Dynamic Resource-Efficient Scheduling in Data Stream Management Systems Deployed on Computing Clouds*

Xunyun Liu[a], Yufei Lin[b], Rajkumar Buyya[c]

[a] *Artificial Intelligence Research Center, National Innovation Institute of Defense Technology, Beijing, 100071, China*
[b] *National Innovation Institute of Defense Technology, Beijing, 100071, China*
[c] *The University of Melbourne, Parkville, VIC, 3010, Australia*

Abstract

Scheduling streaming applications in Data Stream Management Systems (DSMS) have been investigated for years. As the deployment platform of DSMS migrates from on-premise clusters to elastic computing clouds, new requirements have emerged for the scheduling process to tackle workload fluctuations with heterogeneous cloud resources. Resource-efficient scheduling is to improve cost-efficiency at runtime by dynamically matching the resource demands of streaming applications with the resource availability of computing nodes. 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 present a prototype scheduler named D-Storm, which extends the original Apache Storm framework into a self-adaptive MAPE-K (Monitoring, Analysis, Planning, Execution, Knowledge) architecture and validates the efficacy and efficiency of our scheduling algorithm. The evaluation carried out on real-world applications such as Twitter Sentiment Analysis proves that D-Storm outperforms the existing resource-aware scheduler and the default Storm scheduler in terms of the reduction of inter-node traffic and application latency, as well as resulting in a significant amount of resource savings through task consolidation.

Keywords: Stream Processing, Data Stream Management Systems, Task Scheduling, Resource Management

*This is an extension of a conference paper: X. Liu and R. Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," in Proceedings of IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 2017, pp. 485-492.

1. Introduction

Big data processing has gained increasing popularity by proposing new paradigms and infrastructure for handling big-volume and high-velocity data, which greatly extends the capacity of traditional data management systems. High-velocity, as one of the core characteristics of big data, requires processing inputs in real-time to exploit their volatile value. For example, the statistics of share market are collected and analysed in real-time to avoid investment losses, while the ongoing user posts on a social media website are aggregated continuously to suggest trending topics on hype. As the Internet started to connect everything, the generation of high-velocity data is also contributed by a variety of IoT (Internet of Things) driven applications such as smart cities [1], RFID (Radio Frequency Identification) systems [2], and sensor networks [3]. To cater for the real-time processing requirement, stream processing emerges as a new in-memory paradigm that handles continuous, unbounded inputs on a record-by-record basis. Such once-at-a-time processing model performs independent computations on a smallish window of data upon arrival, delivering incremental results with merely sub-second processing latencies.

Despite the diversity of real-time use cases, most of the 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, for example, is a state-of-the-art distributed DSMS that provides a unified data stream management model for semantic consistency and supports the use of an imperative programming language to express user logic. It also offers user-transparent fault-tolerance, horizontal scalability, and state management by providing the abstraction of streaming primitives and simplifying the coordination of distributed resources at the middleware level.

Scheduling of streaming applications is one of the many tasks that should be transparently handled by the DSMSs. As the deployment platform of DSMS shifts from a homogeneous on-premise cluster to an elastic cloud resource pool, new challenges have arisen in the scheduling process to enable fast processing of high-velocity data with minimum resource consumption. First, the infrastructural layer can be composed of heterogeneous instance types ranging from Shared Core to High-CPU and High-memory machines[1], each equipped with different computing power to suit the diverse user needs. Thus, the assumption of homogeneous resources becomes invalid, and the node differences

[1] `https://cloud.google.com/compute/docs/machine-types`

must be captured in the scheduling process to avoid resource contention. Additionally, the distance of task communication needs to be optimised at runtime to improve application performance. In stream processing systems, intra-node communication (i.e. information exchange between streaming tasks within a single node) is much faster than inter-node communication as the former does not involve cumbersome processes of data (de)serialisation, (un)marshalling and network transmission. Therefore, it is up to the dynamic scheduling process to convert as much inter-node communication as possible into intra-node communication. Last but not least, the dynamics of real-time applications lead to unpredictable stream data generation, requiring the processing system to be able to manage elastic resources according to the current workload and improve cost-efficiency at runtime.

Therefore, to maximise application performance and reduce the resource footprints, it is of crucial importance for the DSMS to schedule streaming applications as compact as possible, in a manner that fewer computing and network 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 over/under utilisation and causing execution inefficiency. A few dynamic schedulers have also been proposed recently to reduce the network traffics and improve the maximum application throughput at runtime [4, 5, 6, 7, 8, 9, 10]. However, they all share the load-balancing principle that distributes the workload as evenly as possible across participating nodes, thus ignoring the need of resource consolidation when the input is small. Also, without application isolation, 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 every single node.

To fill in this gap, Peng et al. [11] 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 streaming nature.

2. The scheduler only executes once during the initial application deployment, making it impossible to adapt the scheduling plan to runtime changes. Its implementation is static, which tackles the scheduling problem as a one-time item packing process, so it only works on unassigned tasks brought by new application submissions or 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 algorithm. D-Storm does not require users to statically specify the resource needs of streaming applications; instead, it models 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 customisable scheduling interval, which means that it can free under-utilised nodes whenever possible to save resource costs.

This paper is a significant extension of our previous work [12]. The main **contributions** reported in this paper are summarised as follows:

- We formulate the scheduling problem as a bin-packing variant using a fine-grained resource model to describe requirements and availability. To the best of our knowledge, this work 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 to allocate multidimensional resources. The algorithm is capable of reducing the amount of inter-node communication as well as minimising the resource footprints used by the streaming applications.

- We implement the prototype on Storm and conduct extensive experiments in a heterogeneous cloud environment. The evaluation involving realistic applications such as Twitter Sentiment Analysis demonstrates the superiority of our approach compared to the existing static resource-aware scheduler and the default scheduler.

It is worth noting that though our D-Storm prototype has been implemented as an extended framework on Storm, it is not bundled with this specific platform. The fact that D-Storm is loosely coupled with the existing Storm modules and the design of external configuration make it viable to be generalised to other operator-based data stream management systems as well.

The remainder of this paper is organised as follows: we introduce Apache Storm as a background system in Section 2 to explain the scheduling problem. Then, we formulate the scheduling problem, present the heuristic-based algorithm, and provide an overview of the proposed framework in Sections 3 and 4. The performance evaluation is presented in Section 5, followed by the related work and conclusions in Sections 6 and 7, respectively.

2. Background

This section introduces Apache Storm, explains the concept of scheduling, and uses Storm as an example to illustrate the scheduling process in the state-of-the-art DSMSs. Apache Storm is a real-time stream computation framework built for processing high-velocity data, which has attracted attention from both academia and industry over the recent years. Though its core implementation is written in Clojure, Storm does provide programming supports for multiple high-level languages such as Java and Python through the use of Thrift interfaces. Being fast, horizontally scalable, fault-tolerant and easy-to-operate, Storm is considered by many as the counterpart of Hadoop in the real-time computation field.

Storm also resembles Hadoop from the structural point of view — there is a *Nimbus node* acting as the master to distribute jobs across the cluster and manage the subsequent computations; while the rests are the *worker nodes* with the *worker processes* running on them to carry out the streaming logic in JVMs. Each worker node has a *Supervisor* daemon to start/stop worker processes as per Nimbus's assignment. Zookeeper, a distributed hierarchical key-value store, is used to coordinate the Storm cluster by serving as a communication channel between the Nimbus and Supervisors. We refer to Fig. 1a for the structural view of a Storm cluster.

From the programming perspective, Storm has its unique data model and terminology. A *tuple* is an ordered list of named elements (each element is a key-value pair referred to as a *field*) and a *stream* is an unbounded sequence of tuples. The streaming logic of a particular application is represented by its *topology*, which is a Directed Acyclic

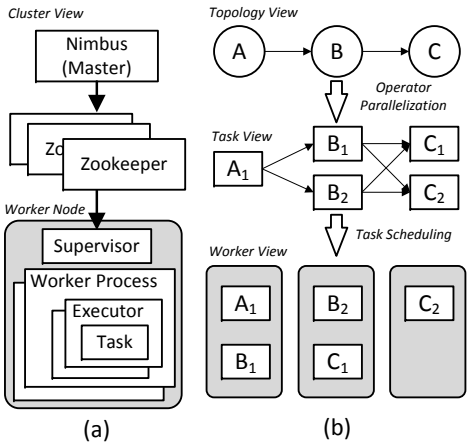


Figure 1: The structural view and logical view of a Storm cluster, in which the process of task scheduling is illustrated with a three-operator application.

