 100 ms and allocate 2 MB for each sub-window. Here, we vary the window size from 0.5 seconds to 2 seconds.

Accuracy of tumbling window (*OTW v.s. TW1 v.s. TW2 v.s. ITW*): As the window size increases, the accuracy of OmniWindow (OTW) is more stable than that of existing tumbling window implementations (TW1 and TW2). In particular, the recalls and precisions of OTW are always close to 100%. But for TW1 and TW2, their recalls drop from 86% to 55% and their precisions decrease from 97%

to 92%. The reason is that TW1 and TW2 allocate memory resources according to a pre-defined window size, which is insufficient to measure additional traffic introduced by the increased window size. On the contrary, OmniWindow measures and allocates resources at the granularity of sub-windows. No matter how user-desired window size changes, our sub-window size remains unchanged. Thus, the resources allocated by OmniWindow can guarantee the measurement accuracy of each sub-window, and hence the accuracy for arbitrary window size.

Accuracy of sliding window (*OSW v.s. SS v.s. ISW*): OmniWindow (OSW) can achieve comparable accuracy as the ideal sliding window (ISW) and higher accuracy than Sliding Sketch (SS). The recalls of Sliding Sketch drop from 96% to 87.6% for MV Sketch as the window size increases. The reason is that more hash conflicts happen as the window grows. Thus, more true heavy hitters will be ignored due to their collision with small flows. The precisions of Sliding Sketch float around 60% for MV Sketch. As discussed in Exp#2, the estimated results of Sliding Sketch actually contain information of more than one sliding window. Some flows are overestimated by Sliding Sketch and the overestimated flows are likely to be wrongly reported as a heavy hitter. Thus, the precisions of Sliding Sketch are small.

Remark. This experiment demonstrates that OmniWindow can realize arbitrary window size with fewer resource overhead and higher accuracy than existing implementations for both tumbling window (TW1 and TW2) and sliding window (Sliding Sketch).

10 RELATED WORKS

Window mechanisms. Recent telemetry solutions atop commodity programmable switches have explored universal measurements (i.e., support various telemetry applications) with limited window mechanism support. One class is query-driven telemetry [24, 29, 38, 80], which provides the expressive telemetry language to customize different applications. Another class is universal sketch-based algorithms [27, 28, 46, 50, 72]. These solutions only support time-based tumbling window due to the resource and programmability constraints of existing switch ASICs.

Some classic algorithms running at end-hosts offer approximate measurement inside the sliding window. They can be divided into three categories: (i) membership query [11, 64, 73]; (ii) frequency estimation [10, 16, 54, 59]; and (iii) heavy hitter detection [4, 30, 43]. However, these algorithms focus on one specific application in the sliding window, hence lacking generality. In addition, they require a lot of memory to achieve fine-grained deletions. Most algorithms cannot be realized in the data plane limited by the resource and programmability of commodity switches.

The most competitive work is Sliding sketch [22], which provides a general framework for data stream processing in sliding window. However, Sliding sketch is not designed for network telemetry atop programmable switches, which lacks an efficient C&R approach. OmniWindow proposes a fast and precise C&R and supports more termination signals.

Window overhead solutions. Existing works propose solutions for different types of window overheads. First, OmniMon [29] uses multiple memory regions to hide the time overhead of C&R operations. OmniWindow further optimizes shared memory regions to reduce the SALU overhead. Second, OmniMon designs a hybrid consistency model which determines the measurement epoch at the end-host, while OmniWindow decides by data-plane consistency model with no modification to end-hosts. Third, many systems [13, 40, 41, 51, 82] take the idea of transferring the states from the switch ASIC to the end-host bypassing heavy switch OS. Among them, ALT [51], DART [41], and DTA [40] also use RDMA to collect the telemetry data. However, ALT only can sequentially write to a block of host memory; and DART incurs higher bandwidth overhead by writing each data to multiple addresses. OmniWindow devises a new RDMA-based collection method, which caches the addresses of hot keys in a MAT. DTA provides a generic and systematic solution to collect telemetry data by introducing the translator and providing switch-level RDMA-extension primitives. We plan to integrate OmniWindow with DTA in our future work.

Window state merging. OmniWindow utilizes AFRs to merge sub-windows. Similarly, TurboFlow [63] generates microflow records (mFRs) at switch ASICs and merges the mFRs into full FRs at switch OSes. The major difference is that TurboFlow aims for generating rich FRs based on the lifetime of flows instead of the window mechanism. Existing studies on the mergeability of sketch-based algorithms [2, 3, 20, 44, 53] focus on the merging of distributed sketch instances. These algorithms are merged in the space dimension, while OmniWindow is in the time dimension.

