RESEARCH-ARTICLE

Accelerating Stream Processing Engines via Hardware Offloading

ZHENGYAN GUO, Tsinghua University, Beijing, China

MINGXING ZHANG, Tsinghua University, Beijing, China

YINGDI SHAN, Tsinghua University, Beijing, China

KANG CHEN, Tsinghua University, Beijing, China

JINLEI JIANG, Beijing National Research Center for Information Science and Technology, Beijing, China

YONGWEI WU, Tsinghua University, Beijing, China

Proceedings of the ACM on Management of Data, Volume 3, Issue 6 (December 2025)
https://doi.org/10.1145/3769754
EISSN: 2836-6573

Accelerating Stream Processing Engines via Hardware Offloading

ZHENGYAN GUO, Tsinghua University, China
MINGXING ZHANG*, Tsinghua University, China
YINGDI SHAN, Tsinghua University, China
KANG CHEN, Tsinghua University, China
JINLEI JIANG, Dept. of Computer Science and Technology, BNRist, Tsinghua University, China
YONGWEI WU, Tsinghua University, China

Modern stream processing engines (SPEs) must handle massive real-time data streams under strict latency and throughput requirements. However, conventional SPEs are constrained by their software parallelization strategies (e.g., queue-based data re-partitioning, high synchronization overheads, etc.), which prevent efficient utilization of modern hardware capabilities, ultimately limiting performance scalability. In this paper, we present FlexStream, a novel SPE that leverages hardware offloading to redesign the parallelization strategies and overcome these limitations. By offloading data re-partitioning to hardware and integrating a coupled network-executor model, FlexStream maximizes resource utilization, achieving up to 95% network bandwidth saturation. To address the load imbalance challenges introduced by this design, we implement a lock-free state backend with efficient state migration mechanisms. Overall, FlexStream achieves throughput improvements of 1.95× −3.35× compared to state-of-the-art SPEs (e.g., LightSaber) across six real-world streaming analytics applications. FlexStream cuts latency spikes by 71.9% and migration time by 66.8% during state migration, highlighting the benefits of hardware-software co-design in SPEs. Our work underscores the potential of hardware-software co-design in SPEs, offering a scalable, elastic solution for real-time analytics.

CCS Concepts: • **Information systems → Stream management**.

Additional Key Words and Phrases: stream management, hardware offloading, elasticity, real-time analytics

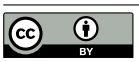
ACM Reference Format:
Zhengyan Guo, Mingxing Zhang, Yingdi Shan, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2025. Accelerating Stream Processing Engines via Hardware Offloading. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 289 (December 2025), 23 pages. https://doi.org/10.1145/3769754

1 Introduction

In recent years, stateful Stream Processing Engines (SPEs) [17, 18, 31, 37, 41, 45] have gained widespread adoption due to the growing demand for real-time analytics in scenarios such as fraud detection, log monitoring, sentiment analysis, and IoT applications [1, 12, 16, 47]. SPEs are expected to be long-running while consistently maintaining low latency and high throughput performance.

*Mingxing Zhang is the corresponding author.

Authors' Contact Information: Zhengyan Guo, Tsinghua University, Beijing, China, guozy22@mails.tsinghua.edu.cn; Mingxing Zhang, Tsinghua University, Beijing, China, zhang_mingxing@mail.tsinghua.edu.cn; Yingdi Shan, Tsinghua University, Beijing, China, shanyd@tsinghua.edu.cn; Kang Chen, Tsinghua University, Beijing, China, chenkang@tsinghua.edu.cn; Jinlei Jiang, Dept. of Computer Science and Technology, BNRist, Tsinghua University, Beijing, China, jjlei@tsinghua.edu.cn; Yongwei Wu, Tsinghua University, Beijing, China, wuyw@tsinghua.edu.cn.



ACM 2836-6573/2025/12-ART289
https://doi.org/10.1145/3769754

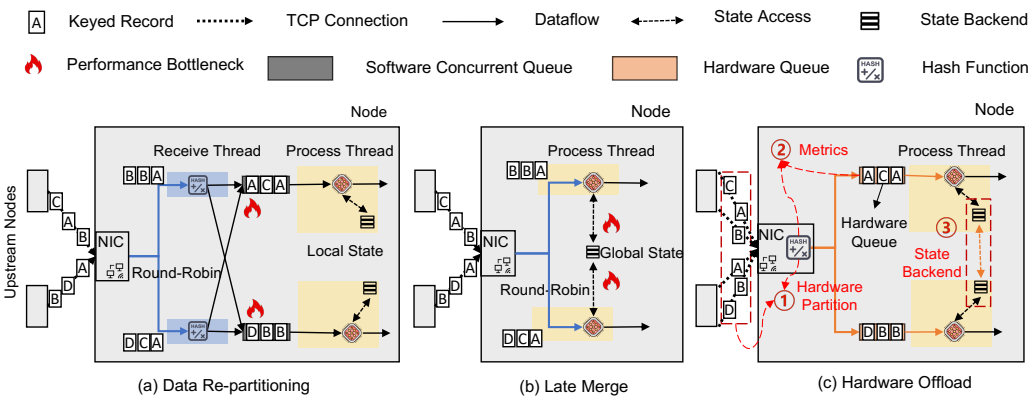


Fig. 1. Comparison of different parallelization strategies.

Although current SPEs can improve overall performance by scaling out the clusters, they fail to fully leverage the advanced hardware capabilities of modern machines, such as high-speed networks and multi-core architectures. According to our evaluation, on a single node, the throughput of state-of-the-art SPEs typically peaks and begins to decline when using more than 8 cores. Additionally, with a 100 *Gbps* Ethernet network bandwidth, they can utilize at most only 40 *Gbps*. Consequently, improving single-machine performance (i.e. scaling up) has emerged as a key research focus in distributed stream processing systems.

Many popular SPEs, such as Apache Flink [17], Apache Spark [41], and Apache Storm [45], rely on parallelization strategies that called data re-partitioning, as shown in Figure 1(a). In this model, receiver threads distribute incoming network data using key-based hash partitioning to multiple-producer-single-consumer (MPSC) queues which is then processed by a dedicated process thread. However, recent works [14, 22, 43, 52] have demonstrated that the excessive use of concurrent queues in SPE introduces significant overhead, becoming a critical performance bottleneck. We conducted experiments using the Yahoo Stream Benchmark [8, 9], a representative stateful stream processing workload. The results, shown in Figure 2, reveal that even with 32 cores, the throughput of the data re-partitioning SPE is limited to just 60 million records per second.

To eliminate the need for software concurrent queues, recent studies [22, 43, 52] have proposed the Late Merge execution strategy, as shown in Figure 1(b). It uses efficient global state backends (e.g., PAT in LightSaber [43]) to replace software queues. As shown in Figure 2(a), under the Late Merge strategy, the SPE can achieve a maximum of 302 million records/s. However, the synchronization overhead among processing threads becomes a severe bottleneck, preventing query performance from scaling effectively beyond 8 cores. Even with additional cores and network bandwidth, its performance does not further improve.

Based on the above analysis, we observe that performance bottlenecks in existing SPEs arise from inefficient software-based concurrent data structures used for data shuffling operations, including mpsc-queues and global state management. Modern hardware, however, offers sophisticated capabilities that can address these limitations more effectively. Receiving Side Scaling (RSS) [3] represents one such advancement—a network driver technology that intelligently distributes network reception workloads across multiple CPU cores in contemporary systems. As illustrated in Figure 1(c), we have developed a novel parallelization strategy that incorporates hardware capabilities into the core design. Our approach changes the inter-node network connection model by replacing multiplexed connections with multiple dedicated connections. This fundamental architectural

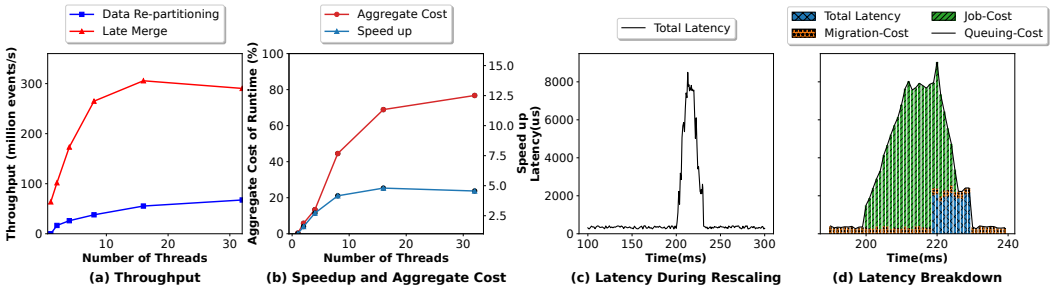


Fig. 2. Micro-benchmark analysis. (a) Throughput vs. number of threads across different parallelization strategies (b) Overhead breakdown of aggregate cost in the late merge parallelization strategies (c) Latency variation during rescaling, with data skew at 200 ms and rescaling completion at 230 ms. (d) Latency breakdown of (c).

redesign enables us to leverage RSS to perform data shuffling directly during network transmission (Figure 1(c)①). By offloading the data shuffle process to hardware, we eliminate the dependency on software concurrent data structures that traditionally limit performance, allowing SPEs to fully utilize both the computational and network resources available in modern hardware architectures.

