Performance-Oriented Deployment of Streaming Applications on Cloud

Xunyun Liu, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Performance of streaming applications are significantly impacted by the deployment decisions made at infrastructure level, i.e., number and configuration of resources allocated for each functional unit of the application. The current deployment practices are mostly platform-oriented, meaning that the deployment configuration is tuned to a static resource-set environment and thus is inflexible to use in cloud with an on-demand resource pool. In this paper, we propose P-Deployer, a deployment framework that enables streaming applications to run on IaaS clouds with satisfactory performance and minimal resource consumption. It achieves performance-oriented, cost-efficient and automated deployment by holistically optimizing the decisions of operator parallelization, resource provisioning, and task mapping. Using a Monitor-Analyze-Plan-Execute (MAPE) architecture, P-Deployer iteratively builds the connection between performance outcome and resource consumption through task profiling and models the deployment problem as a bin-packing variant. Extensive experiments using both synthetic and real-world streaming applications have shown the correctness and scalability of our approach, and demonstrated its superiority compared to platform-oriented methods in terms of resource cost.

Index Terms—Stream Processing, Data Stream Management Systems, Performance Optimization, Resource Management

1 Introduction

Driven by the exponential explosion of unstructured machine-generated data and the accompanying thirst for their timely processing, a new paradigm of in-memory processing — stream processing has emerged, bringing to market a plethora of streaming applications ranging from log monitoring to financial transaction analysis. Contrary to traditional batch paradigm where data are statically preserved for the ensuing manipulation, streaming processing introduces the "process-once-arrival" philosophy: the processing system dynamically accepts possible endless data streams as input, and then immediately processes them using continuous queries that are issued once but run continuously over the current portion of incoming data. Query results are constantly updated while the input data generated from logs, sensors, networks, and connected devices actively traverses through the processing system.

For better programmability and manageability, streaming applications are developed on top of a specialized middleware — Data Stream Management System (DSMS) to exploit the abstraction of processing primitives and simplify the use of distributed computing resources. The imperative programming language provided by DSMS hides the low-level complexity of implementations from applications developers, allowing them to express the continuous query as a direct graph of inter-connected operators (called *topology* hereinafter). In this way, the development burden of complex application logic, e.g. matrix multiplication and iterative data analytic, is significantly relieved. Moreover, the unified data stream management model assists developers with the ability to gracefully partition and route streams between operators, enabling streaming applications to scale-out on a large-scale distributed computing environment in the presence of a higher volume of processing load.

Though the use of DSMS has greatly facilitated the development of streaming logic, the deployment of such layered architecture (streaming logic, DSMS, and underlying hardware) is not transparent to developers. They are responsible for deciding how the streaming logic is carried out in a distributed environment to meet the specific processing requirement, including deciding the number and types of computing resources required by different stages of the streaming application.

Such deployment decisions depend on the type of the target environment. In the past, it was common to run streaming applications in a cluster where a combination of computing nodes were pre-configured [1]–[5]. As developers have assumed a static environment and only tune the application configuration towards higher utilization of on-premise resources, the deployment process in such environment is platform-oriented. With the emergence of cloud computing, an increasing number of streaming applications are being migrated to cloud to exploit the merits of virtualization, such as on-demand self-service, elastic resource pooling and the "pay-as-you-go" billing scheme. Since the cost of using cloud is based on the actual resource usage, a performance-oriented deployment model that customizes the resource provisioning with regard to the actual demands, i.e. enables the streaming application to reach a specific performance[1] target while minimizing the resource consumption is long overdue.

The overall performance of a streaming application is actually determined by the interplay between a variety

• *The authors are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, the University of Melbourne, Parkville, VIC 3010, Australia. E-mail: xunyunl@student.unimelb.edu.au, rbuyya@.unimelb.edu.au*

Manuscript received April 19, 2016; revised September 17, 2016.

1. While the context of performance may very under different types of QoS (Quality of Service), in this paper we refer it as the ability to steadily handle an input stream of throughput T within an acceptable processing latency L.

of contributing factors, which include the implementation of streaming logic, characteristics of workload, parameters of application and DSMS, and the sufficiency of resource provisioning. During the deployment phase, since the logic implementation is already given and the workload characteristics are not to be controlled by the processing system, proper mapping of the streaming logic to the underlying resources is the key for the application to accomplish a predefined performance target. Specifically, there are three optimization problems involved: (1) operator parallelization, which decides the number of running instances (called *tasks* hereinafter) for each operator to partition input load and realise parallel and asynchronous execution (2) resource provisioning, which estimates the resource usage and provisions the right scale computing power to constitute the processing system, and (3) task mapping, which implements the scheduling interface in DSMS, resulting in a task mapping to machines that distributes the computations and transformations derived from the operator logic.

Among the three, operator parallelization and task mapping have been separately investigated in the existing literature for various optimization targets [1], [3], [6]–[8], but resource provisioning is largely ignored in the platform-oriented deployment practice. Manually deciding the required resources makes the deployment plan inefficient nor swift to be applied in a cloud environment.

We overcome this limitation by proposing an automated, performance-oriented deployment framework that tackles these three optimization problems holistically. The deployment framework, called P-deployer, has the following desirable features: (1) based on fine-grained profiling information at the task level, it provisions right-scale execution platform on cloud, as well as decides the operator parallelism and task mapping configurations to guarantee high resource utilization and desired performance outcome; (2) it uses an automatic and iterative approach to deploy streaming application, reducing deployment effort for developers to address the identified performance bottlenecks; and (3) it is transparent to application logic, i.e. the existing application code can be deployed without any changes.

Our main **contributions** are summarized as follows:

- By profiling a real-world streaming application, we illustrate some important observations from the experimentation of different deployment plans, which lay down the foundation for P-deployer.
- We present the design of P-deployer that, to our best knowledge, is the first system to automatically deploy a streaming application on cloud with a predefined performance target.
- We model the resource provisioning problem as a variant of bin-packing problem with tasks being items and machine being bins, where packing together two communicating tasks may make them occupy less volumes than the sum of their individual size. Our heuristic reduces inter-node traffic while ensuring no computing nodes are overloaded.
- We implement a prototype on top of Apache Storm, and conduct a series of deployment experiments using both synthetic and real-world streaming applications to validate the resulted performance and the scalability of our approach. The results confirm that our framework significantly outperforms the state-of-the-art approaches in terms of resource cost.

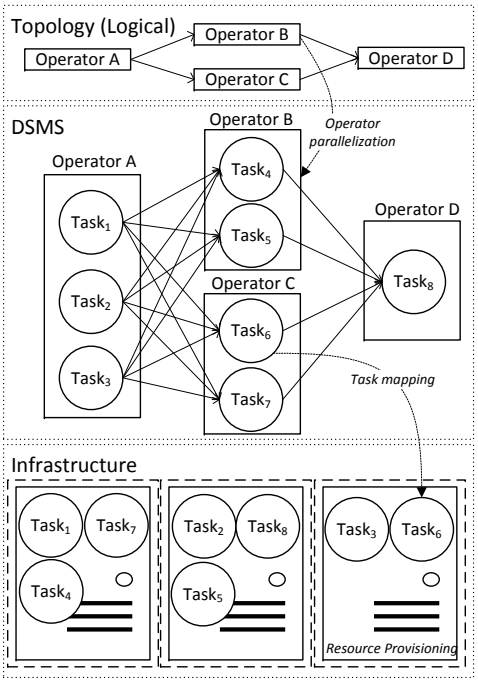

Fig. 1. Layered structure of an example streaming application

2 Background

A streaming application needs to be deployed on a particular execution environment before it can accept and process continuous data streams. In this section, we give a brief introduction to the layered structure of streaming applications and discuss the optimization problems involved in its deployment process.

As shown in Fig. 1, there are three tiers in the streaming application structure. the *topology* lying in the topmost *logical tier* is a directed acyclic graph that defines the streaming logic to be applied on the input data streams. Each vertex is an operator that encapsulates the semantic of a specific operation, such as filtering, join or aggregation; whereas each edge represents the direction of data transfer between upstream and downstream operators.

The *DSMS tier* is a middleware system that manages distributed resource and organises continuous streams to support the upper-level streaming logic defined in the topology. It is the key for the realization of "develop once, use many" concept. Since that a streaming application may need to run in different environment to deal with different volumes of input, DSMS makes the deployment plan adjustable, allowing the parallel execution of each operator to be customized with regard to the specific situation, such as the workload characteristics, performance target, and the capacity of underlying infrastructure.

The first deployment choice incurred in this tier is (1) *operator parallelization*. Specifically, DSMS treats each operator as a dividable logical entity and parallelizes its execution using a number of asynchronous tasks. Each task is logically equivalent as it handles a subset of the operator input and

performs the same type of operation. During the actual execution, DSMS guarantees the correctness of task coordination and makes sure that the subdivided inner streams would follow the pre-defined partition scheme. Therefore, developers are only required to specify the degree of parallelism for each operator so that it can secure a just enough number of tasks to keep up with its inbound load. We illustrate this process in Fig. 1 using a dash arrow labelled as "operator parallelization", which shows that $Operator\ B$ is parallelized into two tasks: $Task_4$ and $Task_5$.

