2018 IEEE 38th International Conference on Distributed Computing Systems

TurboStream: Towards Low-Latency Data Stream Processing

Song Wu*, Mi Liu*, Shadi Ibrahim†, Hai Jin*, Lin Gu*, Fei Chen*, Zhiyi Liu*
* *Services Computing Technology and System Lab, Cluster and Grid Computing Lab*
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
† *Inria, IMT Atlantique, LS2N, Nantes, France*
Email: {wusong, liumi, hjin, lingu, fei_chen_2013, zhiyiliu}@hust.edu.cn, shadi.ibrahim@inria.fr

***Abstract*—Data Stream Processing (DSP) applications are often modelled as a directed acyclic graph: operators with data streams among them. Inter-operator communications can have a significant impact on the latency of DSP applications, accounting for 86% of the total latency. Despite their impact, there has been relatively little work on optimizing inter-operator communications, focusing on reducing inter-node traffic but not considering *inter-process communication* (IPC) inside a node, which often generates high latency due to the multiple memory-copy operations. This paper describes the design and implementation of TurboStream, a new DSP system designed specifically to address the high latency caused by inter-operator communications. To achieve this goal, we introduce (1) an improved IPC framework with OSRBuffer, a DSP-oriented buffer, to reduce memory-copy operations and waiting time of each single message when transmitting messages between the operators inside one node, and (2) a coarse-grained scheduler that consolidates operator instances and assigns them to nodes to diminish the inter-node IPC traffic. Using a prototype implementation, we show that our improved IPC framework reduces the end-to-end latency of intra-node IPC by 45.64% to 99.30%. Moreover, TurboStream reduces the latency of DSP by 83.23% compared to JStorm.**

***Keywords*-distributed computation, data stream processing, inter-operator latency, IPC, operator placement**

I. Introduction

With the rapid development of IoT (Internet of Things), social networks, mobile Internet, etc., there is an eager demand in both scientific and industrial applications for doing real-time analysis on high throughput data streams, i.e., *Data Stream Processing* (DSP) [1]. Several DSP systems have therefore been introduced including Spark Streaming [2], Flink [3], Storm [4], Heron [5], and Alibaba JStorm [6].

As shown in Fig. 1, application of DSP, also called topology, is made up of arbitrary operators with data streams among them. Data streams describe the data dependencies of operators. Each operator can subscribe for any distinct data streams from other operators, and then it can consume the data from subscribed data streams and produce new data streams. Usually, an operator has many instances (i.e., tasks) which are executed in parallel. A DSP cluster may consist of thousands of machines [7]. Each cluster node is configured to use a fixed number of slots, on which workers can be launched. Workers are *Java Virtual Machine* (JVM) processes within which operator instances are executed.

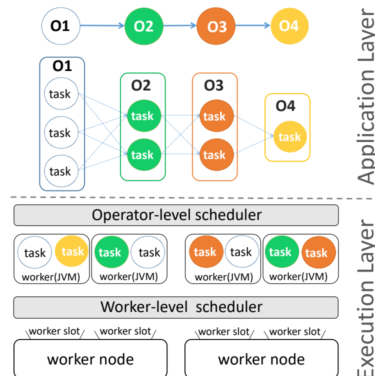


Figure 1: The application layer and execution layer of DSP

In DSP, data is processed in the memory and sent to the next operator immediately without the necessity for storing them. In every shuffling phase, there are large numbers of messages streaming among the operators (i.e., inter-operator communications). If two operators are located in the same worker, the latency of data transmission between them will be very low because they share the same memory space. However, if an operator needs to deliver data to an operator which is located in another worker, the latency will be much higher because current state-of-the-art *inter-process communication* (IPC) framework (i.e., Netty [8]) will invoke serialization, memory copies, and de-serialization when workers reside on the same cluster node and network transmission will be involved when workers reside on different nodes. In practice (see section II), inter-operator communications account for 86% of the total latency of DSP applications and therefore can have a significant impact on their performance.

There are two primary yet complementary ways to optimize inter-operator communications: (i) reduce the cost of intra-node IPC and (ii) reduce the inter-node IPC traffic. Prior efforts have focused mainly on reducing the inter-node IPC traffic by scheduling highly communicating operator instances to workers in the same node [9–12]. However, they may encounter high latency due to the high cost of intra-node IPC, resulting from the memory-to-memory copying inside a cluster node. As an effort to tackle the high cost of intra-node IPC, T-Storm [13] dictates that each node has only ONE available worker for each topology, hence, it avoids the inter-worker traffic inside one node. While this is suitable

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00099

IEEE computer society

for lightly loaded topologies, T-Storm suffers a significant performance degradation in heavily loaded topologies, due to the overhead of JVM *Garbage Collection* (GC) and the competition on CPU resources inside each worker.

To address these limitations, we present the design and implementation of TurboStream, a new DSP system designed specifically for reducing the high latency caused by inter-operator communications. *To the best of our knowledge, TurboStream is the first DSP system that improves IPC when multiple workers reside in the same cluster node (intra-node IPC) and exploits this feature to further improve operator instances scheduling*. To achieve this goal, TurboStream incorporates two complementary components: (1) an improved IPC framework with **OSRBuffer** (***O****ff-heap* ***S****tream* ***R****ing Byte****Buffer***) to reduce memory-copy operations and the waiting time of each single message when transmitting messages between the operators inside one node, and (2) a coarse-grained scheduler that consolidates operator instances and assigns them to nodes to diminish the inter-node IPC traffic. We implemented TurboStream on JStorm — a well known DSP system, developed by Alibaba, to target very low latency DSP applications [6]. Experimental evaluation results show that the improved IPC framework reduces the end-to-end latency of intra-node IPC by 45.64% to 99.30%. Moreover, TurboStream reduces the latency of DSP by 83.23% compared to JStorm. The primary contributions of this paper are as follows:

- By means of experimental evaluation, we provide a deeper look at the latency breakdown of a DSP topology. We find that inter-operator communication accounts for 86% of the total latency. Moreover, we show that current state-of-the-art IPC framework (i.e., Netty) does not scale well under high transmit rate.
- We present TurboStream. It is, to our knowledge, the *first* DSP system which: (1) improves the end-to-end latency of intra-node IPC, with the novel improved IPC framework; and (2) diminishes inter-node IPC traffic, with the coarse-grained scheduler.
- We extend the idea of one memory-copy operation, utilized in MPI to improve IPC [14–16], to DSP systems. To do that, we introduce a novel buffer design, OSRBuffer. It uses the off-heap ring byte buffer to accelerate the message transmission between JVMs. Thus it reduces memory-copy operations and the waiting time of each single message in a DSP topology.
- We propose a generic coarse-grained scheduler, which uses the data dependencies in the topology and the runtime traffic information to consolidate the communicating operator instances before scheduling. It embraces a BFS-based algorithm to assign the consolidated operator instances to cluster nodes to reduce inter-node IPC traffic and strikes load balance in the DSP cluster.

