RDMA-Based Apache Storm for High-Performance Stream Data Processing

Ziyu Zhang, Zitan Liu, Qingcai Jiang, Zheng Wu, Junshi Chen, Hong An

RDMA-Based Apache Storm for high-performance stream data processing

Ziyu Zhang[✉⋆], Zitan Liu, Qingcai Jiang, Zheng Wu,
Junshi Chen, and Hong An[✉]

University of Science and Technology of China, Hefei, China
{zzymm,jauntyliu,jqc,zhengwu,cjuns}@mail.ustc.edu.cn,han@ustc.edu.cn

Abstract. Apache Storm is a scalable fault-tolerant distributed real-time stream-processing framework widely used in big data applications. For distributed data-sensitive applications, low-latency, high-throughput communication modules have a critical impact on overall system performance. Apache Storm currently uses Netty as its communication component, an asynchronous server/client framework based on TCP/IP protocol stack. The TCP/IP protocol stack has inherent performance flaws due to frequent memory copying and context switching. The Netty component not only limits the performance of the Storm but also increases the CPU load in the IPoIB (IP over InfiniBand) communication mode. In this paper, we introduce two new implementations for Apache Storm communication components with the help of RDMA technology. The performance evaluation on Mellanox QDR Cards (40 Gbps) shows that our implementations can achieve speedup up to 5x compared with IPoIB and 10x with 1 Gigabit Ethernet. Our implementations also significantly reduce the CPU load and increase the throughput of the system.

Keywords: Apache Storm, RDMA, InfiniBand, Stream-processing framework, Cloud computing, Communication optimization

1 Introduction

With the increase of Internet users and the development of hardware equipment, processing massive amounts of data in real-time has posed a great challenge to system design. Real-time stream processing frameworks such as Apache Storm [1], Apache Spark [2], and Apache Flink [3] have attracted more attention than batch processing frameworks such as Apache Hadoop [4]. Streaming systems must handle high-speed data streams under strict delay constraints. To process massive data streams, modern stream processing frameworks distribute processing over numbers of computing nodes in a cluster. These systems sacrifice single-node performance for the scalability of large clusters and rely on Java Virtual Machine (JVM) for platform independence. Although the JVM

⋆ The work is supported by the National Key Research and Development Program of China (Grants No.2018YFB0204102).

provides a high-level abstraction from the underlying hardware, the processing overhead caused by its data (deserialization) serialization, the dispersion of objects in memory, garbage collection, and frequent context switching caused by inter-process communication reduce the efficiency of data access provided by the computing framework [5, 6]. Therefore, the overall performance of the scalable computing framework built on the JVM is severely limited in terms of throughput and latency.

In general, the most advanced systems focus on optimizing throughput and latency for high-speed data streams. The data stream model is implemented by utilizing a scalable system architecture based on message passing mechanism [7–9]. From the perspective of the development of modern hardware, the potential performance bottleneck of the scalable streaming computing system is that it cannot fully utilize current and emerging hardware trends, such as multi-core processors and high-speed networks [5]. Due to frequent cross-node communication, the Internet has an important impact on the performance of modern distributed data processing systems. Restricted by the traditional communication protocol, the current streaming computing system cannot fully utilize the 40Gbps network [6]. In addition, the huge load caused by the transmission pipeline based on the data flow model severely limits the computing performance of the CPU [10]. Optimize the execution strategy to solve the problem of unbalanced CPU load in the computing system, such as reducing the size of the partition or simply adding more cores to each network connection can only obtain sub-optimal performance [11].