Graph (DAG) of *operators* standing on continuous input streams. There are two types of operators: *spouts* act as data sources by pulling streams from the outside world for computation, while the others are *bolts* that encapsulate certain user-defined processing logic such as functions, filtering, and aggregations.

When it comes to execution, an operator is parallelised into one or more *tasks* to split the input streams and concurrently perform the computation. Each task applies the same streaming logic to its portion of inputs which are determined by the associated *grouping policy*. Fig. 1b illustrates this process as operator parallelisation. In order to make efficient use of the underlying distributed resources, Storm distributes tasks over different worker nodes in thread containers named *executors*. Executors are the minimal schedulable entities of Storm that are spawned by the worker process to run one or more tasks of the same bolt/spout sequentially, and Storm has a default setting to run one executor per task. The assignment of all executors of the topology to the worker processes available in the cluster is called *scheduling*. Without loss of generality, in this work, we assume each executor contains a single task so that executor scheduling can be interpreted as a process of task scheduling.

Since version 0.9.2, Storm implements inter-node communications with Netty[2] to enable low latency network transmission with asynchronous, event-driven I/O operations.

[2] `https://netty.io/`

However, the data to be transferred still needs to be serialised, then hit the transfer buffer, the socket interface and the network card at both sides of communication for delivery. By contrast, intra-node communication does not involve any network transfer and is conveyed by the message queues backed by LMAX Disruptor[3], which significantly improves the performance as tuples are deposited directly from the Executor send buffer to the Executor receive buffer.

The default scheduler in Storm is implemented as a part of Nimbus function that endeavours to distribute the same number of tasks over the participating worker nodes, where a round-robin process is adopted for this purpose. However, as pointed out in Section 1, such a simple scheduling policy may lead to over/under resource utilisation.

On the other hand, the scheduler proposed by Peng et al. [11] is the most relevant to our work and is widely adopted in the Storm community because it is resource-aware, bin-packing-related, and readily available within the standard Storm release. However, it can only partially tackle the problem of over/under resource utilisation due to the limitation of being static in nature and requiring users to input the correct resource configuration prior to execution. In Section 5, we conduct a thorough comparison of our approach and Peng et al.'s work with performance evaluation in a real environment.

3. Dynamic Resource-Efficient Scheduling

The dynamic resource-efficient scheduling exhibits the following characteristics: (1) each task has a set of resource requirements that are constantly changing along with 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; and (3) the scheduling algorithm is executed on-demand to take into account any runtime changes in resource requirements and availability.

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. In addition, the resource constraints need to be met — the resource requirements of the allocated tasks should not exceed the resource availability in each worker node. Since

[3] `https://lmax-exchange.github.io/disruptor/`

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.

Table 1: Symbols used for dynamic resource-efficient scheduling

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

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 [11], we specify the amount of CPU resources with a point-based system, where 100 points are given to represent the full capacity of a Standard Compute Unit (SCU). The concept of SCU is similar to the EC2 Compute Unit (ECU) introduced by Amazon Web Services (AWS). It is then the responsibility of the IaaS provider to define the computing power of an SCU, so that developers can compare the CPU capacity of different instance types with consistency and predictability regardless of the hardware heterogeneity presented in the infrastructure. As a relative measure, the definition of an SCU can be updated through benchmarks and tests after introducing new hardware to the data centre infrastructure.

In this paper, we assume that the IaaS cloud provider has followed the example of

Amazon to create a vCPU as a hyperthread of an Intel Xeon core[4], where 1 SCU is defined as the CPU capacity of a vCPU. Therefore, every single core in the provisioned virtual machine is allocated with 100 points. A multi-core instance 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 [13], task τ_i's CPU and memory resource requirements can be linearly modelled with regard to the size of the current inputs, which are illustrated in Eq. (1).

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

Note that i and j in Eq. (1) are just two generic subscripts that represent certain values within a range defined in Table 1. Therefore, ξ_{τ_j,τ_i} has a similar meaning of ξ_{τ_i,τ_j} that denotes the size of data stream transmitting from the former task to the latter.

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:

$$\begin{aligned} \text{minimise} \quad & \varkappa(\boldsymbol{\xi}, \boldsymbol{x}) = \sum_{i,j \in \{1,..,n\}} \xi_{\tau_i,\tau_j} (1 - \sum_{k \in \{1,..,m\}} x_{i,k} * x_{j,k}) \\ \text{subject to} \quad & \sum_{k=1}^{m} x_{i,k} = 1, \qquad i = 1, ..., n, \\ & \sum_{i=1}^{n} \omega_c^{\tau_i} x_{i,k} \leq W_c^{\nu_k} \quad k = 1, ..., m, \\ & \sum_{i=1}^{n} \omega_m^{\tau_i} x_{i,k} \leq W_m^{\nu_k} \quad k = 1, ..., m, \end{aligned} \tag{2}$$

where $\boldsymbol{x}$ is the control variable that stores the task placement in a binary form: $x_{i,k} = 1$ if and only if task τ_i is assigned to machine ν_k.

Through the formulation, we quantify the compactness of scheduling by counting the total amount of inter-node communication resulted from the assignment plan, with the optimisation target being reducing this number to its minimal.

[4] `https://aws.amazon.com/ec2/instance-types/`

Specifically, the expression $(1 - \sum_{k \in \{1,..,m\}} x_{i,k} * x_{j,k})$ is a toggle switch that yields either 0 or 1 depending on whether task τ_i and τ_j are assigned to the same node. If yes, the result of $(1 - \sum_{k \in \{1,..,m\}} x_{i,k} * x_{j,k})$ becomes 0 which eliminates the size of the data stream ξ_{τ_i,τ_j} to make sure that only inter-node communication is counted in our objective function.

There are three constraints formulated in Eq. (2): (1) each task shall be assigned to one and only one node during the scheduling process; (2) the CPU resource availability of each node must not be exceeded by the accrued CPU requirements of the allocated tasks; and (3) the memory availability of each node must not be exceeded by the accrued memory requirements of the allocated tasks.

Also, Eq. (3) shows that $\boldsymbol{x}$ can be used to reason the number of used worker nodes as the result of scheduling:

$$\begin{aligned} V_{\text{used}} &= \{\nu_j \mid \sum_{i \in \{1,\ldots,n\}} x_{i,j} > 0, j \in \{1, ..., m\}\} \\ m_{\text{used}} &= |V_{\text{used}}| \end{aligned} \tag{3}$$

3.2. Heuristic-based Scheduling Algorithm

The classical bin-packing problem has proved to be NP-Hard [14], and so does the scheduling of streaming applications [11]. 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 Quality of Service (QoS) constraints on processing time [15], 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 [16] and branch-and-price [17], 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 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

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_{used} is the number of used machines

```
Sort available nodes in descending order by their resource availability as defined in Eq. (4)
m_used ← 0
while there are tasks remaining in τ⃗ to be placed do
    Start a new machine ν_m from the sorted list;
    if there are no available nodes then
        return Failure
    end
    Increase m_used by 1
    while there are tasks that fit into machine ν_m do
        foreach τ ∈ τ⃗ do
            Calculate ϱ(τ_i, ν_m) according to Eq. (5)
        end
        Sort all viable tasks based on their priority
        Place the task with the highest ϱ(τ_i, ν_m) into machine ν_m
        Remove the assigned task from τ⃗
        Update the remaining capacity of machine ν_m
    end
end
return ν⃗
```

de-serialisation procedures that would have been necessary for network transmissions. 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).

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

Secondly, the evaluation of the task priority function 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 node as ν_m, then the task priority function $\varrho(\tau_i, \nu_m)$ can be formulated as a fraction — the greater the resulting value, the higher

priority τ_i will have to be assigned into ν_m.