The underlying *infrastructure tier* is the place where the parallel execution of tasks actually happens. The construction of this tier involves the other two deployment decisions: (2) *resource provisioning* — where developers select the number and types of resources from the elastic resource pool in the cloud. The streaming application in Fig. 1, as an example, has a 3-node virtual cluster provisioned; (3) *task mapping*, which assigns the asynchronous tasks to provisioned machines in an effort to achieve a particular deployment target or optimization goal, e.g. reaching a predefined throughput target, maximizing distributed resource utilization, minimizing the overall data processing latency, and etc. In Fig. 1, $Task_6$ of $Operator\ C$ is assigned to the third node as indicated by the dash arrow.

These three deployment decisions are highly correlated in nature. Our motivation is to iteratively optimize them to achieve automatic, performance-oriented, and cost-efficient streaming application deployment.

3 Preliminaries

We performed a series of deployment experiments on an IaaS cloud to investigate how different deployment decisions affect the performance outcome and resource consumption of a streaming application.

Our proof-of-concepts experiments were conducted on the Nectar Cloud[2] using 4 "m2.medium" instances in the NCI availability zone (each equipped with 2 VCPUs, 6GB RAM and 30GB root disk). On this virtual cluster, we deployed a word count streaming application built on top of the well-known DSMS — Apache Storm[3] 0.10.0 using various deployment plans.

The topology of word count depicted in Fig. 2 consists of four operators: the first operator, Kestrel Spout, pulls data from a message queue server and generates a continuous stream of tweets as its output. The second operator, JSON Parser, parses the stream and extracts the main message body. Sentence Splitter divides the main body of text into a collection of separate words, and finally, Word Counter is responsible for the final occurrence counting.

In order to evaluate the efficiency of different deployment plans, we have set up a profiling environment that feeds the streaming application with a controllable size of input stream and constantly monitors the application behaviours under that given pressure. For the sake of result stability, all the measurements of performance outcomes and resource usages are averaged using 5 consecutive readings of the corresponding value, which are all collected

2. https://nectar.org.au/research-cloud/
3. http://storm.apache.org/

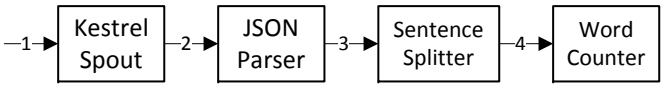

Fig. 2. The topology of word count streaming application, where the solid arrows represent data streams.

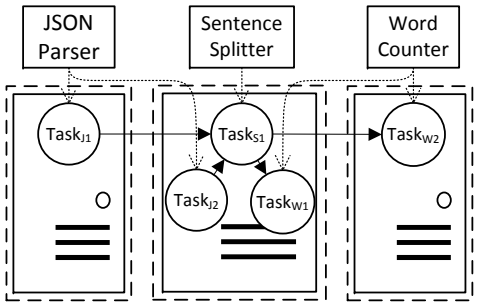

Fig. 3. Partial deployment sketch that shows how to designedly assign the inter-node communication ratio of $Task_{S1}$ to 50%. Solid arrows represent data streams while dash arrows denote the "operator-task" containment relationship.

after the application is stabilized. The observations and corresponding analysis are summarized as follows.

Observation 1: resource consumption of a task is positively correlated with its stream load and the ratio of inter-node communications. We consider the resource consumption in two dimensions: memory and CPU. Unlike memory consumption that can be measured in megabytes; the definition of CPU consumption is puzzlingly vague due to the diversity of operating systems and CPU architectures. Following the method used by the resource-aware scheduler in Storm [9], we adopt a point-based system to describe the amount of CPU resources available on a node or demanded by a particular task. Typically, an available CPU core gets 100 points and a multi-core machine could get $num\_of\_cores * 100$ resources points in total, whereas a task that occupies $x\%$ CPU usages reported by the monitoring system requires x points accordingly. Besides, we define the *stream load* of an operator (or a task) as the amount of stream data that passed through this component during a unit period of time. Due to the DAG organization of the topology, the measurement of stream load can be calculated by summing up the throughputs of all its incoming streams.

Our first experiment tracks the resource consumption of a task when it processes different sizes of stream load. We deployed the word count application using a spread-out strategy, where each operator has only one task and every task is mapped to a separate node. The results showed that, for all the examined tasks, the resource consumption increases almost linearly with the size of stream load. Particularly, those CPU-bound tasks reach their maximum processing capability when they have fully occupied the available CPU resources on a single core.

The second experiment investigates how the resource consumption of a task is influenced by the ratio of inter-node communication. For each examined task, we keep the stream load unchanged at a fixed rate (500 tuples[4]/s)

4. Tuple is a single datum in stream that is processed.

and constantly vary its inter-node communication ratio by parallelizing the upstream and downstream operators and properly placing the spawned tasks on different machines. For example, suppose we would like to probe the resource consumption of a Sentence Splitter task $Task_{S1}$ with a inter-node communication ratio set to 50%, Fig. 3 illustrates how we can deploy the word count application to achieve this goal. Specifically, we parallelize JSON Parser and Word Counter into two tasks ($Task_{J1}$, $Task_{J2}$), ($Task_{W1}$, $Task_{W2}$), but only put $Task_{J2}$ and $Task_{W1}$ on the same node of $Task_{S1}$ collocating with the examined task. If we assume that the load is balanced between the sibling tasks that constitute the same operator, $Task_{S1}$ would have half of its data communication transferred through the inter-node network. Once again, the results have shown a linear increase of resource consumption along with the increasing inter-node communication ratio, indicating that minimizing the inter-node traffic is also quantitatively profitable from the prospective of resource savings.

Observation 2: overly parallelizing an operator on a single machine greatly hurts its performance. We conducted this experiment to break the myth that higher operator parallelism would result in better performance, i.e. fine-grained parallel execution is always encouraged for an operator to incorporate as many tasks as possible to partition its stream load. In this experiment, we chose JSON Parser as the examined operator and tested three parallelization configurations: the first one sets up 2 tasks and puts them on a separate node to avoid interference from other operators; analogously, the second one initializes 8 tasks and the third one creates 16 for comparison. To observe the performance differences resulted from different configurations, we provide the streaming application with a sufficiently large input stream and make sure that other operators will not be the bottleneck of the topology.

From the results we observe that the first configuration yields the best performance among the three (35% higher throughput than the second and 69% higher than that of the third setting). This is because spawning two tasks for JSON Parser is already adequate to make full use of the two-core machine, while having 8 or even 16 tasks on this node only imposes additional resource costs of thread scheduling and context switching, — for the third setting particularly, half of the CPU time is spent running the kernel rather than user space processes, which greatly impairs the trafficability of JSON Parser.

3.1 Assumptions

In addition to the observations we made from the experiments, we make the following assumptions to build an automatic deployment framework:

1) Tasks of the same operator fairly process the same amount of workload. In other words, the stream load of an operator can be equally partitioned to all its constituent tasks.
2) The inter-node communication cost for each task is calculated from its bidirectional bandwidth usages, which is in line with the literature convention [1], [6], [9], [10].
3) The same type of machines are selected on the IaaS cloud to form a homogeneous infrastructure.

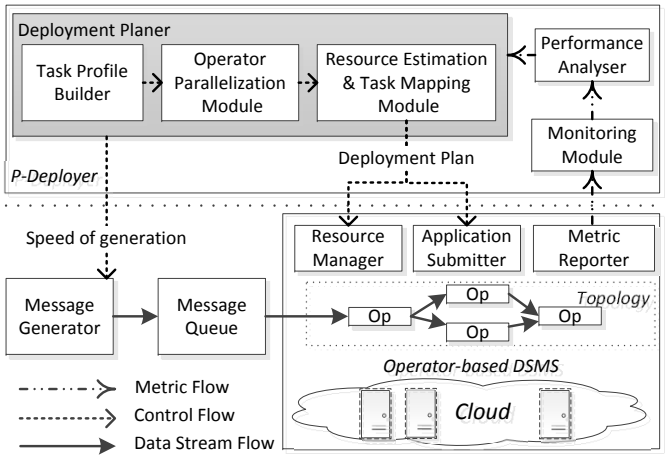

Fig. 4. The Monitor-Analyze-Plan-Execute (MAPE) architecture of P-Deployer and its working environment.

4 P-Deployer Overview

P-Deployer's design follows a typical Monitor-Analyze-Plan-Execute (MAPE) architecture and it works in a profiling environment to find the desirable deployment plan. The construction of profiling environment serves two major purposes: (1) *information collection*: it allows P-Deployer to probe the runtime characteristics of both streaming application and underlying cloud platform, thus collecting necessary information to model the performance behaviour as well as its corresponding resource consumption. (2) *deployment verification*: it verifies the proposed deployment plan against the pre-defined performance target by actual execution, i.e. examining how the deployed streaming application performs under a profiling stream that mimics the situation of processing the maximum throughput.