InfiniBand is an industry-standard switched fabric that is designed for high-speed, general-purpose I/O interconnects nodes in clusters [12]. One of the main features of InfiniBand is Remote Direct Memory Access (RDMA), which allows software to remotely read or update memory contents of another remote process without involving either one's operating system [13]. A direct way to run Apache Storm over InfiniBand is to use the IP over InfiniBand (IPoIB). IPoIB wraps Infiniband devices into regular IP based Ethernet cards, making applications transparently migrated into InfiniBand based device provides a useful feature, that is, using InfiniBand devices easily by IP address is like using Ethernet devices. Cross-node data transmission is realized based on event-driven asynchronous I/O library. Although the asynchronous method can effectively reduce the waiting time during the transmission process, the overhead of context switching cannot be ignored. On the other hand, due to the inherent defects of the TCP/IP protocol stack, the system kernel will copy data buffer multiple times during the message sending and receiving process [11], consuming a lot of CPU resources. As the amount of data and the complexity of the calculation topology increase, the IPoIB mode exacerbates the above problems.

In this paper, we experiment on the cluster of high-performance interconnected hardware devices and prove that the inherent defects of the TCP/IP protocol stack used in the communication component are the main reason for the excessive CPU load. Secondly, We use InfiniBand's remote direct Memory access (RDMA) technology to redesign the data transmission pipeline of Apache

Storm.

The rest of this paper is organized as follows. In Section 2, we examine the overall process of Apache Storm inter-worker communication and the overhead introduced by Netty communication component and its underlying TCP/IP protocol stack. Section 3 presents our optimization on Storm's messaging layer and explains how we design and implement the RDMA-based Storm in detail. Section 4 demonstrates the evaluation result of our newly designed Storm. Finally, Section 5 gives our conclusion.

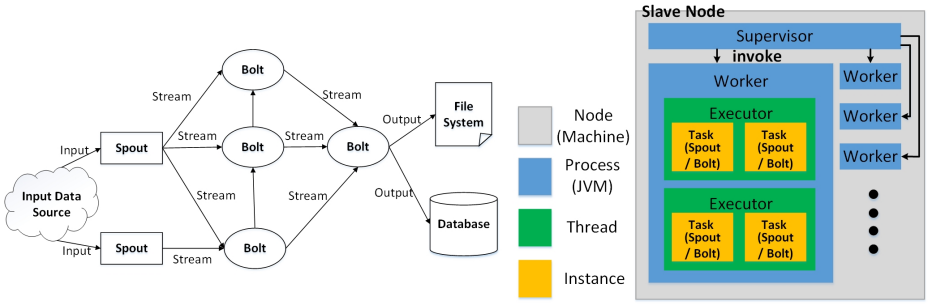

Fig. 1. Computational topology and logical structure

2 Background and Motivation

2.1 Parallel processing of data streams

Parallelism is key to the fast streaming of Storm. Storm handles user-defined computational topology in parallel by creating a large number of Spouts and Bolts as shown in Fig. 1. Spouts get the data stream from the data source, convert the data stream into the smallest data structure tuple, then send them to one or more Bolts. Next, Bolts process the tuple, and then send tuple to the next Bolt or database according to the topology user have specified. By instantiating multiple Spouts and Bolts, the process can be executed in parallel [1]. Fig. 1 shows that a node in the cluster runs multiple Worker processes according to the user's definition and the Worker handles user-defined computational topology. It processes tuples by executing Executor and Executor handles or emits tuples by instantiating Spout or Bolt.

2.2 Message processing structure

Apache Storm applies a distributed producer-consumer pattern and a buffer mechanism to handle data transmission among operators (Spout and Bolt) as shown in Fig. 2. Since different operators are distributed on different nodes in the cluster, the partitioning method requires frequent inter-process communication. A complete communication process is as follows:

1. Each Worker process has a receiver thread (listening to the specified port). When new message arrives, the receiver thread puts the tuple from the network layer into its buffer. When the amount of tuple reaches a certain threshold, the receiver thread sends tuple to the corresponding (one or more) Executors' incoming-queues specified by the task number in the tuples.
2. Each Worker process contains multiple Executors, which are the actual components that process the data. The Executor contains a worker thread with an incoming queue and a sender thread with an outgoing queue. The messages emit by receiver thread gets processed in the worker thread, and then put into the outgoing queue ready to be sent.
3. When the tuples in the outgoing queue of the Executor reaches a certain threshold, the sending thread of the executor will get the tuples in the outgoing queue in batches and send them to the transfer queue of the Worker process, which will be sent to the network by the sending thread then.

