D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications

Xunyun Liu and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: xunyunl@student.unimelb.edu.au, rbuyya@unimelb.edu.au

***Abstract*—Scheduling streaming applications in Data Stream Management Systems (DSMS) has been investigated for years. However, there lacks an intelligent system that is capable of monitoring application execution, modelling its resource usages, and then adjusting the scheduling plan under different sizes of inputs without requiring users' intervention. In this paper, we model the scheduling problem as a bin-packing variant and propose a heuristic-based algorithm to solve it with minimised inter-node communication. We also implement the D-Storm prototype to validate the efficacy and efficiency of our scheduling algorithm, by extending the Apache Storm framework into a self-adaptive MAPE (Monitoring, Analysis, Planning, Execution) architecture. The evaluation carried out on both synthetic and realistic applications proves that D-Storm outperforms the existing resource-aware scheduler and the default Storm scheduler by at least 16.25% in terms of the inter-node traffic reduction and yields a significant amount of resource savings through consolidation.**

1. Introduction

Big data processing has now gained popularity for its ability to deal with data of big volume and high velocity. There are situations in which data received in real time needs to be processed upon arrival to keep its volatile value. For example, an investor must analyse the statistics of share market in time to avoid economic losses and a social media website has to process ongoing user posts to suggest topics on trend. To cater for these real-time requirements, stream processing emerges as a new in-memory paradigm that processes continuous, unbounded inputs on a record-by-record basis. Such once-at-a-time processing model performs independent computations on a smallish window of recent data, delivering results within merely sub-second latency.

Despite the diversity of various use cases, the majority of streaming applications in existence are built on top of a Data Stream Management System (DSMS) to reap the benefits of better programmability and manageability. Apache Storm[1], for example, is a state-of-the-art distributed DSMS implementation with supports for imperative programming language and unified data stream management model. It also offers user-transparent fault-tolerance, horizontal scalability, and state management by providing the abstraction of streaming primitives and simplifying the use of distributed resources at the middleware level.

Scheduling of streaming applications is one of the many things that should be transparently handled by the DSMSs. To maximise application performance and reduce the resource footprints, it is of crucial importance for the DSMS to schedule each application as compact as possible so that fewer resources are consumed to achieve the same performance target. This motivates the needs of resource-aware scheduling, which matches the resource demands of streaming tasks to the capacity of distributed nodes. However, the default schedulers adopted in the state-of-the-art DSMSs, including Storm, are resource agnostic. Without capturing the differences of task resource consumptions, they follow a simple round-robin process to scatter the application tasks over the cluster, thus inevitably leading to execution inefficiency due to over/under utilisation. Recently, a few dynamic schedulers have been proposed to reduce the network traffics and improve the maximum throughput for a single application at runtime [1]–[5]. However, they all share the load-balancing principle that aims to distribute the workload as even as possible. Such practice makes it impossible to consolidate resources when the input is small and the scheduling result may suffer from severe performance degradation when multiple applications are submitted to the same cluster and end up competing for the computation and network resources on each single node.

To fill in this gap, Peng et al. [6] proposed a resource-aware scheduler that schedules streaming applications based on the resource profiles submitted by users at compile time. But the problem is only partially tackled for the following reasons. (1) The resource consumption of each task is statically configured within the application, which suggests that it is agnostic to the actual application workload and will remain unchanged during the whole lifecycle of the streaming application. However, the resource consumption of a streaming task is known to be correlated to the input workload and the latter may be subject to unforeseeable fluctuations due to the real-time nature. (2) The scheduler only schedules once during the initial application deployment, making it impossible to adapt the scheduling plan to runtime changes. The existing scheduler is a static component that regards the scheduling problem as a one-time item packing process, so it only works on unassigned tasks produced by new application submission and worker failures.

In this paper, we propose a dynamic resource-efficient scheduling algorithm to tackle the problem as a bin-packing variant. We also implement a prototype named D-Storm to validate the efficacy and efficiency of the proposed algo-

[1] http://storm.apache.org/

978-1-5386-2129-5/17/31.00 ©2017 IEEE
DOI 10.1109/ICPADS.2017.00070

rithm. D-Storm does not require users to statically specify the resource needs of streaming applications, instead, it evaluates the resource consumption of each task at runtime by monitoring the volume of incoming workload. Secondly, D-Storm is a dynamic scheduler that repeats its bin-packing policy with a customizable scheduling interval, which means that it is able to free under-utilized nodes whenever possible.

The main **contributions** of this work are summarised as follows:

- We propose a dynamic resource-efficient scheduler that, to the best of our knowledge, is the first of its kind to dynamically schedule streaming applications based on bin-packing formulations.
- We design a greedy algorithm to solve the bin-packing problem, which generalises the classical *First Fit Decreasing* (FFD) heuristic. The algorithm is capable of reducing the amount of inter-node communication as well as minimising the resource footprints used by streaming applications.
- We implement the prototype on Storm and conduct extensive experiments to demonstrate the superiority of our approach compared to the existing static resource-aware scheduler.