The profiling environment, as shown in the bottom half of Fig. 4, consists of a message generator, a message queue, and a streaming application to be run in the IaaS cloud environment. The message generator is able to produce data stream with a speed specified by P-Deployer, where all the data used in this stream is collected from the production phase to simulate the real workload. The profiling input is then directed to the message queue, which works as a message buffer to avoid overwhelming the streaming application in case its processing capability cannot catch up with the performance requirement. The streaming application ingests the profiling stream from the message queue with its layered structure shown in the rightmost frame. Alongside the streaming application, there are also some platform dependent modules that connect P-Deployer to the profiling environment. These include the Metric Reporter, the Resource Manager and the Application Submitter, that are respectively responsible for collecting the current performance metrics, provisioning/relinquishing cloud resources and submitting the streaming application to DSMS according to the specific deployment plan.

On top of the profiling environment, P-Deployer has an iterative working flow to implement the MAPE loop, with each iteration proposing a concrete deployment plan and then validating its capability through the profiling of deployed application. This iterative process continues

until the desirable deployment plan is found, or the cost of resource provisioning has already exceeded the user budget. Specifically, by retrieving the output of the Metric reporter, the Monitoring Module measures how the streaming application is performing under the profiling load. The Performance Analyser conducts some boundary checks on the set of metrics and determines whether the performance target has been met or not. If further adjustment is required, it hands in the collected runtime information to the Deployment Planer and indicates the possible cause of performance issue. Then, the Deployment Planer, as shown in the grey box, will update the inaccurate model inputs and make a new deployment plan in an attempt to remedy the performance bottleneck. The executors of P-Deployer are embedded within the cloud platform, following the deployment plan's instruction to control the speed of data generation, manage the cloud resources, and re-submit the streaming application for the next round of evaluation.

The essence of P-Deployer lies in the planning phase of the MAPE loop, which is to propose a holistic deployment plan based on the profiled runtime information. We briefly summarise this phase in three steps:

1) *Building task profile*: modelling the characteristics of each operator by profiling one of its tasks. The task profile essentially depicts the relationship between the desired performance behaviour and the estimated resource consumption at task level, which is the most fine-grained level of DSMS. Note that all the tasks of a single operator share the same task profile due to their homogeneity.
2) *Operator parallelization*: deciding the parallelism degree for each operator based on its stream load and the profile of its tasks. The result of this step is a set of tasks along with their requested resource consumptions.
3) *Resource estimation and task mapping*: formulating the deployment optimization problem as a bin-packing variant, the solution to which will estimate the overall resource demands and produce the mapping result at the same time. The goal is to minimize the number of used machines while satisfying the resource needs of collocating tasks.

5 Deployment Plan Generation

In this section, we discuss in detail how the planning steps are carried out to holistically decide operator parallelization, resource provisioning and task mapping.

5.1 Building Task Profile

As introduced in Sec. 3, we consider two types of resources in this work — memory and CPU, which are separately measured in megabytes and a point-based approach. A deployment plan is considered feasible only if the aggregated resource demands of collocated tasks in each node can be satisfied in both dimensions. Therefore, based on the analysis of collected information, we build a profile for each task to reflect the relationship between the performance behaviour and the corresponding resource consumption. It allows us to estimate how many resources this task would require to reach a specific performance target, i.e. to process a certain size of stream load and to send/receive a certain amount of messages remotely.

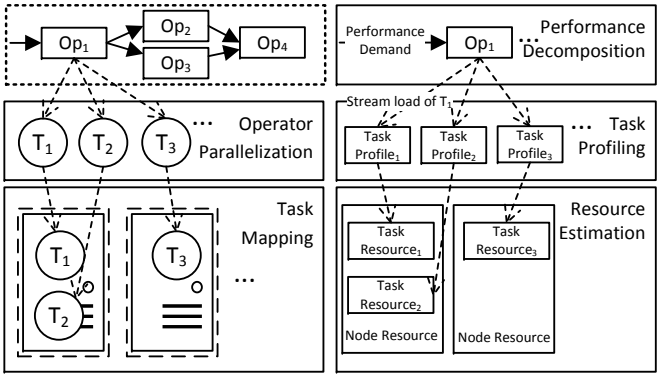

Fig. 5. Translating the performance demand of a streaming application into the estimated resource consumption through its task profiles.

The intuition behind task profile is illustrated in Fig. 5. Performance-oriented deployment has a prerequisite to properly translate the performance demand of a streaming application into its predicted resource consumption. However, there is no theoretical model nor empirical work that capable of directly achieving this goal, mainly because the layered application structure has much complicated the relationship between performance and resource usages. To fill in the gap, we establish this translation by building up a profile for each task, leading to a working process as follows: through operator parallelization, we break down the operators in topology into a set of tasks, meanwhile the application-level performance demand is also decomposed into the performance requirement of each task according to the stream balancing assumption we have made; as the task profile has modelled the resource consumption individually for each task, we estimate the overall resource consumption of the entire application as a result of task mapping by aggregating the resource needs of collocated tasks.

TABLE 1
The attributes of a task profile.

Symbol	Description
t_{soj}	Sojourn time of a tuple for being processed
c_p	CPU consumption of processing a tuple
c_t	CPU consumption of transmitting a tuple
m_{pt}	Memory consumption of processing & transmitting a tuple
n_i, n_o	Number of input (output) streams
S_k	Size (throughput) of the input stream k, $k = 1, ..., n_i$
$S_k^{'}$	Size (throughput) of the out stream k, $k = 1, ..., n_o$

Table. 1 enumerates the attributes of a task profile. Sojourn time t_{soj} is a period of time that a tuple stays within the task for being processed; since we model each task as a single-thread entity, the sojourn time of different tuples cannot be overlapped. The CPU consumption of handling a tuple consists of two parts: the processing cost c_p that is spent on executing the streaming semantic, and the transmission cost c_t that is consumed for network-related activities, such as serialization/deserialization, message buffering and performing the actual send/receive operation. Note that we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2720622, IEEE Transactions on Big Data

only count the inter-node communication in the calculation of transmission cost. This is because intra-node communication normally happens within the shared memory and is backed up by high-performance thread-messaging libraries (e.g. LMAX Disruptor), thus incurring negligible overhead compared to the network-based communication.

On the other hand, m_{pt} denotes the memory footprints of handling a tuple. It is not necessary to differentiate whether the memory is consumed by data processing or transmission, because there is little memory allocated for internal message buffers in order to avoid high queuing latency. Only memory-intensive computations, such as large windowed joins or cache-based analytic algorithms, can result in a considerable amount of memory usages that might define the machine characteristics of provisioned resources.

5.1.1 Modelling task resource consumption

For task τ that has n_i input streams with sizes denoted as $\{S_1, S_2, ..., S_{n_i}\}$ and n_o output streams of sizes in $\{S_1^{'}, S_2^{'}, ..., S_{n_o}^{'}\}$, its CPU consumption C_τ can be modelled in Eq. 1:

$$\begin{aligned} C_\tau &= \sum_{k=1}^{n_i} S_k c_p + (\sum_{k\in\Theta} S_k + \sum_{k\in\Theta} S_k^{'}) c_t \\ C_{\tau,\min} &= \sum_{k=1}^{n_i} S_k c_p \\ C_{\tau,\max} &= \sum_{k=1}^{n_i} S_k c_p + (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S_k^{'}) c_t \end{aligned} \tag{1}$$

where Θ indicates the set of inter-node communication, i.e. $k \in \Theta$ means that the input stream k is received from (or the output stream k is sent to) a task that locates in another node. Depending on the final location of task τ, its CPU consumption varies from the minimum value $C_{\tau,\min}$, when Θ is empty, to the maximum value $C_{\tau,\max}$, when all its communication happens across network.

Analogously, we model the memory consumption of this task in Eq. 2:

$$M_\tau = (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S_k^{'}) m_{pt} \tag{2}$$

5.2 Operator Parallelization

In light of the observation that over-parallelization hurts the operator performance (see Sec. 3), we adopt a minimal parallelism strategy that each operator only spawns the least number of tasks to keep up with its performance requirement. Therefore, the parallelism degree of an operator is determined by two factors: the stream load of this operator under the profiling input, which can be seen as a performance requirement obtained from the monitoring module; and the maximum processing capacity of its constituent task, which can be calculated using the task profile established in the previous step.

Specifically, the maximum processing capacity of a task refers to the largest stream load it can sustain during the runtime, which is determined by the implementation of the stream logic as well as the capability of the execution environment. Having modelled the resource consumption on a per tuple basis, the task profile reveals the confining resources that prohibit the entity from achieving higher throughput on this particular platform. In this sense, we provide a classification for different tasks based on the type of confining resources, and then discuss how to parallelize an operator *op* with tasks in one of these categories to achieve the minimal parallelism objective. The symbols used in this subsection are summarised in Table. 2:

TABLE 2
Symbols used for operator parallelization.