The numerator of this fraction quantifies the increase of intra-node communications as the benefit of assigning τ_i to ν_m. It is 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. It is worth noting that we stop counting the second term in the numerator when the node ν_m is about to be filled up, so tasks capable of bringing more potential intra-node communications will be left for other nodes with more available resources.

On the other hand, the denominator of this fraction depicts the resource costs that ν_m spends on accommodating τ_i. Inspired by the Dominant Resource Fairness (DRF) approach used in Apache Mesos [18], we evaluate the resource costs of τ_i in terms of its usages of critical resource, where "critical resource" is defined as the most scarce and demanding resource type (either being CPU or memory) at the current state. Therefore, tasks occupying less critical resources will be preferred in the priority calculation, and the resulting resource usages are more likely to be balanced across different resource types.

After introducing the rationales behind our priority design, the mathematical formulation of $\varrho(\tau_i, \nu_m)$ is given as follows:

$$
\begin{aligned}
\varrho_1(\tau_i, \nu_m) &= \sum_{j \in \{1,\dots,n\}} x_{j,\nu_m} (\xi_{\tau_i,\tau_j} + \xi_{\tau_j,\tau_i}) \\
\varrho_2(\tau_i, \nu_m) &= \sum_{j \in \Phi_{\tau_i}} (1 - \sum_{k \in \{1,\dots,m\}} x_{j,k}) \xi_{\tau_i,\tau_j} \\
&\quad + \sum_{j \in \Theta_{\tau_i}} (1 - \sum_{k \in \{1,\dots,m\}} x_{j,k}) \xi_{\tau_j,\tau_i} \\
\Re_{\nu_m} &= \max\{ \frac{\sum_{j \in \{1,\dots,n\}} \omega_c^{\tau_j} x_{j,\nu_m}}{W_c^{\nu_m}}, \frac{\sum_{j \in \{1,\dots,n\}} \omega_m^{\tau_j} x_{j,\nu_m}}{W_m^{\nu_m}} \} \\
\varrho_3(\tau_i, \nu_m) &= \Re_{\nu_m}^{x_{\tau_i,\nu_m}=1} - \Re_{\nu_m}^{x_{\tau_i,\nu_m}=0} \\
\varrho(\tau_i, \nu_m) &= \begin{cases} \dfrac{\alpha \varrho_1(\tau_i, \nu_m) + \beta \varrho_2(\tau_i, \nu_m)}{\varrho_3(\tau_i, \nu_m)} & \Re_{\nu_m} \leq D_{\text{Threshold}} \\ \dfrac{\alpha \varrho_1(\tau_i, \nu_m)}{\varrho_3(\tau_i, \nu_m)} & \text{otherwise;} \end{cases}
\end{aligned}
\tag{5}
$$

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, while $\varrho_2(\tau_i, \nu_m)$ denotes the sum of communications that τ_i has with an unassigned peer, which effectively translates to the potential intra-node communication gains in the subsequent task assignments. After that, $\Re_{\nu_m}$ represents the current usage of critical resources in ν_m by the percentage measurement, and $\varrho_3(\tau_i, \nu_m)$ calculates

the difference of $\Re_{\nu_m}$ after and before the assignment to reflect the resource costs of ν_m accommodating τ_i. In the end, $\varrho(\tau_i, \nu_m)$ is defined as a comprehensive fraction of the benefits and costs relating to this assignment. In Eq. (5), α and β are the weight parameters that determine the relative importance of the two independent terms in the numerator, and $D_{\text{Threshold}}$ is the threshold parameter that indicates when the node resources should be considered nearly depleted.

Designing $\varrho(\tau_i, \nu_m)$ in this way makes 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 perspective, which opens only one machine at a time to accept task assignment. The algorithm keeps filling the open node with new tasks until its remaining capacity is depleted, thus satisfying the resource constraints stated in Eq. (2).

3.3. Complexity Analysis

We analyse the work-flow of Algorithm 1 to identify its complexity in the worst case. Line 1 of the algorithm requires at most quasilinear time $O(mlog(m))$ to finish, while the internal while loop from Line 9 to Line 17 will be repeated for at most n times to be either complete or failed. Diving into this loop, we find that the calculation of $\varrho(\tau_i, \nu_m)$ at Line 11 consumes linear time of n, and the sorting at Line 13 takes at most $O(nlog(n))$ time to complete. Therefore, the whole algorithm has the worst case complexity of $O(mlog(m) + n^2log(n))$.

4. Implementation of D-Storm Prototype

A prototype called D-Storm has been implemented to demonstrate dynamic resource-efficient scheduling, which incorporates 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 making scheduling decisions that avoid resource contention and minimise inter-node communication.

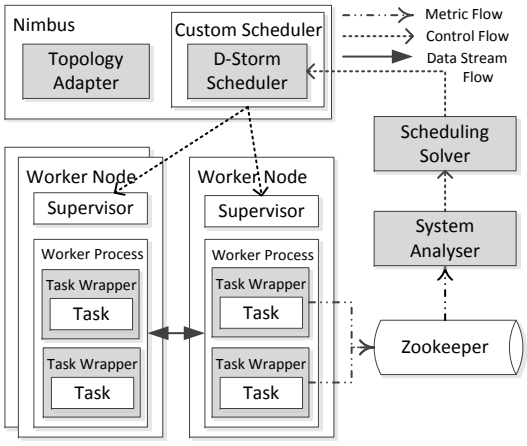


Figure 2: The extended D-Storm architecture on top of the standard Storm release, where the newly introduced modules are highlighted in grey.

- It endeavours to pack streaming tasks as compact as possible without causing resource contention, which effectively translates to the reduction of resource footprints while satisfying the performance requirements of streaming applications.

- It automatically reschedules the application whenever a performance issue is spotted or possible task consolidation is identified.

To implement these new features, D-Storm extends the standard Storm release with several loosely coupled modules, thus constituting a MAPE-K (Monitoring, Analysis, Planning, Execution, and Knowledge) framework as shown in Fig. 2. This architectural concept was first introduced by IBM to design autonomic systems with self-* capabilities, such as self-managing, self-healing [19], and self-adapting [20]. In this work, the proposed MAPE-K framework incorporates self-adaptivity and runtime-awareness into D-Storm, allowing it to tackle any performance degradation or mismatch between resource requirements and availability at runtime.

The MAPE-K loop in D-Storm is essentially a feedback control process that considers the current system metrics while making scheduling decisions. Based on the level at which the metrics of interest are collected, the monitoring system generally reports three categories of information — application metrics, task metrics, and OS (Operating System) metrics.

The application metrics, such as the topology throughput and complete latency[5], are obtained through the built-in Storm RESTful API and used as a coarse-grained interpretation of the application performance. The volume of incoming workloads is also monitored outside the application in order to examine the system's sustainability under the current workload.

The task metrics, on the other hand, depict the resource usages of different tasks and their communication patterns within the DSMS. Acquiring this information requires some custom changes to the Storm core, so we introduce the *Task Wrapper* as a middle layer between the current task and executor abstractions. Each task wrapper encapsulates a single task following the decorator pattern, with monitoring logic transparently inserted into the task execution. Specifically, it obtains the CPU usages in the *execute* method by making use of the *ThreadMXBean* class, and it logs the communication traffics among tasks using a custom metric consumer which is registered in the topology building process.

Apart from higher level metrics, Collectd[6], a lightweight monitoring daemon, is installed on each worker node to collect statistics on the operating system level. These include CPU utilisation, memory usage and network interface access on every worker node. It is worth noting that due to the dynamic nature of stream processing, the collected metrics on communication and resource utilisation are all subject to non-negligible instantaneous fluctuations. Therefore, the monitor modules average the metric readings over an observation window and periodically report the results to Zookeeper for persistence.