It is worth noting that though our D-Storm prototype has been implemented as an extended scheduler on Storm, the fact that it is loosely coupled with the existing Storm modules and the design to be externally configurable makes it viable to be generalised to other operator-based stream processing systems.

The remainder of the paper is organised as follows. We provide an overview of the framework in Section 2, and then formulate the scheduling problem and present the heuristic-based algorithm in Section 3. The performance evaluation is presented in Section 4, followed by the related work and conclusions in Sections 5 and 6, respectively.

2. D-Storm Framework

D-Storm realises dynamic and resource-efficient scheduling by incorporating the following new features into the standard Storm framework:

- It tracks streaming tasks at runtime to obtain their resource usages and the volumes of inbound / outbound communications. This information is critical for the scheduler to avoid resource contention and minimize inter-node communication.
- It makes real-time scheduling decisions to pack tasks as compact as possible, which translates to reducing the resource footprints while satisfying the performance requirements of streaming applications.
- It automatically re-schedules the cluster whenever the resource contention is spotted or it finds a new scheduling plan that yields more resource savings.

To implement these features, D-Storm extends the standard Storm release with several loosely coupled modules as shown in Fig. 1, thus constituting a MAPE (Monitoring, Analysis, Planning, Execution) framework for dynamic application scheduling. Essentially, there is a feedback control process in each MAPE loop that allows for runtime-awareness and system self-adaptivity.

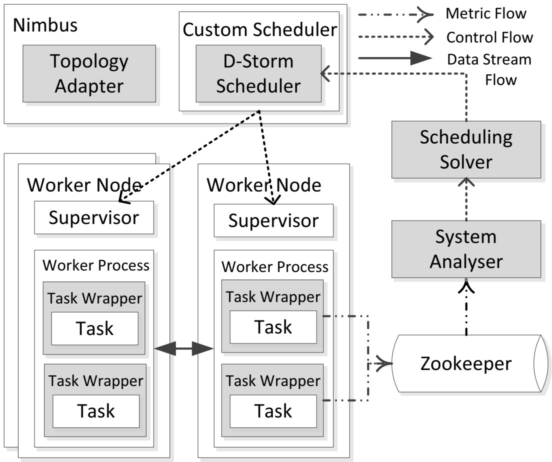


Figure 1. The extended D-Storm architecture on top of the standard Storm release, where the newly introduced modules are highlighted in grey

The feedback control process starts with the *Task Wrapper*, a monitoring module that constantly measures and reports the task resource usages and communications. Each task wrapper only encapsulates a single task following the decorator pattern, which inserts the desired functionalities to the original task logic. Specifically, it monitors the CPU usages in the *execute* method by making use of the *ThreadMXBean* class, and it logs the communication traffics among tasks by registering a custom metric consumer. It is worth noting that the collected metrics on communication and resource usages are subject to non-negligible instantaneous fluctuations due to the dynamic nature of stream processing. Therefore, the task wrapper averages the metric readings over an observation window and periodically reports the result to Zookeeper for persistence.

The analysing phase in the MAPE loop is carried out by the *System Analyser* module, which conducts boundary checks on the collected metrics and determines whether they represent a normal system state. For example, the previously decided schedule plan may result in over-utilization due to the lack of resources, as the resource consumption grows when presented with higher workloads. Analogously, the current scheduling may become an over-kill for processing the off-peak workloads, and tasks should be consolidated into a fewer worker nodes to save costs. The *System Analyser* module is responsible for identifying these over-utilization and under-utilization states through threshold checks and invoking the planning phase when necessary.

The *Scheduling Solver* comes into play when it receives the signal from the system analyser reporting the abnormal system states. It retrieves the runtime resource and performance metrics and then conducts the scheduling calculation using the algorithm elaborated in Section 3. We have designed the scheduler solver in a way that it is not to be invoked more frequently than the predefined scheduling interval, the value of which should be fine-tuned to strike a balance between the system stability and agility.

Once a new scheduling plan is made, the executor in the MAPE loop — *D-Storm Scheduler* takes the responsibility to put the new plan into effect. From a practical perspective, it is a jar file placed on the Nimbus node which implements the *IScheduler* interface to leverage the existing scheduling APIs provided by Storm.

In order to keep our D-Storm scheduling framework transparent to the user-level streaming applications, we also supply a *Task Adapter* module in the Storm core that automatically encapsulates tasks in tasks wrappers. In addition, developers can specify the scheduling parameters through this module, which proves to be an elegant way to cater for the diverse needs of different streaming scenarios.

3. Dynamic Resource-Aware Scheduling

The dynamic resource-aware scheduling in D-Storm exhibits the following characteristics: (1) each task has a set of resource requirements that are constantly changing with regard to the amount of inputs being processed; (2) each machine (worker node) has a set of available resources for accommodating tasks that are assigned to it; (3) the scheduling algorithm is executed on-demand to take into account any runtime changes in task resource requirements.

3.1. Problem Formulation

For each round of scheduling, the essence of the problem is to find a mapping of tasks to worker nodes such that the communicating tasks are packed as compact as possible. Meanwhile, the resource constraints need to be respected that in each node the resource availability is not exceeded by the resource requirements. Since the compact assignment of tasks also leads to reducing the number of used machines, we model the scheduling problem as a variant of the bin-packing problem and formulate it using the symbols illustrated in Table 1.