Symbol	Description
S_{op}	Monitored stream load of operator *op*
P_{op}	Parallelism degree (number of tasks) of *op*
$\left\{\begin{matrix} c_p, c_t \\ m_{pt}, t_{soj} \end{matrix}\right\}$	Task profile attributes that shared between all tasks constituting operator *op*
α	Upper limit on the CPU usages for a single I/O-bound task to perform data transmission
β	Upper limit on the memory usages for a single memory-bound task to occupy

- *CPU-bound*: task of this kind consumes a considerable amount of CPU resources to process a single tuple, such as multiplying small matrices for machine learning algorithms or performing iterative calculation for optimization purposes. CPU-bound task can utilize at most 100 point CPU resources for covering the processing cost c_p due to its single-thread nature, meaning that the maximum processing capacity is reached once it has fully occupied the CPU core for processing streams. Therefore, for an operator *op* whose tasks are CPU-bound and has a stream load of S_{op}, its parallelism degree can be decided as follows:

$$P_{op}(\text{CPU-bound}) = \left\lceil \frac{S_{op}}{100/c_p} \right\rceil$$

- *I/O-bound*: contrary to CPU-bound, I/O-bound tasks spend more time on waiting for the I/O operation rather than performing the actual computation. Its distinctive identifying characteristic is that c_t outweighs c_p significantly in the task profile. Event log processing, for example, incorporates typical IO-bound tasks that ingest a large size of log stream while only applying filtering or some other trivial transformations on it. The previous best practice in platform-oriented deployment has shown that a CPU core should host several IO-bound tasks to make efficient use of the network connection[5], therefore, we model the maximum processing capability for this kind of tasks by limiting the CPU resources it can get for data transmission. If the threshold, denoted as α, is set to 0.1, then at most 10 I/O-bound tasks are permitted to occupy a CPU core for data transmission. Accordingly, we calculate the parallelism degree for I/O-bound operators as follows:

$$P_{op}(\text{I/O-bound}) = \left\lceil \frac{S_{op}}{\alpha/c_t} \right\rceil$$

- *Sojourn time-bound*: some tasks need to invoke an external service to complete a tuple transaction, such

5. http://www.slideshare.net/ptgoetz/scaling-apache-storm-strata-hadoopworld-2014

as connecting to a remote database or calling a third-party API. These operations often require no CPU and memory consumption but may result in a substantial sojourn time for each tuple to be processed. In such case, the maximum processing capacity of each task is bound by the wall clock time, and thus the operator parallelization is conducted as follows:

$$P_{op}(\text{Sojourn time-bound}) = \lceil S_{op} * t_{soj} \rceil$$

- *Memory-bound*: as the stream load increases, some tasks may use a significant amount of memory to maintain a large window cache or store enormous intermediate results. Similar to the IO-bound task, we limit the maximum memory usage for a single task (denoted as β) and in turns deduce that how many tasks are needed for a memory-bound operator to sustain a stream load of S_{op}:

$$P_{op}(\text{Memory-bound}) = \left\lceil \frac{S_{op}}{\beta / m_{pt}} \right\rceil$$

Apart from the decision on the number of tasks, Operator Parallelization also models the resource consumption of each spawned task by taking into account of its share of stream load as well as the associated task profile. However, there is only a range estimation in terms of the CPU usages, as the actual value varies depending on the task location due to different inter-node communication costs.

5.3 Resource Estimation and Task Mapping

The problem we are trying to solve is to assign tasks to machines such that (1) the aggregated resource consumption of collocated tasks is satisfied, and (2) the cost of resource provisioning is minimized. Specifically, each task can be considered as an item of multi-dimensional volumes that are characterised by its resource requirements, while each machine is a bin of the same size as per Assumption 3 made in Sec. 3.1. The optimization target of this problem is to minimize the number of used machines. Therefore, this is a variant of bin-packing problem that can be formalized in the linear programming form.

Table. 3 summarizes the newly introduced symbols in this subsection.

TABLE 3
Symbols used for resource estimation and task mapping.

Symbol	Description
W_c	CPU capacity of provisioned machine
W_m	Memory capacity of provisioned machine
n	Number of tasks
τ_i	Tasks to be assigned, $i = 1, ..., n$
K	Upper bound on the number of machines used
	Symbols used in Eq. 3, 4 and shown in Fig. 6
op_k	Upstream operator k of op, $(k = 1, .., n_i)$
$op_k^{'}$	Downstream operator k of op, $(k = 1, .., n_o)$
S_k	Size of input stream k between op_k and op, $(k = 1, .., n_i)$
$S_k^{'}$	Size of output stream k between op and $op_k{'}$, $(k = 1, .., n_o)$
Adj_k	Number of tasks of op_k collocating with task τ_i
$Adj_k^{'}$	Number of tasks of $op_k^{'}$ collocating with task τ_i

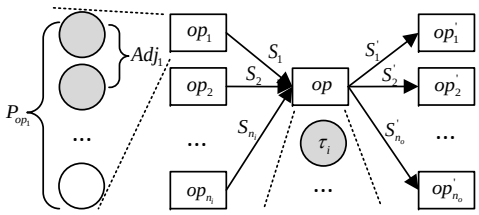

Fig. 6. The illustration of variables used in the calculation of resource consumption for task τ_i, where tasks in grey indicate that they are collocated on the same machine.

5.3.1 Problem Definition

Given a positive integer number of machines (bins) with CPU capacity W_c and memory capacity W_m, and a list of n tasks (items) $\tau_1, \tau_2, ..., \tau_n$ with their CPU demands and memory demands denoted as C_{τ_i}, M_{τ_i} $(i \in 1, 2, .., n)$, respectively, the problem is formulated as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{K} y_k \\
\text{subject to} \quad & \sum_{k=1}^{K} x_{i,k} = 1, && i = 1, ..., n, \\
& \sum_{i=1}^{n} C_{\tau_i} x_{i,k} \leq W_c y_k, && k = 1, ..., K, \\
& \sum_{i=1}^{n} M_{\tau_i} x_{i,k} \leq W_m y_k, && k = 1, ..., K,
\end{aligned}$$

where K is an upper bound on the number of machines needed, and the variables $y_k, x_{i,k}$ are:

$$y_k = \begin{cases} 1 & \text{if machine } k \text{ is used,} \\ 0 & \text{otherwise;} \end{cases}$$

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

The uniqueness of this problem lies in the fact that packing two communicating tasks together on the same machine will result in less resource consumption than the sum of their individual demands. This characteristic is reflected in the calculation of C_{τ_i} as shown in Eq. 3, which is derived from Eq. 1 but making use of the monitored stream loads and the result of operator parallelization to deduce the sizes of input/output streams for task τ:

$$\begin{aligned}
C_{\tau_i} = & \sum_{k=1}^{n_i} \frac{S_k}{P_{op_k} * P_{op}} c_p \quad + \\
& \left(\sum_{k=1}^{n_i} \frac{S_k}{P_{op_k} * P_{op}} (P_{op_k} - Adj_k) + \sum_{k=1}^{n_o} \frac{S_k^{'}}{P_{op_k}^{'} * P_{op}} (P_{op_k}^{'} - Adj_k^{'})\right) c_t
\end{aligned} \tag{3}$$

The variables used in Eq. 3 are illustrated in Fig. 6: τ_i is the examined task affiliated with operator op. According to the application topology, op has n_i upstream operators $\{op_1, ..., op_{n_i}\}$ and n_o downstream operators $\{op_1^{'}, ..., op_{n_o}^{'}\}$. The size of input stream k between operator op_k and op is denoted as S_k, and it can be equally partitioned into communications between constituent tasks of op_k and op based on the stream balancing assumption. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2720622, IEEE Transactions on Big Data

parallelism degree of op_k is denoted as P_{op_k}, and Adj_k is the number of spawned tasks among them that are collocating with τ_i on the same node. Similar notations also apply to the downstream operators for the convenience of presentation.

Using the notations illustrated in Fig. 6, we also present the formulation of memory consumption for task τ as below:

$$M_{\tau_i} = (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S_k^{'}) \frac{m_{pt}}{P_{op}} \quad (4)$$

In order to minimize C_{τ_i}, putting successive tasks on the same node is always encouraged as long as the target node still has sufficient capacity to accommodate their aggregated resource demands. Therefore, by modelling the problem as a special case of two-dimensional vector bin-packing, we have already considered the common requirement of task placement — reducing the inter-node communication whenever possible. The solution to this problem should be a trade-off between consolidating tasks and preventing resource contention in each node.

As for the problem complexity, the classical bin-packing is already NP-Hard as reduced from the PARTITION problem [11]. However, with overlapping tasks whose resource consumption depends on the packing result, our task mapping problem is a more complicated variant that must rely on the use of approximation algorithms to make it tractable.