TurboStream is implemented on JStorm. This is motivated by the wide adoption of JStorm and also due to the very low-latency which JStorm can sustain (see section II). However, it is important to note that, although TurboStream is implemented on (and thus evaluated against) JStorm in this work, our target is more general. In particular, OSRBuffer is *generic* and can be applied to any JVM-based DSP system and covers the two prominent stream processing models: record-at-a-time model and batch model. Moreover, *any* topology-based DSP system can use our coarse-grained scheduler, such as Storm[4], Heron [5], and Flink [3].

The rest of this paper is organized as follows: Section II describes the background and motivation of our study. Section III depicts the overview of TurboStream. Section IV describes the design and implementation of our improved IPC framework for DSP followed by the details of the coarse-grained scheduler in Section V. Section VI presents the evaluation results and corresponding analysis. Section VII discusses the related works. Finally, Section VIII concludes the paper and presents future works.

II. Background and Motivation

In this section, we first describe JStorm, and then we discuss our experimental results to understand the main bottlenecks for achieving low latency in DSP systems.

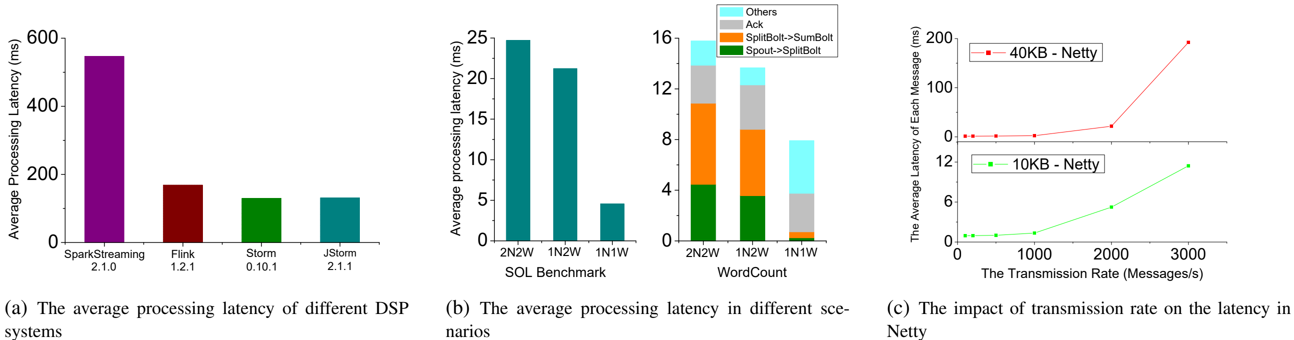
A. A Representative DSP System: JStorm

JStorm is a Java version of Apache Storm and it is said to be faster than Apache Storm [6]. JStorm is widely used in both industry and academia [17]. Operators in JStorm are called Components. The Components that act as the event source of the topology are $Spouts$ while the rest of the topology are called $Bolts$. There are two other kinds of system Components in JStorm: $TopologyMaster$ and $Acker$. $TopologyMaster$ is in charge of backpressure strategy, which can prevent the topology from being overloaded. The function of $Acker$ is to keep track of every source message and it can tell the $Spout$ whether the message has been processed successfully or not. The data stream is made up of continuous tuples. The operator instances of Components are called tasks.

Latency evaluation of DSP systems. We implement TurboStream on JStorm due to its claim of very low latency. Fig. 2(a) shows the average processing latency in different DSP systems. The benchmark we used is from Yahoo Streaming Benchmarks [18] and the input rate is 20,000 events/s. The experimental testbed is described below. All systems use 4 workers for processing data. Both JStorm and Storm outperform other DSP systems in terms of latency.

B. Inter-Operator Latency in DSP

We conduct a series of experiments to analyze the impact of inter-operator latency on the overall latency in JStorm. All the experiments are performed on a cluster of 8 nodes. Each node is equipped with 16x1.2 GHz CPUs, 64 GB of RAM and 210 GB HDD. All the nodes are connected with



(a) The average processing latency of different DSP systems

(b) The average processing latency in different scenarios

(c) The impact of transmission rate on the latency in Netty

Figure 2: Experimental analysis of (a) the latency of different DSP systems, (b) the average processing latency under different deployments, and (c) the impact of transmission rate on the latency in Netty

Table I: The average inter-operator latency of each stage in WordCount

Scenario \ Stage	Spout→SplitBolt	SplitBolt→SumBolt
2N2W	4.46ms	6.40ms
1N2W	3.58ms	5.22ms
1N1W	0.24ms	0.47ms

1Gbps Ethernet. The OS is Redhat Enterprise Linux 6.2 with the kernel version 2.6.32. The JVM version is 1.7.0_79. We deploy JStorm2.1.1 and configure it with 8 worker nodes, each of which has 4 worker slots. The topologies are SOL benchmark and WordCount Topology [19]. In SOL benchmark, there are five operators and each operator has 2 tasks. The input rate is 5,000 tuples/s. In WordCount Topology, $Spout$ produces sentences for the topology, and the input rate is 10,000 tuples/s. $SplitBolt$ splits the sentences into words, which is summed up by the $SumBolt$. We deploy the two topologies in 3 different scenarios (2N2W: deploy the topology to 2 workers in 2 different nodes, 1N2W: deploy the topology to 2 workers inside 1 node, 1N1W: deploy the topology to only 1 worker), and measure the total processing latency as well as the latency of every stage.

As shown in Fig. 2(b), although the intra-node IPC is more efficient than the inter-node IPC in terms of end-to-end latency, the performance difference between 2N2W and 1N2W is not significant in both WordCount Topology and SOL benchmark. As shown in Table I and Fig. 2(b), the proportion of inter-operator latency to total processing latency of WordCount Topology is up to 86.88% and 86.66% in these two scenarios without the effect of $Acker$. When we put all the tasks in only one worker (1N1W in Fig. 2(b)) to remove the inter-worker traffic, there is a sharp drop in total processing latency. As seen in Table I, the total inter-operator latency of WordCount Topology is only 0.71ms. In conclusion, event latency between operators located in different workers is the bottleneck when the topology is deployed on multiple workers (2N2W, 1N2W).