Telemetry architecture. OmniWindow follows the line of recent studies that combine multiple types of network entities to perform measurement collaboratively. Marple [52] realizes an in-network cache for the telemetry key-value store at end-hosts. The switches and end-hosts are coordinated by MOZART [49] to select and monitor desired traffic. Sonata [24] offloads partial stream operations to the switch in order to reduce the computational overhead at the end-host. SwitchPointer [65] views the switch memory as a directory service that records the position of the telemetry data at end-hosts. OmniMon [29] integrates the advantages of switches, end-hosts, and the controller to achieve both resource efficiency and full accuracy.

11 CONCLUSION

OmniWindow is a general and efficient window mechanism framework for network telemetry based on programmable switches. Its design principle is to split the window into more fine-grained sub-windows, which can be merged into different types of windows with variable window sizes. We achieve efficiency with a collaborative architecture which maps the key functionalities of the window mechanism to the switch and the controller, respectively. Experiments show the effectiveness of OmniWindow over both query-driven telemetry systems and sketch-based telemetry algorithms.

ACKNOWLEDGMENTS

We thank the anonymous shepherd and reviewers for their valuable comments. The work was supported in part by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007).

REFERENCES

[1] 2008. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008* (2008).

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems* 38, 4 (2013).

[3] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In *Proc. of ACM STOC*.

[4] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. In *Proc. of IEEE INFOCOM*.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *Proc. of ACM SIGCOMM CCR* 44 (2014), 87–95.

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. of ACM SIGCOMM*.

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proc. of ACM SIGCOMM*.

[8] Caida Anonymized Internet Traces 2018 Dataset. [n. d.]. http://www.caida.org/data/passive/passive_dataset.xml. ([n. d.]).

[9] NVIDIA Mellanox ConnectX-5 EN Card. [n. d.]. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf. ([n. d.]).

[10] Ho-Leung Chan, Tak-Wah Lam, Lap-Kei Lee, and Hing-Fung Ting. 2012. Continuous monitoring of distributed data streams over a time-based sliding window. *Algorithmica* 62, 3 (2012).

[11] F. Chang, Wu chang Feng, and Kang Li. 2004. Approximate caches for packet classification. In *Proc. of IEEE INFOCOM*.

[12] Xiang Chen, Qun Huang, Peiqiao Wang, Hongyan Liu, Yuxin Chen, Dong Zhang, Haifeng Zhou, and Chunming Wu. 2021. MTP: Avoiding Control Plane Overload with Measurement Task Placement. In *Proc. of IEEE INFOCOM*.

[13] Xiang Chen, Qun Huang, Dong Zhang, Haifeng Zhou, and Chunming Wu. 2020. ApproSync: Approximate State Synchronization for Programmable Networks. In *Proc. of IEEE ICNP*.

[14] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proc. of ACM SIGCOMM*.

[15] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms* (2005), 58–75.

[16] Graham Cormode and Ke Yi. 2011. Tracking distributed aggregates over time-based sliding windows. In *Proc. of ACM PODC*.

[17] Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. 2011. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proc. of IEEE INFOCOM*.

[18] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. of ACM SIGCOMM*.

[19] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. of ACM SIGCOMM*.

[20] Joan Feigenbaum, Sampath Kannan, Martin J Strauss, and Mahesh Viswanathan. 2002. An approximate L 1-difference algorithm for massive data streams. *SIAM J. Comput.* 32, 1 (2002).

[21] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosunblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-Grained Clock Synchronization. In *Proc. of USENIX NSDI*.

[22] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. 2020. Sliding Sketches: A Framework Using Time Zones for Data Stream Processing in Sliding Windows. In *Proc. of ACM SIGKDD*.

[23] gRPC. [n. d.]. https://grpc.io/. ([n. d.]).

[24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*.

[25] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In *Proc. of EDBT*.

[26] Qun Huang, Xin Jin, Patrick P C Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*.

[27] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*.

[28] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward Nearly-Zero-Error Sketching via Compressive Sensing. In *Proc. of USENIX NSDI*.

[29] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy. In *Proc. of ACM SIGCOMM*.

[30] Regant YS Hung, Lap-Kei Lee, and Hing-Fung Ting. 2010. Finding frequent items over sliding windows with constant update time. *Inform. Process. Lett.* 110, 7 (2010).

[31] Mobin Javed and Vern Paxson. 2013. Detecting Stealthy, Distributed SSH Brute-Forcing. In *Proc. of ACM SIGSAC*.

[32] Lavanya Jose, Minlan Yu, and Jennifer Rexford. 2011. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Proc of USENIX Hot-ICE*.

[33] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. 2004. Fast portscan detection using sequential hypothesis testing. In *Proc. of IEEE Symposium on Security and Privacy*.

[34] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proc. of ACM SIGCOMM*.

[35] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *Proc. of ACM HotNets*.

[36] Data Plane Development Kit. [n. d.]. https://www.dpdk.org/. ([n. d.]).

[37] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[38] Paolo Laffranchini, Luis Rodrigues, Marco Canini, and Balachander Krishnamurthy. 2019. Measurements As First-class Artifacts. In *Proc. of IEEE INFOCOM*.

