

Packets are Not Pages: Flow-Based Addressing Conserves Memory Bandwidth

AGUR ADAMS, Stanford University, USA

HUI SUB SHIM, Stanford University, USA

COLIN DREWES, Stanford University, USA

BEN O'KEEFE, Stanford University, USA

PHIL LEVIS, Stanford University, USA

ZAKIR DURUMERIC, Stanford University, USA

SmartNICs promise hardware offloading for network applications like traffic analysis and virtual host dispatching. However, due to memory bandwidth limitations, SmartNICs are unable to effectively accelerate applications like intrusion detection and deep packet inspection that require high-speed reassembly. We argue that this limitation is due to an architectural mismatch: variable-sized packets arrive out of order, but a SmartNIC's memory is a contiguous address space broken up into fixed-size pages. This mismatch forces multiple unnecessary copies to translate between the two layouts.

We propose a new memory addressing scheme, flow-based addressing, which enables software to access the reassembled payloads of a flow's packets as a contiguous region in memory. Flow-based addressing allows software to easily operate on reassembled payloads without copies, dramatically reducing memory traffic. Eliminating copies entirely requires hardware that maps logical flow offsets to non-contiguous physical addresses and assembles misaligned payloads into contiguous cache lines at line rate. The new memory addressing scheme introduces three architectural mechanisms: a flow translation table, a private flow cache, and a specialized DMA engine that assembles unaligned, variable-length packets into contiguous cache lines. We describe a prototype design on an FPGA-based SmartNIC and show that, across synthetic microbenchmarks, university campus network traffic, and data center web-search traffic, flow-based addressing reduces DRAM bandwidth use by 77–83%, lowers execution time by 25–62%, and increases per-core throughput by 1.33–2.66×.

CCS Concepts: • **Hardware** → **Networking hardware; Reconfigurable logic and FPGAs**; • **Networks** → **Network architectures**.

Additional Key Words and Phrases: SmartNIC; Flow-based Addressing; Memory Bandwidth; Packet Processing

ACM Reference Format:

Agur Adams, Hui Sub Shim, Colin Drewes, Ben O'Keefe, Phil Levis, and Zakir Durumeric. 2026. Packets are Not Pages: Flow-Based Addressing Conserves Memory Bandwidth. *Proc. ACM Netw.* 4, CoNEXT3, Article 37 (September 2026), 24 pages. <https://doi.org/10.1145/3830392>

Authors' Contact Information: Agur Adams, adamsas@stanford.edu, Stanford University, Stanford, CA, USA; Hui Sub Shim, hsshim@stanford.edu, Stanford University, Stanford, CA, USA; Colin Drewes, drewes@cs.stanford.edu, Stanford University, Stanford, CA, USA; Ben O'Keefe, bokeefe@stanford.edu, Stanford University, Stanford, CA, USA; Phil Levis, pal@cs.stanford.edu, Stanford University, Stanford, CA, USA; Zakir Durumeric, zakir@cs.stanford.edu, Stanford University, Stanford, CA, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2834-5509/2026/9-ART37

<https://doi.org/10.1145/3830392>

1 Introduction

Network stacks consume as much as 60% of host CPU cycles in high-traffic environments [29]. As a result, cloud service providers have increasingly offloaded server network functions to SmartNICs to save energy and improve throughput [10, 18, 26, 41]. Modern SmartNICs include computational accelerators to encrypt and compress data, run regular expression patterns against packets, and perform RDMA transactions [31, 41]. They also have general-purpose cores, which can run arbitrary application code. Offloading computations to these cores can have many system benefits, including reducing the data sent over PCIe [24, 48] and freeing up host CPU processing [21, 27].

On modern SmartNICs, however, computational offloads can quickly run into a bottleneck: memory bandwidth. While link speeds continue to increase, memory bandwidth, in comparison, remains stagnant. For example, a modern BlueField-3 SmartNIC supports 400 Gbps link speeds but only 717 Gbps of aggregate DRAM bandwidth. This 1.8:1 DRAM-to-link bandwidth ratio is less than what is needed to write every packet to DRAM and later read it at network line rate.

This tight DRAM-to-link bandwidth ratio limits what applications can run on a SmartNIC. Packet-level processing, such as in virtualization stacks [12, 17], examines only headers, which constitute a small fraction of traffic. However, applications that process reassembled transport-layer data (e.g., HTTP traffic, RPC requests) quickly run into a memory bandwidth limitation. Stream reassembly requires writing packets to DRAM as they arrive and then later copying their payloads (a read, then a write) into a contiguous region of memory to process (another read). In short, SmartNIC memory is too slow to support line-rate reassembly, hindering hardware acceleration for an important class of applications.

We argue that the memory bottleneck on SmartNICs is due, in part, to a mismatch of abstractions. Application code spends almost all of its cycles and memory bandwidth accessing packets, but memory is organized in pages. Pages have a fixed size, and their structure depends on high locality through a hierarchy of page tables. Packets have low locality, as a link interleaves packets from many flows. For example, prior work has shown how packet buffers scattered through a virtual address space can lead to large numbers of packet drops due to IOTLB misses [38]. Packets also have variable length, as they encapsulate application payloads. This mismatch forces applications to copy payloads into a contiguous buffer before processing: packet payloads occupy scattered, unaligned locations in DRAM, so the CPU must load a full cache line from each fragment regardless of whether it copies data explicitly or accesses it through pointers, exhausting memory bandwidth [39].

This paper proposes a new flow-based addressing scheme and flow cache controller for SmartNICs to use limited memory bandwidth more efficiently and thus enable them to support a broader set of applications. Rather than address packets as disjoint segments within an address space, flow-based addressing presents reassembled segment payloads as a contiguous memory region to support efficient on-NIC computations. Flow-based addressing introduces a new per-core memory address translation table, called the flow translation table. When the SmartNIC CPU wants to access a network stream, it loads a core's flow translation table with the physical addresses of payloads and their lengths. The transport-layer bytestream appears to that core as a contiguous, read-only block of memory, which the core can read directly into its private flow cache. Flow-based addressing shares goals with scatter-gather [20], which allows software to specify disjoint memory regions that hardware merges into a single contiguous buffer in DRAM. However, flow-based addressing differs from scatter-gather in three key ways: (1) the flow cache incorporates bytestream reassembly into the cache hierarchy, (2) the SmartNIC CPU does not have to prepare DMA descriptors, and (3) the flow cache is directly integrated with the instruction pipeline. Scatter-gather provides hardware acceleration for copying disjoint regions of DRAM; flow-based addressing avoids the copies entirely.

Efficiently implementing flow-based addressing has three challenges. First, to process a flow, the SmartNIC CPU must be able to quickly identify the relevant packets and their locations, and populate the flow translation table. To support the SmartNIC CPU's configuration of the flow translation table, our design includes a custom hardware module for parsing, storing, and managing flow metadata as linked lists of packets. Second, an additional addressing scheme introduces memory coherence challenges. Our design makes the private flow cache read-only and non-coherent with page-based addresses; this constraint holds broadly across network workloads because applications generally consume received payload data without modifying it in place. Third, network payloads can have arbitrary offsets and lengths, but memory must properly interact with 64-byte cache lines. Our design leverages a custom data realignment hardware module to correctly map multiple blocks of memory into a single cache line.

We implement flow-based addressing on an AMD Alveo U250 FPGA-based SmartNIC. We show that flow-based addressing in networking applications reduces memory bandwidth use by 77–83%, lowers execution time by 25–62%, and improves per-core throughput by 1.33–2.62 $\times$, showing that flow-based addressing can resolve the imbalance between network and DRAM bandwidth. Flow-based addressing improves SmartNIC performance by removing memory inefficiencies rather than increasing compute throughput.

This paper makes three contributions:

- (1) Identifying memory bandwidth as a key bottleneck in SmartNICs;
- (2) Proposing a novel, flow-based addressing architecture that allows SmartNICs to process reassembled flows without copying;
- (3) Demonstrating flow-based addressing's feasibility while easily integrating with arbitrary software.

Section 2 quantifies the limitations of SmartNIC memory bandwidth when analyzing transport-layer bytestreams. Section 3 presents flow-based addressing's architectural support. Section 4 describes an implementation of flow-based addressing on an AMD Alveo U250 FPGA-based SmartNIC. We evaluate the performance of our prototype implementation in Section 5, compare against relevant work in Section 6, and conclude by discussing challenges and future directions in Section 7.

Our results suggest that the networking community should re-examine memory addressing in high-performance network devices.

2 Motivation

Many network applications analyze reassembled bytestreams [29, 33, 39, 46] requiring developers to copy transport-layer segments into a contiguous DRAM buffer for processing (Figure 1). Beyond the large number of DRAM reads and writes, prior work has shown that noncontiguous packet placement harms locality [39] and that the compensating copy is particularly expensive [8, 36, 42].

The cost of stream reassembly is high on SmartNICs compared to conventional servers because of their limited memory bandwidth relative to their NIC speed. For example, while a Google Cloud

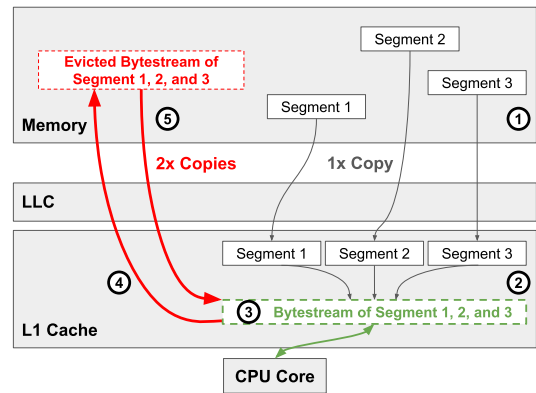


Fig. 1. To assemble an application bytestream, ① the SmartNIC CPU brings transport-layer segments from noncontiguous memory locations into cache and ② makes a copy to ③ scan the entire bytestream. ④ Cache evictions ⑤ move the assembled bytestream in and out of memory.