In this work, the resource consumptions and availability are examined in two dimensions — CPU and memory. Though memory resources can be intuitively measured in terms of megabytes, the quantification of CPU resources is usually vague and imprecise due to the diversity of CPU architectures and implementations. Therefore, following the convention in literature [6], we specify the amount of CPU resources by a point-based system, with 100 points being the full capacity of a single core. Note that a multi-core CPU can get a capacity of *num_of_cores* * 100 points, and a task that accounts for p% CPU usages reported by the monitoring system has a resource demand of p points.

As reported in [7], task τ_i's CPU and memory resource requirements can be linearly modelled with regard to the size of the current input loads, which are illustrated in Eq. (1).

TABLE 1. SYMBOLS USED FOR DYNAMIC RESOURCE-AWARE SCHEDULING

Symbol	Description
n	Number of tasks to be assigned
τ_i	Task i, $i = 1, ..., n$
m	Number of available machines (worker nodes) in the cluster
ν_i	Worker node i, $i = 1, ..., m$
$W_c^{\nu_i}$	CPU capacity of ν_i, measured in a point-based system
$W_m^{\nu_i}$	Memory capacity of ν_i, measured in Mega Bytes (MB)
$\omega_c^{\tau_i}$	Total CPU requirement of τ_i in points
$\omega_m^{\tau_i}$	Total memory requirement of τ_i in Mega Bytes (MB)
$\rho_c^{\tau_i}$	Unit CPU requirement for τ_i to process a single tuple
$\rho_m^{\tau_i}$	Unit memory requirement for τ_i to process a single tuple
ξ_{τ_i,τ_j}	The size of data stream transmitting from τ_i to τ_j
Θ_{τ_i}	The set of upstream tasks for τ_i
Φ_{τ_i}	The set of downstream tasks for τ_i
$\varkappa$	The volume of inter-node traffic within the cluster
m_{used}	Number of used machines in the cluster

$$\begin{aligned} \omega_c^{\tau_i} &= (\sum_{\tau_j \in \Theta_{\tau_i}} \xi_{\tau_j,\tau_i}) * \rho_c^{\tau_i} \\ \omega_m^{\tau_i} &= (\sum_{\tau_j \in \Theta_{\tau_i}} \xi_{\tau_j,\tau_i}) * \rho_m^{\tau_i} \end{aligned} \tag{1}$$

Having modelled the resource consumption at runtime, each task is considered as an item of multi-dimensional volumes that needs to be allocated to a particular machine during the scheduling process. Given a set of m machines (bins) with CPU capacity $W_c^{\nu_i}$ and memory capacity $W_m^{\nu_i}$ ($i \in \{1, .., m\}$), and a list of n tasks (items) $\tau_1, \tau_2, ..., \tau_n$ with their CPU demands and memory demands denoted as $\omega_c^{\tau_i}$, $\omega_m^{\tau_i}$ ($i \in \{1, 2, .., n\}$), the problem is formulated as follows:

$$\text{minimize} \quad \varkappa(\boldsymbol{\xi}, \boldsymbol{x}) = \sum_{i,j \in \{1,..,n\}} \xi_{\tau_i,\tau_j}(1 - \sum_{k \in \{1,..,m\}} x_{i,k} * x_{j,k})$$

$$\begin{aligned} \text{subject to} \quad & \sum_{k=1}^{m} x_{i,k} = 1, && i = 1, ..., n, \\ & \sum_{i=1}^{n} \omega_c^{\tau_i} x_{i,k} \le W_c^{\nu_k} && k = 1, ..., m, \\ & \sum_{i=1}^{n} \omega_m^{\tau_i} x_{i,k} \le W_m^{\nu_k} && k = 1, ..., m, \end{aligned} \tag{2}$$

where $\boldsymbol{x}$ is the control variable that stores the task placement in a binary form:

$$x_{i,k} = \begin{cases} 1 & \text{if task } \tau_i \text{ is assigned to machine } \nu_k, \\ 0 & \text{otherwise;} \end{cases}$$

Eq. (3) shows that $\boldsymbol{x}$ can also be used to reason the number of used machines as a result of scheduling:

$$m_{\text{used}} = \sum_{j \in \{1,...,m\}} \bigvee_{i \in \{1,...,n\}} x_{i,j} \tag{3}$$

3.2. Heuristic-based Scheduling Algorithm

The classical bin-packing problem has proved to be NP-Hard [8], and so does the scheduling of streaming applications [6]. There could be a massive amount of tasks involved in each single assignment, so it is computationally infeasible to find the optimal solution in polynomial time. Besides, streaming applications are known for their strict latency constraints on processing time, so the efficiency of scheduling is even more important than the result optimality to prevent the violation of the real-time requirement. Therefore, we opt for greedy heuristics rather than exact algorithms such as bin completion [9] and branch-and-price [10], which have exponential time complexity.