5.3.2 Heuristics for Solving Bin-packing Problem

We opt for heuristic methods mainly because of efficiency considerations. The scale of the problem increases along with the application performance requirement, which may involve thousands of tasks to be assigned. Having such a huge solution space, it is computational infeasible to search for the optimal result by the use of exact algorithms such as bin completion (BC) [12] and branch-and-price (BCP) [13]. Additionally, packing speed is of crucial importance for the fast deployment of streaming application. P-Deployer may need to invoke the bin-packing process multiple times, with each time the result being verified through profiling as satisfying the model constraint does not necessarily guarantee that it can satisfy the performance requirement. Therefore, we prioritise execution efficiency in the heuristic design, and present a lightweight implementation to make it a good fit for the real-time streaming context.

The proposed solution is analogous to *First Fit Decreasing* (FFD), which is one of the most natural heuristics for bin-packing and is known to be effective in 1-dimensional cases as it is guaranteed to produce a result using less than $\frac{11}{9}OPT+1$ bins (OPT is the optimal solution). FFD is essentially a greedy algorithm that sorts the items in a particular order (normally by descending sizes) and then sequentially places them in the first bin that has sufficient capacity. However, in order to cope with the multi-dimensional and overlapping nature of our problem, this process has to be generalized in three aspects which is shown in Algorithm 1.

As CPU and memory are measured in different metrics, Algorithm 1 first normalizes the task resource demands C_τ and M_τ with regards to machine resource availability W_c and W_m for the ease of comparison in resource scarcity. After this step, each machine can be considered as a bin of unit size and each task is denoted by a 2-d decimal vector.

Algorithm 1: The task mapping and resource estimation algorithm

Input: A task set $\vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be assigned
Output: A machine set $\vec{m} = \{m_1, m_2, \ldots, m_{n_m}\}$ with each machine hosting a disjoint subset of $\vec{\tau}$, where n_m is the number of used machine

```
Normalize resource demands for τ⃗
n_m ← 0
while there are tasks remaining in τ⃗ to be placed do
    Start a new machine m
    Insrease n_m by 1
    while there are tasks that fit in machine m do
        foreach τ ∈ τ⃗ do
            Calculate s(τ, m) (Eq. 5)
            Calculate p(τ, m) (Eq. 6)
        end
        Place the task with the largest p(τ, m) into machine m
        Remove the task from τ⃗
        Update the remaining capacity of machine m
    end
end
return m⃗
```

Secondly, there are two functions introduced to make different tasks comparable in terms of their packing priority. These include: (1) a resource saving function $s(\tau, m)$ that calculates the amount of resource savings if task τ is to be placed on machine m; and (2) a priority function $p(\tau, m)$ that assigns task τ a scalar considering not only the intuition of putting the "largest" item first, but also the inclination to maximize the potential resource savings.

Lastly, since the calculation of $s(\tau, m)$ actually depends on what have already been put into machine m, we adopt a different view to implement this FFD variant. Contrary to the classical "item-centric" FFD implementations that require all tasks to be sorted beforehand and then packed strictly in the pre-calculated order, we implement a different perspective of FFD, called "bin-centric", which allows the packing priority of remaining tasks to be dynamically updated after each task assignment in order to take the current system status into consideration. The task with the largest priority at the moment will be packed next and varied definition of priority would lead to a different FFD heuristic.

Implemented in this way, there is only one machine opened at any time and the algorithm keeps filling it with suitable tasks until there is not enough capacity for any remaining task. The rest of this subsection explains in detail how the algorithm decides the priority order among those "suitable tasks".

Calculating $s(\tau, m)$: when task τ is to be placed on machine m, any previously packed task on this machine that is adjacent to τ will benefit from this assignment by increasing 1 to a particular position of its $\vec{Adj}$ list, which causes their CPU consumption to decrease according to Eq. 3. Note that $\vec{Adj}$ is initialized to all zeros for every task, implying that all unpacked tasks are considered to be external by default when calculating Eq. 3. Analogously, the CPU demand of τ

is also reduced due to task allocation. Therefore, $s(\tau, m)$ can be formulated as follows, where $\tau_k \in \wp(m)$ represents the tasks that have already been packed in machine m:

$$s(\tau, m) = C_{\tau,\max} - C_\tau + \sum_{\tau_k \in \wp(m)} (C_{\tau_k}^{\vec{Adj}} - C_{\tau_k}^{\vec{Adj}+1}) \quad (5)$$

Calculating $p(\tau, m)$**:** the task priority function is designed based on the following considerations:

1) Resource saving is prioritised as it encourages communicating tasks to be packed in a tightly manner.
2) If resource savings are similar or there is no saving from the placement of the rest tasks, the heuristic picks the one that best fills the remaining capacity of the opened machine.
3) Each resource dimension is properly weighted to reflect the relative scarcity, so that when the mapping problem is in essence confined by one type of resource, the heuristic will be dominated by this dimension and reduced to 1-d FFD when necessary. This can be achieved by assigning a larger coefficient to the scarce dimension.

Therefore, $p(\tau, m)$ is formulated as follows:

$$p(\tau, m) = a_r \cdot s(\tau, m) - a_c(C_\tau - r_m^{cpu})^2 - a_m(M_\tau - r_m^{mem})^2 \quad (6)$$

r_m^{cpu}, r_m^{mem} represent the remaining capacities of machine m, and a_r, a_c, a_m are weight coefficients that adjust the importance of each term. While a_r is chosen as a user parameter, a_c, a_m can be calculated as the average task demand in each dimension: $a_c = \frac{1}{n}\sum_{k=1}^{n} C_{\tau_k}, a_c = \frac{1}{n}\sum_{k=1}^{n} M_{\tau_k}$.

6 Implementation

The setup of the profiling environment has been briefly introduced in Sec. 4. More specifically, the Message Generator in Fig. 4 is a Java program that reads the workload file on-demand to emit a particular size of profiling stream; while the Message Queue is a distributed queueing system implemented on Twitter Kestrel[6] that enables controllable message buffering. The use of Thrift interface of Kestrel allows P-Deployer to easily retrieve the length of the message queue and further determine whether the streaming application has been overwhelmed by the profiling data.

Following the MAPE working process, Fig. 7 describes how P-Deployer is integrated with Apache Storm in our prototype. Apache Storm is selected as our target DSMS not only because it is widely adopted in both academia and industry, but also it offers a built-in metric system and Flux – an external configuration reader that greatly facilitates the application of versatile deployment schemes.

Monitor: the Metric Reporter in Fig. 4 is actually implemented by two components that collect system level and application level metrics, respectively. The collected information is then stored in MongoDB[7] and processed in order to determine the task profile attributes listed in Table 1. The Low Level Metric Reporter running in each Worker Node consists of an external statistics collection daemon — collectd[8] and several extended Storm modules. Specifically, collectd runs alongside the Supervisor daemon to probe the CPU and memory utilization of the Worker Process with a resolution of 10 seconds. In the storm-core, we implement the Task Wrapper that encapsulates the task execution with the logic of sampling and reporting resource consumption at the task level: the CPU consumption of operators' *execute* method (c_p) is probed using the *ThreadMXBean* class, while the memory consumption (m_{pt}) is obtained through retrieving the size of the operator state. Recalling the fact that we classify the overall CPU consumption obtained at the process level into processing cost and transmission cost, the CPU consumption of tuple transmission (c_t) is a derived metric that requires the comparison between the task level statistics and the process level statistics. To avoid excessive profiling overhead, P-Deployer sets the default sampling rate to 0.05, i.e. selecting 1 out of 20 tuples to collect and report metrics. Also, we provide the Topology Adapter in storm-core that seamlessly hides the adaptation effort on the application level to leverage this profiling framework.

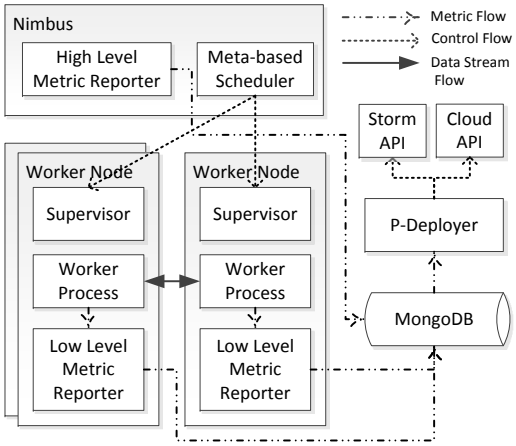

Fig. 7. The integration of P-Deployer into Apache Storm.

The High Level Metric Reporter, on the other hands, collects metrics on the application performance. Some of them are directly obtained from the UI daemon on Nimbus, such as the stream load between different operators (S_k, S_k'), the sojourn time of task profile (t_{soj}), and the application complete latency (average time taken for a tuple and all its offspring to be completely processed by the topology). Some metrics, however, need specific post-processing. For example, there is no default definition for throughput, therefore, the reporter calculates the overall throughput of a streaming application based on its monitoring interval as well as the observations on the accumulative number of acknowledgements or emitted data, depending on whether the application adopts reliable message processing or not.

