An Adaptive Online Scheme for Scheduling and Resource Enforcement in Storm

Shengchao Liu[12], Jianping Weng[12], Jessie Hui Wang[12], Changqing An[12], Yipeng Zhou[3], Jilong Wang[12]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University, China
[2]Beijing National Research Center for Information Science and Technology, China
[3]Department of Computing, Macquarie University, Australia

***Abstract*—As more and more applications need to analyze unbounded data streams in a real time manner, data stream processing platforms, such as Storm, have drawn attention of many researchers, especially the scheduling problem. However, there are still many challenges unnoticed or unsolved. In this paper, we propose and implement an adaptive online scheme to solve three important challenges of scheduling. First, how to make scaling decision in a real time manner to handle fluctuant load without congestion? Second, how to minimize the number of affected workers during rescheduling while satisfying the resource demand of each instance? We also point out that stateful instances should not be placed on the same worker with stateless instances. Third, currently the application performance cannot be guaranteed because of resource contention even if the computation platform implements an optimal scheduling algorithm. In this paper, we realize resource isolation using Cgroup, and then the performance interference caused by resource contention is mitigated. We implement our scheduling scheme and plug it into Storm, and our experiments demonstrate in some respects our scheme achieves better performance than state-of-the-art solutions.**

***Index Terms*—resource allocation, scheduling, stream processing, Storm.**

I. Introduction

Nowadays, various sources such as hardware sensors or software applications are producing continuous data streams of large volume. People from industry and academia would like to be able to collect and analyze these data streams to retrieve valuable knowledge or detect anomalies in a real time manner [1] [2]. Therefore, in these years many stream processing programming models and computation platforms are proposed, such as STREAM [3], Borealis [4], System S [5], StreamBase [6], InfoSphere Streams [7], S4 (Simple Scalable Streaming System) [8], D-Stream [9], Storm [10], Flink [11], Heron [12], and Samza [13]. In most current streaming systems, in order to run an application, one user needs to represent the logic of the application as a directed acyclic graph (DAG) and specify its requirement. An application DAG is a set of interconnected *operators*, with each operator encapsulating the semantic of a specific operation, *e.g.* receiving incoming tuples, conducting a computation, or generating new outgoing tuples. The directed edges in the DAG indicate how the tuple streams are routed to be processed. If the user code of an operator is to compute a result for multiple tuples it receives, the operator needs to maintain some states when it is running, therefore it is defined as a *stateful operator*. Otherwise, the operator is defined as a *stateless operator*.

Corresponding author: J.H. Wang (jessiewang@tsinghua.edu.cn).

The DAG models an application logically. In the physical layer, for parallel execution, each operator needs to be appropriately replicated in multiple instances that split and process the arriving tuples simultaneously. The number of instances is referred to as the *parallelism degree* of the operator. These instances are executed in one or multiple *worker* processes on multiple physical machines, *i.e. worker nodes*.

The capacity of one streaming application, *i.e.* the maximum data rate it can handle without congestion, can be affected by many factors, such as the parallelism degree of each operator, the capacity of each instance, and the assignment of each operator instance to workers and worker nodes. Since the data stream may arrive at the application at a fluctuant rate, naturally we would like to see the capacity of the application can adapt dynamically to the real-time rate of the data stream. In other words, it is valuable for a streaming processing platform to incorporate an automatic and dynamic adaptive scheduling and enforcement algorithm to determine all the factors mentioned above to guarantee the performance of applications and exploit resources of physical infrastructure efficiently.

The scheduling problem has drawn attentions from many researchers. However, the problem has not been well solved yet. For example, currently Storm is not able to adjust the capacity of its applications according to their data arrival rates automatically and adaptively. Although Storm users can set parallelism degrees for operators in their topologies using APIs at runtime, there are several problems. First, it is not transparent to Storm users, *i.e.*, users need to run the API frequently by themselves to change the configuration of their applications. Second, Storm users may not be aware of the optimal parallelism degrees, and the assumption frequently used here by researchers, *i.e.* the capacity is linear with parallelism degree, may not be true. Third, the assignment of instances to workers and worker nodes is done by the default scheduler *EvenScheduler* of Storm periodically each 10 seconds, but the scheduler just assigns executors to the configured number of workers in a round-robin manner with the aim of producing an even allocation, which may not be efficient. Forth, during migrations caused by re-scheduling, the application performance would degrade and the migration cost must be considered. Furthermore, applications of different Storm users may have resource contention, which means their

application performance cannot be guaranteed.

In this paper, we propose an adaptive online scheme for scheduling and resource enforcement on streaming processing frameworks. In particular, we make the following contributions.

- Our scheme can make scaling decisions, *i.e.* determining the amount of resources needed by each instance, in a more timely manner to handle unexpected spikes of load without congestions and resource wastes.
- We pinpoint that instances of stateful operators and stateless operators should not be colocated at the same worker in current implementations of pause-resume migration approaches, and we propose a resource-cost-aware placement algorithm that minimizes the number of affected workers.
- Our scheme can enforce the resource allocation decisions made by our scheduler and mitigate performance interference caused by resource contention by isolating instances using Cgroup [14].

We design our scheduler, implement and integrate it into Storm. Experiments based on a popular topology *WordCount* and a realistic application demonstrate that our algorithm can adapt the capacity of one topology to its stream data arrival rate and produce scheduling and deployment strategies that achieve better performance, *e.g.* shorter completion time, larger throughput and less loss tuples, compared to some state-of-the-art solutions.

The remaining part of the paper is organized as follows. Section II presents an overview of previous related work. Section III describes the scheduling problem formally, and introduces the framework and algorithms of our scheme. In Section IV, we describe how we implement our scheme on Storm, especially how we monitor the running status of topologies and how we guarantee isolation using Cgroup. In Section V, we conduct experiments using WordCount topology and a realistic application topology and present various performance statistics. We also demonstrate the necessity of resource isolation by running an experiment. Section VI concludes our work.

II. Related Work

In recent years, the scheduling problem has drawn the attention of many researchers, and there have been some research efforts to revise the scaling and deployment of topologies periodically at runtime, with the goal to optimize performance for stream processing applications with fluctuant loads. We summarize these research efforts from the following three aspects of the scheduling problem, *i.e.*, scaling decisions, placement strategies and migration issues.

Scaling Decisions. As the arriving load fluctuates, the amount of resources allocated to the streaming processing application should adapt to the load to ensure that the processing performance is acceptable while the resources are utilized efficiently. Current works differ from each other in terms of scaling dimensions and the method to determine the proper scaling decision.

Many works try to scaling in or out by adjusting the number of replications, *i.e.* parallelism degree [15][16][17][18]. The authors of [15] focus on IBM's System S and make its operators to be scalable. In [16], the authors calculate the required parallelism degree for each component based on the estimation of its current capacity and the prediction of data arrival rate in the next time window. In [17], once an operator is found to be a potential bottleneck, the system would try to increase the number of tasks by 1 for the operator. In [18], if the estimated CPU usage per replication of one operator is beyond a predefined threshold region, its parallelism degree would be reconfigured.

There are also some other scaling dimensions. In [19], the authors propose a topology-based scaling mechanism. When a topology is overloaded, it scales by adjusting the number of the cloned topologies or replaced by another new topology with more tasks. In [20], the authors regulate the number of used cores and the CPU frequency through the DVFS (Dynamic Voltage and Frequency Scaling) function offered by modern multicore CPUs.

The reconfigurations of parallelism degrees would cause restarts of involved workers and degrade the application performance. The topology substitution method would also result in a long unavailable time period. Furthermore, because of resource contention, it cannot be guaranteed that the application performance is improved as expected when they adjust parallelism degrees of operators or topologies. In fact, it is regarded to be hard to formally model the resource contention among operators or topologies [17].

The scaling of the CPU frequency proposed in [20] is only suitable for the case in which the underlying architecture is dedicated to the execution of one elastic parallel operator. It cannot work if more than one operators or multiple applications are sharing the resources.