Platform	Cores	NIC Interface	DRAM BW per core	NIC BW per core	Ratio
Google Cloud N2d Server	224	100 Gbps	1.83 GB/s	0.06 GB/s	30.50
2x AMD Milan					
Google Cloud C3 Server	176	200 Gbps	3.49 GB/s	0.14 GB/s	24.93
2x Sapphire Rapids					
Intel IPU E2000 SmartNIC [44]	16	200 Gbps	6.40 GB/s	1.56 GB/s	4.10
NVIDIA BlueField-3 SmartNIC [31]	16	400 Gbps	5.60 GB/s	3.13 GB/s	1.79

Table 1. Aggregate DRAM and network bandwidths of multiple servers and SmartNICs as shown in [31, 32, 44]. As mentioned in [32], the bandwidths reflect theoretical limits and not measured results. For the IPU E2000 and BlueField-3 SmartNICs, the ratio of DRAM bandwidth to NIC interface bandwidth is 6.08 and 13.03 times less than the Google Cloud C3 server, respectively. Memory bandwidth constrains SmartNICs relative to general-purpose servers.

C3 server has a per-core aggregate DRAM bandwidth to NIC bandwidth ratio of 24.93, the ratio on SmartNICs is far lower: 4.10 on the Intel IPU E2000 and 1.79 on the BlueField-3 (Table 1). As network link speeds continue to scale faster than DRAM bandwidth, memory is increasingly the bottleneck for SmartNIC packet processing. Our proposed flow-based addressing removes this bottleneck by altogether eliminating unnecessary copies.

2.1 Execution Time and Throughput Analysis

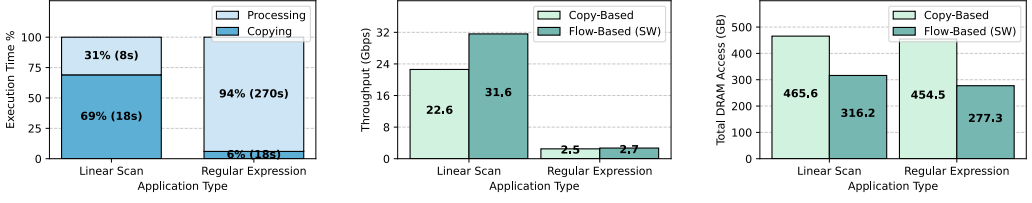
To quantify the cost of copying packets from noncontiguous memory locations in network applications, we measured the time spent copying packets versus processing a flow in two common bytestream analyses representative of intrusion detection [43, 45], network performance characterization [7, 46], and traffic classification [16, 47]. The linear scan searches for the NULL byte and the regular expression matches the pattern `[a-z][0-9]` (i.e., a lower-case letter followed by a digit). Both are representative of common patterns in the Snort [43] intrusion detection ruleset, as Table 2 shows. We defer a more nuanced examination of different computations to Section 5; we use these simple examples to illustrate the hardware bottlenecks in modern SmartNICs.

The experiment uses two NVIDIA BlueField-3 DPUs [31], each equipped with a 16-core Armv8 A78 processor and 32 GB of DDR5 memory; each side uses a single core to eliminate intercore contention. One DPU is the packet transmitter, running DPDK-based `pktgen` [49] to send 1,518-byte packets at line rate. The other DPU served is the receiver, running the target network application task. Figure 2a shows that copying overhead dominates execution time for the memory-bound workload. For linear scan of a single byte pattern, 69% of execution time is spent copying packets into a contiguous buffer, while only 31% is spent on processing. In contrast, for the compute-bound regular expression search using the pattern `[a-z][0-9]`, copying accounts for 6% of the execution time, with 94% spent on processing. These results highlight that when workloads are memory-bound, redundant copies directly reduce per-core throughput. Compute-bound workloads are less sensitive to memory bandwidth but still pay a penalty.

To quantify the potential benefits of eliminating copying and to connect these measurements to our proposed hardware approach, we compared per-core throughput between the baseline (copy-based) implementation and a software-emulated, flow-based addressing primitive (flow-based)

Byte pattern	Occurrences (% of rules)	Symbol / description
00	4,167 (43.63%)	0x00 (null)
22	1,620 (16.96%)	" (double quote)
3B	714 (7.48%)	; (semicolon)
5C	707 (7.40%)	\ (backslash)
3A	367 (3.84%)	: (colon)

Table 2. Top five most common patterns from the public Snort Pulled Pork ruleset. The single null byte is the most common pattern among the 9,549 rules in the ruleset; however, byte patterns can consist of multiple bytes (e.g., `filename=`).



(a) **Execution Time.** Copying accounts for 69% of total execution time in linear scan, showing that removing copying overhead can provide proportional execution time savings.

(b) **Per-Core Throughput.** Eliminating copying increases per-core throughput by 40% for linear scan and 11% for regex search, showing that reducing DRAM accesses boosts processing rates.

(c) **DRAM Access.** Eliminating copying cuts DRAM accesses by 32.1% for linear scan and 39.1% for regex search, benefiting both types of reassembly workloads.

Fig. 2. Execution time, per-core throughput, and DRAM accesses for copy-based and flow-based implementations. Removing copying benefits both workloads, with linear scan seeing the largest throughput gain and both applications showing reduced DRAM accesses.

that accesses payloads in place (Figure 2b) by iterating across a list of packets in software. The linear scan workload sees a per-core throughput increase of 40%, from 22.6 Gbps to 31.6 Gbps while regular expression per-core throughput increases by 11%, from 2.5 Gbps to 2.7 Gbps. These results align with the execution time breakdown, demonstrating that removing copying benefits both memory-bound and compute-bound applications.

2.2 DRAM Bandwidth Analysis

To evaluate the impact of copying overhead on memory traffic, we measure DRAM access counts in both applications and compare them to the software-emulated flow-based addressing implementation (Figure 2c). Since the BlueField-3 architecture does not expose LLC writeback statistics, we collected DRAM read counts using LLC load misses, and approximated DRAM write counts using L2 data cache writeback counts.

For the linear scan workload, removing copies decreases DRAM accesses by 32.1%, from 465.6 GB to 316.2 GB. For regular expression search, DRAM accesses reduced by 39.1%, from 454.5 GB to 277.3 GB. Packet copying generates substantial redundant memory traffic, even in compute-bound applications, constraining the computation possible on a SmartNIC. While our experiments use software-emulation of flow-based addressing, implementing this capability in hardware would enable further gains by incorporating features such as prefetching and flow-aware scheduling.

3 Design

Flow-based addressing provides a contiguous, byte-addressable representation of a flow's bytestream to a SmartNIC's CPU, even when segments are noncontiguous in DRAM.

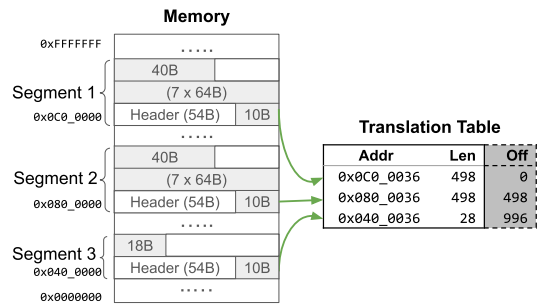


Fig. 3. For each transport-layer segment in a flow, the translation table stores the physical address of the segment, its length, and an offset that begins at zero and increments across segments in the flow. Together these fields let the SmartNIC CPU map any byte position in the flow to its location in memory. Such mapping allows the bytestream to be processed as a contiguous sequence.

As Figure 3 shows, flow-based addressing centers around a per-core flow translation table that maps logical flow offsets to physical addresses. The table allows the SmartNIC CPU to fetch any byte range with a single lookup and the flow cache controller to place the bytestream directly into a stand-alone, directly addressable, per-core flow cache. This eliminates redundant DRAM transfers and the software reassembly overhead of conventional approaches.

Flow-based memory addressing minimizes changes to the existing descriptor ring by reusing packet placement paths, preserving compatibility with existing NIC drivers and software stacks and reducing the engineering cost of deployment. This approach contrasts with prior work such as Ensō [39], which introduces a new CPU-NIC interface, or Pigasus [51, 52], which implements a fixed-function parallel packet processing pipeline on the SmartNIC. In this section, we describe the flow-based addressing mechanism and explain how it differs from conventional scatter-gather.

3.1 Contiguous Access via a Flow Translation Table

Figure 4 depicts the execution of flow-based addressing from packet arrival to flow assembly. When packets arrive from the network, the NIC's ASIC computes a unique identifier based on the packet's unidirectional five-tuple ①, which allows the SmartNIC to append the packet's metadata to the corresponding flow-linked list ②. If a matching list does not exist, the SmartNIC creates a new one. Each list entry contains the physical memory address and payload length, extracted from the packet header during ingress, while the payload itself is stored in DRAM based on the NIC's descriptor receive ring. We call this stage **flow storage**, and it continues until the SmartNIC receives either the last packet of a flow or a timeout marks the flow as complete.

To begin **flow assembly**, the SmartNIC CPU copies metadata from the linked list into the flow translation table ③ and notifies the flow cache controller to assemble the flow ④. As shown in Figure 3, each entry in the flow translation table stores the segment's physical address, length, and a computed offset—the cumulative length of all preceding segments starting from zero. This offset provides a logical byte address within the contiguous bytestream, enabling direct, byte-granular access without the need for the SmartNIC CPU to reconstruct the physical addresses in software every time it wants to read some portion of the bytestream. Once the SmartNIC CPU populates the table and notifies the flow cache controller ④, the flow cache controller issues memory read requests to DRAM to retrieve the required segments to reassemble the bytestream. The flow cache controller loads the retrieved data into the private flow cache in contiguous byte-aligned order and notifies the SmartNIC CPU that the data is ready ⑤. This enables the SmartNIC CPU to have direct access to any portion of the bytestream from the flow cache without additional DRAM transactions ⑥. Since the flow cache is private to its dedicated core, the SmartNIC CPU evicts flow data by overwriting the cache at software-determined points, with no inter-core contention or coherence coordination required.