The proposed algorithm is a generalisation of the classical *First Fit Decreasing* (FFD) heuristic. FFD is essentially a greedy algorithm that sorts the items in a decreasing order (normally by their size) and then sequentially allocates them into the first bin with sufficient remaining space. However, in order to apply FFD in our multidimensional bin-packing problem, the standard bin packing procedure has to be generalised in three aspects as shown in Algorithm 1.

Firstly, all the available machines are arranged in descending order by their resource availability so that the more powerful ones get utilised first for task placement. This step is to ensure that the FFD heuristic has a better chance to convey more task communications within the same machine, thus reducing the cumbersome serialisation and de-serialisation procedures. Since the considered machine characteristics — CPU and memory are measured in different metrics, we define a resource availability function that holistically combines these two dimensions and returns a scalar for each node, as shown in Eq. (4).

$$\wp(\nu_i) = \min\left\{\frac{nW_c^{\nu_i}}{\sum\limits_{j\in\{1,\ldots,n\}} \omega_c^{\tau_j}}, \frac{nW_m^{\nu_i}}{\sum\limits_{j\in\{1,\ldots,n\}} \omega_m^{\tau_j}}\right\} \tag{4}$$

Secondly, the evaluation of the task priority function $\varrho(\tau_i)$ is dynamic and runtime-aware, considering not only the task communication pattern but also the node to which it attempts to assign. We denote the attempted machine as ν_m, then $\varrho(\tau_i, \nu_m)$ can be formulated as a weighted sum of two terms, which are namely: (1) the amount of newly introduced intra-node communication if τ_i is assigned to ν_m, and (2) the amount of potential intra-node communication that τ_i can bring to ν_m in the subsequent task assignments. The mathematical formulation of $\varrho(\tau_i)$ is given as follows:

$$\begin{aligned}
\varrho_1(\tau_i, \nu_m) &= \sum_{j\in\{1,\ldots,n\}} x_{j,\nu_m}(\xi_{\tau_i,\tau_j} + \xi_{\tau_j,\tau_i}) \\
\varrho_2(\tau_i, \nu_m) &= \sum_{j\in\Phi_{\tau_i}} (1 - \sum_{k\in\{1,\ldots,m\}} x_{j,k})\xi_{\tau_i,\tau_j} \\
&\quad + \sum_{j\in\Theta_{\tau_i}} (1 - \sum_{k\in\{1,\ldots,m\}} x_{j,k})\xi_{\tau_j,\tau_i} \\
\varrho(\tau_i, \nu_m) &= \alpha\varrho_1(\tau_i, \nu_m) + \beta\varrho_2(\tau_i, \nu_m)
\end{aligned} \tag{5}$$

Algorithm 1: The multidimensional FFD heuristic scheduling algorithm

```
Input: A task set $\vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be assigned
Output: A machine set $\vec{\nu} = \{\nu_1, \nu_2, \ldots, \nu_{m_{\text{used}}}\}$ with
        each machine hosting a disjoint subset of $\vec{\tau}$,
        where $m_{\text{used}}$ is the number of used machines
Sort available nodes in descending order by their
  resource availability as defined in Eq. (4)
$m_{\text{used}} \leftarrow 0$
while there are tasks remaining in $\vec{\tau}$ to be placed do
    Start a new machine $\nu_m$ from the sorted list;
    if there are no avaiable nodes then
        return Failure
    end
    Increase $m_{\text{used}}$ by 1
    while there are tasks that fit into machine $\nu_m$ do
        foreach $\tau \in \vec{\tau}$ do
            Calculate $\varrho(\tau_i, \nu_m)$ according to Eq. (5)
        end
        Sort all viable tasks based on their priority
        Place the task with the highest $\varrho(\tau_i, \nu_m)$ into
          machine $\nu_m$
        Remove the task from $\vec{\tau}$
        Update the remaining capacity of machine $\nu_m$
    end
end
return $\vec{\nu}$
```

In Eq. (5), $\varrho_1(\tau_i, \nu_m)$ represents the sum of introduced intra-node communication if τ_i is assigned to ν_m, and $\varrho_2(\tau_i, \nu_m)$ denotes the sum of communications that τ_i has with a peer that has not been assigned so far. α and β are the weight parameters that determine the relative importances of these two independent terms. In short, the higher value $\varrho(\tau_i, \nu_m)$ is, the higher priority τ_i will be packed into ν_m.

Designing $\varrho(\tau_i, \nu_m)$ in this way will make sure that the packing priority of the remaining tasks is dynamically updated after each assignment and those tasks sharing a large volume of communication are prioritised to be packed into the same node. This is in contrast to the classical FFD heuristics that first sort the items in terms of their priority and then proceed to the packing process strictly following the pre-defined order.

Finally, our algorithm implements the FFD heuristic from a bin-centric view, which opens only one machine at a time to accept task assignment. The algorithm keeps filling it with new tasks until the remaining capacity is running out, thus satisfying the resource constraints stated in Eq. (2).

4. Performance Evaluation

We evaluate the D-Storm prototype using both synthetic and realistic streaming applications, and compare the heuristic-based scheduling algorithm against the one proposed in the static resource-aware scheduler and the round-robin algorithm in the default Storm scheduler.