The inter-operator latency is no longer the bottleneck of the total processing latency in 1N1W. The total processing latency of SOL benchmark decreases sharply while the total processing latency does not decrease as much as we expect in WordCount Topology, because the former is network sensitive and the later is CPU sensitive. Actually, the backpressure strategy of JStorm is activated in WordCount Topology when we put all the tasks in only one worker as the worker is overloaded. Both the GC overhead and the competition on CPU resources inside each worker significantly degrade the overall performance because one worker takes the total workload. Therefore, we can not put all the tasks in only one worker in real scenario, especially for heavily loaded topologies. In practice, operators are distributed to multiple workers in each node, thus, improving the end-to-end latency of intra-node IPC is an important factor to improve the inter-operator latency of DSP applications.

C. IPC Framework of DSP

Netty [8] is the IPC framework not only for JStorm, but also for Storm, Flink, Spark, etc. Netty socket is more efficient than Java socket for its zero-copy buffer and removing the unnecessary memory copy. We write a server-client Netty application to take a quantitative measurement on the performance of Netty in the context of IPC. The total message latency contains the time of serialization, data transmission, and de-serialization. The latency shown in Fig. 2(c) is the average value of 10,000 messages.

As we increase the transmission rate in Fig. 2(c), the latency of 10KB message and 40KB message increase to 11.43ms and 192.36ms, respectively, when the transmission rate reaches 3,000 messages/s. In Netty, messages will be queued in send buffer and sent in batches when the transmission rate is very high. Such a design is efficient in network transmission but sacrifices the latency of each single message. This batch model is not suitable for intra-node IPC, particularly in DSP systems. Besides, Netty still has three memory-copy operations between heap and off-heap memory in the context of intra-node IPC. We find that the IPC of DSP has many potential optimizations through the analysis above.

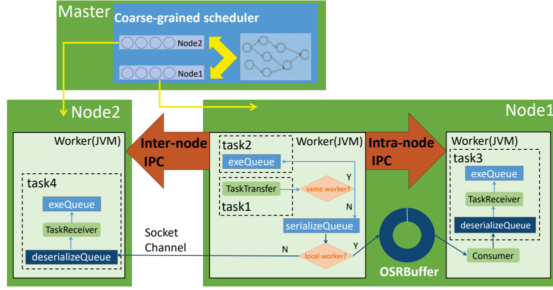


Figure 3: The overview of TurboStream

III. An overview of TurboStream

TurboStream is designed to reduce the inter-operator latency in DSP applications. The reduction in inter-operator latency is due to two important factors: 1) the reduction in the latency of intra-node IPC, and 2) the reduction in inter-node traffic. As shown in Fig. 3, TurboStream consists of two main components, the improved IPC framework with OSRBuffer and coarse-grained scheduler. As mentioned in section II, the design of Netty is not suitable for intra-node IPC in terms of latency. We reduce both memory-copy operations and the waiting time of message in intra-node IPC by using OSRBuffer, which merges read buffer and write buffer into a shared one. At the same time, we provide a producer-consumer model for OSRBuffer to transfer data. In the context of DSP, messages are continuous data streams with high throughput. In order to achieve high efficiency of writing, OSRBuffer designs a ring structure. OSRBuffer does not need read-write lock operations in high-concurrency scenarios, which improves the performance. OSRBuffer can therefore reduce the event latency between operators located in the same node.

The event latency between operators located in the same node decreases through the implementation of OSRBuffer in intra-node IPC. However, the event latency between operators located in different nodes discourages further reduction of the total event processing latency, especially when the proportion of intra-node IPC traffic to total IPC traffic is very low. Therefore, we propose a general coarse-grained scheduler, which uses the data dependencies in topology and the runtime traffic information to consolidate the communicating operator instances (tasks) before scheduling. BFS-based algorithm is designed to assign the consolidated tasks to achieve the reduction in inter-node IPC traffic. Besides, this scheduler can achieve load-balance at the same time. As a result, inter-operator latency is further reduced in TurboStream. It is important to mention that in order to guarantee correct execution after the (re)deployment, TurboStream keeps the same approach, as in JStorm, Storm [20], T-Storm [13]. It shuts down the old workers and starts new workers when a new assignment is needed. This unfortunately brings unavoidable overhead of 10-20s [13, 20, 21]. What's more, the coarse-grained

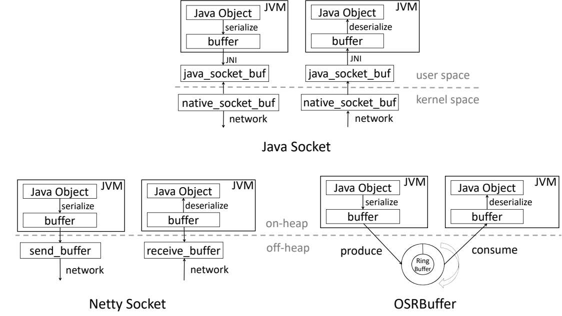


Figure 4: Memory-to-memory data flow when using Java socket, Netty socket, and OSRBuffer

scheduler can operate in an off-line mode, which avoids the above-mentioned overhead but may sacrifice a bit the latency reduction contributed by the scheduler (It uses the data dependencies in topology to consolidate the communicating operator instances off-line).

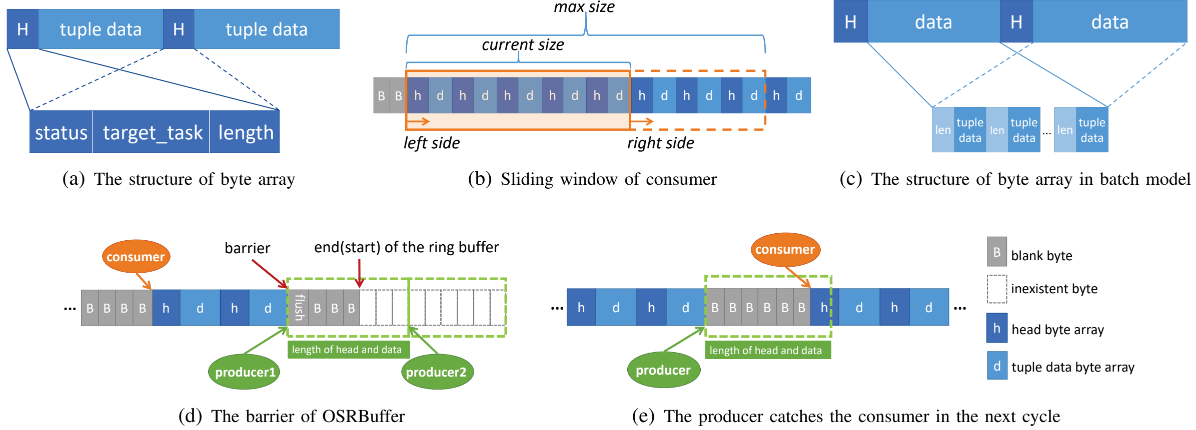
IV. Improved IPC Framework for DSP

Fig. 4 compares the data-flow between two JVMs when using Java socket, Netty socket, and our OSRBuffer. In Java socket, there are three memory-copy operations in both sender and receiver and two times of switching between user space and kernel space. Netty has improved the Java socket in two aspects: (1) Netty uses off-heap memory to buffer the data directly without the unnecessary memory copy. (2) There is no need to switch between user space and kernel space. However, we reduce both memory-copy operations and the waiting time of message in intra-node IPC by using OSRBuffer.