In this article, our scaling dimension is the resources allocated to each replication. Storm version 1.1.1 has integrated a resource-aware scheduler R-Storm, which enables Storm users to specify the resource demand of each task [21]. However, the resource demand should be specified before the topology is started, and it cannot be tuned during running. Furthermore, R-Storm cannot guarantee that each component really gets the specified demand during running. In our work, we exploit Cgroup [14] to isolate instances of components. Our solution can mitigate interferences caused by resource contention among workers and executors, and it can work well for streaming applications running on shared infrastructure such as public clouds.

The method to determine the proper scaling decision can be based on profiling approaches [17][18] or analytical models [20][22]. In [17] and [18], the authors exploit the profiling approach to build the relationship between resource provisioning and performance metrics of application. The profiling results are then used to evaluate different scaling configurations, and the best configuration would be selected. In [20] and [22], the authors predict the performance metrics, *e.g.* mean service time and mean waiting time, under different resource provisioning using queueing theory. Profiling is argued in [17] to be able to provide more reliable results via real experiments than analytical abstract models. But the profiling phase takes time.

Similar to [16], we predict the future load of one operator using two models, *i.e.* the regression of its load in previous sliding windows and the load of its parent operators. However, [16] just assumes that the capacity of one component increases linearly with its parallelism degree, and it does not consider that the amount of resources each replication really acquires may also vary as the parallelism degree changes, while we calculate the resource demand of each replication directly and enforce the allocated resources of each replication using Cgroup. Comparing to the profiling approaches and the queueing theoretic models, our method can adapt to strongly fluctuating load in a more timely manner. The profiling or modelling results can be integrated into our scheme if desired, and the integrated schemes can also enjoy the benefits brought by scaling resources per replication and isolating resource allocations using Cgroup.

Besides calculating resource demands, there are some other issues in making scaling decision. For example, in [23], the authors want to find the right point in time to scale in/out. In [24], the authors focus on the application of spatial preference queries and point out that the auto-scaling system should be with a two-level architecture to cope with fluctuations of different time-scales.

Placement Strategies. The replications of operators should be placed on available physical servers to be executed, and how to place them also impacts the application performance. In summary, placement strategies can be *resource-aware*, *traffic-aware*, and *cost-aware*.

Resource-aware means that only physical servers with sufficient resources are considered when we place a replication. In addition, some placement strategies would like to see that the load is spread evenly among servers, while some other strategies would like to minimize the number of occupied servers. Traffic-aware means that we always want to minimize the latency caused by transferring tuples from one operator to its child operators. Basically, inter-node communication is slower than inter-process communication, which is slower than intra-process communication. Cost-aware means we should consider the cost caused by moving replications or operators from one place to other places.

Many placement strategies are both resource-aware and traffic-aware [25][26][21][27][28]. In [25], Aniello *et al.* notice that in some topologies where the computation latency is dominated by tuples transfer time, limiting the number of tuples that have to be sent and received through the network can contribute to improve the performances. Therefore, they propose to place in the same worker executors that communicate each other with high frequency. The authors of [26] sort executors by their loads and solve the minimization problem for each executor one by one to find its server to minimize inter-node and inter-process traffic while ensuring no worker node is overloaded. They argue there should be a single worker on one worker node for one topology. It may minimize communication traffic, but it places stateful executors and stateless executors on the same worker, which brings out a negative effect of unnecessary restarts of stateful executors. In [21], the authors select the node with the smallest Euclidean distance, which is defined based on resource demand of the replication, resource availability of the target node, and the bandwidth between the target node and the predefined Ref node. In [27], the authors formulate an Integer Linear Programming problem which takes explicitly into account the heterogeneity of computing and networking resources. In [28], the authors formulate a Mixed-Integer Linear Program to find the placement that minimizes the load imbalance among nodes, and they also exploit heuristics to colocate two replications with most frequent communications.

If the placement is re-evaluated and revised periodically, the cost of re-assignment should be considered. As an instance is migrated from its current worker to a new worker, both involved workers must be restarted in Storm, and we need to pause the streams directed to the migrating instance and save the tuples in transit, so to replay them as soon as the migration is completed. Of course during migration the application performance degrades. The migration cost would be more severe if the migrating instance is for a stateful operator, because we have to extract the current state from the old instance and restore it within the new instance. Furthermore, the sequence of migrations must be designed carefully when a placement transition involves migrations of multiple instances or operators [29].

The migration of executors draws attentions of researchers, but they mainly focus on how to guarantee that the migration is completed in a non-destructive way and all states of the bolt should be preserved [30][31][32][33][34]. The work [30] is published in 2004, and it studies the migration problem of a database stream query engine. Roughly speaking, the migration methods can be classified into two categories. The first category exploits a pause-and-resume approach that would block execution during migration[31][32]. In [26], the authors propose to introduce certain delay to ensure old workers are not stopped until new workers are ready, which is helpful for smooth migration procedure. The second category tries to avoid any pause by maintaining more states, which requires more overhead. In either category of methods, the migration would degrade application performance. In [33], the authors focus on SDF (synchronous data flow) based stream programs and propose a suite of compiler and runtime techniques for live reconfiguration without periods of zero throughput. In [34], the authors propose two different migration approaches and conduct experiments to study their performance. However, in [35], the authors study five well-known frameworks and find that none of these frameworks has not addressed all of the issues in state management. Therefore we should avoid unnecessary migrations, especially the instances of stateful operators.

Some research efforts on scaling and placement are cost-aware, *i.e.* taking migration cost into consideration. In [28], the authors assume that the migration cost of one executor is linear with the size of its state. By adding a constraint in the formulated optimization problem, they ensure that the total migration cost is less than a pre-defined threshold. In [36], the authors combine the different costs into a single cost function, such as reconfiguration cost, performance penalty and resource cost, and then the estimated cost and prospective benefit of the reconfiguration are used to update the Q function

in their reinforcement learning approach. Cardellini *et al.* also investigate the problem of how to minimize migration cost while satisfying the application requirements [37]. They formulate an optimization problem which aims to optimize a weighted sum of the normalized QoS attributes including the downtime.

In this work, we just try to minimize the number of affected workers, especially the workers of stateful instances. We design two algorithms to determine how to deal with overloaded nodes and how to scale in.

There are some works focusing on the placement problem in slightly different scenarios. The authors of [38] focus on QoS-aware operator placement in geographically distributed Storm which is operating in a dynamic environment, and they propose a hierarchical distributed architecture. In [39], Xu Le *et al.* propose a scheme named *Stela* to solve the deployment optimization problem when a Storm user wants to add or remove servers (worker nodes). The operator placement issue also appears in Mobile Distributed Complex Event Processing (DCEP) systems. In such systems, there are more challenges caused by mobility, such as limited energy and moving data producer or consumers [40].

III. Scheduling Problem and Solution

In the logical layer, users need to specify each of their applications as a directed acyclic graph (DAG) and submit the DAG to the processing framework. Each DAG can be denoted as $G(N, E, T)$, wherein N represents the set of all operators in this application, E is the set of all directed edges in G, and T is a vector which includes the number of tasks for each operator. Let us assume N_s and N_t are two operators, *i.e.* $N_s, N_t \in N$. The edge from N_s to N_t is denoted by $E_{s,t}$, and $E_{s,t} \in E$ indicates that tuples outputted from N_s would be routed to N_t for further processing. The numbers of tasks for N_s and N_t are specified in T_s and T_t. All these information should be specified by users before the application is started and cannot be changed during running.

After the application is submitted, in the physical layer, it is the responsibility of the streaming processing framework to determine how to schedule and execute all the tasks on its available servers, where each server is a worker node with limited resources. In this work, we focus on computing-intensive applications, therefore we mainly consider CPU resources. The problem of finding a solution to schedule and execute all the tasks on its available servers is referred to as the *scheduling problem*. Each streaming processing framework has one or multiple schedulers to take the responsibility of solving the problem.