Analyse and Plan: P-Deployer, implemented as a single Java program, comprises both the analytical and planning functionalities. It queries the MongoDB for the latest metrics and applies a set of boundary check rules to define the current application state and update the task profile accordingly. The newly updated task profile will trigger

6. https://github.com/twitter-archive/kestrel
7. https://www.mongodb.com/
8. https://collectd.org/

the planning phase to be performed again, resulting in a deployment plan that specifies the number of machines used and the assignment location of each task.

Execute: changes to the virtual infrastructure are made possible by using Apache jclouds[9], where any new provisioned machine is initialized from a image that already has Storm pre-configured. For actual application deployment, P-Deployer uses an external YAML file to specify the parallelism degrees of operators and submit the streaming application to Nimbus by invoking the Storm CLI. The extended Meta-based scheduler guarantees that each task is assigned to its designated Supervisor and Work Node.

7 Performance Evaluation

To validate the correctness and efficiency of the proposed prototype, we have conducted three different sets of experiments:

1) The applicability evaluation validates whether P-Deployer is capable of deploying a variety of streaming applications towards their pre-defined performance targets. The test applications incorporate different topology structures in order to verify the applicability and robustness of P-Deployer.
2) The scalability evaluation shows the runtime behaviour of P-Deployer using relative large test cases, where the application to be deployed has a more complicated topology structure or a higher performance target. The runtime overhead of P-Deployer is also assessed in this experiment.
3) The cost efficiency evaluation compares P-Deployer with the state-of-the-art platform-oriented method in terms of resource usages, as they endeavour to achieve the same performance target in deployment.

7.1 Experiment Setup

The experiment environment is set up on a private cloud supported by OpenStack, which is located in CLOUDS lab at the University of Melbourne. The environment consists of three IBM X3500 M4 machines, and each machine is equipped with 2 x Intel Xeon E5-2620 Processor (6 core@2.0GHz), 64 GB RAM and 2.1 TB HDD. The virtual cluster deployed on the physical environment is composed of a Nimbus Node, a ZooKeeper Node and several on-demand Worker Nodes. All machines are provisioned from the same "m1.medium" template (2 VCPU and 4 GB RAM).

The used streaming applications (a.k.a topologies) and the evaluation methodology are discussed below.

7.1.1 Testing topologies

There are five testing topologies — three synthetic and two drawn from the real-world streaming scenarios.

Micro-benchmark: the synthetic topologies, collectively called *micro-benchmark*, evaluate how P-Deployer generalizes to different topology structures. As shown in Fig. 8, micro-benchmark includes three common structures: *Linear*, *Diamond*, and *Star*, covering the cases where an operator has (1) one-input-one-output, (2) multiple-outputs or multiple-inputs, and (3) multiple-inputs-multiple-outputs, respectively.

9. http://jclouds.apache.org/

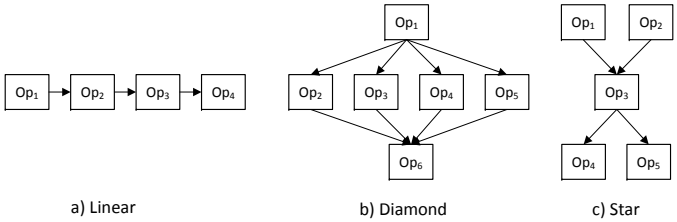

Fig. 8. The synthetic Micro-benchmark topologies, the structures of which are referenced from R-Storm [9].

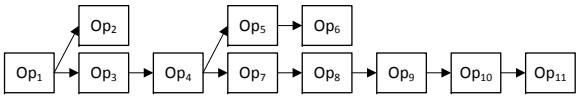

Fig. 9. The Twitter Sentiment Analysis topology.

The *execute* method of each operator is implemented in one of the four patterns in order to reflect different operator types. Specifically, CPU bound operators invoke a random number generation method *Math.random()* 30000 times to generate a significant amount of CPU consumption, while I/O bound operators only apply a JSON parse operation on the incoming tuple for fast processing; Sojourn time-bound operators sleep for 10 milliseconds upon any tuple receipt, and Memory-bound operators temporarily store any message received, maintaining a sliding window with 300 seconds length and 60 seconds sliding interval.

To satisfy the stream balancing assumption, all operators are connected through shuffle-grouping — a stream routing mechanism that evenly partitions internal streams across the receiving tasks. Besides, to generate a relative large internal stream for I/O bound operators to mimic saturated network usages, each operator has a function implemented to adjust its operator selectivity[10] using an external configuration file.

Word Count: please see Sec. 3 for its description.

Twitter Sentiment Analysis: This topology, as shown in Fig. 9, is adapted from a mature open-source project hosted on Github[11]. It has 11 operators constituting a tree style topology that has 8 stages in depth. In terms of processing logic, once a new tweet is pulled into the system (through Op_1, a Kestrel Spout), it is first preserved by a file writer (Op_2) and examined by a language detector (Op_3) to identify which language it uses. If it is written in English, there is a sentiment analysis operator (Op_4) that splits the sentence and calculates the sentimental score for the whole content using AFINN[12], which contains a list of words with their pre-computed sentiment valence from minus five (negative) to plus five (positive). There are also several operators to count the average sentiment result (Op_5, Op_6) and to rank the most frequent hashtags occurring over a specific time window ($Op_7 \sim Op_{11}$).

Note that all the above-mentioned topologies process the same type of workload, as the profiling stream is generated from the same workload file containing 159,620 tweets in

10. Selectivity is an operator property that denotes the number of stream data produced per data consumed.

11. https://github.com/kantega/storm-twitter-workshop

12. http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010

JSON format that collected from 24/03/2014 to 14/04/2014. Topologies are also configured with acknowledgements enabled so as to track the complete latency during the runtime.

7.1.2 Evaluation Methodology

For quantitative comparison of different deployment plans, we introduce the metrics considered as well as the measurement method in our experiments.

Deployment Metrics: throughput and complete latency are the two performance metrics that evaluate the quality of deployment. We deem a deployment plan to be performance satisfactory, as long as it results in a higher throughput than the pre-defined target whilst meeting the latency constraint.

The number of used machines is the only metric that evaluates the cost-efficiency of the proposed deployment plan. In this platform, the user budget of deployment is set to 16 Worker Nodes, which is the maximum capacity of our private cloud if we make sure that each VCPU corresponds to a physical core.

Measurement Method: the application is first deployed on the Storm cluster according to its designated plan. During this process, the number of tasks is set to be same as the number of executers and each Worker Node has only one Worker Process, which conforms to the recommendation settings provided by the Storm community[13]. The deployed topology will execute for 15 minutes to sufficiently stabilize before any measurements are taken. To obtain an average performance result, performance data are collected every 30 seconds for consecutively 5 minutes. Therefore, each iteration of the MAPE loop has a timespan of 20 minutes.

The performance metric needs to report the maximum throughput sustained by the deployed application. To this end, the profiling environment feeds the application with enough size of inputs, lifting the performance to the highest stable point before comparing it with the desired target.

For clarity, the settings of model parameters used in the deployment plan generation are summarised in Table 4.

TABLE 4
Parameter settings used in the deployment planning model

Parameter	Value
α (Upper limit on CPU usages for I/O bound task)	0.1
β (Upper limit on memory usages for Mem-bound task)	512 (MB)
a_r (Coefficient in Eq. 6)	1

7.2 Applicability Evaluation

In this evaluation, we use P-Deployer to deploy all the five topologies towards a designated performance target — a throughput of 2000 tweets/second and a maximum complete latency of 500 ms.

To better examine the applicability of P-Deployer, we have configured the micro-benchmark topology with different time and resource complexities. In particular, the Linear topology includes only CPU-bound operators so that the whole topology is bound by computation; the Star topology, on the contrary, is bound by communication for being made of I/O bound operators with comparatively large internal streams; while the Diamond topology incorporates all types of operators in the middle so as to simulate a hybrid case.

13. http://storm.apache.org/releases/current/FAQ.html

Fig. 10 demonstrates that P-Deployer has been able to steadily improve the topology throughputs and finally realize the pre-defined performance target. The evaluated topologies, despite their different topology structures and resource complexities, have shown a similar scaling pattern during the deployment process: in the first iteration, the micro-benchmark topologies respectively achieve 78%, 68%, and 75% of the performance target; while the Twitter Sentiment Analysis delivers a average throughput of 1237 tweets/second that only accounts for 62% of the requirement. The following MAPE iterations essentially lead to a horizontal scaling up process. P-Deployer gradually exaggerates the unit resource consumption reflected in tasks profiles, resulting in a higher operator parallelism and more Worker Nodes to be added into the Storm cluster. The bin-packing nature of the task mapping algorithm guarantees that only a necessary amount of machines would be introduced and efficiently utilized in the next MAPE iteration. Taking the Linear topology as an example, it initially has a parallelism degree of (1,1,1,1)[14] running on 3 Worker Nodes, but at the end of the scaling process, it spreads over 6 machines with a parallelism degree of (1,2,2,2), and P-Deploy took care not to collocate CPU-bound tasks to fulfil its performance requirement. We also observe that platform scaling is the major source of performance improvement. In case of the Star topology, the performance boost in the third iteration is significantly larger (10.3 x) than the previous one, as it involves a new machine to be added rather than merely adjusting the operator parallelism and task allocation.