The analysing phase in the MAPE-K loop is carried out by the *System Analyser* module, which is implemented as a boundary checker on the collected metrics to determine whether they represent a normal system state. There are two possible abnormal states defined by the comparison of the application and OS metrics. (1) Unsatisfactory performance — the monitored application throughput is lower than the volume of incoming workloads, or the monitored complete latency breaches the maximum constraint articulated in the Quality of Service (QoS). (2) Consolidation required — the majority of worker nodes exhibit resource utilisations below the consolidation threshold, and the monitored topology throughput closely matches the volume of incoming workloads. Note that for

[5]Complete latency: the average time a tuple tree takes to be completely processed by the topology.
[6]`https://collectd.org/`

the sake of system stability, we do not alter the scheduling plan only because the resulting resource utilisation is high. Instead, we define abnormal states as strong indicators that the current scheduling plan needs to be updated to adapt to the ongoing system changes.

The *Scheduling Solver* comes into play when it receives the signal from the system analyser reporting the abnormal system states, with all the collected metrics passed on to it for planning possible amendments. It updates the model inputs with the retrieved metrics and then conducts scheduling calculation using the algorithm elaborated in Section 3. The passive design of invocation makes sure that the scheduler solver does not execute more frequently than the predefined scheduling interval, and this value 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-K 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 scheduling APIs provided by Storm. The result of assignment is then cross-validated with the application metrics retrieved from the RESTful API to confirm the success of re-scheduling.

The Knowledge component of the MAPE-K loop is an abstract module that represents the data and logic shared among the monitoring, analysing, planing and execution functions. For the ease of implementation, D-Storm incorporates the scheduling knowledge into the actual components shown in Fig. 2, which includes background information on topology structures and user requirements, as well as the intelligent scheduling algorithm based on which the self-adaptation activities take place.

In order to keep our D-Storm scheduling framework user-transparent to the application developers, we also supply a *Topology Adapter* module in the Storm core that masks the changes made for task profiling. When the topology is built for submission, the adapter automatically registers the metric consumer and encapsulates tasks in task wrappers with logic to probe resource usages and monitor communication volumes. In addition, developers can specify the scheduling parameters through this module, which proves to be an elegant way to satisfy the diverse needs of different streaming scenarios.

5. Performance Evaluation

In this section, we evaluate the D-Storm prototype using both synthetic and realistic streaming applications. The proposed heuristic-based scheduling algorithm is compared

against the static resource-aware algorithm [11], as well as the round-robin algorithm used in the default Storm scheduler.

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

- Whether D-Storm applies to a variety of streaming applications with diverse topology structures, communication patterns and resource consumption behaviours. Whether it successfully reduces the total amount of inter-node communication and improves the application performance in terms of latency. (Section 5.2)
- How much resource cost is incurred by D-Storm to handle various volumes of workload? (Section 5.3)
- How long does it take for D-Storm to schedule relatively large streaming applications? (Section 5.4)

5.1. Experiment Setup

Our experiment platform is set up on the Nectar Cloud[7], comprising 1 Nimbus node, 1 Zookeeper node, 1 Kestrel[8] node and 12 worker nodes. The whole cluster is located in the availability zone of National Computational Infrastructure (NCI) to avoid cross data centre traffic, and there are various types of resources present to constitute a heterogeneous cluster. Specifically, the 12 worker nodes are evenly created from three different instance flavours, which are (1) m2.large (4 VCPUs, 12 GB memory and 110 GB disk space); (2) m2.medium (2 VCPUs, 6 GB memory and 30 GB disk space) and (3) m2.small (1 VCPUs, 4 GB memory and 30 GB disk space). On the other hand, the managing and coordination nodes are all spawned from the m2.medium flavour. Note that we denote the used instance types as "large", "medium" and "small" hereafter for the convenience of presentation.

As for the software stack, all the participating nodes are configured with Ubuntu 16.04 and Oracle JDK 8, update 121. The version of Apache Storm on which we build our D-Storm extension is v1.0.2, and the comparable approaches — the static resource-aware scheduler and the default scheduler are directly extracted from this release.

[7] `https://nectar.org.au/research-cloud/`

[8] `https://github.com/twitter-archive/kestrel`

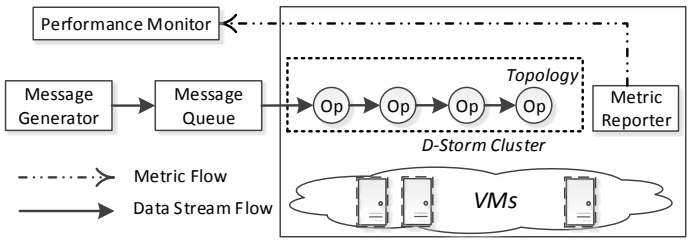


Figure 3: The profiling environment used for controlling the input load. The solid lines denote the generated data stream flow, and the dashed lines represent the flow of performance metrics.

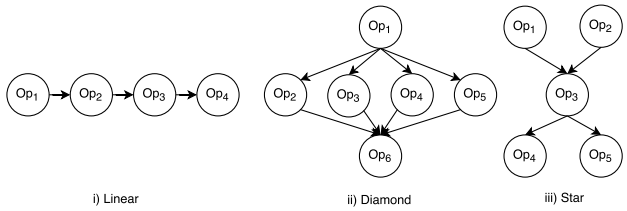
In order to evaluate the performance of D-Storm under different sizes of workload, we have set up a profiling environment that allows us to adjust the size of the input stream with finer-grained control. Fig. 3 illustrates the components of the profiling environment from a workflow perspective.

The *Message Generator* reads a local file of tweets and generates a profiling stream to the Kestrel node using its message push API. The tweet data were collected from 4/03/2014 to 14/04/2014 in JSON format, and the size of the generated data stream is externally configurable. The *Message Queue* running on the Kestrel node implements a Kestrel queue to cache any message that has been received but not pulled by the streaming application. It serves as a message buffer between the message generator and the streaming application to avoid choking either side of them in the case of mismatch.

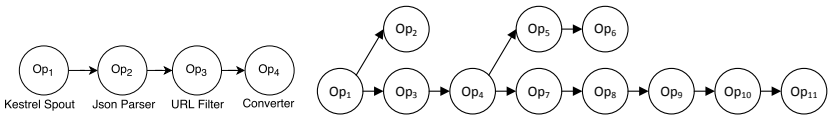
The *D-Storm cluster* runs the D-Storm prototype as well as the streaming application, where different scheduling algorithms are evaluated for efficiency. The *Metric Reporter* is responsible for probing the application performance, i.e. throughput and latency, and reporting the volume of inter-node 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 higher than 500 ms.

5.1.1. Test Applications

The evaluation includes five test applications — three synthetically made and two drawn from real-world streaming use cases. The acknowledgement mechanism is turned on for all these applications to achieve reliable message processing, which guarantees that



(a) The topologies of the micro-benchmark synthetic application



(b) The topology of the URL-converter application

(c) The topology of the twitter sentiment analysis application

Figure 4: The topologies of test applications. Fig. 4a is synthetically designed while the rest two are drawn from realistic use cases.

each incoming tuple will be processed at least once and the complete latency is automatically tracked by the Storm framework. We also set the Storm configuration *MaxSpoutPending*[9] to 10000, so that the resulting complete latency of different applications can be reasonably compared in the same context.

Synthetic applications (Micro-benchmark): the three synthetic applications are collectively referred to as micro-benchmark. They are designed to reflect various topological patterns as well as mimic different types of streaming applications, such as CPU bound, I/O bound and parallelism bound computations.

As shown in Fig. 4a, the micro-benchmark covers three common topological structures — Linear, Diamond, and Star, corresponding to operators having (1) one-input-one-output, (2) multiple-outputs or multiple-inputs, and (3) multiple-inputs-multiple-outputs, respectively. In addition, the synthetic operators used in the micro-benchmark can be configured in several ways to mimic different application types, which are summarised in Table 2.

From the implementation point of view, the configuration items listed in Table 2 have a significant impact on the operator execution. Specifically, C_s determines how many times

[9]The maximum number of unacknowledged tuples that are allowed to be pending on a spout task at any given time.