Let us take Storm as an example to illustrate the physical layer. As shown in Figure 1, in Storm, each worker node is configured with a limited number of *slots*, which are basically ports used by workers to receive messages. The number of slots on each server is pre-configured by Storm operators and cannot be changed after Storm is launched. Typically, it can be set to the number of cores on the server. Each worker must be mapped to a slot to receive message, therefore the number of slots is in fact the maximum number of workers that can be run on this worker node. Please note that there can be multiple topologies running on Storm and the limited number of slots are partitioned by the workers of all these topologies. One topology G can spawn arbitrary number of workers on a worker node, as long as the topology can get free slots. Each worker spawns a number of executors to run the tasks. Each task corresponds to only one executor, but one executor can contain one (by default) or multiple tasks of one operator. We can see that the number of executors has to be smaller than or equal to the number of tasks, otherwise there would be executors without tasks to execute. As a result, Storm users always tend to overestimate the number of tasks when they specify topologies in order to make sure they can increase the number of executors without modifying their topologies when necessary.

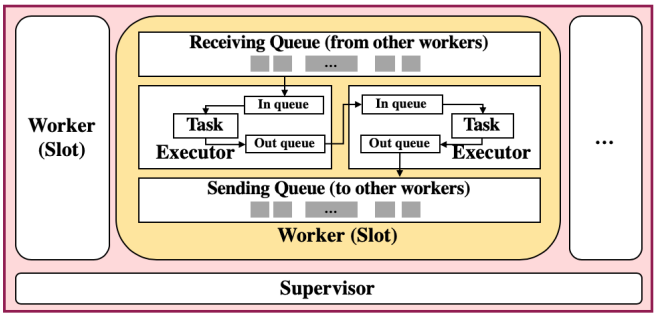

Fig. 1. One worker node in Storm.

Since both the arriving load and running environment can be dynamic, one streaming processing framework may need to change its scheduling decision to optimize the application performance periodically.

After rescheduling, some instances might be assigned to workers/nodes different from the previous assignment. It is called as *migrations*. The application may suffer performance degradation during migrations, *i.e.*, some instances become unavailable due to restarts. Therefore we should reduce the number and influence of migrations as much as possible.

In order to develop algorithms to solve the scheduling problem, we should understand the factors that can affect the running performance of one streaming application. Two key factors are listed as follows.

1) the number of instances for one operator and the amount of resources allocated to each instance.
   It is referred to as *scaling decision* in this paper. There are some constraints when we make scaling decision. The number of instances for one operator has to be smaller than or equal to the number of tasks specified by the DAG. Each instance can use only one CPU core at most. Roughly speaking, in this work we propose to spawn the maximum number of instances to enable the largest amount of resources for future uncertainty and adjust the resources per instance dynamically to accommodate fluctuating loads. We also point out the resource allocation per instance should be enforced to guarantee the system is running as expected in a resource-aware scheduling scheme.

2) the assignment/placement of each instance to workers and worker nodes.
It is referred to as *placement decision* in this paper. There are at least three considerations during placement, *i.e.*, each instance must be able to acquire sufficient resources, the migrations caused by rescheduling should also be minimized, and the cost caused by communication among instances should be minimized if possible. The consideration about communication cost makes us to propose to use least worker nodes and workers, as long as the worker nodes can provide sufficient resources.

Based on the above analysis, we design our algorithms to make scaling decision and placement decision.

A. Scaling Decision

We compute our scaling decision periodically. At t_i, we collect the data of running status during the $(i-1)th$ timeslot, *i.e.* $[t_{i-1}, t_i]$. Now we need to predict the number of tuples each operator needs to process during ith timeslot $[t_i, t_{i+1}]$ and then derive the amount of resources each operator demands during this timeslot.

1) Load Prediction: It is a usual way to predict the future load of an operator C_j from its loads in previous ϕ timeslots with a linear regression model. Let us assume the arrival rate of tuples to C_j in timeslot k is L_k^j. Mathematically, at t_i, we compute the linear coefficients a_i^j and b_i^j by applying the linear least squares method as follows:

$$(a_i^j, b_i^j) = \underset{a,b}{\operatorname{argmin}} \sum_{k=i-\phi}^{i-1} (L_k^j - (a \times t_k + b))^2.$$

Let us define the predicted value of L_i^j as $\bar{L}_i^j$. We compute $\bar{L}_i^j$ as follows:

$$\bar{L_i^j} \approx a_i^j \times t_i + b_i^j.$$

Basically, here we assume the variation of load during the timewindow $[t_{i-\phi}, t_{i-1}]$ is similar with the variation of load during the timewinow $[t_{i-\phi+1}, t_i]$, which is a kind of self-similarity. Many traffic time series in the Internet exhibit characteristics of self-similarity [41] [42] [43] [44].

The above method provides an acceptable way to predict future values when we have no more information about future. For operators except spouts (*i.e.* operators receiving tuples from data sources), it is possible for us to predict more accurately based on more available information. We say that C_p is a parent operator of an operator C_j if tuples outputted from C_p would be routed to C_j for further processing. Basically, it can be noticed that the incoming tuples to one operator are in fact outputted by all of its parent operators. We know that by default a tuple would be considered to be failed if it is not completed by the whole topology within 30 seconds. From it, we can see that in most cases the time period one tuple stays in one operator would not be long. Therefore, we have the following approximation for each non-spout operator C_j:

$$\widetilde{L_i^j} = \sum_{C_p \in \mathbb{P}_j} (\psi_i^p * \overline{\rho}_i^p), \quad where \quad \psi_i^p = L_i^p + \overline{\delta}_i^p. \tag{1}$$

Here, $\mathbb{P}_j$ is the set of all parent components of C_j. ψ_i^p is the number of tuples C_p needs to process in the timeslot i, which can be calculated as the prediction of C_p's incoming load plus the length change of its pending queue, *i.e.*, $L_i^p + \overline{\delta}_i^p$. The upper equation means that the number of incoming tuples to C_j is the sum of tuples produced by all C_j's parent operators. $\overline{\rho}_i^p$ is the average number of tuples C_p emits to next operators for each tuple it processes during $[t_i, t_{i+1}]$. Therefore, the number of tuples C_p sends to its child operator C_j can be calculated as $\psi_i^p * \overline{\rho}_i^p$.

There are two important variables in the equation, δ and ρ, and we estimate their values using the average of historical measurements as follows:

$$\begin{aligned} \overline{\rho}_i^p &\approx \frac{\sum_{k=i-\phi}^{i-1} \rho_k^p}{\phi} \\ \overline{\delta}_i^p &\approx \frac{\sum_{k=i-\phi}^{i-1} \delta_k^p}{\phi}. \end{aligned}$$

In order to avoid congestions as much as possible, we propose to use the maximal value between $\bar{L}_i^j$ and $\widetilde{L_i^j}$ as the final prediction of load, *i.e.*, $L_i^j = \max(\bar{L}_i^j, \widetilde{L_i^j})$.

2) Resource Demand: Statistically, for an operator of a computation-intensive application, its capacity, *i.e.*, the number of tuples it processes, is linear with the amount of CPU resources it utilizes. Let R_k^j denote the amount of CPU resources C_j utilized during kth timeslot, which can be monitored using Cgroup as described later. We have known that the number of tuples C_j has processed in this timeslot is ψ_k^j from monitoring. Therefore, we can estimate the linear coefficient, *i.e.*, the average CPU resources that C_j utilizes to process one tuple during recent ϕ timeslots, as follows:

$$\overline{\kappa}_i^j \approx \frac{\sum_{k=i-\phi}^{i-1} (R_k^j / \psi_k^j)}{\phi} \tag{2}$$

Therefore, the CPU demand of C_j in the timeslot i can be calculated as

$$R_i^j = \psi_i^j \times \overline{\kappa}_i^j. \tag{3}$$

The operator C_j has T_j instances, therefore r_i^j, the amount of resources that each instance requests, is as follows:

$$r_i^j = \frac{R_i^j}{T_j} \tag{4}$$

Allocating exactly the resource share of r_i^j to instances may result two potential issues. As a queue system with a huge number of tuples, the system tends to be less stable if the utilization rate keeps 100 percents. Furthermore, in terms of queue length, small deviations from theoretical ideal values are normal in a practical environment. It would result in

unnecessary minor changes of resource demand. Therefore, we adjust r_i^j in the following ways before enforcement.