While our hardware-accelerated parallelization strategy effectively eliminates software bottlenecks and significantly enhances stream processing engine performance, it introduces a challenge: since each CPU core processes a distinct set of keys bound to specific network connections, severe load imbalance can occur when data skew emerges. This imbalance leaves some CPUs becoming overwhelmed while others remain underutilized, degrading overall throughput and increasing processing latency. Current systems typically address this problem through rescaling techniques that redistribute tasks across cores and migrate state as necessary. However, existing rescaling mechanisms suffer from prolonged state migration periods latency spikes during migration. As illustrated in Figure 2(c), during rescaling, processing latency peaks at 8ms—more than 30 times the latency observed during stable operation.

Our analysis reveals that rescaling latency primarily stems from two phases of the rescaling process: ***(1) Rescaling Trigger Phase:*** This phase begins when data skew occurs and continues until the system detects the imbalance and initiates state migration. Most contemporary SPEs [17, 21, 23] employ Backpressure mechanisms [10, 18, 31] as their primary rescaling trigger signal. While backpressure accurately indicates load imbalance, it leads to substantial record accumulation during the trigger phase, contributing to high queuing-cost. ***(2) State Migration Phase:*** This phase manages state redistribution across cores to restore balance. Since multiple threads must concurrently access the same state during migration, existing SPEs need to implement state migration protocols to ensure state consistency (e.g., Meces [23]). However, constrained by previous parallelization strategies and state backends, these protocols require frequent lock-related operations, resulting in excessive migration-cost and further increasing processing latency.

Based on this comprehensive analysis, we developed a more efficient rescaling mechanism for FlexStream. We implemented a proactive load balancing scheduler that replaces reactive backpressure mechanisms. As shown in Figure 1(c)②, our scheduler actively collects traffic statistics and continuously monitors hardware queue record accumulation rates. By leveraging these metrics, the system can detect data skew more rapidly and provide fine-grained information for precise migration strategies, effectively reducing queuing-cost.

Additionally, we designed a novel state migration protocol and lock-free state backend (Figure 1(c)③) to optimize state migration phase. Our state backend minimizes thread contention by implementing lock-free access through Read-Copy-Update (RCU) mechanisms and Fetch-And-Add (FAA) atomic instructions. Furthermore, our state migration protocol ensures state consistency throughout the migration process, preserving the exactly-once semantics [33] essential for stream processing—guaranteeing that each record modifies state exactly once. This design significantly reduces system impact during migration, ensuring lower latency spikes and faster migration completion times.

To summarize, our work replaces inefficient software-based shuffling with high-performance hardware-accelerated alternatives, substantially improving throughput, while simultaneously addressing the resulting load imbalance challenges through an efficient rescaling mechanism that maintains high elasticity across the entire stream processing pipeline. Evaluation results demonstrate that FlexStream achieves throughput improvements of 3.35× compared to state-of-the-art SPEs across six real-world streaming analytics applications. Concurrently, FlexStream reduces latency spikes by 71.9% and migration time by 66.8% during state migration operations.s

2 Background and Related Work

This section provides the necessary background for our paper and analyzes the limitations of current SPEs when running on modern hardware. Sec. 2.1 introduces the fundamental concepts of stateful stream processing and explores the limitations of various parallelization strategies. Sec. 2.2 discusses rescaling mechanisms in stateful SPEs and identifies shortcomings in existing mechanisms.

2.1 Stateful Stream Processing Engines

Data and query model. As described by Fernandez et al. [6], SPEs treat data as an infinite stream of immutable, unbounded records. Each record contains a timestamp t, a primary key k, and a set of attributes. Timestamp are strict monotonically increasing and used for windowing related operations as well as progress tracking. Streaming queries are modelled as directed acyclic graphs with stateful operators as vertices and data flows as edges. The output of a streaming operator depends on content, timestamp or arrival order of input records, and its intermediate state. In general, an operator must output no result at a timestamp t that is computed using records bearing timestamps greater than t.

Different processing model and parallelization strategies. Current SPEs typically use either a scale-up or scale-out processing model. Scale-out SPEs such as Flink [17] implement a shared-nothing architecture and leverage data re-partitioning [24, 31] as their primary parallelization strategy to achieve scalability. In this distributed processing paradigm, input data streams are segmented into independent substreams based on partition keys, enabling parallel processing across multiple nodes in a cluster. These parallel processing instances communicate with upstream operators through concurrent queues or network-based data channels that follow established exchange patterns. In contrast, scale-up SPEs [27, 36, 44, 53], including Streambox-HBM [35], LightSaber [43], and Grizzly [22], employ a shared-everything architecture. Streambox-HBM is designed for NUMA-aware processing on multi-core machines and focuses on leveraging High Bandwidth Memory (HBM) to accelerate stream processing operator execution. Grizzly and LightSaber enhance processing efficiency through sophisticated parallelization techniques such as task-based parallelization and late merge strategies. Within these architectures, parallel processing instances either concurrently update a shared global operator state or maintain local state updates that the SPE subsequently merges while ensuring consistency guarantees. This architectural distinction shapes how these systems balance throughput, latency, and resource utilization in stream processing

workloads. Notably, recent work on Slash [14] has applied the late merge strategy to scale-out SPEs by leveraging native RDMA acceleration [51, 54].

Limitations of current parallelization strategies. For data re-partitioning strategies, recent research by Zeuch et al. [52] has established that software queues create severe bottlenecks, ultimately limiting the maximum achievable throughput. While Late Merge parallelization strategies attempt to address this limitation by eliminating software queues through thread synchronization, they introduce new inefficiencies. Specifically, the frequent use of locks and atomic instructions required to maintain state consistency during thread synchronization becomes a performance bottleneck itself. As illustrated in Figure 2(b), our experimental evaluation of the Late Merge strategy reveals that performance improvements plateau beyond 8 cores, achieving only approximately a 5× speedup despite additional computational resources. Through detailed performance profiling, we discovered that synchronization overhead grows substantially as thread count increases. At higher parallelism levels, synchronization operations consume nearly 80% of the total execution time, severely limiting throughput scalability.

Challenge ①: Our first research challenge is developing a novel parallel execution strategy that maximizes hardware utilization while minimizing or completely eliminating thread synchronization overhead.

2.2 Prior Rescaling Mechanisms

Stream processing workloads are highly dynamic and often unpredictable. To maintain load balance, SPEs rely on rescaling mechanisms, which typically consist of two phases: rescaling trigger phase and state migration phase.

2.2.1 Rescaling Trigger Mechanisms. The rescaling trigger phase serves to detect data skew and initiate state migration phase. This phase collects metrics used to provide enough information to make fast and accurate decisions as to how to rescale the system. Backpressure is the most commonly used mechanism for triggering state migration in current SPEs: when load imbalancing occurs, records accumulate in record queues, and once these backlogs exceed predefined backpressure thresholds, the system generates backpressure signals. These signals propagate upstream to data shuffling operators, prompting task reallocation to address the imbalance.

Limitations of backpressure mechanisms. While backpressure mechanisms effectively identify operators experiencing performance bottlenecks, they introduce substantial performance degradation during the rescaling process. First, these mechanisms rely on record accumulation in record queues before triggering state migration, which inherently increases end-to-end processing latency. Second, backpressure signals provide only coarse-grained diagnostic information. While sufficient to identify overloaded operators, they lack the granularity needed to determine specific remediation strategies, such as which key states should be redistributed to restore processing balance. As illustrated in Figure 2(c) and Figure 2(d), current SPEs exhibit pronounced latency spikes when data skew occurs. The queuing-cost represents additional processing delay incurred during state migration, primarily due to processing substantial backlogs of accumulated records—a direct consequence of backpressure-based detection. Our analysis reveals that the rescaling trigger phase alone accounts for approximately 62% of the total rescaling process duration, constituting the primary contributor to elevated latencies during rescaling operations. This finding highlights the need for more responsive and fine-grained approaches to workload monitoring and state migration in stream processing systems.

Challenge ②: Our second research challenge is developing fine-grained metrics that provide precise diagnostic information to guide state migration decisions. Such metrics must be coupled

with an efficient and responsive trigger mechanism that can detect and address data skew with minimal latency impact.

2.2.2 State Migration Mechanisms. The critical difficulty in elastic stream processing systems is efficiently migrating state between operator instances while maintaining exactly-once semantics. Traditional SPEs typically adopt a full-restart strategy, where execution is halted, snapshots of all stateful operators are taken, and the parallelism is reconfigured before restarting the stream processing task. However, this approach incurs high overhead because it requires restarting the system for every adjustment. Recent state-of-the-art (SOTA) approaches [13, 25, 28, 48] employ proactive state migration strategies. These methods perform additional operations during normal execution and design specialized migration protocols to reduce migration latency. Notable implementations of such proactive strategies include ChronoStream [50], Meces [23] and Rhino [13].