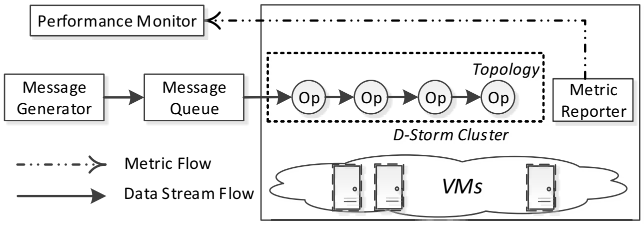


Figure 2. The profiling environment used for controlling the input load. The solid lines denote the generated data stream flow, and the dashed lines represent the flow of performance metrics

Specifically, the performance evaluation focuses on the following independent research questions:

- Whether D-Storm is applicable to different types of streaming applications and capable of reducing the total amount of inter-node communication? (Section 4.2)
- How is D-Storm performing when the input load decreases and the cluster is under-utilized? (Section 4.3)
- How long does it take for D-Storm to schedule relatively large streaming applications? (Section 4.4)

4.1. Experiment Setup

Our experiment platform is set up on the Nectar Cloud[2], comprising 1 Nimbus node, 1 Zookeeper node, 1 Kestrel[3] node and 16 worker nodes. All these machines are spawned from the "m2.medium" flavour, with each equipped with 2 VCPUs, 6 GB memory and 30 GB disk space. For the software stack, all the participating nodes are running Ubuntu 16.04 and Oracle JDK 8, update 121. We built our D-Storm extension on Apache Storm v1.0.2, and the comparable approaches — the static resource-aware scheduler and the default scheduler are directly extracted from this release.

In order to evaluate the performance of D-Storm under different sizes of workload, we have set up a profiling environment that allows us to adjust the size of the input stream with fine-grained control. Fig. 2 illustrates the components of the profiling environment from a working perspective. There is a *Message Generator* that reads the profiling messages from a local file of tweets, and then generates a profiling stream of a given volume to the Kestrel node leveraging the message push API. The *Message Queue* running on the Kestrel node implements a Kestrel queue to cache any messages that have been received but not pulled away by the streaming application. The streaming application is scheduled on the D-Storm cluster to process the profiling stream, which consists of a series of tweets in JSON format that were collected from 4/03/2014 to 14/04/2014. The Metric Reporter is responsible for probing the application performance — such as throughput and latency — using the Storm RESTful API, and reporting the volume of inter-node communication in the forms of the number of tuples transferred and the volume of data streams conveyed in the network. Finally, the *Performance Monitor* is introduced to examine whether the application is sustainably processing the profiling input and if the application performance has satisfied the pre-defined Quality of Service (QoS), such as processing 5000 tuples per second with the processing latency no larger than 500 ms.

4.1.1. Test Applications. The evaluation includes two test applications, one synthetically made and one drawn from real-world streaming use case. The synthetic application is designed to produce different patterns of resource consumption, such as CPU bound and I/O bound computations. There are three synthetic bolts concatenated in serial, following the Kestrel Spout that pulls the input stream from the message queue. The logic of the synthetic bolts is configurable, with all the configuration items listed in Table 2.

TABLE 2. THE CONFIGURATIONS OF THE SYNTHETIC TEST APPLICATION

Symbol	Configuration Description
C_s	The CPU load of each synthetic bolt.
S_s	The selectivity[4] of each synthetic bolt.
T_s	The number of tasks that each synthetic bolt has, also referred to as operator parallelism.

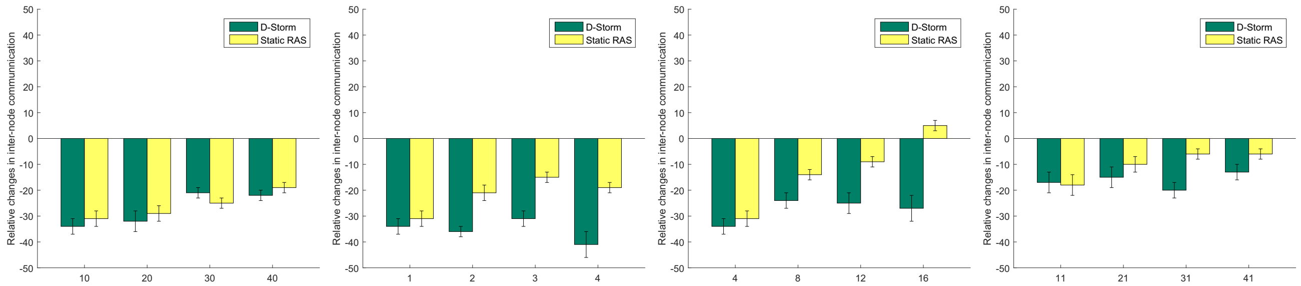
From the implementation point of view, the configuration items listed in Table 2 have a significant impact on the execution logic of the synthetic bolt. Specifically, C_s determines how many times this operator will invoke the method of random number generation *Math.random()* upon any tuple receipt, with $C_s = 1$ representing 100 times invocation. So the higher C_s is set, the larger CPU load the bolt will have. Besides, T_s indicates the operator parallelism during the topology building process, while S_s determines the selectivity of this operator and also the internal size of communication stream between tasks.