Implementing flow storage and reassembly requires addressing three questions: (1) how to size the flow cache and translation table to balance on-chip memory consumption against flow size, (2) how to handle out-of-order TCP segments, and (3) how to manage flow retrieval and the associated per-flow packet metadata.

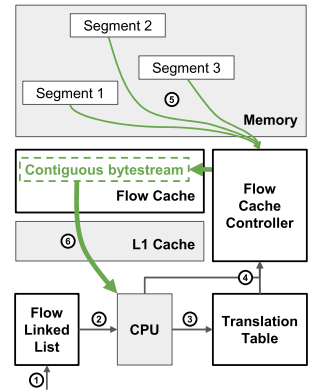


Fig. 4. Flow-based addressing manages per-flow linked lists and uses a translation table to assemble a contiguous bytestream directly in a last-level flow cache for immediate CPU processing.

3.1.1 Cache and Table Sizing. The flow cache and flow translation table require careful sizing to balance on-chip memory consumption against flow size: larger tables reduce the number of iterations required for large flows but consume more silicon area. Ideally, the flow cache should hold the entire active flow bytestream, ensuring all segments remain resident until processing completes, and the translation table should contain enough entries to map every segment in the flow so that the flow cache controller assembles the full bytestream in a single pass without iterating multiple times. In both cases, larger allocations consume proportionally more on-chip silicon area. Yet empirical network measurement studies have shown that TCP flow sizes can vary from a few bytes to tens of megabytes or more [46], making it challenging to determine an optimal flow cache size. To resolve this design question, we considered the cumulative distribution of flow sizes in real-world TCP traffic from a large research university campus network. We measured the total bytes per TCP flow of over 1.6 billion flows in a four-hour period. As shown in Figure 5, while we observed a long tail of large elephant flows, 93.16% of flows are 10 KB or less. Therefore, assuming a TCP maximum segment size of 1460 bytes, we size the flow cache and flow translation table to hold enough entries for a flow of 10 KB as 93% of TCP flows meet this criterion.

Software handles longer flows by loading multiple table-sized windows. For large flows, the SmartNIC CPU can process the bytestream in fixed-size windows that match the translation table capacity. That is, if the table holds W segments, the CPU first populates entries $[0, W - 1]$, requests assembly for the corresponding byte range, and starts processing as soon as the flow cache controller signals completion. While processing that window, the CPU advances a logical window pointer and refills the table with the next group of segments, repeating until it reaches the end of the flow. To preserve correctness for workloads with boundary-sensitive patterns (e.g., regular expression matching), each new window can include a small overlap region from the end of the previous window, then suppress duplicate matches in the overlap during result reporting. This overlap policy avoids missed matches that span window boundaries while keeping per-window metadata small and limiting how much flow-cache space each window uses. As a result, the design supports arbitrarily long flows without requiring a proportionally larger flow cache or translation table.

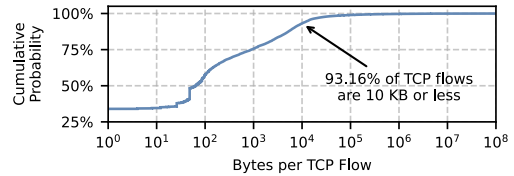


Fig. 5. CDF of total bytes per TCP flow derived from over 1.6 billion TCP flows in a university campus network measured over a four-hour period. Given 93.16% of TCP flows are 10 KB or less, we size the flow cache to 10 KB to handle the common case in a single window. Software manages larger flows by processing the bytestream in successive table-sized windows, advancing a logical window pointer and refilling the translation table until the flow is complete.

3.1.2 Out-of-Order Segments. Past network measurement studies have shown that out-of-order TCP segments occur infrequently (0.3–3%) [13, 46, 52], so we designed our flow translation table to handle in-order segments by default. However, if out-of-order segments do occur, the SmartNIC CPU can reorder them in the linked list and update the flow translation table accordingly. This allows us to maintain a simple and efficient flow assembly mechanism while still handling occasional out-of-order segments.

3.1.3 Flow Retrieval and Packet Metadata. The design supports configurable retrieval strategies: for selective processing, the SmartNIC CPU can retrieve specific offsets of the bytestream. However, for workloads like regular expression search, the SmartNIC CPU can fetch the entire bytestream once and reuse it from the flow cache. In the base case of our design, the SmartNIC CPU performs both flow storage and assembly, but alternative implementations could offload parsing packet headers,

filling and maintaining flow linked lists, and populating the flow translation table to dedicated hardware modules. Such additional hardware support would reduce the SmartNIC CPU’s overhead while enabling optional higher-level optimizations, including prefetching by flow offsets to hide DRAM latency and maintain the flow cache supply, or flow-type filtering to prioritize high-value traffic over low-value or encrypted flows.

Compared to conventional page-based memory systems, the drawback of this design is the management of flow linked lists and their associated metadata. Nevertheless, prior work has shown that managing flow linked lists at line rates of 100 Gbps or more on FPGA-based SmartNICs is achievable in hardware using a combination of insertion and out-of-order engines with shared flow state managed through deamortized cuckoo hashing [35, 51, 52]. We omit optimizing this aspect of the design since it is a well-studied problem. Instead, our work focuses on memory bandwidth utilization. The flow translation table and flow cache are the key innovations of our design, enabling efficient, contiguous access to flow bytestreams without the need for software reassembly or redundant memory transfers.

3.2 Differences from Conventional Scatter-Gather

Scatter-gather [20] copies data from noncontiguous memory locations into a contiguous region in memory. Scatter-gather differs from flow-based addressing in three fundamental ways: copying into a DRAM buffer versus assembling into the flow cache, CPU-driven versus hardware-managed reassembly, and waiting for full assembly before processing versus on-the-fly, pipelined access.

First, flow-based addressing assembles data directly into a private flow cache. This avoids the multiple memory accesses inherent in scatter-gather as shown in Figure 6. On the BlueField-3, these memory accesses exceed the DRAM-network bandwidth ratio, making DRAM the bottleneck and preventing sustained line rate processing. Flow-based addressing stays within this budget by performing a single DRAM read into the flow cache, enabling line-rate processing.

Second, scatter-gather requires the CPU to track physical addresses and prepare DMA descriptors before the DMA engine copies data. With flow-based addressing, hardware maintains the mapping from logical bytestream offsets to physical addresses, so the CPU is abstracted from the physical layout of packets and operates only on a contiguous flow view. This shifts processing from per-packet memory management to a flow-based model.

Third, scatter-gather delays processing until the full bytestream is assembled. Flow-based addressing makes segments available in the flow cache as they arrive, so the CPU can begin immediately and overlap computation with hardware assembly. This pipelining reduces end-to-end latency and improves parallelism.

These differences make flow-based addressing fundamentally distinct from conventional scatter-gather. By eliminating redundant DRAM transfers, abstracting the CPU from the physical layout of packets, and enabling immediate processing of partial bytestreams, flow-based addressing reduces bandwidth pressure and latency while presenting software with a contiguous flow view.

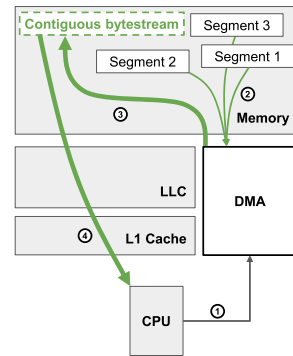


Fig. 6. Scatter-gather uses a DMA engine to copy segments into a contiguous DRAM buffer, requiring the CPU to read them back and adding an extra write-read round trip to DRAM. Flow-based addressing eliminates this redundant traffic by assembling data directly into the private flow cache, reducing latency and improving per-core throughput.

4 Implementation

The previous section described the design concepts and benefits of flow-based addressing. This section details a prototype implementation on the AMD Alveo U250 FPGA-based SmartNIC. Our implementation consists of three main components: (1) a packet handler drives the flow storage process to direct packet placement in DRAM and extract the necessary packet metadata to assemble linked lists; (2) a single-core 64-bit AMD MicroBlaze soft CPU in the FPGA fabric that manages linked-list and flow translation table setup during the flow translation table population process; and (3) a flow cache controller that performs the flow assembly stage (Figure 7).

4.1 Flow Storage

Flow storage is the first stage of the flow-based addressing pipeline (Figure 7). The packet handler receives packets from the NIC, writes each payload to DRAM, and emits a compact metadata record to the MicroBlaze soft CPU. From the L2/L3/L4 headers, the packet handler extracts the five-tuple to compute a flow identifier, captures TCP sequence numbers to maintain in-flow order, and records the payload's DRAM base address and length for later access by the flow cache controller. By extracting only essential metadata, the packet handler emits records at line rate and maintains a clear separation between packet ingress and subsequent stages.

The MicroBlaze soft CPU consumes the metadata stream to maintain per-flow linked lists. For each record, software on the CPU appends the segment to the list with the matching flow identifier, or allocates a new list for a new unidirectional flow. Using software to maintain the per-flow linked lists is a design choice that provides configurability in which metadata fields to extract from packet headers and preserve in the lists. However, list management is inherently repetitive, it could also be offloaded to a custom hardware module to reduce CPU overhead.

The CPU software organizes per-flow linked lists using an open-addressed hash table for per-flow bookkeeping and a shared node pool for segment storage. The hash table maps each unidirectional flow identifier to the head node, tail node, and segment count of that flow's list, enabling $O(1)$ per-flow lookup without scanning all active flows. The hash table resides in on-chip BRAM¹ to minimize lookup latency, while the node pool holds per-segment physical addresses, lengths, and linkage fields in DRAM because its capacity exceeds available on-chip memory (Figure 7). Free pool slots form an intrusive free list through the next-segment field of each node, giving the CPU $O(1)$ allocation and deallocation without heap fragmentation. The hash table resides in a separate BRAM block from the flow cache, so there is no contention when software on the MicroBlaze CPU accesses either storage location.