There are two reasons why the task profiles tend to be underestimated in the first place. (1) The overhead of DSMS and operating system to run the streaming application has not been explicitly considered in our resource consumption model due to the complexity of formulation and measurement. Therefore, as the throughput grows, the increasing overheads of acknowledgement and thread scheduling need to be amortized onto the unit resource consumption. (2) Some operators, especially those performing batch processing or window slide at set intervals, demonstrate a periodical spiky pattern of resource usages even when processing a steady size of stream. Though we should allocate resources to satisfy the peak need of the operator, the spiky period is very short and thus hard to be captured accurately. As a compromise, we enlarge the average value in task profiles so as to ensure the smooth flow of execution.

Another finding from these experiments is that the complete latency consistently grows as the throughput increases. In our measurement, the average complete latency of the Linear topology is 89 ms in the first iteration, but it raises to 159 ms after the deployment process is finalized. We understand the result in the sense that the complete latency is strongly correlated to the number of unacknowledged tuples that are allowed in the system. As there is no latency violation identified in these experiments, P-Deployer currently does not have the ability to throttle the data source. However, such function can be readily implemented using the built-in rate-limiting mechanism[15].

14. From left to right, each number corresponds to the number of tasks for each operator in the Linear topology

15. In Apache Storm, we refer to *max.spout.pending* option for details.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2720622, IEEE Transactions on Big Data

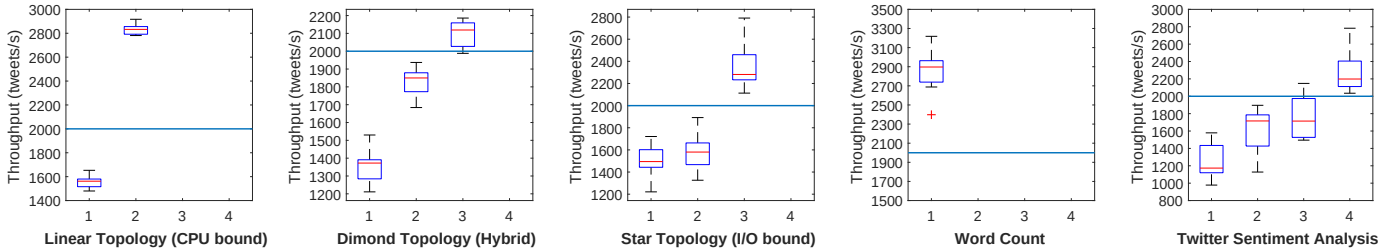

Fig. 10. The monitored throughputs of topologies in the applicability evaluation. Each MAPE iteration corresponds to a plotted box that contains 10 readings of throughputs. P-Deployer finalizes the deployment once the performance target denoted by the horizontal line is reached.

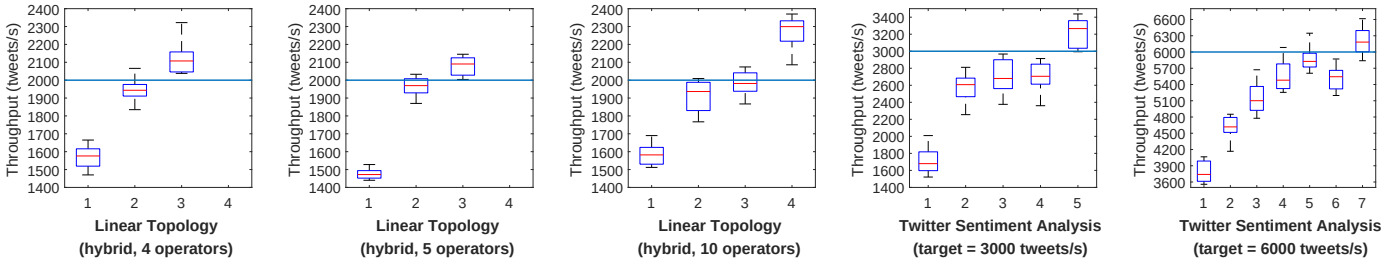

Fig. 11. The monitored throughputs of topologies in the scalability evaluation. Each MAPE iteration corresponds to a plotted box that contains 10 readings of throughputs. P-Deployer finalizes the deployment once the performance target denoted by the horizontal line is reached.

7.3 Scalability Evaluation

Two separate experiments have been conducted to evaluate the scalability of P-Deployer. In the first experiment, the application to be deployed is the Linear topology that consists of various operator types. We further extend the depth of the topology to 5 and 10 so as to generate more complicated topology structures, but the performance target of deployment remains the same as the applicability evaluation.

In the second experiment, we attempt to deploy the Twitter Sentiment Analysis towards higher throughput targets of 3000 tweets/second and 6000 tweets/second, respectively.

Fig. 11 shows that the increasing depths of the Linear topology have little impact on the convergence speed of the P-Deployer, which is the number of iterations required to achieve the desired performance. P-Deployer consistently improves the throughput performance by gradually scaling out the task distribution and solving any bottlenecks caused by resource contention.

However, it takes more efforts for the Twitter Sentiment Analysis to realize a higher performance target and the number of used machines does not linearly scale with it (reaching 3000 tweets/seconds requires 5 nodes while realising 6000 tweets/seconds requires 14 nodes). We also observe a throughput oscillation at the iteration 6 when targeting at the higher throughput. This result is not beyond our expectation due to the fact that P-Deployer works on a best-effort basis and thus cannot guarantee the performance bottleneck to be necessarily solved by the adjustment of deployment. Some DSMS concurrency settings, such as the size of the thread pool and the number of the acker tasks, are also influencing the topology behaviour [14], but they have not been considered in P-Deployer due to the hardness of generalization. Despite this, P-Deployer is still qualified to be a practical deployment tool in the situation where the default settings of DSMS suffice for the performance goal.

During the deployment process in which the Twitter Sentiment Analysis reaches the 6000 tweets/second target, we assessed P-Deployer's runtime overhead, including the profiling cost (the percentage decrease in average throughput after enabling the profiling mechanism) and the running time of the task mapping and resource estimation algorithm. The result is shown in Table. 5, which demonstrates that our profiling mechanism is low-overhead and the FFD heuristic is sufficiently fast for solving the overlapping bin-packing problem.

TABLE 5
The profiling cost (P_c) and the running time of Algorithm 1 (R_t) when deploying the Twitter Sentiment Analysis topology.

Iteration	1	2	3	4	5	6	7
P_c	2.68%	3.14%	3.48%	4.11%	3.72%	2.95%	3.3%
R_t	0.213	0.245	0.312	0.338	0.331	0.351	0.359

7.4 Cost Efficiency Evaluation

7.4.1 Comparable Method

We choose a state-of-the-art platform-oriented approach — Metis scheduler as our comparable method, which outperforms the greedy scheduler [1] in major metrics such as throughput and resource utilization [7]. In general, the Metis scheduler uses the number of machines in the platform as the parallelism degree of each operator; it then builds the task communication graph based on the monitoring of internal streams and translates the task mapping problem into a graph partitioning problem. Metis[16] is used as an external service that solves the partitioning problem with a goal to balance the CPU usage and bandwidth consumption across the platform.

16. http://glaros.dtc.umn.edu/gkhome/views/metis

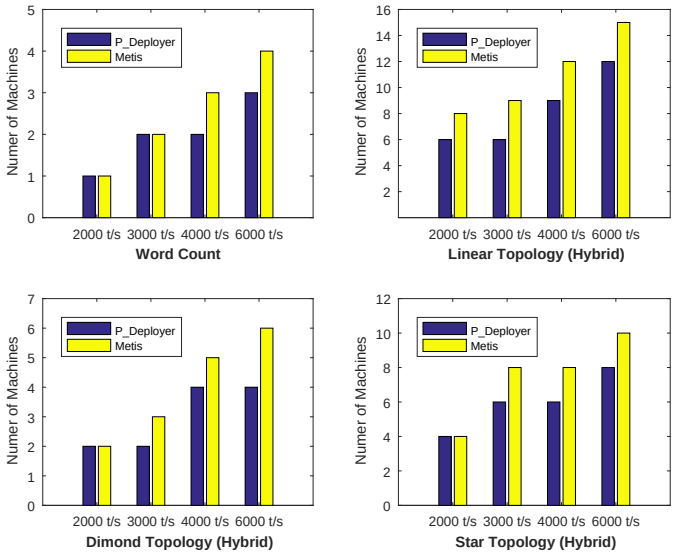

Fig. 12. Comparison of P-Deployer and the Metis scheduler against the resource costs required to reach the same performance target

7.4.2 Result and Analysis

In order to make the result comparable, we control the performance target of deployment and compare P-Deployer and the Metis scheduler against the minimal number of machines required to achieve the same target. The test applications include the Word Count topology and the Micro-benchmark topology with different types of operators.