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

Table 2: The configurations of synthetic operators in the micro-benchmark

Symbol	Configuration Description
C_s	The CPU load of each synthetic operator.
S_s	The selectivity[10]of each synthetic operator.
T_s	The number of tasks that each synthetic operator has, also referred to as operator parallelism.

the method of random number generation *Math.random()* is invoked by the operator upon any tuple receipt (or tuple sending for topology spout), with $C_s = 1$ representing 100 invocations. Therefore, the higher C_s is set, the larger CPU load the operator will have. Besides, S_s determines the selectivity of this operator as well as the size of internal communication stream within the application, while T_s indicates the operator parallelism which is the number of tasks spawned from this particular operator.

URL-Converter: it is selected as a representative of memory-bound applications. Social media websites, such as Twitter and Google, make intensive use of short links for the convenience of sharing. However, these short links eventually need to be interpreted by the service provider to be accessible on the Internet. The URL-Converter is a prototype interpreter, which extracts short Uniform Resource Locators (URLs) from the incoming tweets and replaces them with complete URL addresses in real-time. As depicted in Fig. 4b, there are four operators concatenated in tandem: Op_1 (Kestrel Spout) pulls the tweet data from the Kestrel queue as a stream of JSON strings; Op_2 (Json Parser) parses the JSON string for drawing the main message body; Op_3 (URL Filter) identifies the short URLs from the tweet content; and Op_4 (Converter) completes the URL conversion with the help of the remote service. This application results in significant memory usages, as it caches a map of short and complete URLs in memory to identify the trending pages from the statistics and prevent checking the remote database again upon receiving the same short URL.

Twitter Sentiment Analysis (TSA): the second realistic application is adapted from a comprehensive data mining use case — analysing the sentiment of tweet contents by word parsing and scoring. Fig. 4c shows that 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 [21] for more details of this analysis process and [13] for the application implementation.

5.1.2. Parameter Selection and Evaluation Methodology

In our evaluation, the metric collection window is set to 1 minute and the scheduling interval is set to 10 minutes. These values are empirically determined for D-Storm to avoid overshooting and mitigate the fluctuation of metric observations on the selected test applications. As for the heuristic parameters, we configure $D_{\text{Threshold}}$ to 80%, α to 10, and β to 1, putting more emphasis on the immediate gain of each task assignment rather than the potential benefits. Additionally, the latency constraint of each application is set to 500 ms, which represents a typical real-time requirement for streaming use cases.

For all conducted experiments, we deployed the test application using the same approach recommended by the Storm community[11]. Specifically, the number of worker processes is set to one per machine and the number of executors is configured to be the same as the number of tasks, thereby eliminating unnecessary inter-process communications. Once the test application is deployed, we supply the profiling stream to a given volume and only collect performance results after the application is stabilised.

Also, the performance of the static resource-aware scheduler largely depends on the accuracy of resource profile. As the scheduler required that users submit the static resource profile at compile time [11], we conducted pilot run on test applications, probing their up-to-date resource profile and leading to a fair comparison between the dynamic and static resource-aware schedulers. In particular, we utilised the *LoggingMetricsConsumer*[12] from the *storm-metrics* package to probe the amount of memory/CPU resources being consumed by each operator, and we associate each pilot run with a particular application setting, so that the resource profile can be properly updated whenever the application configuration is changed.

5.2. Evaluation of Applicability

In this evaluation, we ran both the synthetic and realistic applications under the same scenario that a given size of profiling stream needs to be processed within the latency constraint. Different schedulers are compared in two major aspects: (1) the amount of inter-node communications resulted from the task placement, and (2) the complete latency of the application in milliseconds.

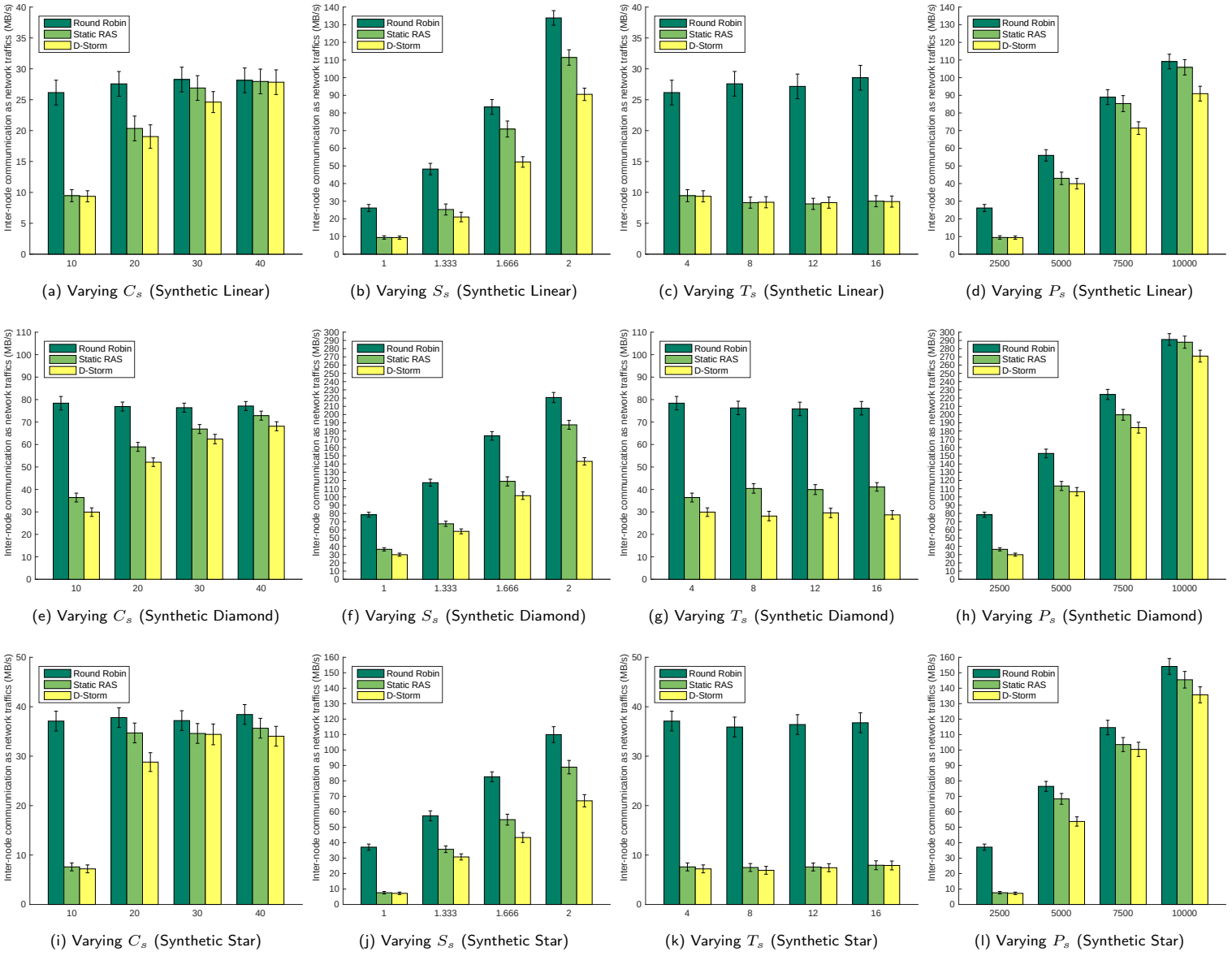
[11] `https://storm.apache.org/documentation/FAQ.html`

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

To better examine the applicability of D-storm, we configure micro-benchmark to exhibit different patterns of resource consumption. These include CPU intensive (varying C_s), I/O intensive (varying S_s) and parallelism intensive (varying T_s). In addition, we alter the volume of profiling stream (P_s) for all the applications to test the scheduler performance under different workload pressures. Table 3 lists the evaluated 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 (defaults in bold)

Configuration	**Value**
C_s (for micro-benchmark only)	**10**, 20, 30, 40
S_s (for micro-benchmark only)	**1**, 1.333, 1.666, 2
T_s (for micro-benchmark only)	**4**, 8, 12, 16
P_s (all applications)	**2500**, 5000, 7500, 10000



(a) Varying C_s (Synthetic Linear) (b) Varying S_s (Synthetic Linear) (c) Varying T_s (Synthetic Linear) (d) Varying P_s (Synthetic Linear)

(e) Varying C_s (Synthetic Diamond) (f) Varying S_s (Synthetic Diamond) (g) Varying T_s (Synthetic Diamond) (h) Varying P_s (Synthetic Diamond)

(i) Varying C_s (Synthetic Star) (j) Varying S_s (Synthetic Star) (k) Varying T_s (Synthetic Star) (l) Varying P_s (Synthetic Star)

Figure 5: The change of the inter-node communication when varying the configurations of the micro-benchmark. We repeated each experiment for 10 times to show the standard deviation of the results. In the legend, RAS stands for the Resource-Aware Scheduler.