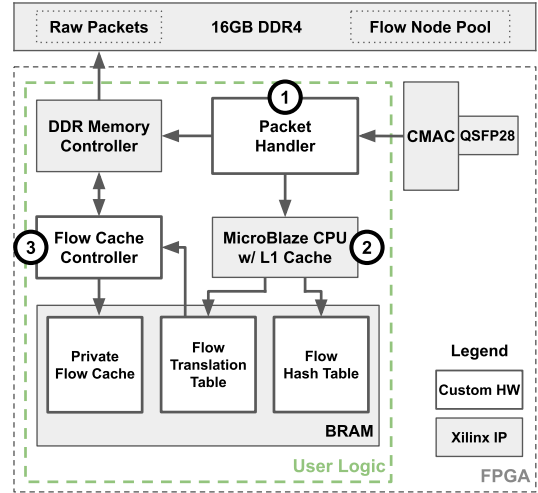


Fig. 7. Flow-Based Addressing System Overview.

The design consists of a custom ① packet handler hardware module, ② Xilinx MicroBlaze CPU with L1 cache, a ③ custom hardware flow cache controller, and a DDR memory controller with access to 16 GB of off-chip DDR4 DRAM.

¹BRAM refers to on-chip SRAM-like memory blocks integrated into FPGAs, offering low-latency storage well-suited for caching and buffering compared to external DRAM.

4.2 Flow Translation Table Population

Figure 7 illustrates how software on the MicroBlaze CPU populates the flow translation table upon final-packet receipt or timeout and, for flows exceeding the table capacity, iteratively advances a logical window pointer to refill the table with the next segment group until the flow cache controller assembles the complete bytestream. Software on the CPU traverses the per-flow linked list, copies each packet's base address and length as provided by the packet handler, and computes the logical offset by accumulating lengths from the first segment. Software on the SmartNIC CPU writes these values into the translation table, with each row corresponding to a single packet in the flow. At any point in time, the flow translation table holds translation entries for a single unidirectional flow, even though the MicroBlaze may simultaneously maintain linked lists for multiple flows.

4.3 Flow Assembly

Once software populates the flow translation table, software on the MicroBlaze CPU issues a `<start_offset, length>` request to the flow cache controller, as shown in Figure 8. The flow cache controller provides flow-based access to DRAM while abstracting physical addresses from the CPU. It consults the translation table to determine the exact addresses and lengths of the segments to be retrieved. Using the translation table, the flow cache controller retrieves the requested bytestream from DRAM, assembles the segments into a contiguous format, and places the reassembled stream in the flow cache within BRAM. The flow cache appears as a contiguous region of directly addressable memory outside of main memory. After the flow cache controller loads the flow cache, the bytestream is ready for software processing.

To assemble flows, the flow cache controller uses two cooperating units: the stream preprocessor and the DRAM reader (Figure 8).

The **stream preprocessor** scans the translation table to locate entries whose bytestream offsets overlap with the requested `<start_offset, length>`. Each entry specifies a contiguous region of physical memory and its offset within the flow's logical bytestream. For each matching entry, the preprocessor computes the exact sub-range to extract, constructs a `StreamDescriptor` with the base physical address, valid byte count, and an end-of-stream flag, and emits these descriptors to the DRAM reader. By performing all offset calculations and range selection up front, the stream preprocessor isolates table interpretation from memory access, allowing the DRAM reader to focus on executing DRAM fetches and alignment.

The **DRAM reader** consumes the descriptors and performs the actual memory reads. On the transmit side, it issues DRAM read requests through AMD's data mover IP core, supplying the physical address and length from each descriptor. As data returns in 64-byte reads, the receive side of the DRAM reader processes both payloads and AXI stream control signals. The AXI control signal `TKEEP` mask identifies valid bytes within each read, which the DRAM reader extracts and appends contiguously to the flow cache in BRAM. The AXI control signal `TLAST` marks the final

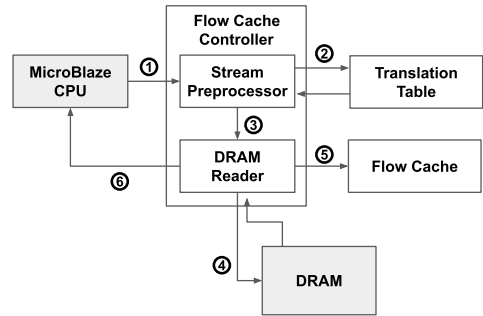


Fig. 8. Flow Cache Controller Architecture. ① The MicroBlaze issues a read request specifying the flow offset and length. ② The Stream Preprocessor queries the translation table to find the physical addresses of payload data. ③ The Stream Preprocessor sends the physical address and length to the DRAM Reader. ④ The DRAM Reader issues read requests to DRAM. ⑤ The DRAM Reader processes the returned data, extracting valid bytes and storing them in the flow cache. ⑥ Once assembled, the flow cache controller signals completion to the CPU.

read of the final descriptor, at which point the DRAM reader signals completion to the MicroBlaze CPU.

The DRAM reader produces a contiguous payload buffer in the flow cache that spans all relevant translation table entries. By using descriptors that precisely define valid ranges and addresses, the DRAM reader fetches only the necessary bytes and merges them directly into a single buffer. The result presents the requested bytestream to the host CPU in one contiguous flow cache, while avoiding unnecessary DRAM overhead.

The flow cache controller supports both on-demand and speculative access patterns. Software on the CPU can fetch only a small portion of the bytestream to decide whether deeper inspection is required, reducing unnecessary DRAM reads. For workloads such as regular expression matching, where repeated access is expected, software on the CPU may prefetch the entire region into the flow cache. In both cases, the flow cache provides low-latency access without incurring additional DRAM traffic.

4.4 Dealing with Unaligned Accesses

A key advantage of flow-based addressing is its ability to assemble transport-layer segments stored at noncontiguous memory locations into one contiguous buffer in the flow cache. This enables the host CPU to process flows directly from its private flow cache without extra DRAM traffic from data copying. However, this benefit introduces the challenge of unaligned accesses. Payloads arriving from the network rarely align with 64-byte AXI read/write access boundaries. Two factors define the problem: the starting offset of the payload and its length, neither of which is guaranteed to match the AXI read size to DRAM. Together these yield four possible alignment cases that require managing arbitrary byte offsets at full throughput. To address this we designed the Flow Cache Data Aligner, which we explain further in Appendix A but omit here for brevity.

4.5 FPGA Resource Utilization

We synthesized and implemented the complete flow-based addressing system on the AMD Alveo U250 FPGA with Vivado 2023.2 [4]. Table 3 summarizes the resource utilization of the flow cache, flow cache controller, and translation table. We implemented the packet handler and flow cache controller as custom hardware modules, while the MicroBlaze CPU manages linked lists and the flow translation table in software. The overall design consumes a modest fraction of the FPGA's resources, leaving ample capacity for additional logic. However, compared to the MicroBlaze CPU and L1 cache, the flow cache and controller appear relatively large, but the MicroBlaze CPU is a soft-core processor highly optimized for use on FPGAs for embedded applications where the designer can omit expensive hardware features [50]. For simplicity, we use a bare-bones MicroBlaze configuration without a floating point unit, barrel shifter, or other advanced features to minimize resource usage. In a production system, designers can use a more powerful CPU, which would reduce the relative resource consumption of the flow cache and controller.

Component	Utilization	FPGA (%)	CPU (%)
MicroBlaze CPU and L1 Cache			
LUTs	24,935	1.44%	100%
Registers	45,699	1.32%	100%
Block RAM Tiles	44	1.64%	100%
Flow Cache and Controller			
LUTs	31,101	1.80%	125%
Registers	43,569	1.26%	95%
Block RAM Tiles	113	4.20%	257%

Table 3. FPGA Resource Utilization: While the design uses a modest fraction of all available FPGA resources, the flow cache and controller consumes 95–257% of the same resources as the MicroBlaze CPU and L1 Cache.

5 Evaluation

We evaluate flow-based memory addressing on an AMD Alveo U250 FPGA-based SmartNIC running at 300 MHz across three experimental setups: microbenchmarks with synthetic traffic, a large university campus network trace, and a datacenter web traffic trace. The campus trace reconstructs Ethernet frames from anonymized per-packet TCP metadata captured on a 100 Gbps optical tap, replacing payload bytes with zeros to preserve campus user privacy, while the datacenter trace generates packets from a flow-size distribution [1, 2]. The campus trace was collected in coordination with our university security, privacy, and networking teams; our IRB has indicated that analyzing fully-anonymized traces that contain no user data to understand network dynamics does not constitute human subjects research.

Each experiment answers three questions:

- How much end-to-end time does the flow-based approach save, and specifically how much does copying time decrease?
- What is the net effect on per-core throughput after accounting for the flow translation table setup cost?
- How does flow-based addressing affect DRAM read and write bandwidth?

Figure 9 shows the experimental system setup. Two servers connect over a 100 Gbps QSFP28 link. The traffic generator transmits packets with pktgen [49] using an AMD Alveo U250 FPGA configured as a standard NIC in AMD's OpenNIC framework [3]. The analysis server receives packets on an AMD Alveo U250 FPGA running the flow-based addressing design with a MicroBlaze soft CPU at 300 MHz as the compute core. Firmware on the FPGA uses a timer to measure execution time per stage.

An AXI hardware performance monitor counts DRAM read and write bytes and transactions.

As in Section 2, we compare the flow-based pipeline against a copy-based baseline. The flow-based pipeline has three stages: flow translation table population, a flow cache controller stage that fetches all payloads into the private flow cache, and a linear scan or regular expression processing. The copy-based baseline has two stages: fetch all payloads into one contiguous DRAM buffer and process them. The experiments exclude packet processing time in both cases because it is identical across pipelines. We also do not include the time nor memory transactions associated with maintaining the flow linked lists as our prototype manages linked lists in software; prior work has shown that this maintenance can instead be offloaded to hardware entirely at line rate [52], and Appendix D provides an analytical estimate of the metadata footprint and DRAM bandwidth cost of the software-managed approach used in our prototype. Across all three experimental contexts, flow-based addressing reduces linear scan execution time by approximately 62%, raises per-core throughput by more than 2.6 $\times$, and reduces DRAM traffic by more than 80%, while regular expression search follows the same Amdahl-bounded pattern with approximately 25% reduction in execution time and 1.34 $\times$ improvement in per-core throughput.