The second test application is taken from a realistic stream processing use case — analysing the sentiment of tweet contents by word parsing. There are 11 operators constituting a tree-like topology, with the sentimental score calculated using AFFINN, a list of words associated with pre-defined sentiment values. We refer to [11] for more details of the analysis process.

4.1.2. Parameter Selection. To avoid overshooting and overcome the fluctuation in metric observations, we set the metric collection window to 1 minute and the scheduling interval to 10 minutes. In addition, we configured α to 10 and β to 1, in order to put a stronger emphasis on the immediate gain of each task assignment rather than the potential benefits. We also set the latency constraint of each application to 500 ms, which represents a typical real-time requirement under streaming scenarios.

[2] https://nectar.org.au/research-cloud/
[3] https://github.com/twitter-archive/kestrel

[4] Selectivity is the number of tuples emitted per tuple consumed; e.g., selectivity = 2 means the operator emits 2 tuples for every 1 consumed.



(a) Varying C_s (Sythetic App) (b) Varying S_s (Sythetic App) (c) Varying T_s (Sythetic App) (d) Varying number of tasks (Real App)

Figure 3. The relative change of the inter-node communication, with the baseline produced by the Default Storm scheduler. We repeated each experiment for 5 times and the result bar also shows the standard deviation of the results. In the legend, static RAS stands for static resource-aware scheduler

4.2. Evaluation of Applicability

In this evaluation, we ran both streaming applications with all three schedulers under the same circumstances that a given size of profiling stream needs to be processed within the latency constraint. For the ease of interpretation, we report the relative changes to the communication volume (MB) while comparing the D-Storm scheduler and the static resource-aware scheduler (static RAS) against the default scheduler.

In order to evaluate the performance of scheduling under different application behaviours, the synthetic topology is configured to exhibit different types of resource consumption patterns, including CPU intensive (varying C_s), I/O intensive (varying S_s) and parallelism intensive (varying T_s). Also, the number of tasks for the twitter sentiment analysis is varied to examine the scheduler applicability to more complex use cases. Table 3 lists the evaluated and default values for the application configurations, where the default values are highlighted in bold. Note that when one configuration is altered, the others are set to their default value for fair comparison.

TABLE 3. EVALUATED CONFIGURATIONS AND THEIR VALUES

Configuration	Value
C_s (synthetic topology)	**10**, 20, 30, 40
S_s (synthetic topology)	**1**, 2, 3, 4
T_s (synthetic topology)	**4**, 8, 12, 16
Number of tasks (realistic application)	**11**, 21, 31, 41[5]

For all the applicability experiments, we set the size of the profiling stream to 2000 tuples / second and only collect communication results after the application is stabilised. Since the static resource-aware scheduler requires users to specify the resource consumptions at compile time, we conducted a pilot run for each application and registered a *LoggingMetricsConsumer*[6] from the *storm-metrics* package to probe the amount of memory/CPU resources being consumed.

In Fig. 3, we present the relative changes of inter-node communication when using D-Storm scheduler and the static resource-aware scheduler, compared to that of the default scheduler as baseline. We find that our D-Storm prototype always performs at least as well as the static counterpart, and often noticeably improves the communication reduction.

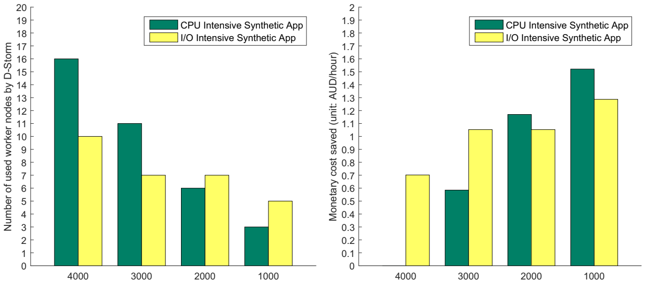
Specifically, a study of Fig. 3a reveals that D-Storm performed similarly to the static RAS when applied to CPU-bound use cases, with an average of communication reduction being 26.63%. Since the communication cost is dominated by the processing cost, scheduling heavy tasks onto a fewer number of nodes reduces to a typical bin-packing problem that is tackled well by both of the two approaches. However, it is worth noting that the communication reduction brought by the static scheduler is based on the correct configurations provided by the pilot run. If this information were not specified correctly, the static resource-aware would lead to undesirable scheduling results that could cause over-utilization and impair the system stability.

Fig. 3b, on the other hand, showcases that D-Storm significantly outperformed the static one by 22% in the most I/O intensive test case. This is credited to the fact that D-Storm specifically takes runtime communications into account during the decision-making process. By contrast, the existing scheduler can only optimise inter-node communication based on the number of task connections, which contains only coarse-grained information and does not reflect the actual communication pattern.

The parallelism intensive test case reveals another flaw of the static scheduler — tasks of the same operator are considered homogeneous in all aspects. As a matter of fact, tasks are not identical in terms of the received processing load, and they usually require a different amount of resources to accomplish the streaming jobs. Besides, as the number of tasks grows, the load distribution on each task as well as the incurred resource consumptions will be further amortised. However, setting the resource demand at the operator level forces all the spawned tasks to share the same resource profile and disregard any load changes. As shown in Figs. 3c and 3d, the performance of the static scheduler significantly degrades when the number of tasks increases, while D-Storm performs consistently well in relatively large

[5] The spout has a single task and each bolt has a parallelism of 1, 2, 3, 4, respectively.

[6] https://storm.apache.org/releases/1.0.2/javadocs/org/apache/storm/metric/LoggingMetricsConsumer.html



(a) The number of used nodes (b) The saved monetary cost

Figure 4. Cost efficiency analysis of D-Storm scheduler as the input load decreases. Fig. 4b is calculated based on the price of m1.medium instances in the AWS Sydney region

test cases, with the average communication reduction being around 27.5% and 16.25%, respectively. It is worth reporting that the static resource-aware scheduler performed even 5% worse than the default scheduler, for which we identified the cause to be the malformed resource profiles causing underutilization and unnecessarily tearing apart the communicating task pairs into different nodes during scheduling.

4.3. Evaluation of Cost Efficiency

Modelling the scheduling problem as a bin-packing variant offers the possibility to consolidate tasks into a fewer nodes when experiencing load valleys. In this evaluation, we apply D-Strom scheduler to the test applications, vary the size of input stream from 4000 tuples / second to 1000 tuples / second, and then examine the minimal resources required to process the given workload without violating the performance QoS.

There are two test applications derived from the synthetic application, mimicking the most intense CPU-bound and I/O-bound scenarios — the first application sets C_s to 40, and the second one sets S_s to 4. The other configurations are not altered during these experiments.

As shown in Fig. 4a, the number of machines used by the D-Storm scheduler steadily reduces when the input load decreases. This trend is more obvious in the first scenario, leading to the reduction of 13 worker nodes. This can be explained by the fact that our testbed is equipped with scarcer CPU resources as compared to the network resources, causing the heavy tasks to be rapidly spread out in the cluster to avoid resource contention. To intuitively illustrate the benefit of task consolidation, Fig. 4b shows the monetary cost saved in AUD if the cluster is composed of m1.medium instances in the AWS Sydney Region[7], where a m1.medium instance is charged at AUD $0.117 per hour.

However, the comparable schedulers such as the static resource-aware scheduler and the default Storm scheduler lack the ability to consolidate tasks when necessary. In these test scenarios, they would occupy the same amount of resources even if the input load dropped to only one-quarter of the previous amount.

TABLE 4. The time consumed in creating schedules by different strategies (unit: milliseconds)

Test Cases / Schedulers	Parallelism Intensive Synthetic App			
	T_s=4	T_s=8	T_s=16	T_s=20
D-Storm	16.4	16.2	16.8	17.2
Static Scheduler	5.2	5.7	5.5	5.9
Default Scheduler	0.72	0.77	0.75	0.79
	Twitter Sentiment Analysis			
	N_t[8]=11	N_t=21	N_t=31	N_t=41
D-Storm	13.4	13.6	13.8	13.9
Static Scheduler	3.41	3.61	3.53	3.91
Default Scheduler	0.61	0.62	0.65	0.62

4.4. Evaluation of Scheduling Overhead

We also examine the time required for D-Storm to calculate a viable scheduling plan, as compared to that of the static RAS scheduler and the default Storm scheduler. In this case, the parallelism intensive synthetic application and the twitter sentiment analysis are chosen for conducting the evaluation.

Studying Table 4, we find that the default Storm scheduler is the fastest among the all three comparable schedulers, which takes less than 1 millisecond to run the round-robin strategy. In contrast, the algorithm proposed in D-Storm is the slowest, as it requires re-sorting all the remaining tasks by their updated priority after each single task assignment. However, considering the fact that the absolute value of the time consumption is at the millisecond level, we conclude our solution is scalable to deal with large problem instances from the real world.

5. Related Work

Scheduling of streaming applications has attracted close attention from both big data researchers and practitioners. This section conducts a multifaceted comparison between the proposed D-Storm prototype and the most related schedulers in various aspects, as summarised in Table 5.

TABLE 5. Related work comparison

Aspects	**Related Works**							**Our**
	[5]	[6]	[2]	[4]	[12]	[13]	[1]	**Work**
Dynamic	Y	N	Y	Y	Y	N	Y	Y
Resource-aware	N	Y	N	N	Y	Y	N	Y
Communication -aware	Y	N	Y	Y	N	Y	Y	Y
Self-adaptive	Y	N	Y	Y	N	N	Y	Y
User-transparent	N	N	Y	Y	N	N	N	Y
Cost-efficient	N	Y	N	N	Y	N	N	Y

Aniello et al. pioneered dynamic scheduling algorithms in the stream processing context [5]. They developed a heuristic-based algorithm that places communicating tasks in pairs to the same node, thus reducing the amount of

[7] https://aws.amazon.com/ec2/pricing/

[8] N_t: The number of tasks for twitter sentiment analysis.

inter-node communication. The proposed solution is self-adaptive, which includes a task monitor to collect metrics at runtime and continuously adapts the scheduling plan to improve overall performance. However, the task monitor is not transparently set up at the middleware level and the algorithm is unaware of the resource demands of each task being scheduled. It also lacks the ability to consolidate tasks into fewer nodes for improving cost efficiency.

By modelling the task scheduling as a graph partitioning problem, Fisher et al. presented that the METIS software is also applicable to the scheduling of stream processing applications, which achieves better results on load balancing and reduction of inter-node communication as compared to Aniello's work [5]. However, their work is also not aware of resource demand and availability, let alone reducing the resource footprints with regard to the input load.

Xu et al. proposed another dynamic scheduler that is not only communication-aware but also user-transparent [4]. The proposed algorithm reduces inter-node traffic through iterative tuning and mitigates the resource contention by rebalancing the workload distribution. However, it does not model the resource consumption and availability for each task and node, thus lacking the ability to prevent resource contention from happening in the first place.

Sun et al. investigated energy-efficient scheduling by modelling the mathematical relationship between energy consumption, response time, and resource utilization [12]. But the algorithm proposed requires modifying the application topology to merge operators on non-critical paths. A similar technique is also seen in Li's work [1], which adjusts the number of tasks for each operator to mitigate performance bottleneck at runtime. Nevertheless, bundling scheduling with topology adjustment scarifies the user transparency and impairs the approach applicability.

The static resource-aware scheduler proposed by [6] has been introduced in Section 1. The main limitation of their work, as well as [13], [14], is that the runtime changes are not taken into consideration during the scheduling process.

6. Conclusions and Future Work

In this paper, we proposed a resource-efficient algorithm for streaming application scheduling and implemented a prototype scheduler named D-Storm to validate its effectiveness. D-Storm tracks each streaming task at runtime to collect its resource usages and communication pattern, which it then uses in the scheduling process to pack communicating tasks as compact as possible. The compact scheduling strategy leads to the reduction of resource usages as well as the amount of inter-node communication that would incur significant serialisation overhead. Our new algorithm overcomes the limitation of the static resource-aware scheduler, offering the ability to adjust the scheduling plan to the runtime changes while remaining sheer transparent to the upper-level application logic.

As for future work, we plan to investigate the use of meta-heuristics to find a better solution for the scheduling problem. Genetic algorithms, simulated annealing and tabu search are among the list of candidates that require further investigation. In addition, we would like to take the network characteristics into account during the scheduling process, in order to put a large volume of task communications on links with higher bandwidth. This research question is well-motivated by the fog computing infrastructure with heterogeneous network resources and the resulting scheduler can find its application in a wide range of Internet of Things (IoT) scenarios.

References

[1] C. Li, J. Zhang, and Y. Luo, "Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm," *Journal of Network and Computer Applications*, vol. 87, pp. 100–115, 2017.

[2] L. Fischer and A. Bernstein, "Workload scheduling in distributed stream processors using graph partitioning," in *Proceedings of the 2015 IEEE International Conference on Big Data*. IEEE, 2015, pp. 124–133.

[3] A. Chatzistergiou and S. D. Viglas, "Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters," in *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*, ser. CIKM '14. ACM Press, 2014, pp. 1579–1588.

[4] 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*. IEEE, 2014, pp. 535–544.

[5] 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 Press, 2013, pp. 207–218.

[6] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-Storm: Resource-Aware Scheduling in Storm," in *Proceedings of the 16th Annual Conference on Middleware*, ser. Middleware '15. ACM Press, 2015, pp. 149–161.

[7] X. Liu and R. Buyya, "Performance-Oriented Deployment of Streaming Applications on Cloud," *IEEE Transactions on Big Data*, vol. 14, no. 8, pp. 1–14, 2017.

[8] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, "Bin Packing Approximation Algorithms: Survey and Classification," in *Handbook of Combinatorial Optimization*, P. M. Pardalos, D.-Z. Du, and R. L. Graham, Eds. Springer New York, 2013, no. January 1998, pp. 455–531.

[9] A. S. Fukunaga and R. E. Korf, "Bin-completion algorithms for multicontainer packing and covering problems," *IJCAI International Joint Conference on Artificial Intelligence*, vol. 28, pp. 117–124, 2005.

[10] J. Desrosiers and M. E. Lübbecke, "Branch-Price-and-Cut Algorithms," in *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., jan 2011, pp. 1–18.

[11] F. A. Nielsen, "A new ANEW: Evaluation of a word list for sentiment analysis in microblogs," *arXiv:1103.2903*, vol. 718, pp. 93–98, 2011.

[12] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, and K. Li, "Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments," *Information Sciences*, vol. 319, pp. 92–112, oct 2015.

[13] T. Li, J. Tang, and J. Xu, "Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing," *IEEE Transactions on Big Data*, vol. 7790, no. 99, pp. 1–12, 2016.

[14] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-aware scheduling of streaming applications using genetic algorithm," *Procedia Computer Science*, vol. 108, no. June, pp. 2240–2249, 2017.