Limitations of current state migration mechanisms. Chrono-Stream [50] introduces a lightweight state-management abstraction that divides state into a fixed number of compute state slices, managing and migrating state during vertical elasticity by rescheduling threads and state slices. It uses a lightweight optimistic readers-writer latch to ensure correctness during concurrent state access by multiple threads. This mechanism is equivalent to Meces's Fetch-on-demand State Accessing Protocol [23] in single-node scenarios, so we select the more recent work Meces as the baseline for evaluating state migration mechanisms. Meces employs a control messaging-based coordination protocol to ensure state consistency during prioritized state migration. However, a significant limitation emerges when multiple threads operate within the same physical machine: the protocol effectively degrades into a lock-based synchronization mechanism where threads must sequentially acquire and release locks on concurrent data structures. This lock contention becomes the primary contributor to migration-cost overhead, as illustrated in Figure 2(d). Our measurements reveal that migration-cost accounts for approximately 38% of the total migration time and is 10 times higher than the job-cost during normal operation. These findings underscore the need for more efficient state migration mechanisms that can minimize synchronization overhead while maintaining state consistency across parallel processing threads.

Challenge ③: The third research challenge we address is the development of an efficient state migration mechanism that capitalizes on single-node architectural characteristics to substantially reduce lock contention and associated overhead during the rescaling process, thereby minimizing disruption to stream processing performance.

3 System Design

As depicted in Figure 3, FlexStream's architecture consists of three core components that enable efficient and elastic stream processing: a hardware-accelerated parallelization strategy, a load-balancing scheduler, and an advanced state migration protocol with a redesigned state backend. These components address the three challenges outlined earlier, respectively.

Hardware offloading parallelization strategy. To address challenge ①, FlexStream leverages modern hardware capabilities through a hardware-software co-design approach (detailed in Sec. 4). We offload data repartitioning overhead to NIC hardware by utilizing the RSS functionality to perform data shuffling directly in hardware. This required redesigning both the network transfer model and the processing model for handling data streams from hardware queues. In our approach, each CPU core processes partition streams segmented by keys, eliminating inter-thread synchronization mechanisms.

Load-balancing scheduler. To address challenge ②, FlexStream implements an innovative load-balancing scheduler (described in Sec. 5) that rapidly responds to load imbalances and facilitates precise migration decisions. Instead of relying on reactive backpressure signals, we employ proactive

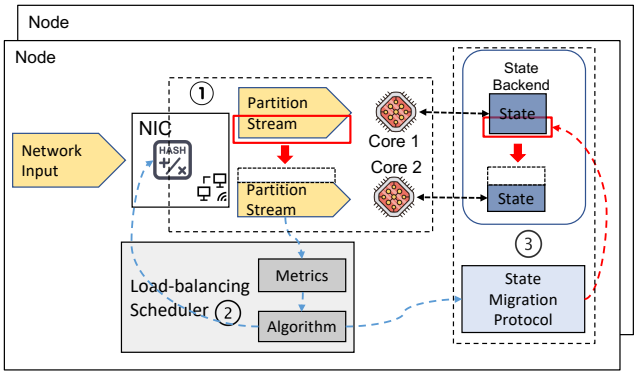


Fig. 3. The architecture of FlexStream.

monitoring of system metrics, including network traffic patterns and packet accumulation rates in hardware queues. This approach accelerates decision-making and provides finer-grained diagnostic information to support migration algorithms, enabling targeted remediation strategies tailored to current workload characteristics.

State migration protocol. To address challenge ③, we introduce a lock-free state backend and a state migration protocol that ensures state consistency during migration (detailed in Sec. 5.4). Our state backend implementation leverages the Read-Copy-Update (RCU) mechanism and atomic instructions, providing two key benefits: efficient core access to states during balanced load conditions, and minimal thread contention during state migration under load imbalances.

FlexStream's architecture effectively addresses all three challenges through these integrated components, enabling high performance, elastic stream processing that fully leverages modern hardware capabilities.

4 Parallelization Strategy

In this section, we present FlexStream's hardware offloading parallelization strategy, designed to achieve high throughput in SPEs. We first outline the design principles and details of our parallelization strategy in Sec. 4.1. We then describe the execution mechanisms required to implement this strategy in Sec. 4.2. This includes modifications to the network transfer model and mechanisms to ensure correctness of SPE processing semantics. Finally, Section 4.3 details the range of stateful operators that our parallelization strategy supports.

4.1 Hardware Offloading

Conventional parallelization strategies in SPEs represent a trade-off between data shuffling overhead and thread synchronization costs [20, 26, 32]. These approaches primarily optimize execution at the software layer, often underutilizing the capabilities of modern hardware. However, integrating hardware acceleration into parallelization strategies presents significant challenges due to the semantic gap between hardware operational characteristics and application-level requirements. For example, the RSS functionality in NICs typically operates on network-level information such as IP addresses and ports. In contrast, stream processing requires partitioning based on application-level keys. This fundamental mismatch creates a barrier to effective hardware utilization in stream processing systems. Bridging this semantic gap requires careful system design that aligns hardware capabilities with the specific requirements of stream processing workloads while maintaining correctness guarantees and processing semantics.

As illustrated in Figure 1(c), FlexStream employs task-parallel transformations to create physically partitioned data flows, ensuring that records with the same key are processed by the same thread. We leverage RSS functionality of NIC to partition streams during network transmission, distributing incoming data across cores based on hashing. SOTA distributed SPEs perform distributed data shuffles in two steps: Step 1, the upstream operator performs hash partitioning, sending data to different processing nodes; Step 2, the receiver thread on the processing node sends the corresponding event to the specific processing core through software queues. Our hardware offloading approach replaces software queues with efficient hardware queues, significantly reducing data shuffling overhead in Step 2 without affecting Step 1, thereby preserving distributed data shuffles in distributed stream processing while improving their efficiency. Our design is motivated by the observation that stream partitioning requires minimal computational resources, making it ideal for hardware offloading. RSS applies hash functions to incoming packets and assigns them to CPU cores based on the hash value. FlexStream adapts the network transfer process to align with RSS characteristics, eliminating thread synchronization overhead while enhancing throughput. FlexStream operates effectively in multi-tenant cloud environments such as Kubernetes [30] and supports virtual environments through hardware SR-IOV [15] capabilities. Nearly all modern NICs from leading vendors and major cloud platforms support both RSS and SR-IOV, ensuring FlexStream's broad portability across diverse deployment scenarios.

4.2 Execution Details

To realize our parallelization strategy, FlexStream must satisfy two key properties: 1) ensuring the correctness of data partitioning during network-based data shuffling, and 2) guaranteeing the correctness of stream processing semantics in subsequent processing threads. Below, we explain how FlexStream achieves these properties.

Network transfer model. Traditional SPEs [2, 43, 53] establish a single network connection between nodes to simplify connection management, relying on multiplexing to transmit data from multiple threads over this connection. However, this model intermingles data from all keys within a single connection, preventing hardware from effectively partitioning the stream by key. To overcome this limitation, FlexStream introduces a connection pool comprising multiple network connections between nodes. The number of connections is configured to be a small multiple of the number of processing cores, ensuring manageable network overhead while maintaining performance. Our network transfer model ensures that records with the same key are consistently routed to the same core at both sender and receiver ends.

At the sender, when a processing core transmits a record to a downstream node, it selects a specific network connection based on the record's key. The connection pool ensures that identical keys are always mapped to the same connection, guaranteeing correctness at the sender. At the receiver, since data for the same key may arrive from multiple upstream nodes, we group incoming connections from different nodes such that connections containing the same key are assigned to the same group. Data within a group is then routed to the same core. This approach ensures data consistency across both sender and receiver, enabling hardware-based data shuffling to correctly partition records by key to the designated core, thus satisfying Property 1.

Correctness of stream processing semantics. Consistent with systems like Apache Flink [17], FlexStream employs a watermark mechanism to ensure correct stream processing semantics. Unlike prior approaches that rely on a single connection between nodes and align watermarks based on the slowest key's progress, FlexStream achieves finer-grained watermark alignment and processing tracking. Our multi-connection model enables watermark alignment at the granularity of connection groups, meaning each key's processing progress depends only on the slowest key

within its group, rather than the slowest key across all connections. This enhances efficiency while preserving correctness.

After the network transmission phase, the input stream is partitioned, allowing each thread to process its data independently without inter-thread communication or synchronization, updating only its local state. However, partitioned streams are susceptible to load imbalances during data skew. We address this through dynamic rescaling, which migrates states between processing threads. Before and after rescaling, the partitioned streams processed by different cores remain orthogonal, ensuring semantic correctness. During rescaling, partitioned streams may exhibit partial overlaps across cores. Our state migration protocol manages these overlaps to ensure semantic correctness. Collectively, these mechanisms ensure the preservation of stream processing semantics, satisfying Property 2.