5.1 Microbenchmarks with Synthetic Traffic

5.1.1 Execution Time. We transmit ten fixed-size 1,510-byte packets in a single flow to derive microbenchmarks. The copy-based design takes 10.2 ms to perform linear scan on these packets, while the flow-based design takes 3.88 ms, a 62% reduction (Figure 10a). Flow translation table

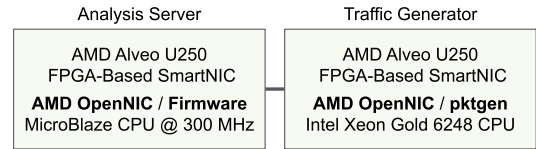
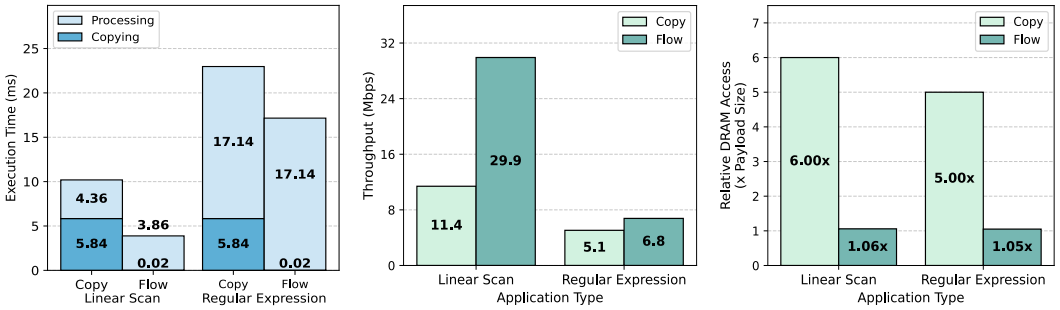


Fig. 9. Experimental setup. The traffic generator sends packets at line rate while the analysis server executes the copy-based or flow-based pipeline through firmware instrumentation, enabling direct attribution of latency and DRAM use to each pipeline stage.



(a) **Execution time breakdown.** Flow-based addressing reduces total execution time by 61.96% for linear scan and 25.33% for regular expression by eliminating the copy stage.

(b) **Per-Core Throughput.** Flow-based addressing improves per-core throughput by 2.62 $\times$ and 1.33 $\times$ for linear scan and regular expression, respectively.

(c) **DRAM Traffic Relative to Payload.** Flow-based addressing reduces DRAM traffic by 82% for linear scan and 79% for regular expression by sourcing data directly from the private flow cache.

Fig. 10. Synthetic Traffic for Microbenchmarks: (a) Execution time, (b) per-core throughput, and (c) DRAM traffic for copy-based and flow-based designs. Flow-based addressing reduces end to end time by 11.47–61.96%, improves per-core throughput 1.33–2.66 $\times$, and reduces DRAM traffic by 79–82%. The gains come from packing unaligned payload segments into the private flow cache and avoiding construction of a DRAM contiguous buffer.

(FTT) population appears only in the flow-based design and adds 12 μ s per table read (Table 4), the expected cost to gain a flow-centric view and processing capability. The largest benefit arises when preparing a contiguous buffer: 5.84 ms in the copy-based design falls to 10 μ s in the flow-based design (Table 4).

Two factors drive this. First, the destination memory differs in the copy-based case (a buffer in DRAM) versus the flow-based case (the flow cache in BRAM). Second, the copy-based design writes via memcpy on the CPU, whereas the flow-based design writes into BRAM through the flow cache controller at hardware speed. The destination memory and the engine that performs the work together account for the near elimination of this step in the flow-based design. Processing time also drops from 4.36 ms to 3.86 ms, an 11% decrease (Figure 10a), because the copy-based design reads from DRAM while the flow-based design reads directly from the private flow cache.

Flow-based addressing yields similar benefits in our regular expression search workload, reducing copying time. The copy-based design takes 22.98 ms while the flow-based design takes 17.16 ms, a 25% reduction (Figure 10a). The flow-based design still incurs the 12 μ s FTT setup cost but eliminates the 5.84 ms contiguous buffer construction step, which accounts for the majority of the time reduction. Consistent with Amdahl’s law [20], this result shows that the flow-based design’s improvement comes from eliminating the copy stage, not from accelerating processing.

5.1.2 Per-Core Throughput. Reducing time spent copying directly increases end-to-end per-core throughput for a fixed amount of work in linear scan. Per-core throughput rises by 162%, from

Stage	Copy-Based	Flow-Based	Reduction
FTT population	0.0 μ s	12 μ s	–12 μ s (–n/a–)
Build buffer	5,836 μ s	10 μ s	5,826 μ s (99.8%)
Linear scan	4,357 μ s	3,861 μ s	496 μ s (11.4%)
Total	10,193 μ s	3,883 μ s	6,310 μ s (61.9%)

Table 4. Per-Stage Execution Time. The flow-based scheme pays a small FTT setup cost of 12 μ s, nearly removes contiguous buffer build, and reduces linear scan time by sourcing data from the private flow cache instead of DRAM, dropping overall time by 61.9%.

11.4 Mbps to 29.9 Mbps (Figure 10b). Although FTT population costs 12 μ s, the combined savings from constructing the contiguous buffer and serving the working set from the private flow cache total 6.31 ms, roughly 525 times the setup cost. The improvement exceeds the trend in Figure 2b, where experiments predicted a 40% gain, because the flow cache controller builds the contiguous buffer at hardware speed and removes the software cost.

Regular expression search throughput follows the same trajectory, rising by 33% from 5.1 Mbps to 6.8 Mbps (Figure 10b). The impact of flow-based addressing is less pronounced for this workload because the processing stage dominates total execution time. The relative gain from eliminating the contiguous buffer construction step is therefore smaller than for linear scan. Although Figure 10a and Figure 10b show execution times on the order of milliseconds and throughput in megabits per second, these values reflect the MicroBlaze CPU's performance at 300 MHz and not the flow cache controller's capabilities. The flow cache controller runs at the same frequency as the MicroBlaze CPU but operates as a dedicated hardware engine that services requests in parallel with the CPU. We measure its throughput in isolation to quantify the flow cache controller's raw performance and its ability to scale.

We vary the number of 1,510 B packets in a single flow from one to forty to quantify peak assembly throughput. We measure the time from the MicroBlaze CPU's notification that the translation table has been populated until the flow cache controller reports completion of bytestream assembly. The flow cache controller saturates at approximately 76.48 Gbps after retrieving forty packets (Figure 11). Throughput and goodput in Figure 11 distinguish total data transferred from valid payload bytes; the difference arises because the data mover performs 64-byte-aligned transfers from DRAM and therefore retrieves invalid bytes at the boundaries of each 1,510 B packet.

Since the Arm Cortex A78 on the BlueField-3 operates at 2.133 GHz [10], approximately 7.11 times the frequency of the MicroBlaze CPU, the flow cache controller meets the demand of high-performance CPUs with ample margin. Assuming a 7.11 $\times$ frequency increase on an ASIC, the flow cache controller could potentially achieve over 543.8 Gbps for a single core, comfortably exceeding the 31.6 Gbps maximum per-core throughput of the BlueField-3 in Figure 2b.

5.1.3 DRAM Access. DRAM traffic falls in proportion to the removal of redundant DRAM accesses. As shown in Figure 10c, DRAM traffic drops from 6.0 $\times$ to 1.05 $\times$ the payload size, an 83% reduction for linear scan. For regular expression search, DRAM traffic falls from 5.0 $\times$ to 1.05 $\times$ the payload size, a 79% reduction. The copy-based baseline reads DRAM to form the contiguous buffer, writes that buffer back to DRAM, and then reads it again for processing. The lower DRAM traffic aligns with the latency and per-core throughput gains in Table 4 and is the main source of the reduction in total time and the ratio of copying time.

In summary, flow-based addressing replaces software-driven, DRAM-backed buffer construction with hardware-executed assembly into the flow cache. The processing stage then reads from BRAM rather than DRAM. Together, these changes reduce DRAM traffic by 83% for linear scan

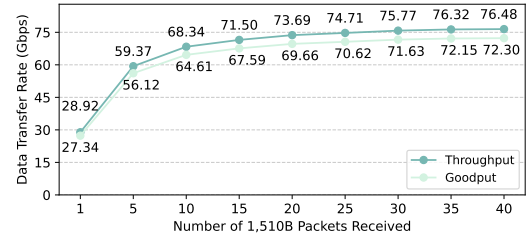
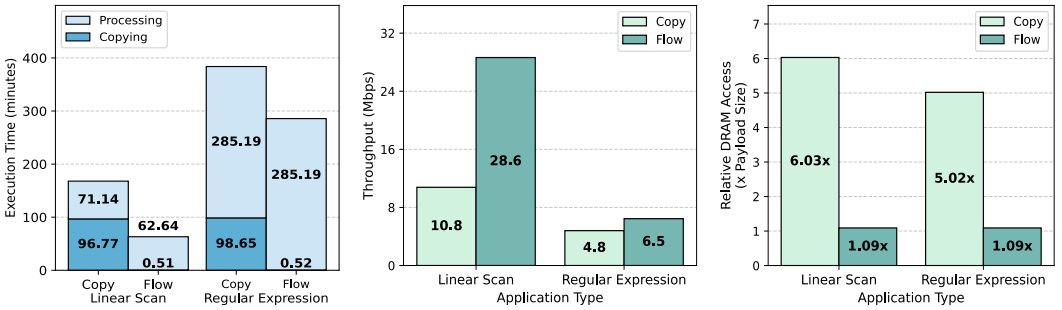


Fig. 11. Flow Cache Raw Throughput. The MicroBlaze CPU limits the overall execution time and throughput of the network application. In isolation, the flow cache controller can achieve approximately 76.48 Gbps at 300 MHz when receiving payload data from forty 1,510 B packets over DRAM (53,382 B total). Given the BlueField-3's Arm Cortex A78 at 2.133 GHz needs approximately 31.6 Gbps of throughput when running linear scan (Figure 2b), the flow cache controller's design amply satisfies a single core's needs.



(a) **Execution time breakdown.** Flow-based addressing reduces total execution time by 62% for linear scan and 26% for regular expression by eliminating the copy stage. (b) **Per-Core Throughput.** Flow-based addressing improves per-core throughput by 2.66 $\times$ and 1.34 $\times$ for linear scan and regular expression, respectively. (c) **DRAM Traffic Relative to Payload.** Flow-based addressing reduces DRAM traffic by 82% for linear scan and 78% for regular expression by sourcing data directly from the private flow cache.

Fig. 12. Campus Network Traffic: (a) Execution time, (b) per-core throughput, and (c) DRAM traffic for copy-based and flow-based designs. Flow-based addressing reduces end-to-end time by 26–62%, improves per-core throughput 1.34–2.66 $\times$, and reduces DRAM traffic by 78–82%, matching the synthetic results.

and 79% for regular expression search. Regular expression search gains less than linear scan because the processing stage, not copying, dominates its runtime. Amdahl’s law bounds throughput improvement to 1.33 $\times$ for regular expression search versus 2.62 $\times$ for linear scan. Across both workloads, the consistent reduction in DRAM traffic confirms that reorganizing data movement around flows removes the redundant copy-write-read cycle and shifts execution time from moving data to processing it.

5.2 Large Flows and Real-World Traffic

We evaluate two additional network traffic traces to test whether the gains observed in microbenchmarks extend beyond artificial single-flow conditions. The first trace uses anonymized TCP packet metadata collected from a large university campus network over a five-minute window, capturing 14.64 million packets across 100,211 flows dominated by bulk PSH+ACK data transfers. The second uses a synthetic datacenter web-search workload [1, 2] with 18,607 maximum-transmission-unit packets from 10,000 flows, spanning flow sizes from 2 KB to 100 MB. Appendix B and Appendix C provide more detailed summaries of the trace construction process and characterization for the campus and datacenter traces, respectively. The sending server replays each trace using DPDK-based pktgen [49], and the receiving server runs the flow-based design. The following subsections show the impact of these two traces on execution time, per-core throughput, and DRAM accesses for both the flow-based design and the copy-based baseline, using the same linear scan and regular expression search workloads as in Section 5.1’s microbenchmarks.

5.2.1 Large University Campus Network. As shown in Figure 12a, the copy-based design takes 167.91 min to process all flows with linear scan, while the flow-based design completes in 63.16 min, a 62% reduction. Copying time falls 99% (from 96.77 min to 0.51 min) and processing drops 12% (from 71.14 min to 62.64 min) by sourcing data from the flow cache. Per-core throughput rises 166%, from 10.77 Mbps to 28.64 Mbps (Figure 12b), matching the 162% synthetic gain. DRAM traffic drops from 6.03 $\times$ to 1.09 $\times$ the payload size, an 82% reduction for linear scan (Figure 12c). Regular expression

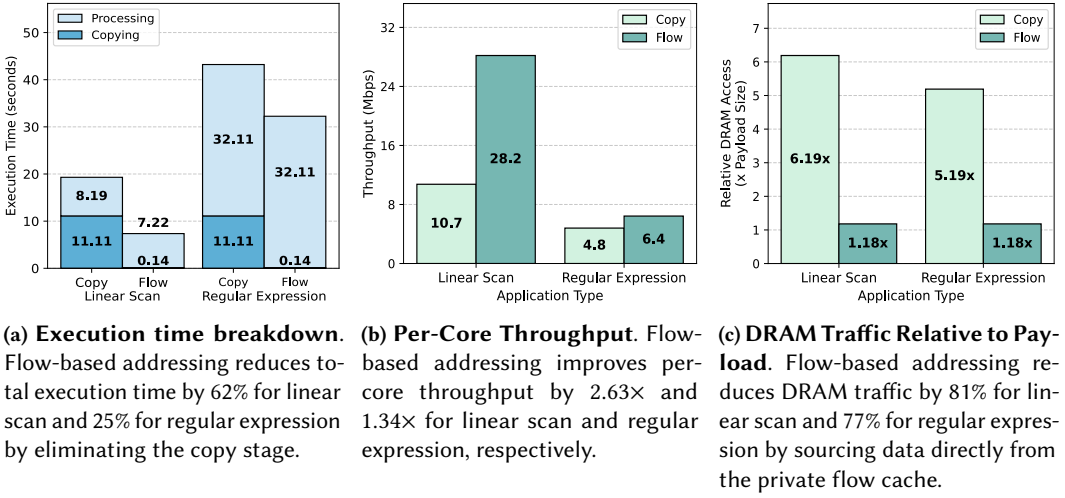


Fig. 13. Datacenter Web-Search Traffic: (a) Execution time, (b) per-core throughput, and (c) DRAM traffic for copy-based and flow-based designs. Flow-based addressing reduces end-to-end time by 25–62%, improves per-core throughput 1.34–2.63×, and reduces DRAM traffic by 77–81%, consistent with the synthetic and campus results.

search follows the same Amdahl-bounded pattern: execution time falls 26% (from 383.84 min to 285.71 min), per-core throughput rises 34%, and DRAM traffic falls 78%.

The campus results confirm that the benefit of flow-based addressing does not depend on the uniformity of a controlled single-flow experiment. The copy stage accounts for the same dominant fraction of execution time regardless of flow size distribution because the inefficiency is structural: software-driven buffer construction reads each payload segment from DRAM, writes it into a contiguous buffer, and reads it again for processing, independent of traffic composition. Flow-based addressing eliminates this redundant access sequence in every flow. Given the top one percent of flows carry ninety-six percent of all traffic volume in the campus trace, flow-based addressing eliminates the cost of copying precisely where the majority of bytes concentrate.

5.2.2 Datacenter Web Traffic. The copy-based design takes 19.30 s for linear scan and the flow-based design completes in 7.36 s, a 62% reduction (Figure 13a). Consistent with the synthetic and campus results, copying falls 99% (from 11.11 s to 0.14 s) and processing drops 12%. Per-core throughput rises 163% (Figure 13b). DRAM traffic drops from 6.19× to 1.18× the payload size, an 81% reduction for linear scan (Figure 13c). Regular expression search execution time falls 25% from 43.22 s to per-core throughput rises 34%, and DRAM traffic falls 77%.

The datacenter trace spans five orders of magnitude in flow size, from 2 KB query flows to 100 MB background transfers, presenting the two conditions under which flow-based addressing faces its most demanding constraints. Short flows require the flow translation table setup cost to remain negligible relative to the copy-stage savings it enables, while large flows require the flow cache controller to sustain consistent throughput across many payload segments within a single flow. The datacenter results match the campus and synthetic results across all three metrics, confirming that the design meets both constraints simultaneously and that its gains do not depend on any particular flow size distribution.

Across all three experimental contexts, Table 5 reveals a consistent three-way pattern: flow-based addressing removes redundant DRAM accesses, reduces copying time, and raises per-core throughput simultaneously. For linear scan, execution time falls approximately 62%, per-core

throughput rises approximately 2.6 $\times$, and DRAM traffic falls approximately 81% in every case. For regular expression search, Amdahl's law bounds the gains to approximately 25% in execution time and 1.34 $\times$ in throughput. These results confirm that reorganizing data movement around flows removes redundant DRAM accesses, shifts time from moving data to processing it, and scales to realistic network traffic patterns.

6 Related Work

Flow-based addressing builds on three bodies of prior work. First, on NIC-CPU throughput, prior work has identified bottlenecks in PCIe communication [30], descriptor receive rings [34], and the mismatch between packetized NIC interfaces and arbitrary-length application streams [39]; this paper contributes the SmartNIC memory bandwidth bottleneck arising from payload reassembly. Second, on zero-copy networking, Clark and Tennenhouse identified copy cost as a fundamental CPU throughput constraint as early as 1990 [11]; Cornflakes [36] and ZeroNIC [42] subsequently address scatter-gather bookkeeping overheads and page-alignment constraints in socket APIs. ZeroNIC's direct placement of payload data from the NIC into application buffers most closely parallels the flow cache proposed here, though flow-based addressing specifically targets transport-layer bytestreams rather than arbitrary accelerator destinations. Third, on stateful packet processing, managing flow entries at line rate grows harder as link speeds increase [15, 19, 37, 46], with competing hardware and software solutions for insertion, deletion, and out-of-order handling [33, 35, 51, 52]; prior work also cautions against rigid TCP hardware implementations [33, 40]. This paper adopts linked-list-based flow tracking as one viable implementation and acknowledges that more efficient schemes may emerge.

7 Conclusion

Hardware acceleration in NICs today focuses on making common or pervasive computations faster or more efficient, such as pushing and popping VLAN tags [22], checksums [14], and flow load balancing [5, 23]. OpenFlow and software defined networking (SDN) pushed this one step further, providing a rich set of hardware primitives for software to control and program a network [6, 9, 28]. By pushing these operations into hardware, NICs and their surrounding systems can process packets at line rate without using many CPU cycles.