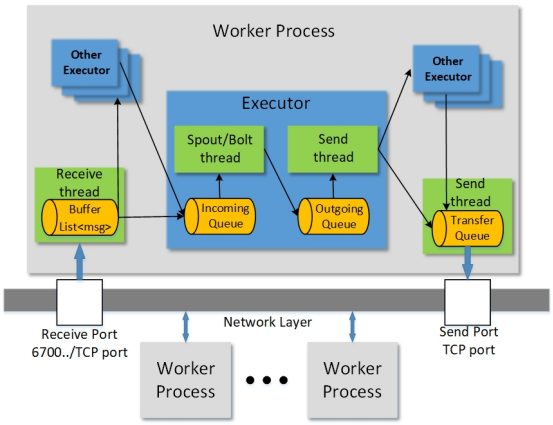

Fig. 2. Communication inside and outside worker processes

2.3 InfiniBand and RDMA

The streaming system on modern hardware must effectively use three key hardware resources to achieve high resource utilization: CPU, main memory, and network [14, 15]. However, Trivedi [6] and Steffen [5] prove that the large load caused by the data transmission pipeline has become the bottleneck of the CPU, and the streaming computing system cannot currently fully utilize the high-performance interconnect. The development of network hardware technology makes the network communication speed may exceed the memory bandwidth in the future. The commonly used Ethernet technology provides 1, 10, 40, or 100 Gbit bandwidth [5]. InfiniBand, an industry-standard switched fabric that is designed for High Performance Computing (HPC) cluster interconnection, on the other side, provides bandwidth comparable or even faster than the main memory and its main feature, Remote Direct Memory Acess (RDMA) allows

software to read or update the remote memory contents without any CPU involvement [12]. This important trend will lead to drastic changes in streaming system design. In particular, the traditional idea of network layer and kernel-user boundary data encapsulation is inconsistent with the future development trend of network technology. Therefore, the speed of using high-performance interconnection networks to transfer data between nodes is greater than or equal to the speed of memory access will bring new challenges to the design of future streaming computing systems [16]. Our work presents a possible solution to the above problem.

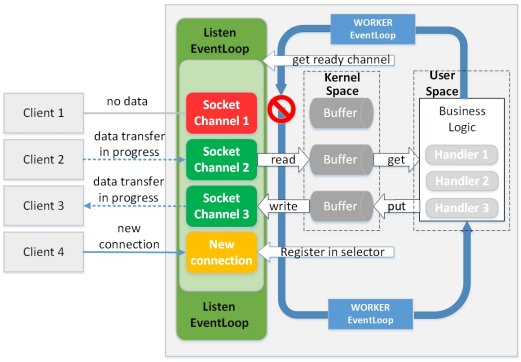

Fig. 3. Netty's NIO communication model

2.4 Analysis

The Storm communication module is implemented by Netty framework currently. Based on NIO (non-blocking IO), Netty uses Reactor as its multiplexing model. During a communication, the server uses a thread pool to receive client connections concurrently as shown in Fig. 3. When the TCP connection gets accepted, the server registers the newly created Socket-Channel (an abstraction of network operations, including basic I/O operations) to a thread in the I/O thread pool. In order to handle service requests delivered to a service handler by one or more inputs concurrently, Netty uses the Reactor multiplexing model as shown in Fig. 3. The Listen-EventLoopGroup (bossGroup) is responsible for listening sockets. When new clients arrive, it allocates corresponding Channels and routes them to the associated request handlers.