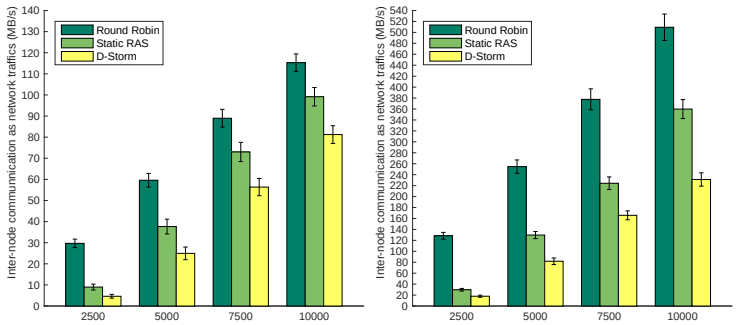
Fig. 5 presents the changes of inter-node communication while altering the micro-benchmark configurations listed in Table 3. We find that our D-Storm prototype always performs at least as well as the static counterpart, and often results in significant communication reduction as compared to the default scheduler.

Specifically, a study of Figs. 5a, 5e and 5i reveals that D-Storm performs slightly better than, or at least similarly to the static Resource-Aware Scheduler (RAS) when applied to CPU-intensive use cases. In most instances, D-Storm achieves the similar communication reduction as the static RAS, with the difference also reaching as high as 17% when C_s is set to 20 for the Star topology. We interpret this performance similarity in that all streaming tasks are created homogeneously by their implementation, which makes the job easy for the static method to probe accurate resource profiles through a pilot run. Moreover, as the scheduling of CPU-intensive application reduces to a typical bin-packing problem, both resource-aware approaches performed well in the beginning by utilising the large nodes first for assignment. On average, they saved 63.9%, 57.7%, and 80.1% inter-node traffic compared to the default scheduler in the linear, diamond and star topology, respectively.

This also explains the variation trend we see in the figures: as the applications become increasingly computational-intensive, the performance gain of being resource-aware is gradually reduced (from on average 67.2% less to almost identical). This is because the streaming tasks are forced to spread out to other nodes as they become more resource-demanding, thus introducing new inter-node communications within the cluster.

However, it is worth noting that the communication reduction brought by the static RAS is based on the correct resource profile provided by the pilot run. If this information were not specified correctly, the static resource-aware scheduler would lead to undesirable scheduling results, causing over-utilisation and impairing the system stability.

Figs. 5b, 5f and 5j, on the other hand, showcase the communication changes as the selectivity configuration is altered, which creates heterogeneous and intensive communications on the tailing edges of the topology. The results demonstrate that D-Storm outperforms the static RAS in terms of the communication reduction by on average 15.8%, 17.4%, and 16.2% for the linear, diamond and star topology, respectively. This is credited to the fact that D-Storm is able to take runtime communications into account during the decision-making process. By contrast, the existing resource-aware scheduler can only optimise inter-node communication based on the number of task connections,



(a) Varying P_s (URL-converter) (b) Varying P_s (TSA)

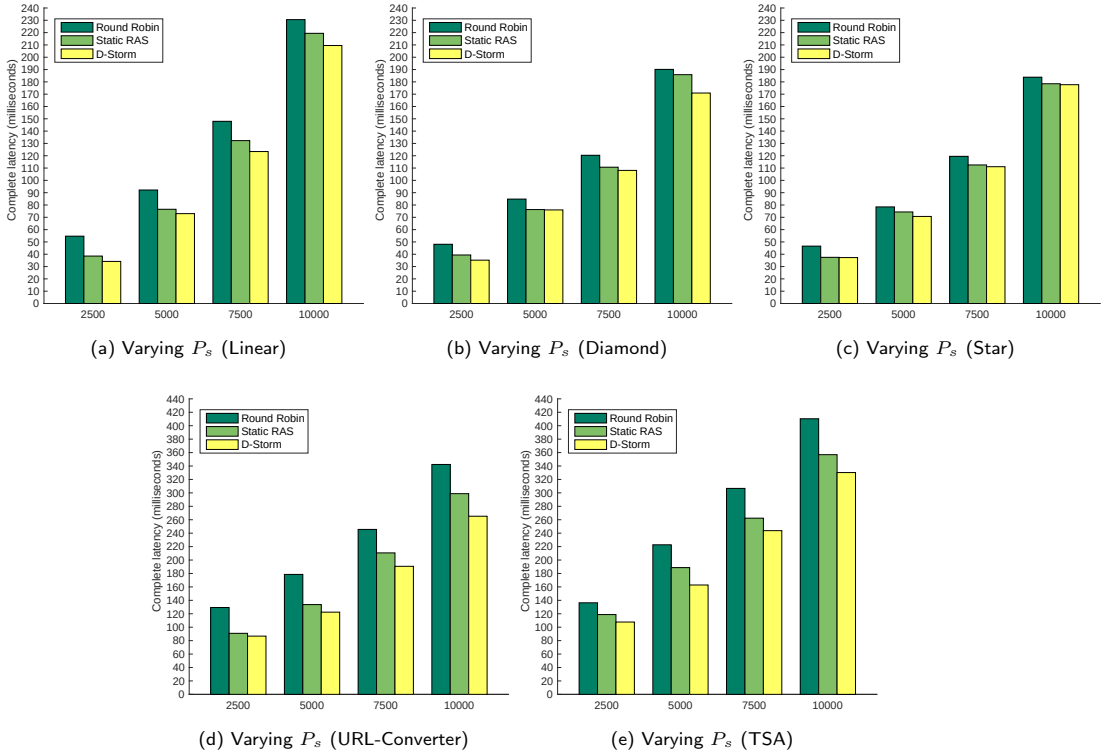
Figure 6: The change of the inter-node communication when varying the input size of realistic applications. We repeated each experiment for 10 times to show the standard deviation of the results.

which contains only coarse-grained information and does not reflect the actual communication pattern. We also find out that the amount of inter-node communication increases rapidly along with the growing selectivity. Especially, the four-operator linear topology exhibits 5.1, 11.8, and 9.7 times network traffic increase under all the three schedulers when the selectivity configuration doubles, which proves that the internal communication has been magnified exponentially by the concatenated selectivity settings.

Figs. 5c, 5g and 5k compare the three schedulers in parallelism intensive test cases. The analysis of results discovers that the amount of inter-node communication is relatively insensitive to the variations of the parallelism settings. We found it is because a single worker node can accommodate more tasks for execution, as the resource requirement of each streaming task reduces effectively in inverse proportion to the parallelism increase. However, it is also worth reporting that in Fig. 5g the static RAS performs noticeably worse than D-Storm, causing an average of 10.4 MB/s more network traffic in the four test cases. We interpret this result as the static scheduler having overestimated the resource requirement for Op_6, which results in the use of more worker nodes in the cluster. As a matter of fact, the additional overhead of thread scheduling and context switching increases along with the parallelism setting, which would be hard for the static RAS to estimate prior to the actual execution.