This paper takes a different approach from this prior work—rather than accelerate an operation through a new offload, it aims to remove the operation entirely. Flow-based memory addressing changes the memory architecture so software no longer has to copy packets, improving per-core throughput and making it possible for modern SmartNICs to process application streams at line rate with limited memory bandwidth. SmartNICs inherited page-based memory management from the processor architectures they incorporate, despite the fact that they have a very specialized use case with a very different structure. Our key idea is to make addressing work in the way the applications think and use memory in the units that matter (i.e., packets are not pages) rather than relying on a level of indirection (e.g., scatter-gather and descriptors). As memory is increasingly

Context	Execution	Throughput	DRAM
Linear Scan			
Synthetic	62%	2.62 $\times$	83%
Campus	62%	2.66 $\times$	82%
Datacenter	62%	2.63 $\times$	81%
Regular Expression Search			
Synthetic	25%	1.33 $\times$	79%
Campus	26%	1.34 $\times$	78%
Datacenter	25%	1.34 $\times$	77%

Table 5. Relative Performance Gains. The near-identical results across synthetic, campus, and datacenter traces confirm that the benefit scales with traffic volume rather than with any particular flow size distribution.

the bottleneck in computing systems [25], our work suggests that there may be untapped potential in re-examining the memory addressing of high-performance devices.

Acknowledgments

This work was supported in part by a Sloan Research Fellowship and the National Science Foundation under Grant Number #2319080. A Stanford Graduate Fellowship in Science and Engineering, Sequoia Capital, Samsung, ASML, Google, EMD Electronics, the Stanford Differentiated Access Memories Project, and the U.S. Marine Corps Doctorate of Philosophy Technical Program also supported the authors of this work. We would also like to thank Mr. Greg Watson, whose expertise and guidance with FPGA development were instrumental. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

- [1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 63–74. doi:10.1145/1851275.1851192
- [2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (Hong Kong, China) (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. doi:10.1145/2486001.2486031
- [3] AMD. 2022. The OpenNIC Project. <https://github.com/Xilinx/open-nic>
- [4] AMD. 2026. AMD Vivado Design Suite. <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html>
- [5] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. doi:10.1145/3359989.3365412
- [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 *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. doi:10.1145/2486001.2486011
- [7] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2019. Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article 56 (Dec. 2019), 25 pages. doi:10.1145/3366704
- [8] Qizhe Cai, Shubham Chaudhary, Midhul Vuppapapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the ACM SIGCOMM 2021 Conference*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3452296.3472888
- [9] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.* 37, 4 (Aug. 2007), 1–12. doi:10.1145/1282427.1282382
- [10] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. 2024. Demystifying Datapath Accelerator Enhanced Off-path SmartNIC. arXiv:2402.03041 [cs.NI] <https://arxiv.org/abs/2402.03041>
- [11] D. D. Clark and D. L. Tennenhouse. 1990. Architectural considerations for a new generation of protocols. *SIGCOMM Comput. Commun. Rev.* 20, 4 (Aug. 1990), 200–208. doi:10.1145/99517.99553
- [12] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauch Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCaboote, Marc de Kruijff, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI'18: Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*. 373–387. <https://www.usenix.org/conference/nsdi18/presentation/dalton>
- [13] Sarang Dharmapurikar and Vern Paxson. 2005. Robust TCP Stream Reassembly in the Presence of Adversaries. In *Proceedings of the 14th USENIX Security Symposium*. <https://www.usenix.org/conference/14th-usenix-security-symposium/robust-tcp-stream-reassembly-presence-adversaries>

- [14] Linux Kernel Documentation. 2025. Checksum Offloads. <https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html>
- [15] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3445814.3446724
- [16] Michael Finsterbusch, Chris Richter, Eduardo Rocha, Jean-Alexander Muller, and Klaus Hanssgen. 2014. A Survey of Payload-Based Traffic Classification Approaches. *IEEE Communications Surveys and Tutorials* 16, 2 (2014), 1135–1156. doi:10.1109/SURV.2013.100613.00161
- [17] Daniel Firestone. 2017. VFP: a virtual switch platform for host SDN in the public cloud. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 315–328.
- [18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Anepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. <https://www.usenix.org/conference/nsdi18/presentation/firestone>
- [19] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *NSDI'22: Proceedings of the 19th USENIX Conference on Networked Systems Design and Implementation*. 807–827. <https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni>
- [20] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Cambridge, MA.
- [21] Jack Tiger Humphries, Neel Natsu, Kostis Kaffes, Stanko Novaković, Paul Turner, Henry M. Levy, David Culler, and Christos Kozyrakis. 2025. Wave: Offloading Resource Management to SmartNIC Cores. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Rotterdam, Netherlands) (ASPLOS'25)*. Association for Computing Machinery, New York, NY, USA, 264–281. doi:10.1145/3676642.3736113
- [22] IEEE. 1999. IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-1998* (March 1999), 1–214. doi:10.1109/IEEESTD.1999.89204
- [23] Intel. 2016. Receive-Side Scaling (RSS). <https://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf>
- [24] M. R. Siavash Katebzadeh, Antonios Katsarakis, and Boris Grot. 2025. Reliable Replication Protocols on SmartNICs. arXiv:2503.18093 [cs.DC] <https://arxiv.org/abs/2503.18093>
- [25] Peijiang Li, Muhammad Shahir Abdurraman, Rachel Cleaveland, Sergey Legtchenko, Philip Levis, Ioan Stefanovici, Thierry Tambe, David Tennenhouse, and Caroline Trippel. 2025. Towards Memory Specialization: A Case for Long-Term and Short-Term RAM. arXiv:2508.02992 [cs]. doi:10.48550/arXiv.2508.02992
- [26] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. 2021. Performance Characteristics of the BlueField-2 SmartNIC. arXiv:2105.06619 [cs]. doi:10.48550/arXiv.2105.06619
- [27] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (Beijing, China) (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. doi:10.1145/3341302.3342079
- [28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. doi:10.1145/1355734.1355746
- [29] YoungGyou Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92. <https://www.usenix.org/conference/nsdi20/presentation/moon>
- [30] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (Budapest, Hungary) (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. doi:10.1145/3230543.3230560
- [31] NVIDIA. 2024. NVIDIA BlueField Networking Platform. <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>

- [32] Seo Jin Park, Ramesh Govindan, Kai Shen, David Culler, Fatma Özcan, Geon-Woo Kim, and Hank Levy. 2025. Lovelock: Towards Smart NIC-Hosted Clusters, In *ACM SIGENERGY Energy Informatics Review*. *SIGENERGY Energy Inform. Rev.* 4, 172–179. doi:10.1145/3727200.3727226
- [33] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafir. 2021. Autonomous NIC offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual, USA) (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 18–35. doi:10.1145/3445814.3446732
- [34] Boris Pismenny, Adam Morrison, and Dan Tsafir. 2025. Disentangling the Dual Role of NIC Receive Rings. In *OSDI '25: Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*. 651–669. <https://www.usenix.org/conference/osdi25/presentation/pismenny>
- [35] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI'19: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. 531–548. <https://www.usenix.org/conference/nsdi19/presentation/pontarelli>
- [36] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 200–215. doi:10.1145/3600006.3613137
- [37] Thea Rossman, Diana Qing, Gerry Wan, and Zakir Durumeric. 2026. Iris: Expressive Traffic Analysis for the Modern Internet. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (Renton, WA, USA) (NSDI '26)*. USENIX Association, USA.
- [38] Benny Rubin, Saksham Agarwal, Qizhe Cai, and Rachit Agarwal. 2024. Fast & Safe IO Memory Protection. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, 95–109. doi:10.1145/3694715.3695943
- [39] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Enso: A Streaming Interface for NIC-Application Communication. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 1005–1025. <https://www.usenix.org/conference/osdi23/presentation/sadok>
- [40] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. USENIX Association, Renton, WA, 87–102. <https://www.usenix.org/conference/nsdi22/presentation/shashidhara>
- [41] Justine Sherry. 2024. The I/O Driven Server: From SmartNICs to Data Movement Controllers. *SIGCOMM Computer Communication Review* 53, 3 (Feb. 2024), 9–17. doi:10.1145/3649171.3649174
- [42] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelman, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. 2024. High-throughput and flexible host networking for accelerated computing. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (Santa Clara, CA, USA) (OSDI'24)*. USENIX Association, USA, Article 22, 19 pages.
- [43] Snort. 2025. <https://www.snort.org/>. <https://www.snort.org/>
- [44] Naru Sundar, Brad Burres, Yadong Li, Dave Minturn, Brian Johnson, and Nupur Jain. 2023. 9.4 An In-depth Look at the Intel IPU E2000. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. 162–164. ISSN: 2376-8606. doi:10.1109/ISSCC42615.2023.10067333
- [45] Suricata. 2025. <https://www.suricata.io/>. <https://www.suricata.io/>
- [46] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. 2022. Retina: analyzing 100GbE traffic on commodity hardware. In *Proceedings of the ACM SIGCOMM 2022 Conference (Amsterdam, Netherlands) (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 530–544. doi:10.1145/3544216.3544227
- [47] Gerry Wan, Shinan Liu, Francesco Bronzino, Nick Feamster, and Zakir Durumeric. 2025. CATO: end-to-end optimization of ML-based traffic analysis pipelines. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (Philadelphia, PA, USA) (NSDI '25)*. USENIX Association, USA, Article 81, 18 pages.
- [48] Xingda Wei, Rongxin Cheng, Yuhang Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 987–1004. <https://www.usenix.org/conference/osdi23/presentation/wei-smartnic>
- [49] Keith Wiles. 2026. Pktgen: A DPDK-based Packet Generator. <https://github.com/pktgen/Pktgen-DPDK>
- [50] Xilinx. 2024. MicroBlaze Processor Reference Guide (UG984). https://docs.amd.com/r/en-US/ug984-vivado-microblaze-ref?tocId=TgJTXF9qLhuPO_9p3XVnZQ
- [51] Zhipeng Zhao, Joseph Melber, Siddharth Sahay, Shashank Obla, Eriko Nurvitadhi, and James C. Hoe. 2023. Exploiting the Common Case When Accelerating Input-Dependent Stream Processing by FPGA. *IEEE Trans. Comput.* 72 (May

2023), 1343–1355. doi:10.1109/TC.2022.3200576

- [52] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1083–1100. <https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng>

A Dealing with Unaligned Cases

The flow cache controller reads payload data from DRAM in 64-byte bursts over an AXI bus. Given transport-layer payloads follow variable-length headers, their starting offsets and lengths rarely align to burst boundaries. Two factors define the alignment relationship between a payload and DRAM bursts: the starting offset of the payload and its length. Together, these factors yield four distinct cases that the flow cache controller must handle correctly and efficiently (Figure 14). This appendix details each case and the strategy used to address it.