A straightforward way to run existing applications on InfiniBand is to use IP over InfiniBand (IPoIB). However the existing TCP/IP stack will cause frequent context switching, which consumes lots of CPU resources as throughput gets bigger. A typical stream processing application involves bunches of data transfer between workers, and the problem grows worse with higher parallelism or smaller message size. The excessive use of CPU could also result in a throughput decrease since workers have to wait for the context switching to finish. To sum up, stream processing applications like Apache Storm cannot fully utilize InfiniBand and its communication modules need targeted optimization.

3 Design

3.1 Messaging Transport Layer and Basic Model

The message component of the Storm source code defines several interfaces to describe the communication needs between workers. Based on these, we wrote our own components to support RDMA communication. The relevant interfaces are indicated in Fig. 4.

- TransportFactory: Used to create the communication context, which is used by WorkerState in the Worker process.
- Context: Used to provide actual context for server and client communication component for different hardware devices (Ethernet or InfiniBand) according to different configuration parameters.
- Connection Callbacks: Receive messages asynchronously and notify the receiver thread for subsequent processing

The communication is one-way for workers, that is, the clients actively connect to the server and send their message. The server is required to implement *registerRecv()* call, and call when new messages arrive, and the client is required to implement *send()* call.

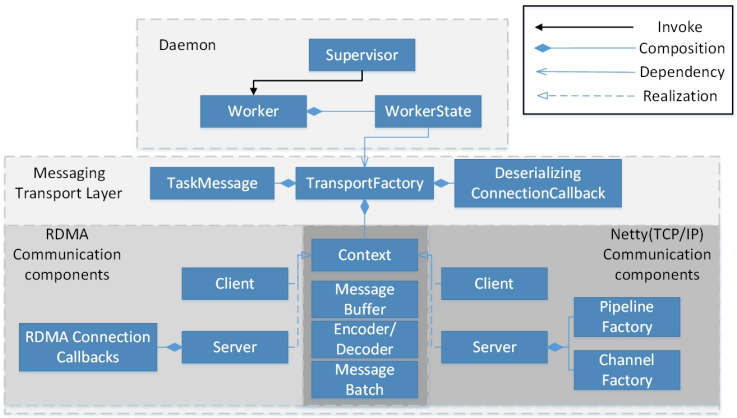

Fig. 4. Communication module components of Storm

3.2 JXIO-based implementation

JXIO is a high-performance asynchronous reliable messaging library optimized for hardware acceleration. Fig. 6 shows the communication timing logic based on JXIO. Fig. 5 shows the main classes involved and their respective functionalities are as follows:

- The Server class is responsible for Initializing parameters, binding IP addresses, setting listening ports, registering memory pools, and initializing event handling queues to handle client connections and IO events.

– ServerPortal is for listening to incoming connections. It can accept/reject or forward the new session request to another portal. Two event handlers for ServerPortal are required to implement, namely onSessionNew for a new session arrival notification and onSessionEvent for events like CLOSED.
– ServerSession is for receiving Message from client and sends responses. ServerSession handles three events on his lifetime, namely onRequest for requests from the client, onSessionEvent for several types of session events, and onMsgError when error occurs.