Although works [14–16] have achieved significant performance improvement in intra-node IPC, it is still hard to share data between workers under high concurrency. Most DSP systems are JVM-based frameworks, where user can not manage the memory space explicitly because JVM has its own Memory Manager. We use off-heap memory to break through the limitation of Memory Manager in JVM. However, there are some key issues in message transmission through off-heap memory: (1) There is no Java object but only raw byte array in off-heap memory. (2) We need to handle the failure of messages. (3) There are synchronous and asynchronous issues among workers.

A. Producer-Consumer Model

To solve the problems above, we first design a novel producer-consumer model for OSRBuffer. For each tuple, we add a head including the key information for it, with which consumer can consume the continuous tuples (message) from boundless raw bytes. The structure of the head is shown in Fig. 5(a). The producer pushes the byte array of the tuple with a head to OSRBuffer while the consumer will process the byte array according to the head information. The head has three fields: $status$, $target_task$, $length$.



(a) The structure of byte array (b) Sliding window of consumer (c) The structure of byte array in batch model

(d) The barrier of OSRBuffer (e) The producer catches the consumer in the next cycle

Figure 5: The design of OSRBuffer in improved IPC framework

$status$ tells consumer the status of current tuple, which has five types: $isWriting$, $isReady$, $hasRead$, $flush$, $other$. $target_task$ is the id of the task that current tuple will be sent to. $length$ is the length (byte) of the byte array. When consumer reaches a new tuple, it first checks the status. If the status is $isReady$, it will read the byte array and push to the target task. If the status is $isWriting$, it can not consume the byte array, because the current tuple is being written or it has failed. Our producer-consumer model also supports a batch model, the structure of which is shown in Fig. 5(c). In batch model, we serialize each tuple to byte array with an inner head which only contains the length of tuple. Then, the byte arrays of all tuples will be merged as a whole byte array.

B. Sliding Window

When the current tuple is being written or has failed, the status will be $isWriting$ and the consumer can not consume it. If it has failed, the consumer just skips it. However, the consumer should wait on the tuple when it is being written. In order to distinguish between these two cases without impacting the speed of consumer, we provide a sliding window for consumer. Fig. 5(b) depicts the details of our design. When the consumer reaches a new tuple in OSRBuffer, it first checks the status. If the status of current tuple is $isWriting$, the left side of the window will wait on the tuple while the right side will keep on consuming the following tuples. When the status of current tuple turns $isReady$, the left side will consume it and reach the following tuple. If the status of current tuple remains unchanged, the left side will wait until the window reaches the maximum size and then skip to the following tuple. The following tuple has three kinds of status: 1) $isWriting$, it will repeat the step of waiting. 2) $isReady$, it will process the tuple. 3) $hasRead$, which means the tuple has been processed by the right side before, it just skips the tuple. When the sliding window reaches the maximum size, the left side will have to skip the current tuple no matter what the status is. Therefore, the right side will never wait. Such a design of sliding window can reduce the number of missing tuples without impacting the consuming speed of consumer.

C. Asynchronization and Synchronization Issues

We use asynchronous communication in OSRBuffer for data transmission between the producer and consumer. Usually, the length of tuple byte array is different, so there might be some blank bytes in OSRBuffer (bytes between barrier and end of the ring buffer in Fig. 5(d)). When the producers reach the end of the buffer or the remaining space is not enough to hold a whole tuple, they have to jump to the starting position. The producer (producer 1 in Fig. 5(d)), which is the first one to detect that the remaining space is not enough, will put the status $flush$ to the current position to inform the consumer about the barrier of the ring buffer. The design of sliding window has accelerated the speed of consumer, but there might be a special situation when producer is still much faster than consumer. In this situation, the producer might catch the consumer in the next cycle, as shown in Fig. 5(e). When the producer is going to overwrite the unconsumed tuples, it needs to wait on the current position until there is enough blank space. Thus, there is no need for synchronization between producer and consumer.

In our design, every task in a worker can push byte array to OSRBuffer. Thus, OSRBuffer needs to handle the synchronization issues among producers. To synchronize access to shared OSRBuffer, a simple and direct approach is to use locks. However, the lock-based approaches cause many problems, such as diminished parallelism caused by mutual exclusion and lock contention. We use a lock-free synchronization algorithm for producers. It relies on well-known hardware synchronization primitive, that is *Compare-And-Swap* [22] (CAS). We use a global shared variable to record the write address where the new byte array can be

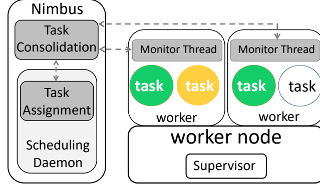


Figure 6: The architecture of coarse-grained scheduler

pushed to. Each time the producer is going to push byte array to OSRBuffer, it will fetch the write address and reserve enough space (This is a CAS-based atomic operation). Such design can prevent the attempt to push byte array to an address where a previously invoked producer has yet to complete its writing. Thus, producers can push byte arrays to OSRBuffer concurrently, which is very efficient in writing.

V. COARSE-GRAINED SCHEDULER

We first present an overview of the coarse-grained scheduler and then describe in details its key components.

A. Overview

The architecture is shown in Fig. 6. The key components of the scheduler are on-line monitor, $TaskConsolidation$, and $TaskAssignment$. On-line monitor gets both the traffic and workload information of each task. $TaskConsolidation$ takes into account the communication patterns among operator instances (tasks) to consolidate the tasks before scheduling. The consolidated tasks will be assigned to the same node by the scheduler. As they do not bring inter-node IPC traffic overhead, the inter-operator latency will further decrease. After the step of $TaskConsolidation$, $TaskAssignment$ chooses available nodes for the consolidated tasks. As the step of $TaskConsolidation$ has tried to consolidate the communicating tasks together, $TaskAssignment$ mainly targets balancing the load among nodes. We design a BFS-based algorithm in $TaskAssignment$ to assign consolidated tasks, which can achieve both load balance and reduction in the inter-node IPC traffic. The scheduler only needs to assign the consolidated tasks to nodes rather than workers, so our scheduler is **coarse-grained**.