Case 1 (Offset Aligned, Length Aligned):

This is the ideal scenario. As shown in Figure 14a, the flow cache controller can read 64-byte bursts directly from DRAM and store them into the flow cache without any additional processing. For instance, a request at offset 0 with length 128 results in two clean accesses of DRAM reads that map directly into the cache. In practice, this case is rare. Transport-layer payloads usually follow variable-length headers (e.g., Ethernet, IP, TCP, etc.), which prevent alignment to burst boundaries. On the Alveo U250, payloads typically begin after 54 bytes of headers, making a perfectly aligned start offset unlikely.

Case 2 (Offset Aligned, Length Unaligned): The payload begins on a burst boundary, but its length is not a multiple of the burst size. The flow cache controller must therefore handle a partial burst at the end of the transfer. Given that payload starts are typically unaligned, this case seldom occurs in practice. Both Case 1 and Case 2 can thus be dismissed as uncommon scenarios for the flow-based addressing scheme.

Case 3 (Offset Unaligned, Length Aligned):

— This case reflects common traffic patterns, where payloads begin at unaligned offsets (e.g., 54 bytes) but still have lengths that align to access boundaries. For instance, a request starting at offset 54 and spanning 128 bytes requires three DRAM bursts, producing 10B, 64B, and 54B of valid data as shown in Figure 14b.

Case 4 (Offset Unaligned, Length Unaligned) — Flow Cache Data Aligner: This is the most general and complex scenario, where

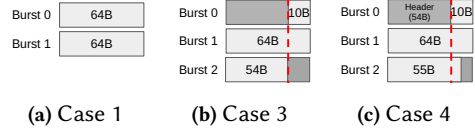


Fig. 14. Examples of DRAM unalignment cases for flow cache writes. (a) Aligned case: Data starts and ends on 64 B boundaries, allowing direct burst writes. (b) Partially unaligned case: The payload starts at an unaligned offset but ends on a boundary, enabling partial reads at the start and end to be merged with adjacent data for efficient storage. (c) General unaligned case: Both start offset and length are unaligned, requiring the Flow Cache Data Aligner to handle valid-byte extraction and sequential writing. Since most real-world traffic falls into the unaligned cases, hardware support for partial-burst handling is essential to maintain correctness and throughput.

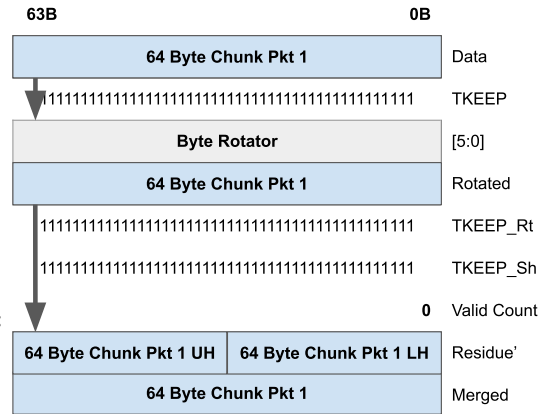


Fig. 15. Flow Cache Data Aligner Operation: Using the AXI control signal TKEEP, the Data Aligner identifies valid bytes in each read from DRAM, rotates, and merges them with any residual bytes from the previous read to form contiguous 64B writes into the flow cache.

					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		
					Count		
					TCP Flags		

Metric	Value	Payload Range	Count	(%)	
Payload size per packet	1,460 B	0 B (header-only)	0	0.00	
Packets written	18,607	1–100 B	0	0.00	
Unique flows	10,000	101–500 B	0	0.00	
Avg. packets per flow	1.9	501–1000 B	0	0.00	
PCAP files	2	1001–1400 B	0	0.00	
Total PCAP size	28.5 MB	1401+ B	18,607	100.00	
					TCP Flags Count (%)
					ACK 18,607 100.00

(a) Trace Summary. (b) Payload Size Distribution. (c) TCP Flag Distribution.

Fig. 18. Characterization of the synthetic datacenter web-search traffic trace. (a) 18,607 packets across 10,000 flows, averaging 1.9 packets per flow. (b) 100% of frames carry a full 1,460-byte payload. (c) All 18,607 packets carry only the ACK flag (100%).

Reconstruction. A Python reconstruction script merges the 16 per-core binary streams using a min-heap keyed on nanosecond timestamps, requiring $O(C)$ memory proportional to capture core count C . For each record, the script synthesizes a complete Ethernet/IPv4/TCP frame with independently verified checksums and a zero-filled payload matching the recorded length, preserving frame sizes, inter-packet timing, and flow structure while satisfying anonymization requirements. The script partitions output into PCAP files capped at 16,384 packets to respect the MicroBlaze CPU's FIFO queue depth. Both tools are available in the online repository.

Trace characterization. The 5 min capture window yields 14.64 million packets from 100,211 flows totaling 15.73 GB of metadata across 894 PCAP files (Figure 17a). Payload sizes confirm bulk-transfer dominance: 71% of frames carry more than 1,000 B (Figure 17b). Figure 16 shows that the top 1% of flows each exceed 600 KB and carry 96% of all traffic volume. ACK and PSH+ACK frames together account for 98% of packets (Figure 17c). This flag distribution exercises all three firmware segment reassembly paths: PSH+ACK continuation, FIN termination, and RST recovery, confirming the trace faithfully represents a real-world, bulk-transfer-dominated workload.

C Datacenter Web-Search Network Traffic Trace

Trace construction. We derive the datacenter trace from the web-search flow-size distribution characterized by DCTCP experiments [1, 2], which spans 2 KB latency-critical query flows to 100 MB throughput-sensitive background update flows (see Figure 19). Measurements from production web-search clusters document that more than 95% of all bytes originate from the 30% of flows exceeding 1 MB [1, 2]. This distribution simultaneously exercises the short-flow case, where flow translation table setup cost must remain negligible relative to total execution time, and the byte-dominant large-flow case, where the flow cache controller must sustain high throughput across many payload segments and amortize that setup cost across a complete reassembled bytestream. A purpose-built generation script samples the empirical CDF to produce 10,000 directional flows and generates complete Ethernet/IPv4/TCP frames at maximum transmission unit size for each sampled flow, yielding 18,607 packets averaging 1.9 packets per flow.

Reconstruction. The generation script constructs each frame from a 14-byte Ethernet header, a 20-byte IPv4 header, and a 20-byte TCP header with independently verified checksums followed by

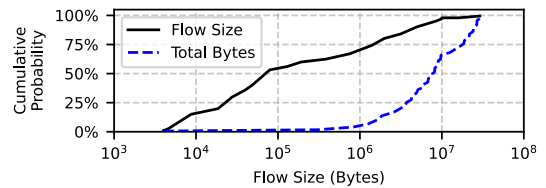


Fig. 19. CDF of datacenter web-search flow sizes. More than 95% of bytes originate from the 30% of flows exceeding 1 MB, confirming elephant-flow dominance. Recreated from data reported in [2].

a full 1,460-byte payload, and partitions the output into two sequential PCAP files each bounded at 16,384 packets to respect the MicroBlaze CPU's slower 300 MHz clock speed. The script verifies Ethernet/IPV4/TCP checksums across all 18,607 frames prior to replay.

Trace characterization. Figure 18a summarizes the trace: 18,607 packets across 10,000 flows averaging 1.9 packets per flow, totaling 28.5 MB across 2 PCAP files. Figure 18b confirms that all 18,607 packets carry a full 1,460-byte payload, placing every frame in the highest payload range and presenting the flow cache controller with a uniform stream of maximum transmission unit segments. Figure 18c shows that all packets carry only the ACK flag, which exercises the firmware's sliding window mechanism across an uninterrupted sequence of contiguous segments and confirms the trace presents a uniform, high-density workload that stresses the flow cache controller's high-throughput segment assembly path.

D Linked-List Metadata Footprint and DRAM Bandwidth Cost

Section 5 excludes the time and memory transactions associated with maintaining the flow linked lists because our prototype manages linked lists in software on the MicroBlaze soft CPU rather than in hardware. This appendix provides an analytical estimate of the metadata footprint and DRAM bandwidth cost of this software-managed approach, grounded in the firmware's own struct layouts and constants (Section 4).

Metadata footprint. Each linked-list node (`flow_node_t`) stores a 4-byte flow identifier, the 40-byte packet-metadata record (`pkt_meta_t`, sized by the fixed `PKT_META_SIZE` constant), and 2-byte next/previous segment pointers, for 48 B per received packet (up to 56 bytes under worst-case compiler alignment). Since we cap the node pool at 20,000 entries in our implementation, the aggregate metadata occupies at most 1 MiB of DRAM regardless of trace length, which is under 0.01% of the 16 GB DDR4 capacity we dedicate for our experiments.

Per-packet DRAM bandwidth cost. Each list append issues one 48-byte node write, which fits within a single 64-byte DRAM burst matching the flow cache controller's own DRAM access granularity (Section 4), plus a read-modify-write of the previous node's link field, bounding the added DRAM traffic to between 64 B and 192 B per packet.

Applied to our traces. Applied to our traces (10 synthetic, 14.64 million campus, and 18,607 datacenter packets), this adds at most 1.9 KB, 2.81 GB, and 3.57 MB of DRAM traffic, respectively – 4–12% on top of the flow-based design's already-measured DRAM traffic in every case. Folding this into the measured ratios raises them from 1.05–1.18 $\times$ payload size to at most 1.09–1.31 $\times$, still a 79–82% reduction versus the copy-based baseline's 5.0–6.19 $\times$ and within a few percentage points of our measured 81–83% reduction for linear scan. Prior work has also shown that linked-list maintenance can be offloaded to hardware entirely at line rate [52], which would eliminate even this small residual cost.

Received 12 December 2025; revised 29 May 2026; accepted 22 June 2026