First, we increase the resource share aggressively but decrease conservatively. If r_i^j is larger than r_{i-1}^j, we immediately implement r_i^j. Otherwise, we stay at r_{i-1}^j as long as the accumulated decrease is less than $\theta/2$. Second, we always assign a little more resources to each executor to improve the stability of the system. In details,

$$\widehat{r}_i^j = \left\lceil \frac{r_i^j + \frac{\theta}{2}}{\theta} \right\rceil \times \theta. \tag{5}$$

It can be viewed as we use θ percent of a single CPU core as the granularity of resource allocation. Roughly speaking, decreasing the granularity of resource allocation should be beneficial for reducing resource waste, but increase the risk of instability in extreme cases. θ can be set according to the maximum acceptable (affordable) resource waste per instance.

3) The Enforcement of Resource Allocation: It is still a problem whether the instance can get its allocation share during running, because there are many instances from one topology or multiple topologies on one worker node and these instances are competing for resources. Therefore, we have to enforce our allocation and avoid the uncertainty caused by resource contention. We know that Cgroup can control and monitor the resource usage of all threads. We propose to use Cgroup to make sure that any instance cannot use more resources than its allocation share. Since all instances only use their own shares, performance interference caused by resource contention can be removed.

B. Placement Considering Resource and Migration

Now we need to decide the placement of these instances of different operators, *i.e.*, the mapping of each instance to worker node and worker.

There are two cases in which the placement should be revised. First, some nodes would be overloaded in t_i due to the increasing demand of their own instances. Second, the number of nodes can be reduced due to the decreasing total demand of instances. Roughly speaking, the first case happens when the arriving load is becoming heavier, while the second case happens when the arriving load is becoming lighter.

The revision of placement would result in migration of some instances. As one instance is migrated, its source worker and destination worker would be restarted, and the application performance would degrade. Such kind of migration cost must be considered, therefore we should determine the placement of t_i based on its placement of t_{i-1} and try to minimize the number of affected workers.

Furthermore, the migration of stateful instances takes longer time than migration of stateless instances [32], because their states have to be saved and restored before and after migrations [45] [46]. Therefore, we should try to avoid migrations of stateful instances.

Besides the above consideration, there are some research efforts that have pointed out we should reduce the unnecessary inter-node and inter-process communications to improve the performance of one topology [26]. Therefore the total number of nodes/workers should be minimized. As a result, we propose that *there should be two workers on one worker node for one application, one for stateful instances and one for stateless instances*. In [26], in order to minimize communication cost, the authors suggest that there should be only a single worker on one worker-node. Considering the following scenario. One stateless instance e_1 needs to be migrated to other nodes, and this migration oughts to be a stateless migration. However, stateful instances are also placed on the same worker as e_1, which makes the migration turning to a costly stateful migration. Our proposal can avoid such kind of unnecessary stateful migrations.

In summary, we propose the following placement guidelines.

1) an instance can be placed on one node only if the node can provide sufficient resources;
2) there should be two workers on one node, for stateful and stateless instances separately, in order to avoid unnecessary stateful migrations;
3) the number of affected workers should be minimized during revising the placement;
4) the number of used nodes should be minimized, to minimize monetary cost and also the communication cost.

Based on these guidelines, we propose heuristic algorithms to make the placement decision. Algorithm 1 is to deal with instances that cannot get sufficient resources without migration. Algorithm 2 presents three important functions invoked by Algorithm 1 to find suitable target workers for migrations. Algorithm 3 is to find whether the number of used nodes can be reduced. We explain our ideas in these algorithms detailedly as follows.

1) migration due to overload: For an overloaded node, at least one worker would be affected to move out some instances. We try these possible plans in sequence, *i.e.*, disrupting its stateless worker, disrupting its stateful worker, or disrupting both workers. We can see that we prefer to disrupting stateless workers when necessary and we prefer to disrupting only one worker than two workers for one node.

After all overloaded nodes are checked, we can get a list of disrupted workers caused by overload directly. The resources of these disrupted workers are assumed to be released and we will determine their placement locations and allocate resources for them later. The next problem is where to place these disrupted instances to solve the resource shortage.

We first try to avoid affecting any other workers. If a node currently has no stateful (stateless) worker, or its stateful (stateless) worker is in the list of disrupted workers, it would be a good target for placing stateful (stateless) instances since no extra worker is disrupted. It is shown as the function *FindZeroAffected()* in our algorithms.

Then we try the case in which only the target worker is disrupted, which is shown as the function *FindOneAffected()*. The target worker should be also be added to the list. Note that for one node, we do not move a worker out to make room for the other worker to accommodate more instances, since in this case two workers are disrupted. As the last step, if we

still need more resources, we have to scale out and use more nodes, which is shown as the function *FindFreeNodes()*.

Now we have found sufficient resources, and we can place all instances of disrupted workers. In this step, we sort migrated instances by breadth-first search of the DAG and place sorted instances in sequence. In this way, neighboring instances are likely to be placed closely. In this article, we focus on computation-intensive applications in which communication cost minimization is with low priority.

2) try scale in: In each period, we check if the number of used nodes can be reduced. We iteratively check the least loaded node and see if we can move its workers and merge them with workers on other nodes. Note that in order to limit the migration cost, we would select only one target worker for one migrated worker.

Algorithm 1 Migrating Instances on Overloaded Nodes

$n.sl$ and $n.sf$ denote the stateless worker and stateful worker of node n. $D(w)$ denotes the resource demand of worker w.

```
//step 1: find disrupted workers due to overload.
for n ∈ usedNodes do
    if D(n.sl) + D(n.sf) > R_n then    // n is overloaded
        if D(n.sf) < R_n then
            migrateSL.add(n.sl)
        else if D(n.sl) < R_n then
            migrateSF.add(n.sf)
        else
            migrateSL.add(n.sl)
            migrateSF.add(n.sf)
RTotal^sl ← Σ_{∀w∈migateSL} D(w)
RTotal^sf ← Σ_{∀w∈migateSF} D(w)
//step 2: find target workers.
FindZeroAffected(RTotal^sf, RTotal^sl)
if RTotal^sf > 0 or RTotal^sl > 0 then
    FindOneAffected(RTotal^sf, RTotal^sl)
if RTotal^sf > 0 or RTotal^sl > 0 then
    FindFreeNodes(RTotal^sf, RTotal^sl)
//step 3: assign disrupted instances to usable nodes.
orderedInstances ← sort instances of migrateSF and migrateSL by bread-first search on DAG
orderedNodes ← sort nodes by nodeID
for e ∈ orderedInstances do
    place e on the first node with usable and sufficient resources
```

IV. Implementation in Storm

In this section, we would introduce how we implement our scheduler in Storm. Figure 2 presents the architecture of our implementation. It is mainly composed of four modules, *i.e.*, topology monitoring, scaling decision, placement decision, and resource monitor and enforcement.

- topology monitoring: it monitors the realtime demand and performance of the topology (application) and collects data to learn its running status, such as the incoming data rate, outgoing rate, execution time of tuples, and internal structure *etc*.
- scaling decision: based on the data collected, it computes the amount of resources demanded by instances.
- placement decision: it is responsible for assigning instances to slots on servers.

Algorithm 2 Find target workers to acquire resources to deal with overloaded servers.

```
function FindZeroAffected(RTotal^sf, RTotal^sl)
    for n ∈ usedNodes do
        // stateful demand first
        if RTotal^sf > 0 and (n.sf = NULL or n.sf ∈ migatedSF) then
            if n.sf = NULL then
                createWorker(n.sf)
                migrateSF.add(n.sf)
            CpuAcquire = min(RTotal^sf, n.remainingRes)
            update RTotal^sf and n.remainingRes
        // now stateless demand
        if RTotal^sl > 0 and (n.sl = NULL or n.sl ∈ migatedSL) then
            do the similar thing for stateless demand

function FindOneAffected(RTotal^sf, RTotal^sl)
    for n ∈ usedNodes do
        if RTotal^sf > 0 and n.remainingRes > 0 then
            migrateSF.add(n.sf) // n.sf would be affected.
            update RTotal^sf and n.remainingRes accordingly
        do the similar thing for stateless demand

function FindFreeNodes(RTotal^sf, RTotal^sl)
    for n ∈ notUsedNodes do
        if RTotal^sf > 0 and n.remainingRes > 0 then
            create and migrateSF.add(n.sf)
            update RTotal^sf and n.remainingRes accordingly
        do the similar thing for stateless demand
```

Algorithm 3 Try scale in to minimize the number of used servers.