B. Task Consolidation

Before scheduling, we first get the relationship of tasks and then begin the step of $TaskConsolidation$. The consolidation algorithm is shown in Algorithm 1. In this algorithm, the communicating tasks will be consolidated to a $parentTask$. Tasks of the same parent will be assigned to the same node in $TaskAssignment$. The $parentTasks$ will be consolidated in the same way until the task number is smaller than $taskNumThreshold$, which is a parameter in our algorithm. There will be multi-level task consolidation when the topology is very large. For each task: First, sort its neighbour tasks in descending order by the amount of traffic between them (line 4-5 in Algorithm 1). Second,

Algorithm 1 Consolidation of tasks

Input: the set of tasks of the topology $taskSet$; the set of data streams of the topology $dataStreams$;
Output: the set of parent tasks $parentTaskSet$

```
1: parentTaskSet ← Φ, parentDataStreams ← Φ
2: for v in taskSet do
3:     if v hasn't been visited then
4:         neighbors ← getInputOutputTasks(v)
5:         sort(neighbors)
6:         consolidated ← false
7:         for u in neighbors do
8:             if u hasn't been visited then
9:                 consolidate u and v to a parent task p
10:                put p to parentTaskSet
11:                consolidated ← true
12:                break
13:            end if
14:        end for
15:        if consolidated ≡ false then
16:            u will produce a parent task p for itself
17:            put p to parentTaskSet
18:        end if
19:    end if
20: end for
21: for e in dataStreams do
22:     source ← e.source_task, target ← e.target_task
23:     p1 ← source.parent(), p2 ← target.parent()
24:     if p1 ≠ p2 then
25:         put e to parentDataStreams
26:     end if
27: end for
```

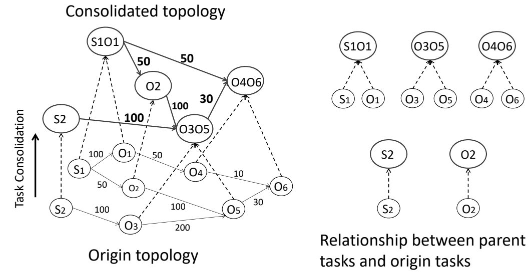


Figure 7: One-level task consolidation

get the first unvisited neighbour task and consolidate them to the $parentTask$ (line 7-14 in Algorithm 1). If all its neighbour tasks have been visited before, it will consolidate itself alone to a $parentTask$ (line 15-18 in Algorithm 1). Once two tasks have been consolidated to a $parentTask$, the traffic overhead between them can be ignored (line 21-27 in Algorithm 1).

Fig. 7 is a simple example of one-level task consolidation. Numbers between operators refer to inter-operator traffics and ones in **bold** refer to the traffic between parent tasks. We assume that the parallelism of each operator is one in current example. After each level of consolidation, the relationship will be established like Fig. 7. When the scale of topology is very large, there might be multiple levels of consolidation. After multiple levels of consolidation, the mapping between the origin tasks and parent tasks will become a relational binary tree, which will be used in the next step of $TaskAssignment$.

Algorithm 2 Split the set of parent tasks into two subsets

Input: $taskSet$, used node number N, $dataStreams$
Output: RES_SET

1: $totalWeight$ of the tasks in $taskSet$
2: $N1 \leftarrow Math.ceil(N/2)$, $N2 \leftarrow Math.floor(N/2)$
3: $target \leftarrow N1/N$, $minDataStream$;
4: **for** $task$ in $taskSet$ **do**
5: $travelQueue \leftarrow \Phi$, $gatherTasks \leftarrow \Phi$
6: $unvisited.putAll(taskSet)$, $visited \leftarrow \Phi$
7: $balance \leftarrow 0$, $gatherWeig \leftarrow 0$
8: $travelQueue.put(task)$
9: **while** $travelQueue$ && $balance < target$ **do**
10: $c_task \leftarrow travelQueue.poll()$
11: $tempWeig \leftarrow c_task.weight + gatherWeig$
12: $tempBlance \leftarrow tempWeig/totalWeight$
13: $dif1 \leftarrow |tempBlance - target|$
14: $dif2 \leftarrow |balance - target|$
15: **if** $dif1 < dif2$ **then**
16: gather c_task in $gatherTasks$
17: update $gatherWeig$, $balance$
18: push unvisited neighbors to $travelQueue$
19: **end if**
20: update $unvisited$, $visited$
21: $dif3 \leftarrow |balance - target|$
22: **if** $!travelQueue$ && $dif3 > factor$ **then**
23: put an unvisited task to $travelQueue$
24: **end if**
25: **end while**
26: **if** $|balance - target| < factor$ **then**
27: **if** $cutDataStream < minDataStream$ **then**
28: $minDataStream \leftarrow cutDataStream$
29: $RES_SET \leftarrow gatherTasks$
30: **end if**
31: **end if**
32: **end for**

C. Assignment of Consolidated Tasks

$TaskAssignment$ will assign the consolidated tasks (parent tasks from consolidation algorithm) to available nodes to achieve workload balance. We first divide the consolidated tasks into two parts. Each part is assigned a number of nodes according to its total workload, that is, the nodes number is proportional to the total workload of each part. We continue dividing each part until the nodes number of the part is only one. This is the key step of the algorithm of $TaskAssignment$, so we show its details in Algorithm 2. Every time we visit a task, we try to gather the task in the $gatherTasks$ and check if it will result in better load balance (line 10-14 in Algorithm 2). If it will get a better load balance, we put current task in the $gatherTasks$ and put all its unvisited neighbour tasks in the $travelQeue$, otherwise we skip to the next task in $travelQeue$ and repeat the steps until reaching the balanced partition result (line 15-25 in Algorithm 2). We can get a balanced partition result in each iteration and check whether the current balanced partition result has the least inter-node traffic (line 27-30 in Algorithm 2). With the balanced partition result that has the least inter-node traffic, and the relational binary tree in consolidation algorithm, the scheduler will know exactly which node each task should be assigned to. The total cost of these two algorithms is $O(|N|^3)$. N is the number of tasks in topology. As the intra-node IPC has been improved by our IPC framework, the intra-node transfer is no longer the bottleneck. Thus, we use a simple round-robin strategy to assign the tasks to available workers within corresponding node. Note that, to simplify the description of Algorithm 2, we assume that the number of available workers is the same in each node. However, our implementation considers that the number of available workers in each node is different. This is important as it is common to have multiple concurrent topologies deployed in a cluster, and therefore the number of available workers for current topology in each node might be different.

D. Online Monitor

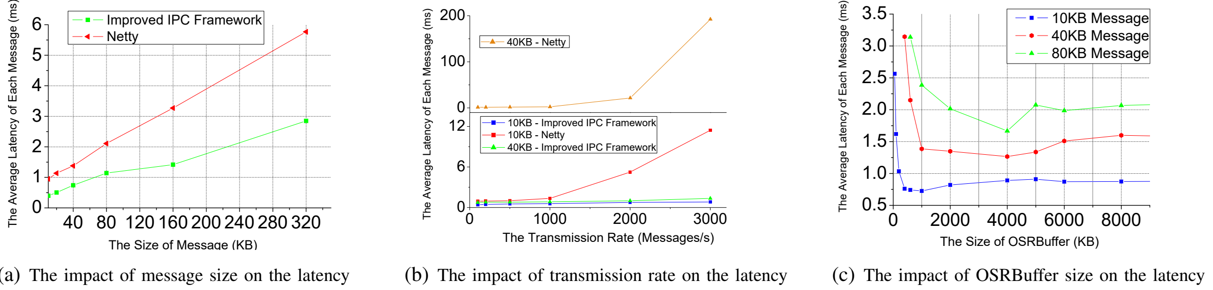
In order to obtain the runtime information of the workload and the traffic of every task, we add a monitor module with a timer in each worker. The monitor is a long-running thread and in charge of runtime information of all the tasks in current worker. During each cycle time T, the monitor records two information for each task: (1) The number of messages sent to every output task. (2) The usage of CPU resource (Hz), which includes the CPU resource used for all the threads in current task. With the runtime information, the on-line strategy will start the step of $TaskConsolidation$ and $TaskAssignment$. Once the on-line strategy gets a better assignment, the redeployment will start. As mentioned in Section III, similar to JStorm, Storm [20], and T-Storm [13], we shut down the old workers and then start up workers with new assignment in current implementation. Another approach is to use task migration, however, it – if successful – also brings a noticeable overhead due to the migration time [23–26]. Moreover, if a failure in the on-line messaging occurs during the migration, the job fails. It is one of our future works to reduce the overhead of the redeployment.

VI. Evaluation

We use the same experimental setup as in Section II. The experiments are divided into three parts: (1) We first evaluate the performance of the improved IPC framework in the context of end-to-end message transmission. (2) We conduct two groups of experiments on JStorm and TurboStream to measure the reduction in the total event processing latency. (3) We evaluate JStorm against TurboStream under different input rate.

A. Performance of Improved IPC Framework

We use the same server-client application in Section II to evaluate the performance of improved IPC framework. We conduct three groups of experiments to evaluate the impact of message size, transmission rate, and OSRBuffer size on the average message latency. In order to show the impact of message size on the latency, we vary the size of message from 10KB to 320KB. The size of OSRBuffer is set to 2MB and the transmission rate is set to 100 messages/s. Though the latency increases as the size of message increases, our



(a) The impact of message size on the latency

(b) The impact of transmission rate on the latency

(c) The impact of OSRBuffer size on the latency

Figure 8: Experimental analysis of (a) the impact of message size on the latency, (b) the impact of transmission rate on the latency, and (c) the impact of OSRBuffer size on the latency

improved IPC framework still achieves better performance compared to Netty. As shown in Fig. 8(a), the average message latency of Netty remains about two times that of our improved IPC framework when the message size varies from 10KB to 320KB, which is caused by the multiple memory-copy operations.

As we increase the transmission rate in Fig. 8(b), our improved IPC framework is still able to sustain low average message latency, with the OSRBuffer, which not only reduces memory-copy operations but also reduces the waiting time in buffer. In contrast, Netty suffers of high average message latency when the transmission rate reaches 3,000 messages/s. This is due to the long waiting time as messages are queued in the send buffer when the transmission rate is very high. This in turn aggravates the latency of each single message. In summary, we observe that improved IPC framework reduces the end-to-end latency of intra-node IPC by 45.64% to 99.30% as shown in Fig. 8(a) and Fig. 8(b).

In the third group of experiments, we try to find out the most suitable OSRBuffer size for different sizes of message. We fix the transmission rate as 3,000 messages/s. As shown in Fig. 8(c), a small size of OSRBuffer results in high latency due to the overhead of the frequent waiting of the Consumer and Producer. On the other hand, large size of OSRBuffer will obviously waste a lot of memory space. Our results in Fig. 8(c) indicate that an appropriate size of the OSRBuffer is about 50-100 times the size of the message.

B. Processing Latency

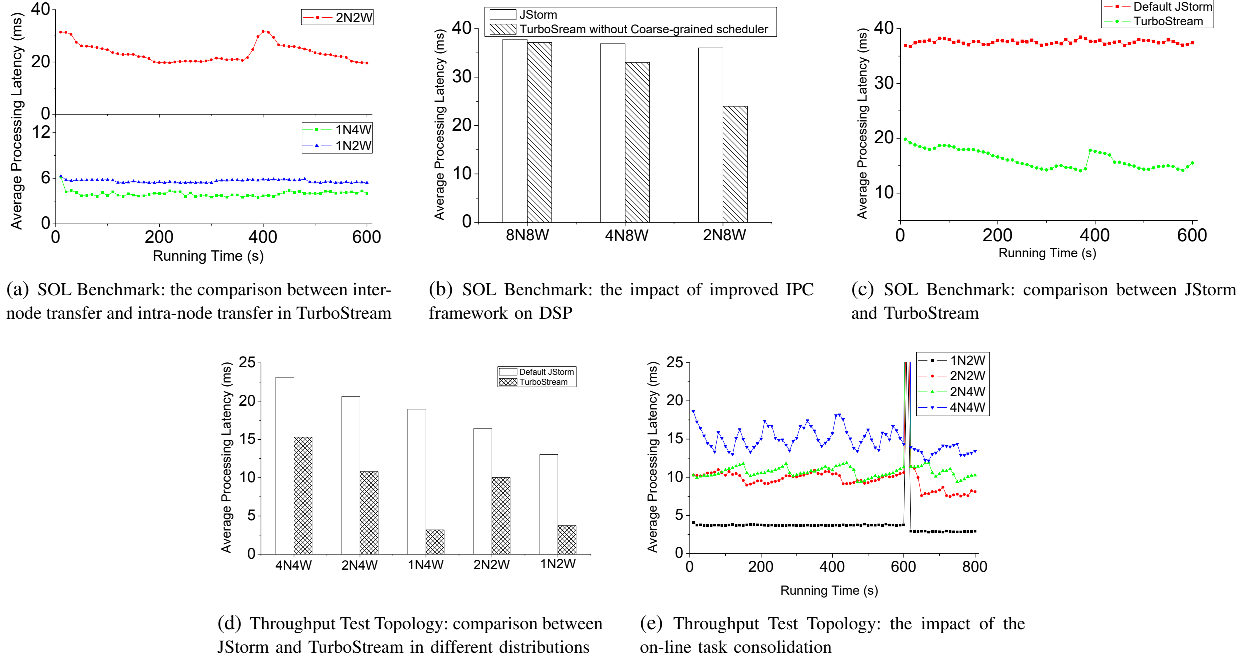
We conduct two groups of experiments on two commonly-used benchmarks that represent the characteristics of a broad class of topologies. The first group of experiments is on SOL Benchmark [19]. It is one of the most commonly-used network sensitive benchmarks, which has one Spout and five Bolts. We configure the topology with 5 tasks for each Bolt. The input rate is about 5,000 tuples/s. The average processing latency, shown in Fig. 9(b) and Fig. 9(d), is the average value of messages in 10 minutes while the average processing latency, shown in Fig. 9(a), Fig. 9(c), and Fig. 9(e), is the average value of messages in every 10 seconds.

Fig. 9(a) shows a comparison between the inter-node transfer and the intra-node transfer in TurboStream. As shown in Fig. 9(a), assigning one worker per node (i.e, 2N2W, as in T-Storm [13]) results in higher latency compared to the case when multiple workers are assigned per node (i.e., 1N2W, 1N4W). This is due to the inter-node transfer. More importantly, TurboStream achieves even lower latency when operators are distributed to 4 workers inside each node (i.e., 1N4W) as the intra-node IPC is no longer the bottleneck in TurboStream.

Fig. 9(b) shows the average processing latency in both JStorm and TurboStream — without enabling the coarse-grained scheduler. As expected, both DSP systems achieve the same latency in the 8N8W scenario. However, unlike JStorm, TurboStream further reduces the average processing latency when there are more than 1 worker in each node. In particular, the average processing latency decreases by 12.71% and 36.68% in 4N8W and 2N8W, respectively. This performance improvement results from our improved IPC framework with OSRBuffer as it targets only improving the intra-node IPC and thus it works in the case of 4N8W and 2N8W. Importantly, the performance improvement strongly depends on the proportion of intra-node IPC traffic to total IPC traffic. As shown in Table II, the proportions of intra-node IPC traffic to total IPC traffic are 14.29%, and 42.86% in 4N8W and 2N8W, respectively. Finally, enabling the coarse-grained scheduler side by side with the improved IPC framework further reduces the average processing latency. As shown in Fig. 9(c), TurboStream reduces the average processing latency by 59.84% compared to JStorm, when we configure the topology with 8 workers and distribute them to 2 nodes.

In the second group of experiments, the test topology is Storm Throughput Test Topology [27]. We deploy the topology on JStorm cluster and TurboStream cluster, respectively. In order to see the impact of improved IPC framework and coarse-grained scheduler, we keep the configuration the same in JStorm cluster and TurboStream cluster.

The results are shown in Fig. 9(d). Although, the proportion of intra-node IPC traffic to total IPC traffic is 0% in



(a) SOL Benchmark: the comparison between inter-node transfer and intra-node transfer in TurboStream

(b) SOL Benchmark: the impact of improved IPC framework on DSP

(c) SOL Benchmark: comparison between JStorm and TurboStream

(d) Throughput Test Topology: comparison between JStorm and TurboStream in different distributions

(e) Throughput Test Topology: the impact of the on-line task consolidation

Figure 9: Processing latency testing in DSP

Table II: The proportion of intra-node IPC traffic to total IPC traffic in different distributions

Distribution / IPC type	8N8W	4N8W	2N8W	1N1W
Intra-node IPC	0%	14.29%	42.86%	0%
Inter-node IPC	100%	85.71%	57.14%	0%
Distribution / IPC type	**4N4W**	**2N4W**	**2N2W**	**1N2W**
Intra-node IPC	0%	33.33%	0%	100%
Inter-node IPC	100%	66.67%	100%	0%

both 4N4W and 2N2W, there is still a significant decrease in the total processing latency. In particular, the average processing latency decreases by 33.85% and 38.90% in 4N4W and 2N2W, respectively. This is due to the coarse-grained scheduler. However, when there are more than 1 worker in each node, TurboStream results in even more reductions in the average processing latency compared to JStorm. The average processing latency decreases by 47.64%, 83.23% and 71.35% in 2N4W, 1N4W, 1N2W, respectively. We can also see that the average processing latency of 1N4W is lower than that of 1N2W in TurboStream, which is consistent with the experimental results in Fig. 9(a).

In summary, the event latency between operators located in different nodes becomes the bottleneck when the proportion of intra-node IPC traffic to total IPC traffic is very low, such as 4N4W and 2N2W. Hence, the improved IPC framework does little work. However, TurboStream reduces the total processing latency by up to 38.90% compared to JStorm. This demonstrates the important role of our coarse-grained scheduler in reducing the inter-node traffic.

All the experiments above use the task consolidation off-line. Although off-line task consolidation is quite effective in reducing the inter-node IPC traffic, the on-line task consolidation can make further performance improvements as it can handle traffic and workload changes during the execution of the DSP application. Precisely, the inter-node IPC traffic will be reduced as the task consolidation considers the current traffic information. As shown in Fig. 9(e), in 1N2W, where there is no inter-node IPC traffic, there is no change in the observed latency. However, the average processing latency in 2N2W and 4N4W is decreased by 20.89% and 12.91%, respectively. In both scenarios, the reduction in latency is observed after the redeployment is completed. The re-deployment is triggered by the on-line task consolidation at 600s. Moreover, we can clearly see the impact of the redeployment, as the processing latency increases sharply during the redeployment. As we discussed in section V, in current implementation, we choose shutdown-start approach to redeploy the topology with new assignment, which will take about 15-20 seconds.

C. Throughput Test: JStorm vs. TurboStream under Different Input Rate

We use the Throughput Test Topology [27] and compare the latency of both JStorm and TurboStream under different input rates. The results are shown in Fig. 10. The average processing latency of JStorm is about 2.16 times that of TurboStream when the input rate is about 1,000 tuples/s. As the input rate continues to increase, the average processing latency increases up to 142.53ms in JStorm, when the input rate reaches 40,000, while it stays below 20ms in

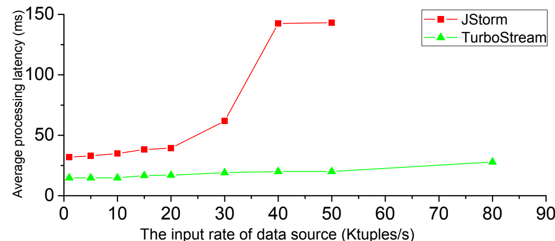


Figure 10: Comparison of the average processing latency of JStorm and TurboStream under different input data rates

TurboStream. However, as the input rate reaches 80,000 tuples/s, JStorm activates the backpressure strategy, which (by default) controls the input rate at 50,000 tuples/s, and new messages will be queued in memory. TurboStream, on the other hand, suffers a slight increase in the average processing latency (27.90ms) when the input rate reaches 80,000 tuples/s. This experiment result shows that TurboStream can cope with very high input rate and therefore achieves much better throughput than JStorm. This is due to the improved IPC framework and the coarse-grained scheduling which can jointly improve resource utilization of cluster nodes and the processing latency.