4.3 Expressiveness of Hardware Offloading

Our design supports any operator based on key partitioning, including key-based window aggregations and joins. We discuss the operator implementations in two specific scenarios: non-rescaling and rescaling operations.

Non-rescaling scenario: With hardware offloading, each core processes a distinct set of keys. Each core maintains and updates only its local state. This isolation ensures that operators execute correctly without coordination overhead between cores.

Rescaling scenario: Rescaling changes the mapping between keys and cores, which can result in the same key being processed by different cores simultaneously. We address this challenge using the state migration protocol described in Sec. 5.4. This protocol ensures consistent state updates across cores and maintains operator correctness during rescaling.

Beyond the operators mentioned above, FlexStream also supports complex UDFs (User-Defined Functions). Our system can implement the diverse set of UDFs officially supported by Apache Flink [17], including Scalar, Table, Aggregate, Table Aggregate, Async Table, and Process Table functions. Consider Flink's Table Function as an example. This function allows users to transform one row of input data into multiple output rows based on custom logic. In a joinLateral operation, users specify which field should be used to combine data from different sources. FlexStream takes advantage of this user-specified field by using it as the partitioning key for hardware acceleration.

5 Rescaling Mechanisms

In this section, we detail FlexStream's dynamic rescaling mechanisms, which enable efficient adaptation to changing workload characteristics and data skew patterns. Our approach centers on two key components: (1) Load-Balancing Scheduler to trigger rescaling: Sec. 5.1 provides a detailed analysis of our metrics collection methodology, while Sec. 5.2 presents our algorithmic approach to key-based traffic redistribution among processing cores. (2) Efficient State Migration Mechanism: we implement a novel lock-free state backend, described in Sec. 5.3. Sec. 5.4 then elaborates on the complete state migration protocol, demonstrating how FlexStream preserves processing semantics and minimizes performance degradation during redistribution events. Finally, Section 5.5 explains how FlexStream ensures fault tolerance during node failures.

5.1 Rescaling Trigger Metrics

Designing rescaling trigger metrics for hardware offloading parallelization strategy presents several key challenges. First, detecting load imbalances early enough to prevent significant performance degradation, while avoiding overreactions to short-lived fluctuations, requires precise tuning. Second, developing fine-grained metrics that accurately capture processing load in the context of stream processing semantics is non-trivial.

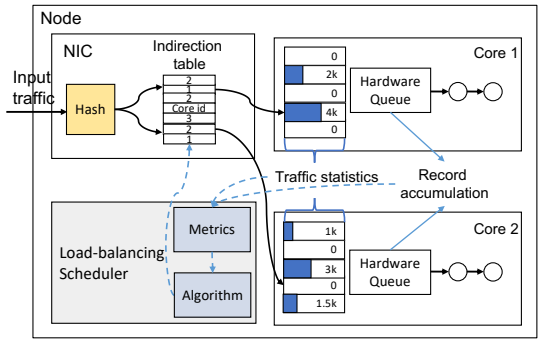


Fig. 4. Load-balancing scheduler overview.

Figure 4 provides an overview of the load-balancing scheduler in FlexStream. By offloading the data re-partitioning process to the NIC, the variation in traffic directed to different cores reflects the degree of data skewness among different keys in the input dataset. However, since processing capacities differ across cores, relying solely on traffic differences does not fully capture the load imbalance between them. To address this, the record accumulation metric we collect offers a direct measure of the disparity between a core's processing capacity and the input rate of its data. When record accumulation occurs at certain cores, we integrate this with the packet count of the input data to perform traffic redistribution, thereby achieving load balancing.

Our approach requires no modifications to the NIC itself. Instead, it leverages standard NIC functionality, specifically the RSS hash embedded in packet metadata. Each core monitors the number of packets received per connection, maintaining a table of counters sized according to the number of connections. The lower bits of the hash serve as the index to this table, with the corresponding counter incremented for each packet received. Redistribution is accomplished by reprogramming the RSS indirect table via the NIC's standard API.

Compared to the metrics employed by previous SPEs, our metrics offer finer granularity and eliminate the need to wait for records to accumulate beyond a specific threshold. Furthermore, the data statistics collected via the NIC provide more precise information for state migration, enabling faster, more accurate, and finer-grained load balancing.

5.2 Load Balancing Among Active Cores

Problem. We formalize the problem of balancing the load among CPU cores as follows: K is the set of all cores and N is the set of connections. M_i is the current load of underloaded cores and zero for overloaded cores. L_i is the fractional load of each connection N_i, A_i represents the record accumulation rate for each connection and M is the average load that each core should reach. $T_{i,j}$ is the assignment matrix of core i to connection j and $T_{i,j}^{old}$ a copy of the current assignment matrix, thus one can formulate an optimization problem that migrates the imbalance to achieve the best spreading of the load as follows:

$$\min \quad \sum_{i \in K} \left(\alpha_1 (T * L^\top)_i + \alpha_2 (T * A^\top)_i - M_i\right)^2 + \beta * \sum_{i \in K, j \in N} |T_{i,j} - T_{i,j}^{old}|$$

$$\text{s.t.} \quad \sum_{j \in N} T_{i,j} = 1 \quad \forall i \in K, \tag{1}$$

$$T_{i,j} \in \{0, 1\} \quad \forall i \in K, j \in N \tag{2}$$

The first constraint ensures that a connections is assigned to one and only one core, while the second makes the problem binary, as we cannot migrate parts of a connection to different cores. The minimization is done using a multi-objective function. The first term of the objective function minimizes the resulting squared load imbalance by applying the assignment described by T, considering both traffic load and record accumulation, the problem is similar to multi-way number partitioning [29]. The second term minimizes the number of connection transfers. Each migration has a cost, that should be reduced as much as possible. The parameters α_1 and α_2 control the relative importance of traffic statistics versus record accumulation metrics (with $\alpha_1 + \alpha_2 = 1$). The scalarization of the two objectives is done with the β factor. The selection of weighting parameters α_1, α_2, and β is based on system characteristics and experimental validation, which can be adjusted according to your specific experimental environment.

Algorithm. We adapted the heuristic algorithm from RSS++ [5], which uses an iterative approach to solve the multi-way partitioning problem. The algorithm begins by identifying overloaded and underloaded cores, then strategically moves the heaviest connection from overloaded cores to underloaded ones. It employs multiple iterations to find an optimal balance point, avoiding both insufficient and excessive migrations. Our key modification to this algorithm is the treatment of migration units. Instead of migrating individual connections as in the original algorithm, FlexStream migrates all connections associated with the same key as a single atomic unit. This approach guarantees that stream semantics are preserved during rebalancing, as related data remains processed by the same core. Additionally, we incorporated record accumulation metrics alongside traffic statistics to better account for varying processing capacities across cores. The convergence speed is exceptionally fast, with an average completion time of approximately 25 microseconds, and the algorithm effectively balances the load across cores. By preserving stream semantics and efficiently redistributing workloads, FlexStream achieves both correctness and high performance in load balancing.

5.3 Lock-Free State Backend

Implementing the lock-free state backend in FlexStream presents significant challenges, as it must simultaneously address two key requirements: (1) enabling concurrent state access during state migration without performance degradation, and (2) ensuring that state access during unrescaling period remains unaffected while addressing the first requirement.

When data skew occurs, we modify the RSS indirection table to redistribute traffic as detailed in Section 5.2. This creates a scenario where multiple cores may simultaneously access the same state as new records are directed to different cores while original cores continue processing accumulated records. Our lock-free design allows threads to process state for the same key concurrently without blocking, illustrated using the window operator example.

As depicted in Figure 5(a), our state backend maintains locally computed results for each thread, with incremental updates performed when a thread receives a watermark signal triggering window computation. The architecture comprises three key components: (1) **Thread-Local State:** each thread maintains its own local state, reflecting the current progress of that thread's processing. (2) **RCU-Based Boolean Array:** we employ a boolean array maintained via an RCU-based mechanism [34], where each index corresponds to a unique window ID for each key's window. The boolean value indicates whether that window is currently undergoing state migration. (3) **Atomic Integer Array:** This array tracks the number of threads involved in a given window that have completed their computations. Only the final thread to modify a window generates the conclusive window aggregate, ensuring correctness during migration.

In our architecture, worker threads primarily perform read/write operations on their local state, read operations on the boolean array, and atomic operations on the integer array. Write operations

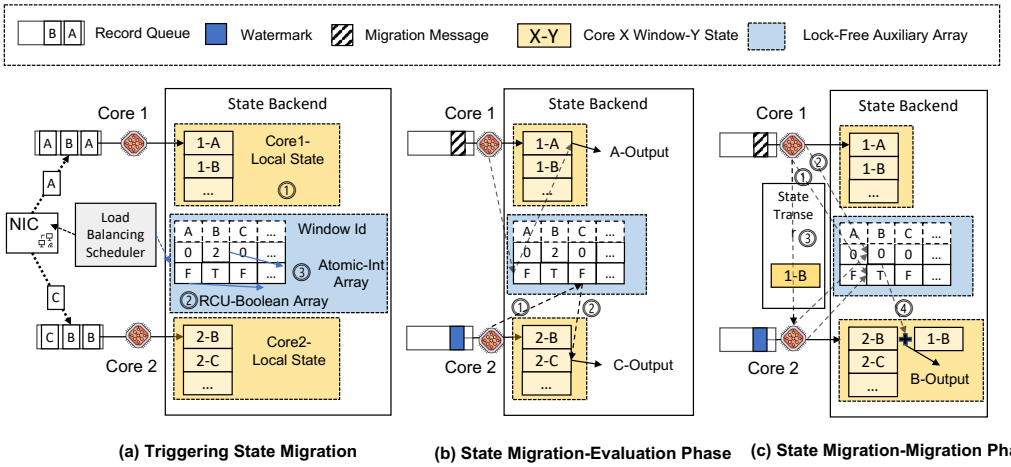


Fig. 5. State migration mechanisms: (a) Lock-free state backend overview and how to trigger state migration; (b),(c) Two phases of state migration protocol.

on the boolean array occur only during migration initialization by the load balancing scheduler. The RCU mechanism ensures minimal overhead for read operations, avoiding expensive synchronization primitives in the critical path.

This design achieves near lock-free access throughout the operation of the SPE, significantly enhancing efficiency compared to traditional lock-based approaches. By eliminating contention and blocking operations during both normal processing and migration scenarios, our state backend provides substantial performance improvements, particularly under high-throughput conditions and during dynamic rescaling.

5.4 State Migration Protocol

To maintain stream processing semantics during rescaling, we implement an efficient state migration protocol. Figure 5 shows a scenario where core 1 is overloaded and requires migrating key B's data to core 2. In this example, both watermarks and migration messages can trigger computation results. The migration process follows this protocol:

(1) **Evaluation Phase (Figure 5(b)):** When a thread processes a special record triggering a Window Aggregate, it first determines the Window ID to be processed and examines the corresponding value in the boolean array. If no state migration is required, it directly computes the Window Aggregate result. Otherwise, it proceeds to the migration phase.
(2) **Migration Phase (Figure 5(c)):** The thread employs a Fetch-and-Add (FAA) atomic instruction to determine whether it is the last thread affecting the result. If the retrieved value is non-zero—indicating other threads are still processing the window—the current thread forwards its state to those threads. If the value is zero, the thread is the final one processing the result and combines its local state with any received states from other threads to compute and emit the final result.

Correctness guarantee. This protocol ensures system correctness for state updates during both normal operation and rescaling scenarios. Stream processing progress is governed by watermarks [4]. During a Migration Stage where state associated with key k is redistributed among active cores, each incoming record influences the final result exactly once. Let t_1 denote the timestamp

when migration begins and t_2 the timestamp when it concludes. Prior to t_1, records for k are sent exclusively to the pre-migration thread. After t_2, they are directed solely to the post-migration thread. During the interval $[t_1, t_2]$, records may be processed by both threads, but they affect the final result only once, thereby preserving the exactly-once semantics of stream processing.

Expressiveness guarantee. In the window aggregate example discussed above, states satisfy distributive and commutative properties, allowing the state migration protocol to execute efficiently. However, many use cases, such as complex event processing, involve order-sensitive operators that do not meet these properties. The state migration protocol can still execute correctly in these cases, but requires additional storage space. For example, when transferring states of order-sensitive operators from core 1 to core 2, core 1 does not compute partial results for incoming events. Instead, it stores the raw records in their original order. During inter-thread synchronization, core 1 transfers these raw records as state to core 2. Core 2 then replays these events to compute the final results. The TCP connection ensures event ordering between core 1 and core 2. Therefore, our state migration protocol supports the implementation of complex queries.

Our survey of numerous related works [18, 21, 22, 27, 40, 43, 50] and real-world use cases [8, 9, 38, 39, 42, 46, 49] shows that most queries in mainstream benchmarks exhibit key-partitionable, commutative, and mergeable properties. These queries span applications including auction systems, time-series analysis, network monitoring, and financial risk control, demonstrating the broad applicability of our approach.

5.5 Fault Tolerance Guarantees

In SPEs, fault tolerance is critical for ensuring elasticity. FlexStream inherits fault-tolerance guarantees from the underlying SPEs. However, our hardware offloading parallelization strategy requires minor modifications to the host SPE's fault tolerance mechanisms.

For example, Flink provides fault tolerance through distributed state snapshots. This approach captures persistent operator states at specific intervals. Upon failure, the system restores operator states from the latest checkpoint. A common approach for consistent distributed state snapshots is the Chandy-Lamport algorithm [7] or its variants. In Flink, operators persist their states upon receiving barriers from software queues, enabling globally consistent snapshots. However, FlexStream eliminates software-based queues, requiring a different approach. In FlexStream, processing threads wait for checkpoint barriers from all network input connections before persisting their states. FlexStream also records additional information in snapshots, including network connection details for each processing thread and RSS indirection table information. This enables reconstruction of the key-to-core hash mapping during node failure recovery. These modifications guarantee consistent and reliable system behavior while maintaining data consistency throughout the job lifecycle.

6 Evaluation

In this section, we experimentally validate FlexStream's system design through comprehensive end-to-end experiments and micro-benchmarks. First, we describe our experimental setup in Sec. 6.1. In Sec. 6.2, we compare FlexStream against Grizzly and LightSaber and Apache Flink on end-to-end queries. Second, we compare FlexStream against Meces, a current SOTA state migration approach, to evaluate latency performance (Sec. 6.3). Finally, we conduct a detailed performance analysis to identify the sources of FlexStream's performance advantages (Sec. 6.4).

6.1 Experimental Setup

6.1.1 Hardware and Software. We run the experiments on an inhouse, 16-node cluster. Each node is equipped with a 24-core, 2.40 Ghz Intel Xeon Gold 6240R CPU, 32 GB of main-memory, and a single-port Mellanox Connect-X6 EDR 100Gb/s NIC. Each NIC is connected to a 100 Gbits InfiniBand

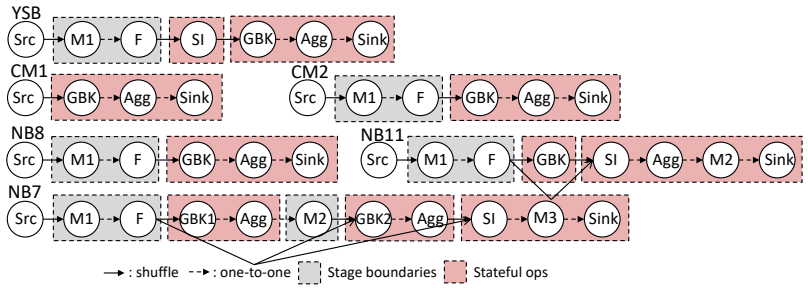


Fig. 6. A simplified application DAG of stream queries used in our evaluation. M and F are map and filter operators, GbK is a stateful group-by-key operator for incremental aggregation on windows, and SI is a non-mergeable stateful operator for the join operation.

EDR switch by Mellanox and has RSS support. Every node runs Ubuntu Server 22.04. We disable hyper-threading and pin each thread to a dedicated core. Unless stated otherwise, every hardware component is configured with factory settings.

Software Configuration. We evaluate four systems as Systems under Test (SUTs) - FlexStream, Grizzly [22], LightSaber [43], and Apache Flink 1.20. We select Apache Flink as a representative of SPE that employ data re-partitioning parallelization strategy, while Grizzly and LightSaber represent engines that use late merge parallelization strategy. We follow Flink's configuration guidelines [19]. On each node, we allocate half of the cores for processing and the other half for network I/O. FlexStream is compiled with O3 compiler optimization and native CPU support using gcc 9.3 and configured to use all physical cores.

6.1.2 Workloads. To experimentally validate our system design, we select the *Yahoo! Streaming Benchmark* (YSB) [8, 9], the *NEXMark benchmark suite* (NB) [46], and the *Cluster Monitoring* (CM) benchmark [49]. We choose YSB and NB, as they are commonly-used benchmarks that represent real-world scenarios [13, 52]. We select CM, as it is based on a publicly-available, real-world dataset provided by Google. Figure 6 illustrates the simplified original DAG of stream queries used for our evaluation. In the workloads, window sizes and slides are measured in Milliseconds.

YSB. The YSB assesses the performance of windowed aggregation operators. A record is 78-bytes large and stores an 8-bytes primary key and an 8-bytes creation timestamp.

NB. The NB simulates a real-time auction platform with three logical streams: an auction stream, a bid stream, and a seller event stream. Records are 206 (seller), 269 (auction), and 32 (bid) bytes large. Each record stores an 8-bytes primary key and an 8-bytes creation timestamp.

CM. The CM benchmark executes a stateful aggregation over a stream of timestamped records containing the traces from a 12.5K-nodes cluster at Google. Each tuple is 64 bytes large and stores an 8-bytes primary key and an 8-bytes timestamp.

6.2 End-to-end Performance

In this section, we quantify FlexStream's throughput advantages over existing stream processing systems. Each experiment is repeated multiple times and average throughput is reported.

6.2.1 Methodology. We generate data in real-time and ingest it through network, accurately reflecting production-grade streaming environments. During execution, we measure query processing throughput, defined as the number of records processed per second by the SUTs.

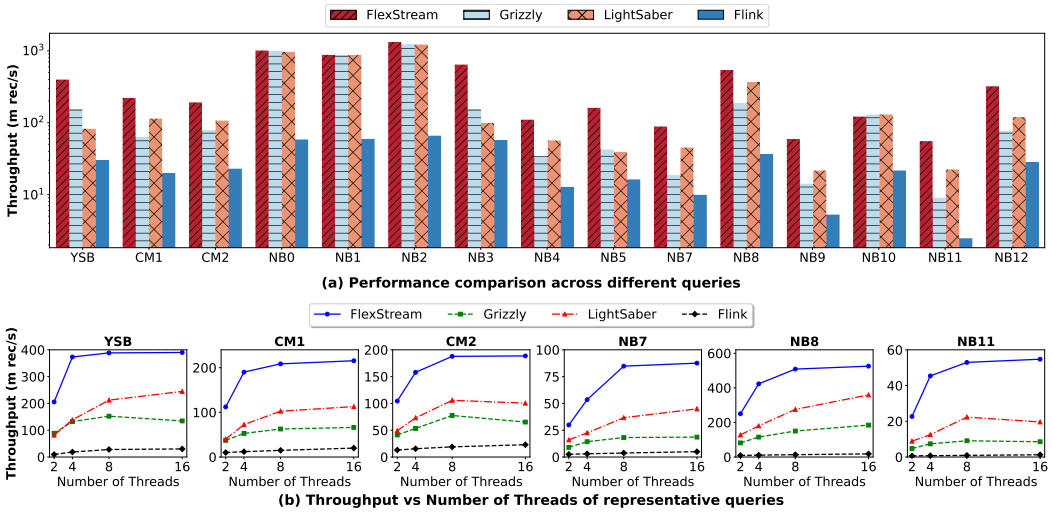


Fig. 7. Throughput for YSB, CM, and NB (in records/s) of Flink, Grizzly, LightSaber, and Flexstream on single node.

For scaling-up performance evaluation, all operators are deployed on a single node. Data shuffling between operators is performed using the node's NIC. Each SUT is configured to utilize 16 threads per node to ensure consistent resource allocation across comparative systems.

For scale-out experiments, we deploy the streaming job across up to 16 nodes. For NEXMark, we selected queries NB7, NB8, and NB11 because they collectively represent all types of stateful operators in the benchmark suite.

6.2.2 Scaling-up Performance. FlexStream consistently outperforms all baseline systems across most queries, as shown in Figure 7(a). The exceptions are queries NB0-2 and NB10, where FlexStream shows comparable performance to the baselines. This is because queries NB0-2 involve only stateless operations, while NB10 primarily tests system output performance without requiring data shuffling. Since our optimization strategies target stateful operations and data movement bottlenecks, they do not provide performance improvements for these specific query types.

Figure 7(b) further analyzes how throughput varies with thread number for individual queries. We focus our analysis on NB7-8, and NB11 because they comprehensively cover all scenarios in the NEXMark benchmark. FlexStream achieves up to 1.59×, 2.89×, and 12.91× higher throughput on the YSB compared to LightSaber, Grizzly, and Flink, respectively. On the CM dataset, FlexStream demonstrates even more substantial gains, with throughput increases of up to 1.90×, 3.24×, and 10.79× over the same systems. For queries NB0-12, FlexStream delivers up to 3.28×, 6.40×, and 43.84× higher throughput.

These results confirm that FlexStream's hardware offloading parallelization strategy significantly enhances SPE throughput. This superior performance stems from two key architectural advantages. First, by offloading data repartitioning to hardware, FlexStream eliminates software-mediated queuing bottlenecks that constrain traditional systems. Second, both LightSaber and Grizzly rely on fine-grained synchronization primitives such as atomic operations and concurrent data structures to coordinate worker threads. These synchronization mechanisms limit their ability to fully utilize available network bandwidth. In contrast, FlexStream achieves near-linear scalability up to four threads. Beyond this point, scalability benefits diminish as query execution transitions from being

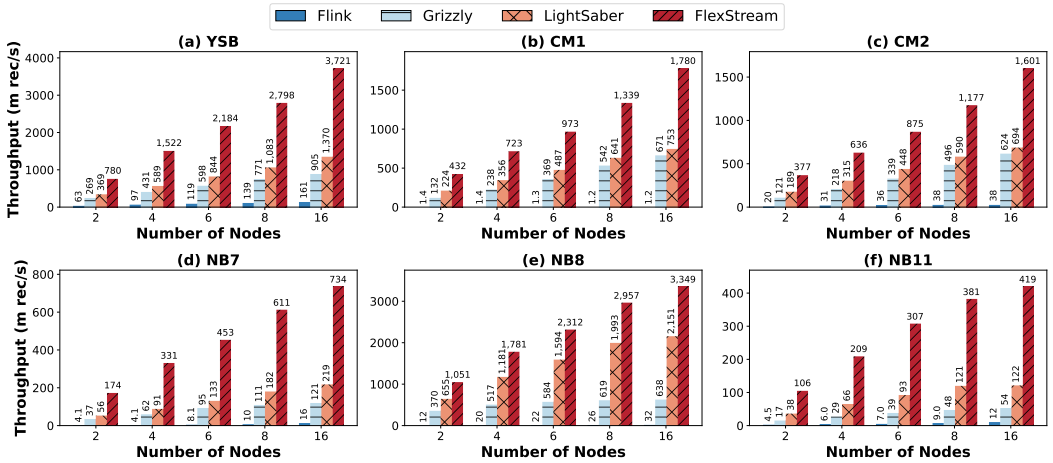


Fig. 8. Throughput for YSB, CM, and NB (in records/s) of Flink, Grizzly, and Flexstream on 2, 4, 6, 8, and 16 nodes.

compute-bound to network bandwidth-bound. We provide a detailed analysis of these network bandwidth limitations in subsequent sections.

6.2.3 Scaling-out Performance. As shown in Figure 8, FlexStream exhibits near-linear scalability across all queries, consistently outperforming all systems under test. On YSB, FlexStream achieves up to 2.71×, 4.11×, and 23.11× higher throughput compared to LightSaber, Grizzly, and Flink, respectively. On the CM dataset, FlexStream delivers up to 2.36×, 2.65×, and 42.13× higher throughput relative to the same systems. The performance advantage is particularly pronounced for complex analytical queries, with FlexStream achieving throughput improvements of up to 3.43×, 7.90×, and 45.88× for queries NB7, NB8, and NB11, respectively. These results demonstrate FlexStream's potential for deployment in large-scale production environments where processing demands fluctuate rapidly and unpredictably.

We excluded Flink from detailed comparative analysis in this section due to its partitioning approach, which incurs prohibitive runtime overhead at scale. For Grizzly and LightSaber, they encounter performance bottlenecks when implementing data shuffling operators across nodes. Unlike single-node deployments where a thread can process an entire stream processing task independently, distributed processing environments frequently require stateful operators to access data from multiple nodes, introducing additional coordination overhead that constrains scalability. In contrast, FlexStream's parallelization strategy, which optimizes data partitioning during the network transmission phase, inherently aligns with the operator distribution requirements of scale-out SPEs.

Our experimental analysis reveals that FlexStream maintains strong scalability across 16-node cluster configurations, sustaining significant performance advantages over baseline systems. For most queries, FlexStream scales effectively with increased cluster size. However, the scalability of queries NB7 and NB8 is constrained by the limited parallelism of source operators in these benchmark queries. The restricted data ingestion capability from external sources prevents full utilization of subsequent nodes, which limits the scalability potential of the distributed setup.

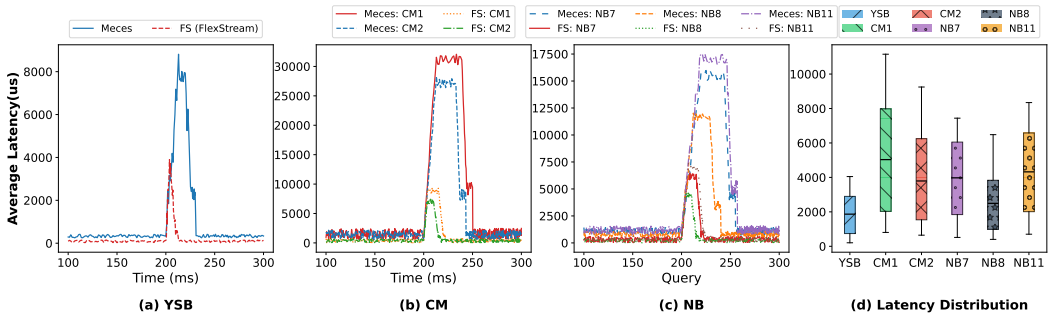


Fig. 9. End-to-end latency of YSB, CM, and NB.

6.3 End-to-end Latency

This section evaluates the latency characteristics of FlexStream during dynamic rescaling. We first outline our methodology in Sec. 6.3.1, then benchmark FlexStream's improvements over state-of-the-art approaches in Sec. 6.3.2.

6.3.1 Methodology. We select Meces [23], a current state-of-the-art method, as our baseline. To isolate the performance contributions of core mechanisms (e.g., dynamic scaling), our experiments used fixed empirical values to avoid scheduler parameter interference. In our experiment, we chose to set the algorithm parameters as $\alpha_1 = 0.3, \alpha_2 = 0.7, \beta = 1$. All experiments described below are conducted using a single-node configuration. We first generate stable input streams where the input rate is stable and does not fluctuate. At a specific point (e.g., $t = 200ms$ in our evaluation), we dynamically alter key distribution skewness to simulate sudden load imbalances. We focus on two metrics in measurement during rescaling:

(1) Latency: To evaluate the end-to-end latency of SPE, we configure the stream generators to periodically insert marker records. We denote latency as the time difference between these markers entering and leaving the SPE.

(2) State Migration Time: Duration between skew detection and completion of state redistribution. This metric quantifies the system's elasticity in adapting to workload shifts.

6.3.2 Latency Performance during Rescaling. As depicted in Figure 9(a)-(c), FlexStream effectively reduces latency peaks during migration and shortens the duration of the state migration process. We use average latency to clearly reflect overall performance improvements. In the YSB benchmark, FlexStream achieves a 61.8% reduction in latency peaks and a 66.8% decrease in state migration time. For the CM dataset, it delivers a 71.9% reduction in latency peaks and a 54.5% decrease in migration duration. Similarly, in the NB dataset, FlexStream lowers latency peaks by 57.1% and reduces migration time by 64.0%. Moreover, Figure 9(d) shows the latency distribution for each query, including P95 and P99 percentiles. The results demonstrate that our migration strategy achieves superior tail latency performance compared to the baselines while maintaining consistently low latency levels.

The enhanced performance can be attributed to two key architectural innovations. First, our proactive metrics collection and analysis framework directly reducing processing latency during rescaling and substantially shortening the trigger phase. Second, our state migration protocol leverages incremental state updates to optimize the migration process.In contrast, Meces exhibits frequent inter-thread contention, resulting in suboptimal performance characterized by higher latency spikes and extended migration periods. In Section 6.4.3, we provide a detailed analysis

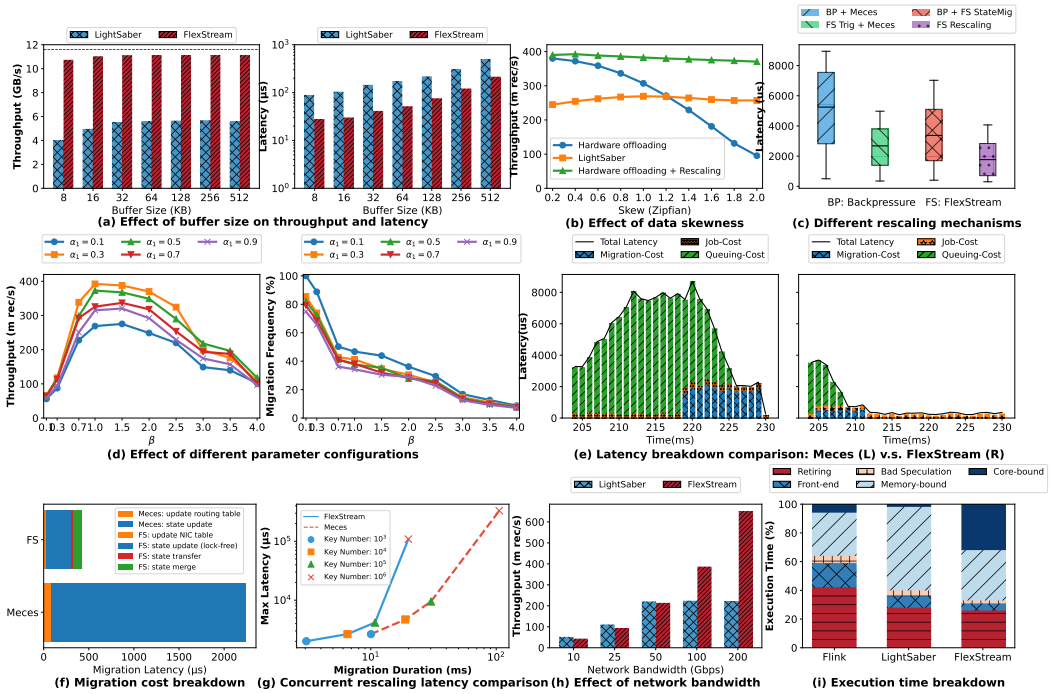


Fig. 10. Drill-down analysis of FlexStream, LightSaber and Flink on YSB.

of the sources of these latency improvements and examine latency behavior during concurrent rescaling operations.

6.4 Performance Drill-down

This section analyzes the factors contributing to FlexStream's performance improvements from both workload-related and hardware-related perspectives. Workload-related aspects (Sections 6.4.1, 6.4.2, and 6.4.3) include analysis of data characteristics, parameter sensitivity, and rescaling dynamics. Hardware-related aspects (Section 6.4.4) analyze the impact of network bandwidth on throughput and use micro-architectural analysis to quantify execution behavior. All experiments use the YSB query on a single-node configuration for direct comparison with LightSaber.

6.4.1 Data Characteristics. This subsection examines how workload characteristics, specifically event batching policy and data skewness, impact FlexStream's performance.

Event Batching Policy. As shown in Figure 10(a), we test the impact of different buffer sizes on throughput and latency. The buffer size represents the number of events in a batch during TCP transmission between nodes. FlexStream nearly saturates the theoretical bandwidth limit of the NIC using a 32 KB buffer size, while LightSaber utilizes only up to 48% of the available bandwidth due to synchronization overheads. Furthermore, the 32 KB buffer size configuration has minimal impact on overall SPE processing latency. Smaller buffer sizes result in more frequent I/O operations that fail to fully utilize network bandwidth, while larger buffer sizes significantly increase latency.

Data Skewness. As shown in Figure 10(b), using partitioning keys generated from a Zipfian distribution with skewness parameter $z = 0.2 - 2$, we observe that LightSaber's round-robin partitioning strategy remains largely insensitive to data skewness. However, a parallelization

strategy relying solely on hardware offloading suffers significant throughput degradation (up to 25.1%) under data skew. Our rescaling mechanism effectively mitigates this impact, enhancing FlexStream's robustness and enabling it to handle data skew efficiently.

6.4.2 Parameter Sensitivity Analysis. We evaluate three key parameters (α_1, α_2 and β) defined in our rescaling algorithm in Sec. 5.2. Parameter α_1 controls the system's sensitivity to accumulating burst events, while parameter α_2 determines the scheduler's preference for long-term load trends versus short-term fluctuations. Parameter β represents the ease of state migration. Values that are too high or too low for β lead to performance degradation.

As shown in Figure 10(d), we vary the values of β and α_1 (where $\alpha_1 + \alpha_2 = 1$) to observe changes in overall system throughput and state migration frequency. The results show that the system achieves peak throughput at $\alpha_1 = 0.3, \alpha_2 = 0.7, \beta = 1.0$, while maintaining reasonable migration frequency control. Additionally, when α_1 ranges between $0.3 - 0.7$ and β ranges between $0.7 - 2.0$, the system maintains good overall throughput performance. This indicates the robustness of our system under suboptimal parameter settings. For specific types of stream processing tasks, such as high-burst workload applications or stable workload applications, optimal performance can be achieved through fine-tuning these parameters experimentally.

6.4.3 Rescaling Dynamics. This subsection evaluates FlexStream's rescaling latency and provides a detailed breakdown of migration cost components, as well as testing the impact of different concurrency levels on state migration.

Rescaling Latency. Figure 10(c) evaluates various combinations of rescaling strategies and state migration mechanisms, confirming that FlexStream consistently outperforms prior rescaling approaches across key latency metrics. The detailed breakdown in Figure 10(e) reveals the underlying factors contributing to FlexStream's superior performance. Our proactive trigger metrics replace conventional backpressure mechanisms, significantly reducing the number of queued events during rescaling operations, which directly minimizes queuing-cost overhead. Additionally, our state migration mechanism substantially reduces inter-thread state conflicts. As a result, FlexStream achieves a flatter latency profile during rescaling operations and completes state migrations in significantly less time compared to alternative approaches.

Migration Cost Breakdown and Concurrency Testing. As shown in Figure 10(f), migration cost consists of four main components: state update, NIC table updates, state transfer, and state synchronization. Our optimization primarily stems from replacing Meces' thread-conflicted state updates with lock-free state updates (64.1%) combined with state merge operations (21.4%). State transfer overhead is minimal since it only involves passing pointers and ownership between threads. Furthermore, we evaluated migration cost overhead under different key numbers. As the number of keys increases, more keys require migration, resulting in higher concurrent rescaling parallelism. Figure 10(g) shows that FlexStream outperforms Meces across all key numbers, with optimization benefits increasing as the key count grows. With 10^6 keys, FlexStream lowers latency peaks by 72.2% and reduces migration time by 82.3%.

6.4.4 Analysis of Hardware-Related Aspects. We analyze hardware-related performance factors from two perspectives: network bandwidth limitations and hardware performance counter analysis.

Network Utilization. As shown in Figure 10(h), we conduct experiments using a 200 Gbps network card and test performance impact by limiting network bandwidth rates. The experimental results demonstrate that FlexStream's throughput increases with network card bandwidth, allowing it to more effectively utilize available network bandwidth. This confirms that FlexStream's throughput in Section 6.2.2 is indeed limited by network bandwidth. In contrast, LightSaber can

only utilize up to 50 Gbps of network bandwidth. At lower bandwidths (10/25 Gbps), FlexStream underperforms LightSaber because our approach relies on network cards for hardware offloading.

Micro-architectural Analysis. Our micro-architectural behavior analysis is guided by Intel's optimization reference [11], which identifies stalls in the CPU pipeline. The metrics we use are categorized into five types: (i) front-end stalls, caused by instruction fetch bottlenecks; (ii) core-bound stalls, due to constraints in execution units; (iii) memory-bound stalls, resulting from memory subsystem delays; (iv) bad speculation, stemming from branch mispredictions; and (v) retiring cycles, representing successful instruction execution.

As depicted in Figure 10(i), Flink exhibits up to 15% front-end stalls due to its large instruction footprint, indicating inefficiencies in instruction delivery. LightSaber's profile is predominantly memory-bound, driven by prolonged access to concurrent state data structures, which introduces inter-thread contention and significant memory subsystem delays. In contrast, FlexStream's execution architecture maximizes network bandwidth utilization and eliminates state conflicts between threads during execution. Consequently, FlexStream's profile is more core-bound, relying on pause instructions during network input processing rather than concurrent data access. This reduces the prominence of memory-bound stalls, leading to enhanced system performance and optimization.

7 Conclusion

This paper presents FlexStream, a novel SPE that achieves exceptional performance through hardware offloading techniques and an efficient state migration mechanism. FlexStream demonstrates linear scalability and sub-second elasticity even under highly dynamic workloads, addressing critical limitations in current stream processing systems. Our comprehensive evaluation shows that FlexStream delivers up to 3.14× higher throughput compared to state-of-the-art systems like LightSaber. During rescaling operations, FlexStream reduces latency peaks by 61.8% and decreases state migration time by 66.8%, enabling seamless adaptation to changing workload conditions without compromising performance.

The design principles demonstrated in FlexStream highlight the importance of co-designing stream processing systems with modern hardware capabilities. Our hardware offloading strategy provides key-level operator support, and future work could explore leveraging more sophisticated hardware such as smartNICs to enable support for more complex operations. Additionally, integrating AI-driven solutions into our elasticity design presents promising research opportunities. For example, reinforcement learning or deep learning models could be employed to anticipate data skew patterns and proactively adjust the RSS indirection table to minimize migration costs. We believe the hardware-software co-design approach pioneered in FlexStream can be extended to other aspects of stream processing, potentially enabling even greater performance improvements as hardware capabilities continue to evolve.

Acknowledgments

We thank our shepherd and all the reviewers for their valuable comments and suggestions. This work is supported in part by the National Key Research and Development Program of China (2022YFB2404200), and the Beijing National Research Center for Information Science and Technology.

References

[1] 2020. Twitter sentiment analysis: A tale of stream processing. https://towardsdatascience.com/twitter-sentiment-analysis-a-tale-of-stream-processing-8fd92e19a6e6/

[2] 2021. flink-rescalable-state. https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/

[3] 2024. Receive Side Scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[4] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. *Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow.* Technical Report. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States).

[5] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, 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*. 318–333.

[6] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 725–736.

[7] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[8] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, et al. 2015. Benchmarking streaming computation engines at yahoo. *Tech. Rep.* (2015).

[9] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

[10] Rebecca L Collins and Luca P Carloni. 2009. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*. 205–214.

[11] Intel Corporation. 2012. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation. Volume 3: System Programming Guide.

[12] Debezium. 2019. Building audit logs with change data capture and stream processing. https://debezium.io/blog/2019/10/01/audit-logs-with-change-data-capture-and-stream-processing/

[13] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[14] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking stateful stream processing with rdma. In *Proceedings of the 2022 International Conference on Management of Data*. 1078–1092.

[15] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[16] Exastax. 2017. Real-time stream processing for internet of things. https://medium.com/@exastax/real-time-stream-processing-for-internet-of-things-24ac529f75a3/

[17] Apache Flink. 2015. https://flink.apache.org/

[18] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[19] Apache Foundation. 2015. Apache Flink Configuration. https://ci.apache.org/projects/flink/flink-docs-master

[20] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541.

[21] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. 2019. {EdgeWise}: A better stream processing engine for the edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 929–946.

[22] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2487–2503.

[23] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 539–556.

[24] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

[25] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[26] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316.

[27] Anand Jayarajan, Wei Zhao, Yudi Sun, and Gennady Pekhimenko. 2023. Tilt: A time-centric approach for stream query optimization and parallelization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 818–832.
[28] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 783–798.
[29] Richard E Korf. 2009. Multi-way number partitioning.. In *IJCAI*, Vol. 9. 538–543.
[30] T Kubernetes. 2019. Kubernetes. *Kubernetes. Retrieved May* 24 (2019), 2019.
[31] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*. 239–250.
[32] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.
[33] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. {StreamScope}: Continuous Reliable Distributed Processing of Big Data Streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 439–453.
[34] Paul E McKenney and Jonathan Walpole. 2007. What is RCU, fundamentally? *Linux Weekly News (LWN. net)* (2007).
[35] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2019. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–181.
[36] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2017. {StreamBox}: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 617–629.
[37] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.
[38] Jiapu Pan and Willis J Tompkins. 2007. A real-time QRS detection algorithm. *IEEE transactions on biomedical engineering* 3 (2007), 230–236.
[39] Robert B Randall and Jerome Antoni. 2011. Rolling element bearing diagnostics—A tutorial. *Mechanical systems and signal processing* 25, 2 (2011), 485–520.
[40] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 301–314.
[41] Apache Spark Streaming. 2013. https://spark.apache.org/streaming/
[42] Adrian Ţăran-Moroşan. 2011. The relative strength index revisited. *African Journal of Business Management* 5, 14 (2011), 5855–5862.
[43] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. Lightsaber: Efficient window aggregation on multi-core processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2505–2521.
[44] Georgios Theodorakis, Fotios Kounelis, Peter Pietzuch, and Holger Pirk. 2021. Scabbard: Single-node fault-tolerant stream processing. *Proceedings of the VLDB Endowment* 15, 2 (2021), 361–374.
[45] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.
[46] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. Nexmark–a benchmark for queries over data streams (draft). *Technical report* (2008).
[47] Virtuslab. 2020. Preventing fraud and fighting account takeovers with kafka streams. https://www.confluent.io/blog/fraud-prevention-and-threat-detection-with-kafka-streams/
[48] Li Wang, Tom ZJ Fu, Richard TB Ma, Marianne Winslett, and Zhenjie Zhang. 2019. Elasticutor: Rapid elasticity for realtime stateful stream processing. In *Proceedings of the 2019 International Conference on Management of Data*. 573–588.
[49] John Wilkes. 2011. More Google Cluster Data. https://github.com/google/clusterdata
[50] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 723–734.
[51] Seokwoo Yang, Siwoon Son, Mi-Jung Choi, and Yang-Sae Moon. 2019. Performance improvement of apache storm using infiniband rdma. *The Journal of Supercomputing* 75 (2019), 6804–6830.

[52] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment* 12, 5 (2019), 516–530.

[53] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 international conference on management of data*. 705–722.

[54] Ziyu Zhang, Zitan Liu, Qingcai Jiang, Junshi Chen, and Hong An. 2021. RDMA-based apache storm for high-performance stream data processing. *International Journal of Parallel Programming* 49, 5 (2021), 671–684.

Received April 2025; revised July 2025; accepted August 2025