Finally, we evaluate the communication changes as the test application handles different volumes of workload. The results of micro-benchmark are shown in Figs. 5d, 5h and 5l, while the results of the realistic applications are presented in Fig. 6. Specifically, D-Storm and the static RAS performed similarly when applied to the micro-benchmark,

which demonstrates that the static method works well on applications with homogeneous operators. On the other hand, D-Storm achieved much better performance than its static counterpart in the realistic applications — Fig. 6 shows that D-Storm improves the communication reduction by 14.7%, 21.3%, 18.7% and 15.5% in the URL-Converter, and by 9.3%, 18.8%, 15.5% and 25.3% in the Twitter Sentiment Analysis when P_s is varied from 2500 to 10000. Part of this performance improvement is credited to D-Storm being able to handle uneven load distributions, which is a common problem in realistic applications due to the hash function based stream routing. As a contrast, the static scheduler configures resource profile at the operator-level, deeming the spawned streaming tasks homogeneous in all aspects. Consequently, the increasingly unbalanced workload distribution among the same-operator tasks is ignored, which leads to the performance degradation of the static scheduler.



(a) Varying P_s (Linear) (b) Varying P_s (Diamond) (c) Varying P_s (Star)

(d) Varying P_s (URL-Converter) (e) Varying P_s (TSA)

Figure 7: The application complete latency under different volumes of profiling streams. Each result is an average of statistics collected in a time window of 10 minutes, so the error bar is omitted as the standard deviation of latency is negligible for stabilised applications

We also collected the metric of complete latency to examine the application respon-

siveness while using different schedulers. As observed in Fig. 7, the complete latency is strongly affected by the combination of two factors — the size of the profiling stream and the volume of inter-node communications. First of all, the higher the application throughput, the higher the complete latency is likely to be yield. If we calculate an average complete latency for these three schedulers, we can find out the results for the linear, diamond, star, URL-Converter, and Twitter Sentiment Analysis have increased to 5.2, 4.4, 4.3, 2.9 and 3.1 times the original values, respectively. It also shows that more resources are required for the Twitter Sentiment Analysis to handle higher throughput without violating the given latency constraint, as the complete latency has reached as high as 410.4 milliseconds in our evaluation.

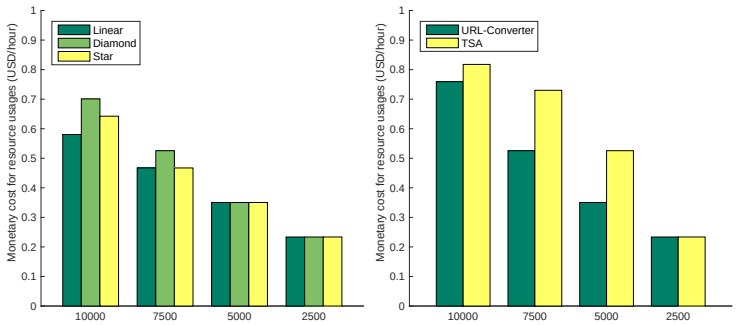
Besides, we notice that reducing the inter-node communication is also beneficial to improving the application responsiveness. As shown in Figs. 7d and 7e, packing communicating tasks onto fewer worker nodes allows D-Storm to reduce the communication latency to on average 72.7% and 78% of that of the default scheduler in the URL-Converter and Twitter Sentiment Analysis, respectively. These results confirm the fact that conducting communication on networks is much more expensive than inter-thread messaging, as the later avoids data serialisation and network delay through the use of a low-latency, high-throughput message queue in memory.

5.3. Evaluation of Cost Efficiency

Modelling the scheduling problem as a bin-packing variant offers the possibility to consolidate tasks into fewer nodes when the volume of incoming workload decreases. In this evaluation, we examine the minimal resources required to process the given workload without violating the latency constraint. Specifically, D-Strom scheduler is applied to the test applications, with the size of input stream (P_s) varied from 10000 tuples / second to 2500 tuples / second. To intuitively illustrate the cost of resource usages, we associate each worker node created in the Nectar cloud with the pricing model in the AWS Sydney Region[13]. In particular, a small instance is billed at $0.0292 per hour, a medium instance costs $0.0584 per hour, and a large instance charges $0.1168 per hour.

All five test applications introduced in Section 5.1.1 are included in this evaluation, in which the synthetic topologies have configured their settings to the default values. As shown in Fig. 8, the cost of resources used by the D-Storm scheduler steadily reduces

[13] `https://aws.amazon.com/ec2/pricing/`



(a) Decreasing P_s (Micro-benchmark) (b) Decreasing P_s (Realistic apps)

Figure 8: Cost efficiency analysis of D-Storm scheduler as the input load decreases. The pricing model in the AWS Sydney region is used to calculate the resource usage cost.

when the input load decreases. Specifically, the diamond topology is the most resource-consuming synthetic application in the micro-benchmark, utilising 4 large nodes and 4 medium nodes to handle the profiling stream at 10000 tuples / second. Such resource configuration also demonstrates that D-Storm avoids using smaller instances for scheduling unless the larger nodes are all depleted, which helps minimise the inter-node communication and improve the application latency. As the volume of the profiling stream drops from 10000 tuples / second to 2500 tuples / second, the resource consolidation is triggered and the usage cost of the diamond, linear, and star topology reduces to 33.3%, 40.2%, 36.4% of the original values, respectively.

This same trend is also observed in the evaluation of realistic applications, with consolidation resulting in 69.2% and 71.43% cost reduction for the URL-Converter and Twitter Sentiment Analysis, respectively. In particular, Twitter Sentiment Analysis only requires two large instances to handle the reduced workload whereas it used to occupy the whole cluster for processing.

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, which results in under-utilisation and significant resource waste.

5.4. Evaluation of Scheduling Overhead

We also examine the time required for D-Storm to calculate a viable scheduling plan using Algorithm 1, as compared to that of the static RAS scheduler and the default Storm

scheduler. In this case, the synthetic applications are evaluated with various parallelism settings, as well as the realistic applications under the default size of the profiling stream.

Specifically, we utilised the *java.lang.System.nanoTime* method to probe the elapsed time of scheduling in nanosecond precision. To overcome the fluctuation of results, we repeated the clocking procedure for 5 times and present the average values in Table 4.

Table 4: Time consumed in creating schedules by different strategies (unit: milliseconds)

Schedulers \ **Test Cases**	**Linear Topology**			
	T_s=4	T_s=8	T_s=12	T_s=16
D-Storm	15.49	18.07	26.52	32.29
Static Scheduler	3.01	3.91	4.25	4.51
Default Scheduler	1.20	1.64	1.98	2.04
	Diamond Topology			
	T_s=4	T_s=8	T_s=12	T_s=16
D-Storm	19.08	22.90	35.89	39.64
Static Scheduler	3.29	3.62	5.78	6.01
Default Scheduler	1.77	1.51	2.84	2.83
	Star Topology			
	T_s=4	T_s=8	T_s=12	T_s=16
D-Storm	13.80	23.66	28.71	32.59
Static Scheduler	3.11	5.38	5.76	5.27
Default Scheduler	1.36	1.78	2.17	2.47
	Realistic applications			
	URL-Converter		TSA	
D-Storm	18.17		42.25	
Static Scheduler	5.56		5.91	
Default Scheduler	1.55		2.99	

Studying Table 4, we find that the default Storm scheduler is the fastest among all three comparable schedulers, which takes less than 3 milliseconds to run the round-robin strategy for all test applications. Its performance is also relatively insensitive to the increasing parallelism configuration, as there are no task sorting or comparison involved in the scheduling process.

On the other hand, the static resource-aware scheduler usually takes 3 to 6 milliseconds to run its greedy algorithm. Compared to the default round-robin scheduler, it consumes

roughly twice the time of the former to make sure that the number of communication connections across different worker nodes is minimised.

In contrast, the algorithm proposed in D-Storm is the slowest among the three, as it requires dynamically 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, and the analysis in Section 3.3 has shown that the algorithm is at worst in quadratic time complexity, we conclude our solution is still efficient and scalable to deal with large problem instances from the real world.