VII. Related Work

A. IPC Improvement

Given that many big data processing systems are based on JVM, such as Hadoop [28], Spark, Storm, Flink, there is a rising interest in JVM for high performance. Some research efforts have demonstrated the inefficiency of Java socket in delivering high performance data analytics and thus implement hardware specific solutions, based on *Remote Direct Memory Access* (RDMA) to improve the performance of big data applications [29, 30]. As DSP has a more stringent requirement for processing latency, Netty [8] has been introduced as an alternative to Java sockets in many DSP systems, however, as discussed in section II, it still leads to suboptimal performance in terms of latency.

Much work has focused on improving intra-node IPC in MPI. The copy-in/copy-out algorithm is used [31, 32] to transport the messages between MPI process while kernel assisted memory copy is used [14–16], it utilizes one-copy memory and the experimental results show significant benefit in the context of end-to-end communications. Inspired by these works, we find that intra-node IPC has many potential optimizations in DSP. However, it is hard to use the aforementioned technologies directly. Most DSP systems are JVM-based frameworks, where user can not manage the memory space explicitly because JVM has its own Memory Manager. Besides, there are synchronous and asynchronous issues and fault-tolerant issues in message transmission in DSP systems.

B. Operator Placement and Task Scheduling in DSP

Operator placement has been extensively studied in DSP systems [4–6, 10, 13, 21]. SBON [10] introduces a tool for DSP system to make placement decision in a virtual cost space and then the space coordinates will be mapped to physical node. The traffic-based on-line scheduler in [21] reduces both inter-worker and inter-node traffic, but it requires modifications to the user code in spouts and bolts to enable constant monitoring of the data load. In contrast, our coarse-grained scheduler is transparent to users. T-Storm [13] requires that each node has only one available worker for each topology to avoid the inter-worker traffic inside one node which is only suitable for lightly loaded topologies. There is a fair number of research work on addressing heterogeneity in terms of resource and input rate in DSP [9, 12, 33, 34]. For instance, Buddhika et al. [33] introduce an on-line scheduler to mitigate the impact of interference on the performance of DSP. Liu et al. [12] proposes a runtime scheduler to redistribute the tasks according to the changes in the input rate and the computational capability. In contrast to the above-mentioned work, TurboStream is the first DSP system to optimize the intra-node IPC and exploit this feature to enable coarse-grained assignments of consolidated tasks to nodes.

VIII. Conclusion

Inter-operator communication is crucial for the performance of DSP applications. We, for the first time, highlight the two main factors contributing to the inter-operator latency, specifically the intra-node IPC and inter-node IPC traffics. We show that the latency of current state-of-the-art IPC framework is high due to the multiple memory-copy operations and the buffering mechanism. We therefore introduce TurboStream, a new DSP system for low latency DSP applications. TurboStream reduces the end-to-end latency of intra-node IPC by reducing memory-copy operations and waiting time of each single message, thanks to its novel off-heap buffer design, i.e. OSRBuffer. Moreover, it exploits the dependencies between operators to consolidate the communicating tasks before assigning them to nodes, thus reduces the inter-node IPC traffic. We implemented TurboStream on JStorm. Experimental results show that TurboStream substantially improves latency of DSP applications. As future work, we plan to investigate alternative approaches to the shutdown-start approach and to study TurboStream with more applications and on real cloud platforms.

Acknowledgment

This research is supported by National Key Research and Development Program under grant 2016YFB1000501 and the ANR KerStream project (ANR-16-CE25-0014-01). Shadi Ibrahim and Song Wu are the corresponding authors of the paper.

References

[1] Data Never Sleeps, https://www.domo.com/learn/data-never-sleeps-4-0/.

[2] Apache Spark Streaming Project, http://spark.apache.org/streaming/.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'14. New York, NY, USA: ACM, 2014, pp. 147–156.

[5] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'15. New York, NY, USA: ACM, 2015, pp. 239–250.

[6] Alibaba JStorm Project, http://www.jstorm.io/.

[7] JStorm in Alibaba, https://yq.aliyun.com/articles/62693.

[8] Netty Project, https://netty.io/.

[9] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware'15. New York, NY, USA: ACM, 2015, pp. 149–161. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814808

[10] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 49–49.

[11] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer, "Soda: An optimizing scheduler for large-scale stream-based distributed computer systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware'08. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 306–325.

[12] Y. Liu, X. Shi, and H. Jin, "Runtime-aware adaptive scheduling in stream processing," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 14, pp. 3830–3843, 2016.

[13] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ser. ICDCS'14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 535–544.

[14] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 462–469.

[15] H.-W. Jin and D. K. Panda, "Limic: Support for high-performance MPI intra-node communication on linux cluster," in *Proceedings of the 2005 International Conference on Parallel Processing*, ser. ICPP'05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 184–191.

[16] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Locality and topology aware intra-node communication among multicore CPUs," in *Recent Advances in the Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 265–274.

[17] Who Are Using JStorm, https://github.com/alibaba/jstorm/issues/81.

[18] Yahoo Streaming Benchmarks, https://github.com/yahoo/streaming-benchmarks.

[19] Intel Storm Benchmark, https://github.com/intel-hadoop/storm-benchmark.

[20] M. Yang and R. T. Ma, "Smooth task migration in apache storm," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'15. New York, NY, USA: ACM, 2015, pp. 2067–2068. [Online]. Available: http://doi.acm.org/10.1145/2723372.2764941

[21] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS'13. New York, NY, USA: ACM, 2013, pp. 207–218.

[22] J. D. Valois, "Lock-free linked lists using compare-and-swap," *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 214–222, 1995.

[23] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161036

[24] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "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*, ser. SIGMOD'13. New York, NY, USA: ACM, 2013, pp. 725–736. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465282

[25] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 49–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643640

[26] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS'14. New York, NY, USA: ACM, 2014, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611294

[27] Storm Throughput Test, https://github.com/stormprocessor/storm-benchmark.

[28] Apache Hadoop Project, http://hadoop.apache.org/.

[29] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over infiniband," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[30] Y. Wang, C. Xu, X. Li, and W. Yu, "JVM-bypass for efficient hadoop shuffling," in *Proceedings of 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 569–578.

[31] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 521–530.

[32] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[33] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, Dec 2017.

[34] Z. Weng, Q. Guo, C. Wang, X. Meng, and B. He, "Adastorm: Resource efficient storm with adaptive configuration," in *Proceedings of 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 1363–1364.