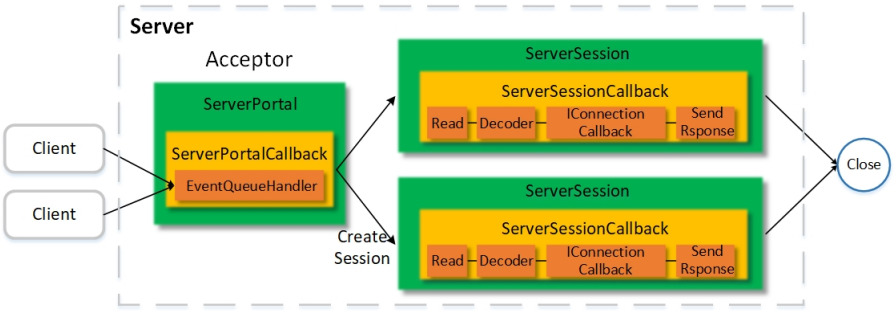

Fig. 5. Basic model for JXIO-Based RDMA communication

3.3 DiSNI-based implementation

DiSNI (former known as jVerbs [17]) is a Java library for direct storage and networking access from userspace. The DiSNI-based component consists of two parts: the Server and the Client. Fig. 6 shows the communication timing logic based on DiSNI. On the Server side, we implemented our own Endpoint by inheriting RdmaEndpoint class and overriding *init*() and *dispatchCqEvent*() call. Since calls to *serverEndpoint.accept*() are blocked, a separate thread (ServerMain) for server logic is necessary. Once connection established, the *init*() method will set up several recv Verbs call, whose number is configured by recvCallInitialized. Once messages arrive, *dispatchCqEvent*() will be called, and the normal TaskMessages are passed to upper layer by calling IConnectionCallback registered by *registerRecv*() call. On the Client side, Client.Endpoint is responsible for client logic. Once *send*() is issued by upper layer, messages are put into the task message queue, waiting for client thread to take.

4 Performance Evaluation

4.1 Experimental Setup

We use 8 Sugon nodes with dual Intel Xeon E5-2660 processors and 128GB RAM each for our evaluation cluster setup. Each node runs CentOS Linux release 7.3.161, and are connected with both 1GigE and Mellanox 40G QDR InfiniBand via an Ethernet and an InfiniBand switch, respectively. We use our modified

Storm based on v2.0.0 and Zookeeper-3.4.5 in our experiments. The Zookeeper is deployed on all 8 nodes of the cluster, so that the Storm nimbus and supervisor daemon can share the state information at a minimal cost. The Storm nimbus daemon is deployed on node1 and Storm supervisor daemon is deployed on the remaining 7 nodes.

To find the best buffer size of Storm performance, we run Storm on 4 nodes at first, adjusting the message size by setting configuration parameter *topology.transfer.buffer.size*, and determine the best size with the minimal processing delay and CPU load. Then, we use *the transferred tuple per second* to benchmark the performance of Storm under the best buffer size tested. The topology we use is designed to maximize inter-worker communication, therefore broadcast grouping is used. We use 4 worker processes per node for all the tests mentioned.

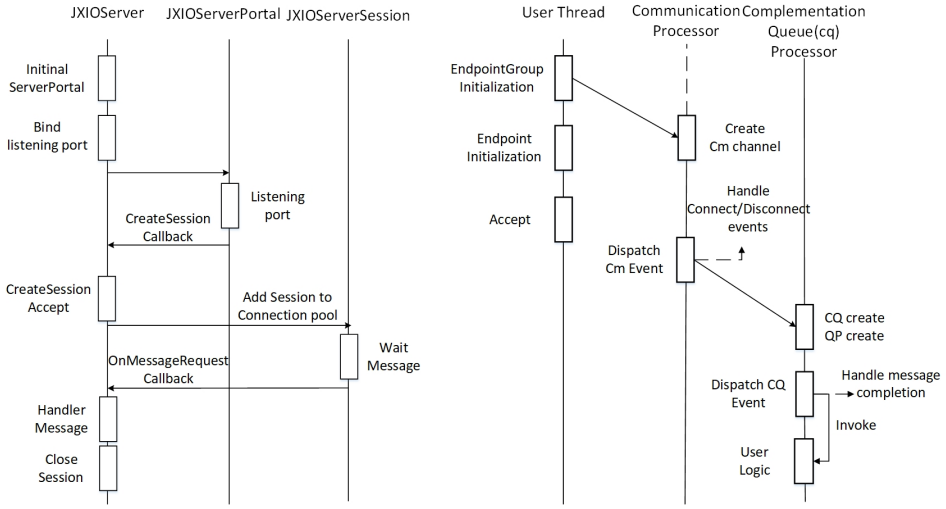

Fig. 6. RDMA Communication Diagram (left: JXIO, right: DiSNI)
Each of the horizontal timelines corresponds to a representative object with its name given above. The rectangle marks significant events that happened during the process.