6. 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**
	[10]	[11]	[7]	[9]	[22]	[23]	[24]	[4]	**Work**
Dynamic	Y	N	Y	Y	Y	Y	N	Y	Y
Resource-aware	N	Y	N	N	Y	Y	Y	N	Y
Communication -aware	Y	N	Y	Y	N	N	Y	Y	Y
Self-adaptive	Y	N	Y	Y	N	N	N	Y	Y
User-transparent	N	N	Y	Y	N	N	N	N	Y
Cost-efficient	N	Y	N	N	Y	Y	N	N	Y

Aniello et al. pioneered dynamic scheduling algorithms in the stream processing context [10]. They developed a heuristic-based algorithm that prioritises the placement of communicating tasks, thus reducing the amount of inter-node communication. The proposed solution is self-adaptive, which includes a task monitor to collect metrics at runtime and conducts threshold-based re-scheduling for performance improvement. 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. [7] demonstrated that the METIS software is also applicable to the scheduling of stream processing applications, which achieves better results on load balancing and further reduction of inter-node communication as compared to Aniello's work [10]. However, their work is also not aware of resource demand and availability, let alone reducing the resource footprints with regard to the varying input load.

Xu et al. proposed another dynamic scheduler that is not only communication-aware but also user-transparent [9]. The proposed algorithm reduces inter-node traffic through iterative tuning and mitigates the resource contention by passively 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 utilisation [22]. They also studied reliability-oriented scheduling to trade-off between competing objectives like better fault tolerance and lower response time [23]. But the algorithms proposed in these two papers require modifying the application topology to merge operators on non-critical paths. A similar technique is also seen in Li's work [4], which adjusts the number of tasks for each operator to mitigate performance bottleneck at runtime. Nevertheless, bundling scheduling with topology adjustment sacrifices the user transparency and impairs the applicability of the approach.

Cardellini et al. [25] proposed a distributed QoS-aware scheduler that aims at placing the streaming applications as close as possible to the data sources and final consumers. Differently, D-Storm makes scheduling decisions out of the resource-saving perspective and regards the minimisation of network communication as its first-class citizen. Papageorgiou et al. [26] proposed a deployment model for stream processing applications to optimise the application-external interactions with other Internet-of-Things entities such as databases or users, while our work focuses entirely on reducing network traffic among streaming operators.

The static resource-aware scheduler proposed by [11] has been introduced in Sections 1 and 2. The main limitation of their work, as well as [24, 27], is that the runtime changes to the resource consumptions and availability are not taken into consideration during the scheduling process.

7. Conclusions and Future Work

In this paper, we proposed a resource-efficient algorithm for scheduling streaming applications in Data Stream Management Systems and implemented a prototype scheduler named D-Storm to validate its effectiveness. It tackles new scheduling challenges introduced by the deployment migration to computing clouds, including node heterogeneity, network complexity, and the need of workload-oriented task consolidation. D-Storm tracks each streaming task at runtime to collect its resource usages and communication pattern, and then it formulates a multi-dimensional bin-packing problem in the scheduling process to pack communicating tasks as compact as possible while respecting the resource constraints. The compact scheduling strategy leads to the reduction of inter-node communication and resource costs, as well as reducing the processing latency to improve the application responsiveness. 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 conduct scheduling research in an Edge and Fog environment, where heterogeneous and geographically-distributed resources are typically connected using wireless technologies that are subject to network constraints such as limited bandwidth, longer communication delays, and unpredictable interruptions. The proposed scheduling algorithm should take the networking dimension into consideration to generate an efficient scheduling strategy, which avoids over-utilising the power-limited Edge devices and puts a large volume of task communications over links with higher network performance.

References

References

[1] N. Mitton, S. Papavassiliou, A. Puliafito, K. S. Trivedi, Combining Cloud and Sensors in a Smart City Environment, EURASIP Journal on Wireless Communications and Networking 2012 (1) (2012) 247–256.

[2] A. Puliafito, A. Cucinotta, A. L. Minnolo, A. Zaia, Making the Internet of Things

a Reality: The WhereX Solution, in: The Internet of Things, Springer New York, New York, NY, 2010, pp. 99–108.

[3] A. Cuzzocrea, G. Fortino, O. Rana, Managing Data and Processes in Cloud-Enabled Large-Scale Sensor Networks: State-of-the-Art and Future Research Directions, in: Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGRID '13, IEEE, 2013, pp. 583–588.

[4] C. Li, J. Zhang, 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 87 (3) (2017) 100–115.

[5] D. Sun, R. Huang, A Stable Online Scheduling Strategy for Real-Time Stream Computing over Fluctuating Big Data Streams, IEEE Access 4 (2016) 8593–8607.

[6] L. Cheng, T. Li, Efficient data redistribution to speedup big data analytics in large systems, in: Proceedings of the 23rd IEEE International Conference on High Performance Computing, HiPC '16, 2016, pp. 91–100.

[7] L. Fischer, 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.

[8] A. Chatzistergiou, 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, CIKM '14, ACM Press, 2014, pp. 1579–1588.

[9] J. Xu, Z. Chen, J. Tang, S. Su, T-storm: Traffic-aware online scheduling in storm, in: Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems, ICDCS '14, IEEE, 2014, pp. 535–544.

[10] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, ACM Press, 2013, pp. 207–218.

[11] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R. Campbell, R-storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Conference on Middleware, Middleware '15, ACM Press, 2015, pp. 149–161.

[12] X. Liu, R. Buyya, D-storm : Dynamic resource-efficient scheduling of stream processing applications, in: Proceedings of the 23rd International Conference on Parallel and Distributed Systems, IEEE, 2017, pp. 1–8.

[13] X. Liu, R. Buyya, Performance-oriented deployment of streaming applications on cloud, IEEE Transactions on Big Data 14 (8) (2017) 1–14.

[14] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin packing approximation algorithms: Survey and classification, in: P. M. Pardalos, D.-Z. Du, R. L. Graham (Eds.), Handbook of Combinatorial Optimization, Springer New York, 2013, pp. 455–531.

[15] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, O. F. Rana, Resource Management for Bursty Streams on Multi-Tenancy Cloud Environments, Future Generation Computer Systems 55 (2016) 444–459.

[16] A. S. Fukunaga, R. E. Korf, Bin-completion algorithms for multicontainer packing and covering problems, in: Proceedingds of the 2005 International Joint Conference on Artificial Intelligence, IJCAI '05, ACM Press, 2005, pp. 117–124.

[17] J. Desrosiers, M. E. Lübbecke, Branch-price-and-cut algorithms, in: Wiley Encyclopedia of Operations Research and Management Science, John Wiley & Sons, Inc., 2011, pp. 1–18.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness : Fair allocation of multiple resource types, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI '11, 2011, pp. 323–336.

[19] S. Caton, O. Rana, Towards Autonomic Management for Cloud Services Based upon Volunteered Resources, Concurrency and Computation: Practice and Experience 24 (9) (2012) 992–1014.

[20] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[21] F. A. Nielsen, A new ANEW: Evaluation of A Word List for Sentiment Analysis in Microblogs, in: Proceedings of the ESWC2011 Workshop on 'Making Sense of Microposts': Big Things Come in Small Packages, Springer, 2011, pp. 93–98.

[22] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, K. Li, Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments, Information Sciences 319 (2015) 92–112.

[23] D. Sun, G. Zhang, C. Wu, K. Li, W. Zheng, Building a Fault Tolerant Framework with Deadline Guarantee in Big Data Stream Computing Environments, Journal of Computer and System Sciences 89 (2017) 4–23.

[24] T. Li, J. Tang, J. Xu, Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing, IEEE Transactions on Big Data 7790 (99) (2016) 1–12.

[25] V. Cardellini, V. Grassi, F. L. Presti, M. Nardelli, On qos-aware scheduling of data stream applications over fog computing infrastructures, in: Proceedings of the IEEE Symposium on Computers and Communication, IEEE, 2015, pp. 271–276.

[26] A. Papageorgiou, E. Poormohammady, B. Cheng, Edge-Computing-Aware Deployment of Stream Processing Tasks Based on Topology-External Information: Model, Algorithms, and a Storm-Based Prototype, in: Proceedings of the 5th IEEE International Congress on Big Data, IEEE, 2016, pp. 259–266.

[27] P. Smirnov, M. Melnik, D. Nasonov, Performance-aware scheduling of streaming applications using genetic algorithm, Procedia Computer Science 108 (6) (2017) 2240–2249.