The scheduler executes this algorithm to update the placement according to the predicted resource demands of executors.

```
while True do
    n ← the least loaded node //n is the node with the highest possibility to be removed
    orderedUsedNodes ← sort nodes by their remaining resources in ascending order
    for n' ∈ orderedUsedNodes do
        if n' ≠ n and n'.remainingRes > D(n.sl) then
            targetSL = n'.sl
            update n'.remainingRes accordingly
        if n' ≠ n and n'.remainingRes > D(n.sf) then
            targetSF = n'.sf
            update n'.remainingRes accordingly
    if targetSL ≠ 0 and targetSF ≠ 0 then    //remove n
        place n.sf on targetSF
        place n.sl on targetSL
    else
        return //exit because no node can be removed.
```

- resource monitor and enforcement: this module is responsible for enforcing the resources allocated to each instance by the scaling decision module.

We plugin our scheduler in Storm by implementing the IScheduler interface and modifying the *storm.scheduler* configuration in *storm.yaml* to replace the default scheduler by our scheduler. Storm has two types of processing nodes: Nimbus and supervisor. The first three modules are running on Nimbus node and complete their responsibilities using Nimbus APIs. The last module, *i.e.* resource monitor and enforcement, should

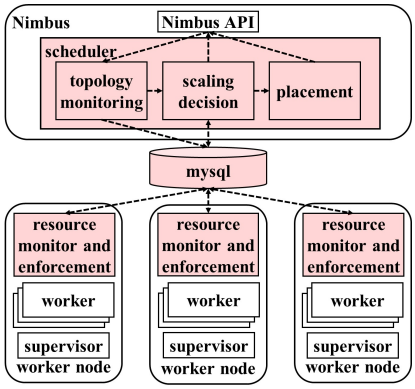

Fig. 2. Architecture of Our Implementation.

be deployed on each supervisor node.

The design of our scaling decision and placement decision algorithms has been described in Section III. This section mainly focuses on how we monitor the running status and enforce our scheduling decisions using Storm APIs.

A. Topology Monitoring

As we have stated, we need to understand the internal structure of the topology under study, monitor its running, and collect various performance statistics. The information collected by our topology monitoring module would be used to predict future load to find corresponding optimal scheduling solution (in Section III) and evaluate the performance of a scheduling algorithm (in Section V).

TABLE I
NIMBUS APIS USED IN THIS WORK

Nimbus API		Functions
C_j.get_inputs()		all operators sending tuples to C_j.
C_j.get_common()		all neighboring operators of C_j.
C_j.get_emitted()		the number of emitted tuples by C_j.
bolt	C_j.get_executed()	the number of tuples processed by C_j.
spout	C_j.get_complete_ms_avg()	average time a tuple stays in Storm.
	C_j.get_acked()	the number of tuples successfully completed.
	C_j.get_failed()	the number of tuples that timeout.

Table I lists all Nimbus APIs used in our work. Please note that Nimbus API provides statistics from the start of the topology to the time point when the API is invoked. Therefore, in order to know the statistics during $[t_i, t_{i+1}]$, we should subtract the data collected at t_i from the data collected at t_{i+1}. For ease of reading, we skip this step in the following paragraphs.

1) The internal structure of the topology: We say that C_p is a parent operator of an operator C_j if tuples outputted from C_p would be routed to C_j for further processing. Let $\mathbb{P}_j$ be the set of all parent operators of C_j, which can be get using Nimbus API as follows,

$$\mathbb{P}_j = C_j.get\_inputs().$$

In the other direction, let $\mathbb{C}_j$ be the set of all child operators of C_j. We can get $\mathbb{C}_j$ as follow,

$$\mathbb{C}_j = C_j.get\_common() - C_j.get\_inputs().$$

2) Statistics of Each Operator: We monitor the number of tuples processed and the number of tuples emitted by each operator C_j, using Nimbus API C_j.*get_executed()* and C_j.*get_emitted()*.

The tuples emitted by C_p would be sent to its child operator C_j, then we can calculate the number of tuples C_j receives as follows,

$$L^j = \sum_{C_p \in \mathbb{P}_j} C_p.get\_emitted().$$

Please note L^j should be monitored by injecting measurement logic into the codes of instances if its parent has multiple child operators. If tuples are arriving with a faster speed than C_j's capacity, some tuples would be put in C_j's queue. We monitor the changes of queue length as follows,

$$\delta^j = C_j.get\_executed() - L^j.$$

For each incoming tuple, there might be different number of tuples emitted after it is processed by C_j. It is determined by the content of the incoming tuple and the code logic of the operator. We monitor the ratio as follows,

$$\rho^j = \frac{C_j.get\_emitted()}{C_j.get\_executed()}.$$

3) Statistics of the Topology: In order to evaluate the performance of different scheduling algorithms, we need to know the running statistics of the whole topology. In this work, we monitor its average latency (complete time), throughput (the number of acked) and the number of timeout (failed). Their corresponding Nimbus APIs are listed in Table I.

B. The Enforcement of Resource Allocation

We depend on *Cgroup*, a function provided by the operating system (OS) of servers, to enforce the desired resource allocation computed by us. Each executor of Storm is in fact a thread running on a JVM of one worker node. It has three identifiers, *i.e.*, executor ID assigned by Storm, thread ID assigned by JVM, and thread ID assigned by the OS of the worker node. Cgroup runs in Linux kernel and it controls the resource usage of all threads based on thread IDs assigned by OS. However, during running, by directly calling Java library API, Storm can only get thread ID assigned by JVM. Then we have to develop a function to help Storm find out the thread ID assigned by OS for each executor.

So we choose to ask each executor to report its own thread ID assigned by the OS. After reading the code of Storm project, we notice that when a thread is spawned, a function *prepare* of the operator is invoked [47]. We rewrite the *prepare* function to include a code segment where the thread ID is reported to our database.

Now the challenge is how one executor obtains its own thread ID assigned by the OS correctly. The prepare function is written in Java and runs on jvm (java virtual machine). The API to get thread ID in Java library returns a thread ID on the jvm, which is not the thread ID assigned by the operating system. The way we get the thread ID assigned by

OS is as follows. In the prepare function, we use the JNI (Java Native Interface) to call a function written in C programming language, wherein we invoke the system call $\_\_NR\_gettid$ to obtain the thread ID assigned by OS. This task only needs to be run once when the thread is spawned, so it does not affect the performance of the topology.

After the executor reports its thread ID, the information would be stored in the database for Cgroup to use. During running, our scheme obtains a list of executor IDs on each worker node using Nimbus API, finds the thread ID for each executor ID from the database, and then controls and monitors resource usage of the executor using its thread ID.

Figure 3 illustrates our method described above to obtain and use thread IDs of executors for resource allocation enforcement. The two solid lines (with a single arrow) represent the procedure of one executor to obtain its own thread ID and report to database. It is invoked only once when the executor is spawned. The two dotted lines (with double arrows) represent the procedure of our enforcement module (Cgroup agent) to control and monitor resource usage of executors on the worker node. This procedure is invoked periodically.

V. Experiments and Performance Evaluation

We implement our proposed scheduling framework in Storm released version 1.1.1, and then we deploy the system on a cluster with three worker nodes connected using a 1000Mbps switch. Each worker node has two Intel Xeon E5-2620 v4 CPUs, *i.e.* in total each node has 16 physical CPU cores. We assign six cores for running Storm. The memory size of each node is 128G.

In the following experiments, our topology monitor keeps running and collects data every 10 seconds, which is also the time interval to run the scheduling algorithm once. ϕ is set to be 5, which means we predict the load of next timeslot from previous five time slots, *i.e.*, 50 seconds. The scaling granularity θ is set to be 20.

We conduct experiments to evaluate our scheduling algorithm using well-known data processing application (topology), namely *WordCount Topology*. As shown in Figure 4, the topology is with a linear structure which consists of one Spout and two Bolts [48]. The spout component is the source of data steaming, and it receives data from Kafka by subscribing the topic of Kafka [49] and sends lines of a text file to Split. Split bolt splits the data it receives into words and sends these words as tuples to Count bolt using fields grouping. Count bolt is responsible for counting up the occurrence of each word. Here, Spout and Split are stateless components and Count is a stateful component.

We also conduct experiments for a more realistic application and a real data stream. We monitor all http requests going through the border router of one dormitory region of a campus network. The monitor agent keeps sending tuples to our topology, and each tuple includes the information of one http request. The topology of our application is shown in Figure 5. The *Spout* receives arriving tuples from the agent on the router. The *Retrieve* component retrieves a vector of *(timestamp, destination IP)* from each tuple. All vectors are sent to each of *Country*, *ISP*, and *Prefix* components, which find out the geographical location (country), service provider (ISP) and $/16$ prefix of the destination IP respectively. The Country component emits *(timestamp, destination country)* to the *Stat.Ctry* component. Stat.Ctry counts the number of visits to each country in a recent timeslot and predicts a "normal" value based on historical data using the sliding window algorithm. If the visit number to one country within the recent timeslot is too smaller or too bigger than the normal value, it would send the information to the *Alarm* component. The component *Stat.ISP* and *Stat.Pfx* work similarly as Stat.Ctry but focus on the visit numbers to each ISP and each prefix respectively.

As the campus network is providing critical network service for tens of thousands of students and staff, we are not allowed to connect our system to the router directly. We get a collected dataset of 33 hours of December 2017 and develop a simulator to replay them for our experiments. Furthermore, in order to compare different schedulers under the same load, we also have to replay HTTP traffic.

A. Evaluation Framework and Metrics

In this work, we compare scheduling performance of our scheme with R-Storm [21]. The main reason is that R-Storm is the only resource-aware scheduler which has been built into Storm. We do not compare with the default scheduler because the default scheduler has clear disadvantages. With default scheduler, when one user submits a topology to Storm, he should specify the number of workers used by the topology and the number of executors for each component. The default scheduler would put these workers and executors evenly on all available worker nodes, which means load balancing is the most important consideration in its scheduling. In [26], the authors have pointed out the default scheduling did not consider the negative effect of communication traffic across nodes and processes and always used all available nodes regardless of workload which makes it impossible to save operational cost and electricity cost by consolidating worker nodes. In [21], the authors also pointed out the default scheduling disregarded resource demands and availability and therefore it can be inefficient at times.

In the following evaluation, the parallelism degree of components is set to be 4 for both our scheduler and R-Storm. For R-Storm, we simulate two cases. In the first case, we set the resources of each executor as 30% of one CPU core, while the resources of one executor are set to be 70% in the second case.

We simulate fluctuations of the data streaming arrival rate by controlling the speed of writing data to Kafka topic. We use the following metrics to evaluate the performance of a scheduler.

- complete time: in the past 10 seconds, the average time each tuple takes from its arrival at spout to the time point when the tuple is processed successfully by the last component.
- throughput: the number of tuples that are processed successfully in the past 10 seconds.

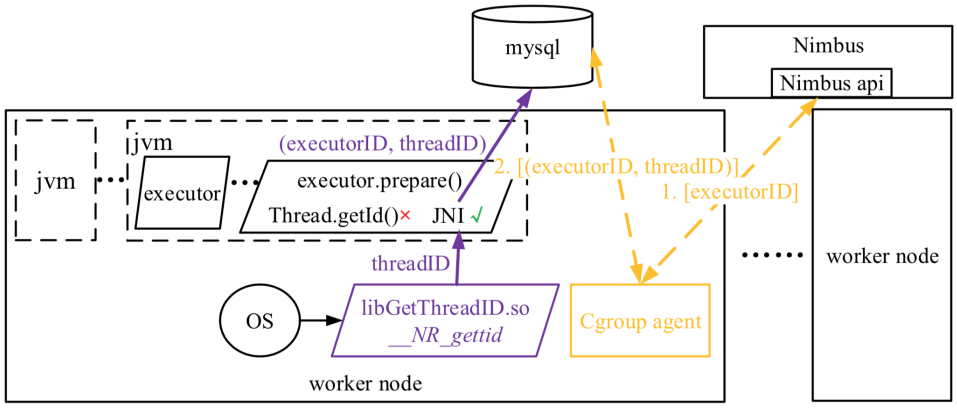

Fig. 3. Obtain Thread ID of Executors for Resource Enforcement.

Fig. 4. WordCount topology.

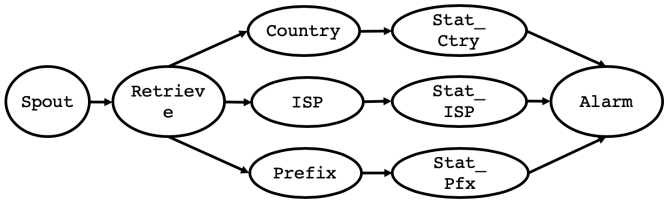

Fig. 5. The topology of HTTP traffic monitor and analyzer.

- number of failed: the number of tuples that are not completed within 30 seconds (default value of Storm) in the past 10 seconds.
- the number of workers and worker nodes the topology used in past 10 seconds.

We depend on Nimbus APIs to collect statistics of the above metrics every 10 seconds. The Nimbus APIs are *get_complete_ms_avg()*, *get_acked()*, and *get_failed()*.

B. WordCount, Regular Fluctuations

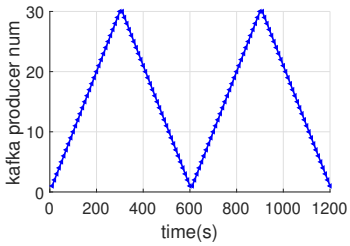

Fig. 6. Input stream with regular fluctuations.

In this experiment, the streaming rate is changing as shown in Figure 6. During the first 5 minutes, we increase the data rate linearly every 10 seconds; during the second 5 minutes, we decrease the data rate every 10 seconds. The input stream repeats these changes in the latter 10 minutes.

Figure 7 presents our experiment results of their scheduling performance. R-Storm(equal) means we configure R-Storm's parameters to make each of its operator using equal resources as our scheme. Please note in reality it is not easy for users to specify demand properly. For clarity, we further calculate the average of all metrics and show them in Table II.

From Figure 7 and Table II, we can see that our scheduler clearly outperforms R-Storm(30) in terms of all three metrics. The average complete time decreases from $13115.4ms$ to $7.6ms$, which shows an improvement of almost 1725 times. The throughput increases from 93897 to 140995, where the improvement is 1.5 times. The number of failed tuples decreases from 4629 among 93897 to 699 among 140995, where the improvement is about 10 times.

Allocating more resources to each executor obviously can improve the performance of the topology. Comparing our scheduler with R-Storm(70) and R-Storm(equal), we can see that they have smaller number of failed tuples, but our scheduler achieves a smaller complete time and a similar throughput. The failed tuples of our scheme is caused by migrations, and incorporating a loss-free migration mechanism can solve the issue. On the other hand, R-Storm(70) consumes much more resources than our scheme. R-Storm(70) always allocates 70% of a CPU core to each executor, therefore it needs $12 \times 0.7 = 8.4$ (3 components, each with 4 executors) CPU cores in total during running, which means two worker nodes are always demanded.

In terms of the number of workers and worker nodes, we can see with our scheduler the topology only needs one worker node and two workers during most of the running time. In average, it uses 2.6 workers and 1.6 worker nodes. Using less workers and worker nodes can reduce inter-node and inter-process communication cost.

Figure 8 presents CPU utilizations of executors of three components during the running of our scheduler. For executors of one component, we plot our prediction (Equation 4), our allocation (Equation 5), and its real utilization rate (collected by our monitor). We can see they are trying to adapt to the fluctuating incoming data rates.

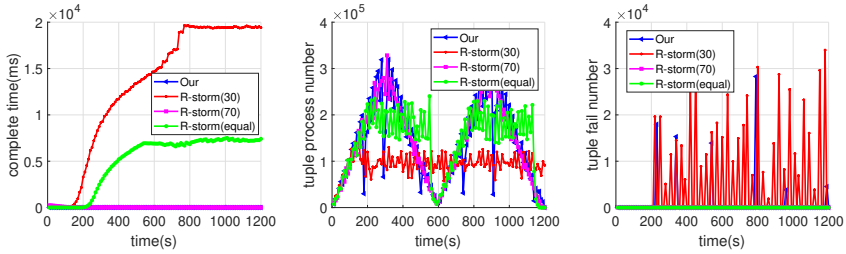

Fig. 7. Performance statistics of three schedulings (Regular).

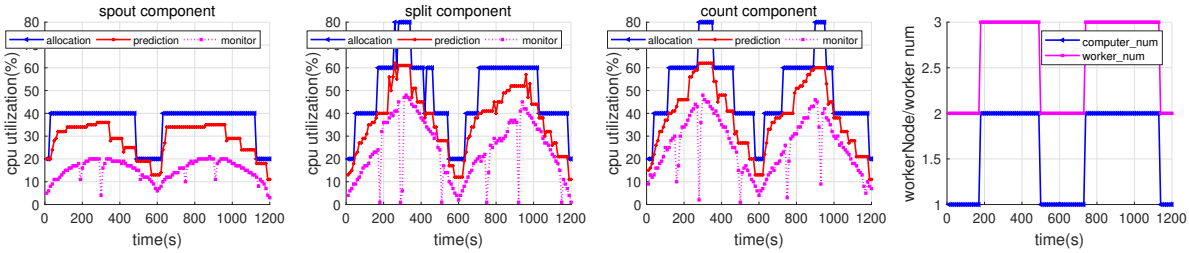

Fig. 8. Resource allocation and real usage of three components (Regular).

TABLE II
AVERAGE PERFORMANCE METRICS (REGULAR)

	complete time	throughput	fail number
our method	7.6ms	140995	699
R-Strom(30)	13115.4ms	93897	4629
R-Strom(70)	23.9ms	142710	0
R-Storm(equal)	5079ms	143897	0

In summary, under regularly fluctuating incoming data stream, our methods can adapt resource allocation to arrival data rate, and thus it can achieve good performance with fewer resources.

C. WordCount, Random Fluctuations

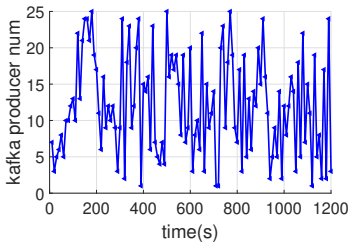

Fig. 9. Input stream with random fluctuations.

In this experiment, the streaming rate is changing as shown in Figure 9. Every 10 seconds, we generate a random integer in $[1, 25]$, which is the number of Kafka producers. In this way, we get an input stream with random fluctuations. Figure 10 presents the experiment results of scheduling performance. For clarity, we further calculate the average of all metrics and show them in Table III.

From Figure 10 and Table III, we can see that our scheduler achieves a significant improvement comparing with R-Storm(30) in terms of all performance metrics. Our scheduler experiences a longer complete time than R-Storm(70), but it has been 30 times better than R-Storm(equal). The throughput is similar. Again, the failed tuples of our scheme is caused by migrations, and incorporating a loss-free migration mechanism can solve the issue. In terms of the number of workers and worker nodes, with our scheduler the topology only needs one worker node and two workers during most of the running time. In average, our method uses 2.3 workers and 1.3 worker nodes, which means it causes less inter-node and inter-process communication cost and has a positive influence on the overall performance, such as complete time and throughput.

TABLE III
PERFORMANCE METRICS (RANDOM)

	complete time	throughput	fail number
our method	66.5ms	97474	1225
R-Strom(30)	14263ms	85097	2830
R-Strom(70)	4.8ms	107120	0
R-Strom(equal)	1857.1ms	106940	0

D. HTTP Monitor and Analyzer

The left plot of Figure 11 shows the number of HTTP requests within consecutive 33 hours, which is the arriving load of our HTTP monitor. The performance statistics of experiments is presented in Figure 11 and Table IV.

In order to make comparison, we run R-Storm with three different parameter settings. We first run R-Storm with all available resources (6 cores $\times 3$ servers) and monitor the CPU usages during running. In this way, we can get roughly perfect knowledge of the resource demand of each operator at any

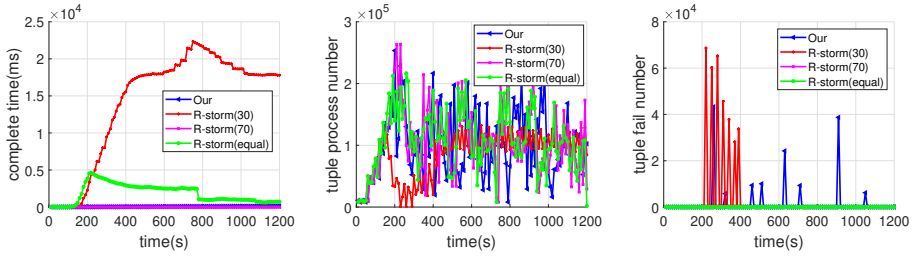

Fig. 10. Performance statistics of three schedulings (Random).

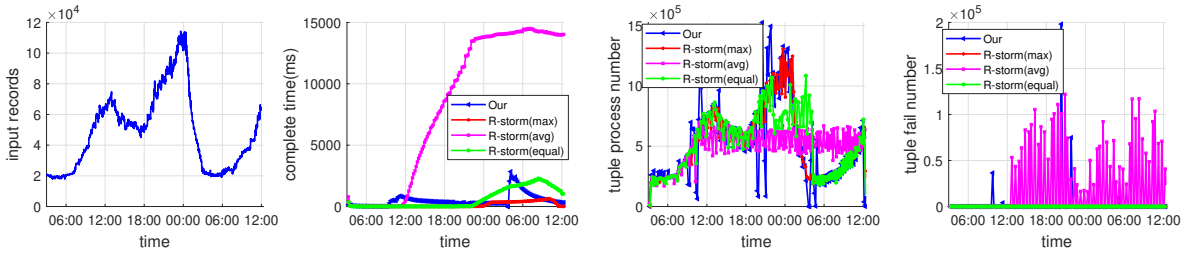

Fig. 11. Performance statistics of HTTP monitor and analyzer.

TABLE IV
AVERAGE PERFORMANCE METRICS (HTTP)

	complete time	throughput	fail number
our method	499ms	545530	1881
R-Strom(avg)	8385ms	497897	17391
R-Strom(max)	154.5ms	530859	0
R-Strom(equal)	638ms	530330	0

timepoint, which can be taken as a good reference to determine settings for R-Storm. For each operator, we calculate its average and the maximum resource demand during running, and we set them as parameters for R-Storm(avg) and R-Storm(max). Then we run our scheduler and make a record of resource allocation at any time point. We calculate the total allocated resources for each operator and set parameters to make R-Storm consuming the equal amount of resources as our scheduler. This experiment is named as R-Storm(equal).

Please note here we are assuming R-Storm has perfect knowledge on resource demand, which is impossible in real-world. To some extend it means our evaluation has been in favor of R-Storm. As we have stated before, although R-Storm allows users to specify their resource demands explicitly when they submit their own topologies, what's the optimal value to set is still an important challenge.

Our method does not require any knowledge of traffic load in advance. The improvement of complete time is 1.2 times, comparing R-Storm(equal) with our method. The throughput is even slightly better than R-Storm(max). Due to migrations without any special handling, the failed tuples of our scheme is larger, but incorporating a loss-free migration mechanism can solve the issue.

E. Necessity of Resource Enforcement

There can be multiple topologies running on one Storm platform at the same time. Currently, Storm has no mechanism to guarantee one topology really acquires the resources it requests, especially when there is resource contention among multiple topologies. That is why we propose to use Cgroup to enforce our resource allocation scheme. With Cgroup, we in fact implement the resource isolation among topologies.

We conduct an experiment as follows. In Storm, we run two topologies, A and B. The input streaming of A is with a constant rate, and we specify a sufficient amount of resources for its executors using R-Storm. B serves as a background topology in this experiment, and its input streaming rate is changing dynamically. B also specifies its resource demand using R-Storm before it is started. Our goal is to study the influence of B's changing load on A's performance, *i.e.*, the performance interference caused by resource contention.

The resulting performance statistics of A are presented in Figure 12. Obviously, the experiment with Cgroup achieves a much better performance than the experiment without Cgroup. With Cgroup, A achieves shorter complete time, larger throughput, and less failed tuples. Furthermore, these performance metrics are more stable than the experiment without Cgroup.

When there is no Cgroup for resource allocation, A's performance varies a lot, which demonstrates its performance is affected by B's dynamic load. Furthermore, although A requests a sufficient amount of resources, it still experiences a serious latency and loss rate. It is because the resources which are expected to be used by A are in fact taken over by B. Therefore, resource enforcement and isolation are necessary for us to guarantee performance of a topology.

VI. Conclusion and Future Work

In this work, we propose an adaptive online scheme to schedule and enforce resource allocation in stream processing

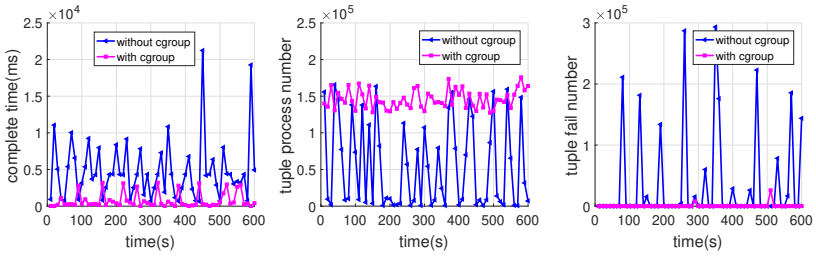

Fig. 12. Performance statistics of A with and without resource isolation/enforcement.

systems. Our goal is to make sure that the system can achieve good performance under fluctuant load, *e.g.*, less congestion under heavy load and less resource waste under light load. Particularly, we calculate the desired resource amount of each executor in a more accurate and timely manner by taking both historical data rates and internal structure of the topology into consideration. As the load fluctuates, our heuristic placement algorithm can deal with overloaded nodes and try scale-in properly which can minimize the number of affected workers and used nodes. Moreover, as far as we know, we are the first to notice the negative effect of colocating stateful executors and stateless executors. Therefore, we pinpoint that the placement algorithm should spawn two workers, instead of a single worker, for each topology, then the migration of stateless executors would not result in the restart of stateful executors. We also implement a way to enforce that executors can really acquire their resource shares allocated by our scheduler and the performance uncertainty caused by resource contention is mitigated.

We implement our scheduling scheme and plug it into Storm, and our experiments demonstrate in some respects our scheme achieves better performance than state-of-the-art solutions. We hope our research can provide some valuable insights into the scheduling problem of stream processing systems. We recognize that the mechanism to support loss-free migrations is an important research direction, since it can significantly improve the performance of adaptive schedulers.

Acknowledgment

The authors thank the editors and anonymous reviewers for taking time to review this paper and for their suggestions that helped improve this paper. This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB0801302 and the National Natural Science Foundation of China under Grant No. 61202356.

References

[1] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems & Software*, vol. 127, pp. 205–216, 2017.

[2] M. Nardelli, M. Nardelli, M. Nardelli, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *ACM Sigmetrics Performance Evaluation Review*, vol. 44, no. 4, pp. 11–22, 2017.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *Stream: The Stanford stream data manager*. Springer, Berlin, Heidelberg, July 2016, pp. 317–336.

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *CIDR*, vol. 5, no. 2005, 2005, pp. 277–289.

[5] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 431–442.

[6] "TIBCO StreamBase and the TIBCO accelerator for Apache Spark," TIBCO Software Inc., Tech. Rep., 2017.

[7] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM infosphere streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1093–1104.

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2010, pp. 170–177.

[9] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing, HotCloud'12*. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342773

[10] (2013) Storm project. [Online]. Available: http://www.storm-project.net/

[11] "Apache flink," 2018. [Online]. Available: http://flink.apache.org/

[12] "Apache heron," 2018. [Online]. Available: http://incubator.apache.org/projects/heron.html

[13] "Apache samza," 2018. [Online]. Available: http://samza.apache.org/

[14] Cgroups. [Online]. Available: https://en.wikipedia.org/wiki/Cgroups

[15] 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 (IPDPS)*. IEEE, 2009, pp. 1–12.

[16] R. K. Kombi, N. Lumineau, and P. Lamarre, "A preventive auto-parallelization approach for elastic stream processing," in *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1532–1542.

[17] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya, "A stepwise auto-profiling method for performance optimization of streaming applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 4, pp. 24:1–24:33, 2018. [Online]. Available: https://doi.org/10.1145/3132618

[18] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 572–585, March 2018.

[19] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti, "A topology-based scaling mechanism for Apache Storm," *International Journal of Network Management*, vol. 27, no. 3, p. e1933, 2017.

[20] T. D. Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, vol. 127, pp. 302 – 319, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216301467

[21] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-Storm: Resource-aware scheduling in Storm," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 149–161.

[22] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang,

"DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3338–3352, Dec 2017.

[23] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proceedings of the IEEE 30th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2014, pp. 296–302.

[24] G. Mencagli, M. Torquati, and M. Danelutto, "Elastic-PPQ: a two-level autonomic system for spatial preference query processing over dynamic data streams," *Future Generation Computer Systems*, vol. 79, pp. 862 – 877, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1730938X

[25] 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*. ACM, 2013, pp. 207–218.

[26] 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 (ICDCS)*. IEEE, 2014, pp. 535–544.

[27] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. New York, NY, USA: ACM, 2016, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/2933267.2933312

[28] K. G. S. Madsen, Y. Zhou, and J. Cao, "Integrative dynamic reconfiguration in a parallel stream processing engine," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 227–230.

[29] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '18. New York, NY, USA: ACM, 2018, pp. 136–147. [Online]. Available: http://doi.acm.org/10.1145/3210284.3210292

[30] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 431–442. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007617

[31] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[32] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in Storm," in *Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016, pp. 583–590.

[33] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe, "Gloss: Seamless live reconfiguration and reoptimization of stream programs," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 98–112. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173170

[34] A. Shukla and Y. Simmhan, "Toward reliable and rapid elasticity for streaming dataflows on clouds," in *Proceedings of 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 1096–1106.

[35] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *The VLDB Journal*, vol. 27, no. 6, pp. 847–872, Dec 2018.

[36] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171 – 185, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17326821

[37] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4334, 2018.

[38] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "Distributed QoS-aware scheduling in Storm," in *Proceedings of the 2015 ACM International Conference on Distributed Event-Based Systems*, 2015, pp. 344–347.

[39] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 22–31.

[40] F. Starks, V. Goebel, S. Kristiansen, and T. Plagemann, *Mobile Distributed Complex Event Processing—Ubi Sumus? Quo Vadimus?* Cham: Springer International Publishing, 2018, pp. 147–180.

[41] J. Aracil, R. Edell, and P. Varaiya, "A phenomenological approach to Internet traffic self-similarity," in *Proceedings of the 35th Annual Allerton Conference on Communication, Control and Computing*, 1996, pp. 1–24.

[42] R. D. Smith, "The dynamics of Internet traffic: Self-similarity, self-organization, and complex phenomena," *Advances in Complex Systems*, vol. 14, no. 6, pp. 905–949, 2011.

[43] R. Donthi, R. Renikunta, R. Dasari, and M. Perati, "Study of delay and loss behavior of Internet switch-markovian modelling using circulant markov modulated poisson process (CMMPP)," *Applied Mathematics*, vol. 5, no. 3, pp. 512–519, 2014.

[44] C. Dabrowski, "Catastrophic event phenomena in communication networks: A survey," *Computer Science Review*, vol. 18, pp. 10 – 45, 2015.

[45] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya, "E-Storm: Replication-based state management in distributed stream processing systems," in *Proceedings of the 2017 International Conference on Parallel Processing*, 2017, pp. 571–580.

[46] 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*. ACM, 2015, pp. 2067–2068.

[47] P. T. Goetz and B. O'Neill, *Storm Blueprints: Patterns for Distributed Realtime Computation*. Packt Publishing Ltd, 2014.

[48] "Wordcounttopology." [Online]. Available: http://t.cn/RajHPwZ

[49] Kafka. [Online]. Available: http://kafka.apache.org/