4.2 The effect of message size on performance

We run Storm on 4 nodes, and 4 Worker processes per node, a total of 16 processes running on the cluster. Each Worker process contains 5 Executors, and each Executor chooses to be a Spout or a Bolt. We conclude our experiment in two aspects according to the experimental data as shown in Fig. 7 and Fig. 8:

– The increase in send buffer size will result in an increase in processing delay, which has the most significant impact on TCP and has less impact on RDMA implementations. Both RDMA and IPoIB achieve significant improvements over TCP. Compared with TCP, RDMA can achieve an acceleration ratio of

around 10, and compared with IPoIB, RDMA can achieve an acceleration ratio of around 5. The DiSNI version of Storm has a slightly lower processing latency than the JXIO version.

– The send buffer size will also affect the CPU load. Because IPoIB takes advantage of the hardware, it can process more packets than TCP at the same time. However, this exacerbates the shortcomings of TCP/IP protocol stack, that is, frequent context switching and memory copying, which will increase the CPU load. RDMA significantly reduces CPU load compared to IPoIB because there is no need to interrupt the operating system for memory copying.

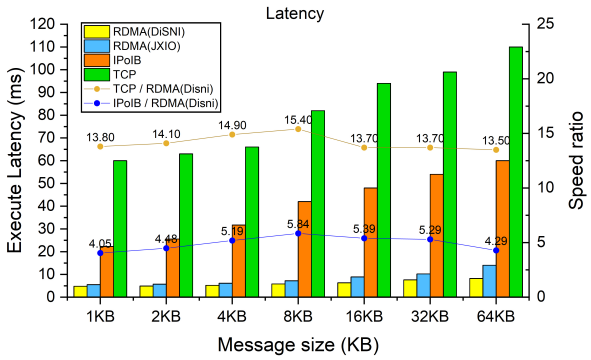

Fig. 7. Experimental result of Execute latency

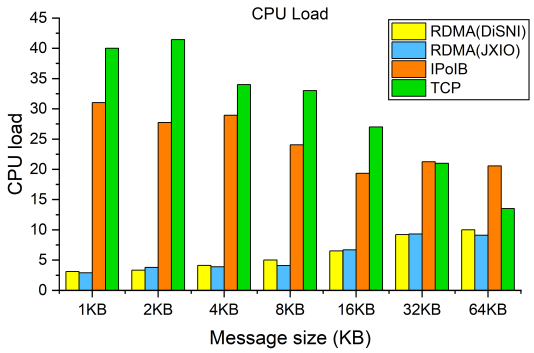

Fig. 8. Experimental result of CPU load

4.3 The impact of distributed scale on performance

We test the scalability of RDMA based Apache Storm, testing on 2, 4, 6, and 8 nodes, respectively. The result we measure is the total number of tuples in a certain length of time. This data is used to represent the network's total throughput.

As shown in Fig. 9, when the number of nodes in the cluster is 2 and the total

number of working processes is 8, the number of connections between processes is 8*8. At this time, the throughput of systems based on RDMA (DiSNI), RDMA (JXIO), IPoIB, and TCP are 894 Tuples/s, 660 Tuples/s, 316 Tuples/s, and 444 Tuples/s. When the cluster size increased to 8 nodes and the number of processes increased to 32, the throughput rates of the three modes are 3482 Tuples/s, 3138 Tuples/s, 1900 Tuples/s and 1058 Tuples/s. After the scale is expanded by 4 times, the system throughput based on RDMA, IPoIB, and TCP modes has increased by 4 times, 6 times, and 2 times respectively.

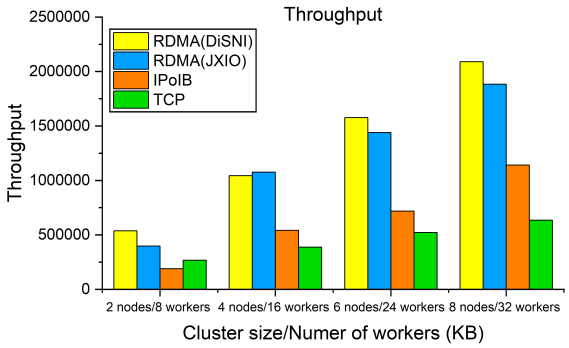

Fig. 9. Experimental result of Transferred Tuples

5 Conclusion

In this paper, we reconstruct the communication module of the streaming processing framework Storm with two different RDMA-based Java interfaces, JXIO and DiSNI, respectively. The experimental results show that the optimized Storm achieves significant performance improvement. When complex stream topologies are specified, the frequent memory copying of the TCP/IP protocol stack causes the CPU to perform context switching frequently, which will increase the CPU load. The accelerated transmission of IPoIB's underlying hardware exacerbates the above consequences. The experiment shows that the optimized Storm can effectively use RDMA technology to reduce CPU utilization significantly. When the message buffer size increases, the RDMA version of the Storm shows good acceleration characteristics. It can significantly reduce the processing delay and Bolt can process and send more Tuples. The throughput of the topology network increases linearly when the cluster size increases. Experimental results show that the scalability of the optimized Storm is improved compared to the TCP version.

6 Related Work

The work is inspired by Seokwoo Yang's [11], which implements JXIO acceleration on an earlier version of Storm. Our work differs from Yang's in two ways:

First, we have implemented JXIO acceleration in the current version of Storm, and lots of interfaces have to be reconsidered since the messaging interface has been changing a lot. Second, JXIO is based on a high-level, request-response based communication paradigm, which limits the possibility for further optimizations. The native Verbs call and SVC design in DiSNI [17] are brought to address and solve such problems, and we have achieved lower CPU load and latency with the help of the direct JNI interface the DiSNI have provided.

References

1. Apache Storm. https://storm.apache.org/, 2019.
2. Apache Spark. http://spark.apache.org/, 2019.
3. Apache Flink. https://flink.apache.org/, 2019.
4. Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
5. Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12(5):516–530, 2019.
6. Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir]relevance of network performance for data processing. In Austin Clements and Tyson Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.
7. Dawei Sun, Shang Gao, Xunyun Liu, Fengyun Li, and Rajkumar Buyya. Performance-aware deployment of streaming applications in distributed stream computing systems. *Int. J. Bio Inspired Comput.*, 15(1):52–62, 2020.
8. Xunyun Liu and Rajkumar Buyya. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Comput. Surv.*, 53(3):50:1–50:41, 2020.
9. Gayashan Amarasinghe, Marcos Dias de Assunção, Aaron Harwood, and Shanika Karunasekera. Ecsnet++ : A simulator for distributed stream processing on edge and cloud environments. *Future Gener. Comput. Syst.*, 111:401–418, 2020.
10. Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 659–670. IEEE Computer Society, 2017.
11. Seokwoo Yang, Siwoon Son, Mi-Jung Choi, and Yang-Sae Moon. Performance improvement of apache storm using infiniband RDMA. *J. Supercomput.*, 75(10):6804–6830, 2019.
12. Patrick MacArthur, Qian Liu, Robert D. Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. An integrated tutorial on infiniband, verbs, and MPI. *IEEE Commun. Surv. Tutorials*, 19(4):2894–2926, 2017.
13. Elena Agostini, Davide Rossetti, and Sreeram Potluri. Gpudirect async: Exploring GPU synchronous communication techniques for infiniband clusters. *J. Parallel Distributed Comput.*, 114:28–45, 2018.

14. Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 705–722. ACM, 2019.
15. David Corral-Plaza, Inmaculada Medina-Bulo, Guadalupe Ortiz, and Juan Boubeta-Puig. A stream processing architecture for heterogeneous data sources in the internet of things. *Comput. Stand. Interfaces*, 70:103426, 2020.
16. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.
17. Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. jVerbs: Ultra-low latency for data center applications. *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*, 2013.