[39] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978).

[40] Jonatan Langlet, Ran Ben-Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. 2023. Direct Telemetry Access. In *Proc. of ACM SIGCOMM*.

[41] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. 2021. Zero-CPU Collection with Direct Telemetry Access. In *Proc. of ACM HotNets*.

[42] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proc. of ACM SIGCOMM*.

[43] Lap-Kei Lee and HF Ting. 2006. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proc. of ACM PODS*.

[44] Jakub Lemiesz. 2021. On the Algebra of Data Sketches. *VLDB Endowment* 14, 9 (2021).

[45] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: Datacenter Sockets Can Be Fast and Compatible. In *Proc. of ACM SIGCOMM*.

[46] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of USENIX NSDI*.

[47] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proc. of ACM CoNEXT*.

[48] W. Liu, W. Qu, J. Gong, and K. Li. 2016. Detection of Superpoints Using a Vector Bloom Filter. *IEEE Trans. on Information Forensics and Security* 11 (2016), 514–527.

[49] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. 2016. MOZART: Temporal Coordination of Measurement. In *Proc. of ACM SOSR*.

[50] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*.

[51] Ziyuan Liu, Zhixiong Niu, Ran Shu, Wenxue Cheng, Peng Cheng, Yongqiang Xiong, Lihua Yuan, Jacob Nelson, and Dan RK Ports. 2022. A Disaggregate Data Collecting Approach for Loss-Tolerant Applications. In *Proc. of ACM APNet*.

[52] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*.

[53] Jelani Nelson and David P Woodruff. 2010. Fast manhattan sketches in data streams. In *Proc. of ACM PODS*.

[54] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-based Querying of Distributed Sliding-Window Data Streams. *VLDB Endowment* 5, 10 (2012).

[55] PktGen. [n. d.]. https://pktgen-dpdk.readthedocs.io. ([n. d.]).

[56] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM*.

[57] Intel Architecture Instruction Set Extensions Programming Reference. [n. d.]. https://www.intel.com/content/dam/develop/external/us/en/documents/319433-024-697869.pdf. ([n. d.]).

[58] Intel SSE4 Programming Reference. [n. d.]. https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-705230.pdf. ([n. d.]).

[59] Nicoló Rivetti, Yann Busnel, and Achour Mostéfaoui. 2015. Efficiently Summarizing Data Streams over Sliding Windows. In *Proc. of IEEE NCA*.

[60] InfiniBand Architecture Specification Release 1.2.1 Annex A17: RoCEv2. [n. d.]. https://cw.infinibandta.org/document/dl/7781. ([n. d.]).

[61] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[62] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*.
[63] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information rich flow record generation on commodity switches. In *Proc. of ACM EuroSys*.
[64] Rajath Subramanyam, Indranil Gupta, Luke M. Leslie, and Wenting Wang. 2015. Idempotent Distributed Counters Using a Forgetful Bloom Filter. In *Proc. of IEEE ICCAC*.
[65] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*.
[66] Lu Tang, Qun Huang, and Patrick PC Lee. 2019. MV-Sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *Proc. of IEEE INFOCOM*.
[67] Lu Tang, Qun Huang, and Patrick PC Lee. 2020. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Transactions on Networking* 28, 5 (2020), 2350–2363.
[68] Lu Tang, Qun Huang, and Patrick P. C. Lee. 2020. SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders. In *Proc. of IEEE INFOCOM*.
[69] Tofino. [n. d.]. https://www.barefootnetworks.com/products/brief-tofino/. ([n. d.]).
[70] Steve T. Watt, Shankar Achanta, Hamza Abubakari, Eric Sagen, Zafer Korkmaz, and Husam Ahmed. 2015. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)*.
[71] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. 1990. A linear-time probabilistic counting algorithm for database applications. *Proc. of ACM TODS* 15 (1990), 208–229.
[72] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*.
[73] MyungKeun Yoon. 2010. Aging Bloom Filter with Two Active Buffers for Dynamic Sets. *IEEE TKDE* 22, 1 (2010).
[74] Minlan Yu. 2019. Network Telemetry: Towards a Top-down Approach. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019).
[75] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*.
[76] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proc. of ACM SIGCOMM*.
[77] Ying Zhang. 2013. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *Proc. of ACM CoNEXT*.
[78] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets.. In *Proc. of USENIX NSDI*.
[79] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: Enabling on-the-Fly Task Reconfiguration for Network Measurement. In *Proc. of ACM SIGCOMM*.
[80] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2020. Newton: Intent-Driven Network Traffic Monitoring. In *Proc. of CoNEXT*.
[81] Yang Zhou, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. 2022. Evolvable Network Telemetry at Facebook. In *Proc. of USENIX NSDI*.
[82] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. of ACM SIGCOMM*.