Fig. 12 demonstrates that, in most cases, P-Deployer occupies less resources than the Metis scheduler does. As seen in the deployment of the Word Count topology, P-Deployer is able to use 1 less Worker Nodes than that of Metis scheduler to reach the 6000 tweets/second target. When the Metis scheduler is applied, we observed that the capacities[17] of the last three operators reached up to 0.808, 0.494, and 0.505, respectively, but the CPU utilization of each Worker Node barely exceeded 35%. This leads to a conclusion that the operator parallelism is underestimated using the number of available machines, and the insufficient operator parallelization would in turn impede the high utilization of the underlying platform.

On the other hand, disregarding the operator type in task mapping also caused significant performance degradation. We have compensated the insufficient parallelism (multiplied by 5) for Sojourn time-bound operators when using the Metis schedule to deploy the synthetic topologies, yet still the cost of processing is noticeably higher than that of P-Deployer, with $2 \sim 3$ more machines required in each case to reach the highest target. The performance disparity is attributed to load imbalance that resulted from the inaccurate workload modelling. The Metis scheduler only approximates the Worker Node load by counting the number of tuples being transmitted, as compared to our approach that firstly models the resource consumption on the fine-grained task level, and then deduces the workload of each machine by additively accruing the resource consumption of collocating tasks.

17. Capacity represents the percentage of the time in the observation time window that the operator spent executing inputs.

8 Related Work

There is a rich body of literature on the deployment of streaming applications, each associated with different optimization targets.

Some papers aim to improve the throughput and resource utilization. Fisher et al. employed Bayesian optimization to tune the operator parallelization and other configurations for achieving higher throughputs [14]. However, treating the streaming application as a blackbox function does not properly take advantage of the domain knowledge, and it results in a comparatively long convergence time. The Metis scheduler, proposed by the same group, avoids the convergence problem by building up a task communication graph and solving it with full-fledged partitioning software [7]. Nevertheless, as shown in our experiments, the resource utilization and workload balancing could still suffer without considering the operator type and parallelism holistically. There is also a resource-aware scheduler that allows users to specify the resource demand for each task in order to optimize workload distribution and inter-node communication [9]. However, our experience has suggested that the resource demand of a task strongly correlates to its stream load and communication pattern, which can only be obtained at runtime through application profiling.

Some works, on the other hand, investigated adaptive deployment to build elastic streaming applications. To maintain high resource utilization, Vanderveen et al. [15] employed MAPE loops to elastically scale the streaming application along with the workload changes. But the adopted threshold method much resembles the auto-scaling policy in Amazon Web Services (AWS) and is inadequate to model the particularity of target applications. To honour the latency constraint during scaling, Fu et al. [8] proposed a dynamic resource scheduler that tailors the deployment plan to the fluctuating workload. The performance model is a rigorous queueing network that ignores network transmission costs, which makes it only applicable to computationally intensive streaming applications.

Workload estimation and latency modelling has also been discussed in the literature to make the adaptation process more pro-active, yet still the issue of cost-efficient scaling remains without optimizing the deployment holistically. Zacheilas et al. [16] predicted the workload characteristics in order to choose the appropriate transitions on parallelism, but they did not consider the operator communication pattern to reduce network overhead. Li et al. [4] presented a predictive scheduling framework, utilizing Support Vector Regression to predict the complete latency under certain deployment arrangement. But they left out operator parallelization, and the proposed scheduling algorithm is essentially an exhaustive search to traverse all feasible solutions. Nevertheless, the established optimization techniques can be integrated with P-Deployer to choose the right time of scaling and minimize the negative impacts of dynamic workload adaptation. With the knowledge of predicted workload distribution, Mayer et al. [17] investigated dynamic stream partitioning using queueing theory models and time series analysis. Their work can be integrated with our operator parallelization model to dynamically adjust the operator parallelism, providing probabilistic guarantees on the buffering limit. By modelling the latency spike created

by operator movements, Heinze et al. [18] proposed an elastic placement algorithm that reduces the number of latency violations. This work can be used in tandem with P-Deployer, replacing the proposed bin-packing model that solely commits to reducing the resource cost. Cardellini et al. [19] investigated the optimal operator placement as an Integer Linear Programming problem. The computed result can be used to evaluate the quality of our heuristic solution. Also, to speed up the process of finding optimal solutions to resource allocation/placement problems, evolutionary algorithms [20], [21] and machine learning based frameworks [4], [22] have been investigated in the literature.

There are also several papers on reducing inter-node communication [1], [6], [10], [19] and improving system manageability and energy-efficiency [2], [23]. However, all the above-mentioned efforts fall short when the deployment of streaming application has been commissioned with a specific performance target, the case of which is commonly seen in the cloud computing context.

9 Conclusions and Future Work

In this work, we proposed P-Deploy – an automated, cost-efficient and performance-oriented deployment framework for running streaming applications on cloud. Inspired by the observations that drawn from real experiments, P-Deployer adopts a Monitor-Analyze-Plan-Execute (MAPE) architecture working in a profiling environment that gradually scales the streaming application to approach its performance target. In each iteration, P-Deployer builds the relationship between performance behaviour and resource consumption at the fine-grained task level, decides the operator parallelism based on its profile and performance demand, and then solves the task mapping and resource estimation problem as a variant of bin-packing. The evaluations demonstrated that P-Deployer is able to deploy a variety of streaming applications towards their performance target, and it outperforms the state-of-the-art platform-oriented approach in terms of cloud resource usages. Since the cloud environment benefits from the emerging architecture used in data centres, it is of crucial importance to make the deployment process aware of the hardware heterogeneity and allow customized resource provisioning that considers application characteristics and requirements. P-Deployer is our first attempt to advance the research of this topic.

In the future, we plan to extend P-Deployer with the ability to dynamically adapt streaming applications to the workload fluctuations, which includes autonomously adding/releasing resources according to the varying performance requirement, and taking advantage of heterogeneous resources to exhibit full auto-scaling characteristics.

References

[1] 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. 208–218.

[2] P. Basanta-Val, N. Fernández-García, A. Wellings, and N. Audsley, "Improving the predictability of distributed stream processors," *Future Generation Computer Systems*, vol. 52, pp. 22–36, 2015.

[3] L. Eskandari, Z. Huang, and D. Eyers, "P-Scheduler: adaptive hierarchical scheduling in apache storm," in *Proceedings of the Australasian Computer Science Week Multiconference*. ACM Press, 2016, pp. 1–10.

[4] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *Proceedings of the IEEE International Conference on Big Data*. IEEE, 2015, pp. 333–338.

[5] Y. Liu, X. Shi, and H. Jin, "Runtime-aware adaptive scheduling in stream processing," *Concurrency and Computation: Practice and Experience*, vol. 28, pp. 1–14, 2015.

[6] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-Aware Online Scheduling in Storm," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 535–544.

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

[8] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 411–420.

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

[10] 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*. ACM, 2014, pp. 1579–1588.

[11] M. R. Garey and D. S. Johnson, *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[12] A. S. Fukunaga and R. E. Korf, "Bin-completion algorithms for multicontainer packing and covering problems," *Journal of Artificial Intelligence Research*, vol. 28, pp. 393–429, 2007.

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

[14] L. Fischer, S. Gao, and A. Bernstein, "Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization," in *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 22–31.

[15] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *Proceedings of the 2015 IEEE First International Conference on Big Data Computing Service and Applications*. IEEE Computer Society, 2015, pp. 154–161.

[16] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *Proceedings of the 2015 IEEE International Conference on Big Data*. IEEE, 2015, pp. 213–222.

[17] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *IEEE Internet of Things Journal*, vol. 2, no. 4, pp. 274–286, 2015.

[18] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[19] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM Press, 2016, pp. 69–80.

[20] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu, "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1230 – 1242, 2013.

[21] E. Feller, L. Rilling, and C. Morin, "Energy-aware ant colony based workload placement in clouds," in *Proceedings of the IEEE 12th International Conference on Grid Computing*. IEEE, 2011, pp. 26–33.

[22] X. Zuo, G. Zhang, and W. Tan, "Self-adaptive learning pso-based deadline constrained task scheduling for hybrid iaas cloud," *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 2, pp. 564–573, 2014.

[23] T. Matteis and G. Mencagli, "Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing," in *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, pp. 1–12.

Xunyun Liu received the B.E. and M.E degree in Computer Science and Technology from the National University of Defense Technology in 2011 and 2013, respectively. He is currently working towards the PhD degree in Computer Science at University of Melbourne. His research interests include stream processing and distributed systems.

Rajkumar Buyya is a Fellow of IEEE, Professor of Computer Science and Software Engineering. He is also a Future Fellow of the Australian Research Council and the Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia.