Robust Resource Management in Distributed Stream Processing Systems

Xunyun Liu

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

April 2018

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

Robust Resource Management in Distributed Stream Processing Systems

Xunyun Liu
Principal Supervisor: Prof. Rajkumar Buyya

Abstract

Stream processing is an emerging in-memory computing paradigm that ingests dynamic data streams with a process-once-arrival strategy. It yields real-time insights by applying continuous queries over data in motion, giving birth to a wide range of time-critical applications such as fraud detection, algorithmic trading and health surveillance.

Resource management is an integral part of the deployment process to ensure that the stream processing system meets the requirements articulated in the Service Level Agreement (SLA). It involves the construction of the system deployment stack over distributed resources, as well as its continuous adjustment to deal with the constantly changing runtime environment and the fluctuating workload. However, most existing resource management techniques are optimised towards a pre-configured deployment platform, thus facing a variety of challenges in resource provisioning, operator parallelisation, task scheduling, and state management to realise robustness, i.e. maintaining a certain level of performance and reliability guarantee with minimum resource costs.

In this thesis, we investigate novel techniques and solutions for robust resource management to tackle arising challenges associated with the cloud deployment of stream processing systems. The outcome is a series of research work that incorporate SLA-awareness into the resource management process and relieve the burden of the developers to monitor, analyse, and rectify the performance and reliability problems encountered during execution. Specifically, we have advanced the state-of-the-art by making the following contributions:

1. A stepwise profiling and controlling framework that improves application performance by automatically scaling up the parallelism degree of streaming operators. It also ensures proper resource allocation between data sources and data sinks to avoid processing backlogs and starvation.

2. A resource-efficient scheduler that monitors the application execution, models the resource consumption, and consolidates the task placement for improving cost efficiency without causing resource contention.

3. A replication-based state management framework that masks state loss in the cases of node crashes and JVM failures, which also reduces the fault-tolerance overhead by eliminating the need of synchronisation with a remote state storage.

4. A performance-oriented deployment framework that conducts iterative scaling of the streaming application to reach its pre-defined targets on throughput and latency, regardless of the initial amount of resource allocation.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Xunyun Liu, 15 April 2018

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2- 6 and are based on the following publications:

- **Xunyun Liu** and Rajkumar Buyya, "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review and Future Directions," *ACM Computing Surveys*, ACM Press, 2018 (under review).

- **Xunyun Liu**, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu and Rajkumar Buyya, "A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, Volume 12, Issue 4, Pages: 1-33, ACM Press, 2017.

- **Xunyun Liu** and Rajkumar Buyya, "Performance-Oriented Deployment of Streaming Applications on Cloud," *IEEE Transactions on Big Data (TBD)*, Accepted, In press, DOI:10.1109/TBDATA.2017.2720622 Pages: 1-14, IEEE, 2017.

- **Xunyun Liu**, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein and Rajkumar Buyya, "E-Storm: Replication-based State Management in Distributed Stream Processing Systems," *in Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, Bristol, UK, Pages: 1-10, IEEE, 2017.

- **Xunyun Liu** and Rajkumar Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," *in Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, China, Pages: 1-8, IEEE, 2017.

- **Xunyun Liu** and Rajkumar Buyya, “Dynamic Resource-Efficient Scheduling in Data Stream Management Systems Deployed on Computing Clouds,” *ACM Transactions on Internet Technology (TOIT)*, ACM Press, 2017 (Under review).

- **Xunyun Liu**, Amir Vahid Dastjerdi and Rajkumar Buyya, “Stream Processing in IoT: Foundations, State-of-the-Art, and Future Directions,” *Internet of Things: Principles and Paradigms*, Pages: 145-161, Morgan Kaufmann, 2016.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my principal supervisor, Professor Rajkumar Buyya, for guiding me tirelessly and providing valuable research insights and continuous support throughout my PhD, without which I could never reach this milestone of writing acknowledgement. In addition, I wish to thank my co-supervisor, Dr. Benjamin Rubinstein, for generously sharing with me his knowledge and vision when I need a pair of professional eyes to analyse the real nature of the problem. I also sincerely thank Dr. Rodrigo N. Calheiros for being my co-supervisor while working at the University of Melbourne. My supervisors helped me to grow from a junior researcher, and their encouragement is indispensable to the completion of this thesis. I had many fruitful discussions with them that shed lights on the future research as well.

I would like to thank the chair of my PhD committee Professor Shanika Karunasekera for her support and guidance. I also want to express my gratitude to Dr. Aaron Harwood, Dr. Adel Nadjaran Toosi, and Dr. Amir Vahid Dastjerdi for their time generously spent on answering my questions and proofreading my papers.

Special thanks go to Dr. Chenhao Qu. As a senior student at CLOUDS lab, he offered me guidance when I was struggling to choose an approach for implementing feasible solutions. Besides, I would like to thank all members of the CLOUDS lab and acknowledge the support of the fellow PhD graduates and students. Thank you to Deepak Poola, Nikolay Grozev, Maria Rodriguez, Bowen Zhou, Yaser Mansouri, Jungmin Jay Son, Safiollah Heidari, Caesar Wu, Minxian Xu, Sara Moghaddam, Muhammad H. Hilman, Redowan Mahmud, Md. Tawfiqul Islam, Shashikant Ilager, TianZhang He, etc.

I acknowledge the China Scholaship Council, the University of Melbourne, and the Nectar Cloud for providing me with financial support and appropriate facilities to pursue this doctoral degree. I am also grateful to the administrative staff at the School of Computing and Information Systems, especially Rhonda Smithies, Madalain Dolic, and Julie Ireland, for their support in my numerous applications.

I would like to thank my parents who have encouraged me to embark on this wonderful journey, and my girlfriend, Miss Yi Fan, for her company as we taste all the sweets and bitters of pursuing a PhD. I cannot imagine a greater fortune other than having them in my life. Their love and care are indispensable to any of my achievements.

Xunyun Liu
Melbourne, Australia
April 2018

Contents

List of Figures

List of Tables

Chapter 1
Introduction

THE recent development of Information and Communications Technology (ICT) has led us to the Big Data era, where a wide range of Internet-scale web applications and the growing number of interconnected intelligent devices have created a tremendous data explosion. Take the prevalence of Internet of Things (IoT) as an example, Gartner estimated that there were approximately 8.4 billion of connected devices in 2017, and this figure is expected to reach 20.4 billion by 2020.[1] International Data Corporation (IDC) further asserted that the amount of data being generated in a year will mushroom from 4.4 zettabytes in 2013 to 44 zettabytes in 2020, exhibiting a ten-fold increase within seven years of development.[2] With unstructured machine-generated information being created at such an unprecedented scale and speed, new challenges have arisen in different phases of the data life cycle, spanning across collection, aggregation and transmission, to processing, updating and visualisation.

The explosive growth of data generation has also been accompanied by the surging demands for real-time data processing, which poses unprecedented challenges to the underlying IT infrastructure and processing paradigms. In many established databases and MapReduce frameworks, the batch processing approach has been adopted for handling large volume of data chunks with scalability and accuracy. However, this store-and-process model is not suitable for processing real-time data streams due to the limitation of storage capacity and the strict latency constraint. Therefore, stream processing, a new in-memory paradigm that supports in harnessing the potential of transient data in motion, has emerged to cope with the velocity requirement of big data. Instead of applying one-off queries to the static data as a series of batch jobs, stream processing adopts

[1] https://www.gartner.com/newsroom/id/3598917
[2] https://www.emc.com/leadership/digital-universe/2014iview/index.htm

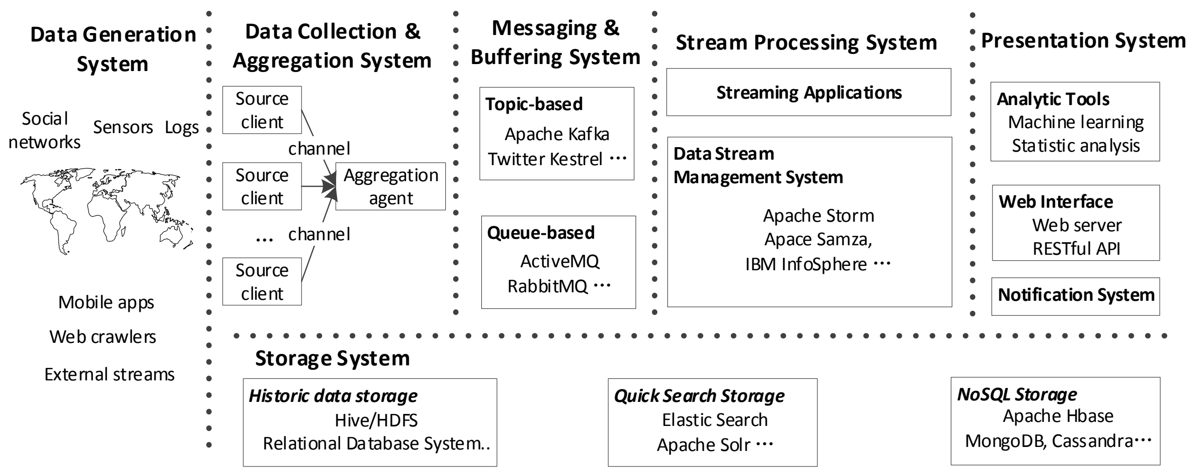


Figure 1.1: The data lifecycle in a complete stream processing scenario

the process-once-arrive philosophy to achieve low processing latencies on volatile data streams, where massively parallel architectures are also investigated to power real-time data analysis in a distributed environment.

Since stream processing is an umbrella term that describes the activities related to the collection, integration, analysis, visualisation, and system integration of stream data, it is necessary to define the scope of a distributed stream processing system. Fig. 1.1 depicts the complete data life cycle in stream processing from its generation to consumption. Based on the particular functionality, the whole data path has been broken into six separate streaming components that are responsible for data generation, collection, buffering, processing, storage and presentation, respectively. This thesis mainly focuses on the resource management in the processing component, which is conveniently referred to as a stream processing system in the rest of this thesis.

Fig. 1.2 further illustrates a distributed stream processing system as a layered structure consisting of streaming applications, middleware, and a distributed infrastructure. The kernel of a streaming application is a continuous query that is submitted by the developer upon the completion of the development cycle. Its implementation is a set of inter-connected operators that stand on the incoming streams, filtering data continuously unless being explicitly terminated. In the example streaming application, the vertices denote different operators encapsulating certain streaming logic such as split and join; whereas the edges represent dynamic data streams that concatenate the upstream

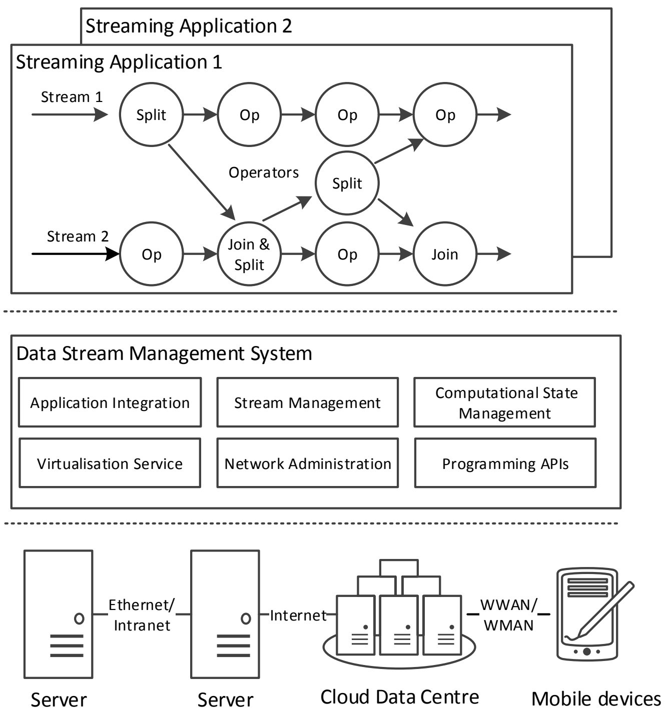


Figure 1.2: An example of a distributed stream processing system

and downstream operators. The relevant query result is incrementally updated as the application inputs traverse through the processing platform, producing a timely response with sub-second latencies. Lying at the middleware layer is a Data Stream Management System (DSMS), which serves a similar role as the conventional database management systems (DBMS) in a batch processing framework to bridge the user application with the operating systems of different computing nodes. As middleware, a DSMS provides application integration, stream management and a variety of other services to the developers, while abstracting away the complexity of dealing with the concurrent infrastructure elements and heterogeneous network structures. At the bottom of the structure, the infrastructure of a distributed stream processing system may include various resource types ranging from mobile devices to virtual and physical servers, the location of which could be geographically distributed to cater to the particular streaming scenarios.

Resource management is an integral part of the deployment process to improve system performance and reduce the execution cost. During the process of building the layered structure shown in Fig. 1.2, resource management determines where and how the user-defined streaming logic is executed in a real-time distributed environment. Specifically, it involves the construction and configuration of the DSMS and the infrastructure layer, as well as managing the cross-layer relationships such as the parallelisation and mapping of abstract streaming logic to the concrete threads and processes. It should be noted that managing resources in a distributed stream processing system is markedly different to what we have been familiar with in a batch processing framework. Traditionally, the one-off queries are optimised towards the execution platform for performance improvement, while the main objective of resource management used to be controlling access to the critical resources to prevent resource leaks and contentions. However, it is cumbersome to update continuous queries of stream processing in the presence of possible fluctuating inputs and strict latency requirements, so ideally the resource management needs to be customised to suit the particular needs of streaming logic as well as catering to the varying characteristics of workloads.

With the advent of cloud computing, an elastic, seemingly endless resource pool is made available to its customers through a subscription model. Cloud computing provides a new level of flexibility for resource management and enables runtime resource adjustments for a distributed stream processing system. Particularly, the motivation of robust resource management is to maintain the articulated Service Level Agreement (SLA) in performance and reliability, while minimising the cost of resource consumption with a collection of profiling, modelling, and decision-making techniques. Nevertheless, there are many challenges left to reaching this target, which we summarise in the following.

1.1 Challenges in Robust Resource Management

Maintaining the articulated Service Level Agreement (SLA) in performance and reliability with reduced cost has brought many challenges. We discuss them in a top-down order following the layered structure of a distributed stream processing system.

1.1.1 Challenges in Application Profiling

Accurate profiling of application execution provides the basis for decision-making in the resource management process. However, the existing approaches profiled either raw resource utilisation metrics such as CPU and memory usages that do not faithfully reflect the performance of streaming application, or high-level application metrics such as throughput and latency that are too general to be used to track internal stream bottlenecks. A low-overhead profiling technique capable of identifying runtime operator congestions and network bottlenecks is needed at the fine-grained thread level to depict the application features, the computing power of provisioned resources, and the characteristics of incoming workload.

1.1.2 Challenges in Operator Parallelisation

Operator parallelisation divides a parallel operator into several functionally equivalent replicas, each handling a subset of the whole operator inputs to accelerate data processing. Since the way how operators are partitioned into streaming tasks affects the distributed execution of a streaming application, it is important to select appropriate parallelism degrees to avoid performance bottlenecks and resource wastage. Nevertheless, this is a challenging work due to a combination of factors. The selectivity[3] of an operator could be varying at runtime leading to the fluctuation of workload to its downstream operators. The capability of a single task may be mis-estimated, thus causing under-parallelisation and over-parallelisation with overloaded tasks and excessive management overhead, respectively. Lastly, it is challenging to maintain the balance of data sources and data sinks for the system to achieve performance synergy: an overly powerful data source may cause severe backlogs in data sinks, while an inefficient data source would starve the subsequent operators and encumber the overall throughput.

1.1.3 Challenges in Task Scheduling

Task scheduling decides the placement of streaming tasks across horizontally scaled resources to carry out streaming logic at different locations simultaneously and independently. The first challenge is to reduce the amount of inter-node communication which involves message serialisation and network transfer, while intra-node communication can be reduced to the passing of a pointer in memory which is more efficient and reliable. The second challenge is to avoid resource contention among collocated tasks, which

[3]Selectivity: an operator metric that describes the number of data tuples produced as outputs per tuple consumed in inputs.

is one of the leading causes of performance deterioration. It is also a challenge to make task scheduling more adaptive to the fluctuating workloads and resource availability.

1.1.4 Challenges in State Management

State management in a distributed stream processing system is essential to support dynamic scaling and mask state loss in case of failures. The existing state management approaches rely heavily on the checkpointing method that commits states regularly and recovers from the last checkpoint if the execution is interrupted. However, this method involves a remote data store for state preservation and access, resulting in significant overheads to the performance of error-free execution. It is also hard to tune the frequency of checkpointing – a small interval would bring significant state synchronization overhead; while a large interval would risk losing state between checkpointing and being unable to replay failed messages. A novel state management mechanism is required to reduce the runtime overhead while providing enough support for tolerating different types of failures.

1.1.5 Challenges in Resource Provisioning

In the literature, it is common to have resources provisioned prior to the deployment of the stream processing system. Therefore, the streaming application and the data stream management system need to be optimised towards the pre-configured computing nodes in order to improve the resource utilisation and performance. However, such platform-oriented optimisation is conducted on a best-effort basis and can provide little guarantee in achieving the desired performance and reliability targets. With robust resource management, we are interested in performance-oriented resource provisioning that enables the streaming application to reach a specific performance target with minimised resource consumption.

1.2 Research Problems and Objectives

This thesis focuses on the robust resource management in a distributed stream processing environment, with a purpose of maintaining the satisfactory SLA in performance and reliability using a minimal amount of resources. In order to tackle the above-mentioned challenges, this thesis has identified and investigated the following research problems:

- **How to profile the streaming application and properly decide the parallelism degree for different type of operators?** The most common approach used to determine operator parallelism is to gradually measure the execution capacity of each operator and adjust the degree of parallelism according to the expertise of the developer. This method involves a huge number of man-hours and may result in a suboptimal configuration. An automatic application profiler is needed to help decide the operator parallelism considering the application features and the platform computing power.

- **How to monitor runtime application execution, model its resource usages, and then automatically adjust task scheduling under different sizes of inputs?** The default scheduler in many DSMSs is agnostic about matching the resource requirements with availability, while the existing resource-aware scheduler is static and oblivious to the runtime changes of workload pressure. This means that the scheduling plan would inevitably lead to load imbalance and resource competition. Thus, a dynamic resource-efficient scheduler is required to tackle runtime variations and perform task consolidation when needed to save resource costs.

- **How to manage internal operator states to protect them from various types of failures?** Without proper state management, JVM and node crashes would cause the loss of states and eventually the incorrect processing results. As discussed before, the existing check-pointing method incurs excessive runtime overhead and is hard to tune the backup interval to suit the real-time requirement. A new state management mechanism is required to enhance the system reliability with less execution overhead.

- **How to provision and manage resources for the stream processing system to**

achieve a pre-defined performance target? The current resource management practices are mostly platform-oriented, meaning that the resource allocation, task scheduling, and operator parallelism are decided to fit a static resource-set environment regardless of the actual performance requirement. To make full use of the various types of resources in clouds, a performance-oriented resource management framework is required to depict the relationship between resource provisioning and performance scalability, making sure that the right-scale execution platform is created to meet the specific computing requirements.

1.3 Evaluation Methodology

The proposed approaches in this thesis were evaluated with both synthetic streaming applications and real-world applications in either public clouds (Nectar[4]) or private clouds (OpenStack at Clouds lab, The University of Melbourne). The synthetic applications cover different types of communication patterns, different requirements of resource consumption, and different time-space operator complexities. The real-world streaming applications include Word Count, Twitter Sentiment Analysis[5] — an established streaming application to judge the positivity and negativity of tweets, and URL-extraction, a memory-intensive application extracting short Uniform Resource Locators (URLs) from incoming tweets and replacing them with complete URL.

Throughput and processing latency are the two dominant metrics evaluating the overall performance of a stream processing system. The reliability aspect is examined by the runtime overhead on performance and the recovery time it takes to restore the system back to functioning. We have also designed and implemented a profiling environment to make sure that the evaluation of concerned metrics is controllable and repeatable. This framework has been extensively used in all of our research chapters, and statistical techniques such as Lilliefors Test have been applied as well to clearly indicate the improvements in performance and reliability.

Notably, we have implemented a different prototype system for each of our research work to evaluate the feasibility and efficacy of the proposed approaches. These proto-

[4]https://nectar.org.au/research-cloud/
[5]https://github:com/kantega/storm-twitter-workshop

types are all extended on Apache Storm, implementing a Monitor-Analyze-Plan-Execute (MAPE) architecture to be runtime-adaptive in resource management. To enable repetition of our experiments, we have released the source code of these prototypes and the test applications to the research community[6].

1.4 Thesis Contribution

The **key contributions** of this thesis are listed below:

1. A survey and taxonomy of resource management and task scheduling in distributed stream processing systems.

2. A stepwise auto-profiling method for performance optimisation of streaming applications.

    - A mathematical model that describes the relationship between resource provisioning and application performance metrics.
    - A profiling strategy implemented as a feed-back control loop that allows for self-adaptivity, scalability, and general applicability to a wide range of streaming applications.
    - An operator parallelisation mechanism that automatically scales up the streaming application on a given platform.

3. A resource-efficient and application-transparent task scheduler for stream processing systems deployed on computing clouds.

    - A system model and a cost model to formulate the task scheduling problem as a bin-packing variant.
    - A greedy algorithm to solve the bin-packing problem, which generalises the classical First Fit Decreasing (FFD) heuristic to allocate multidimensional resources.

[6]`https://github.com/xunyunliu`

- A prototype on Storm that conducts dynamic resource-efficient scheduling, reducing the amount of inter-node communication as well as minimising the resource footprints used by the streaming applications.

4. A replication-based state management framework in distributed stream processing systems:

    - A replication-based state management mechanism for achieving state persistence in the case of failures, which exposes a concise fluent-style interface and works transparently to the upper-level logic.
    - A failure recovery protocol that guarantees the application integrity when failover occurs.
    - A prototype implementation that operates at the lowest thread level and is seamlessly integrated to Storms execution flow. The replication of state is also autonomous and high-performance, which allows multiple state transfers to occur concurrently.

5. A performance-oriented deployment framework for automatic resource management under certain performance requirement:

    - An empirical study that describes how application performance is affected by resource provisioning, task scheduling and operator parallelisation.
    - A task scheduling algorithm that reduces inter-node traffic while ensuring no computing nodes are overloaded, which further considers the collocation effect that packing together two communicating tasks in a single node may make them occupy less resources than the sum of their individual size.
    - A self-adaptive resource management framework that allows eventually reaching the predefined performance target regardless of the initial resource allocation.

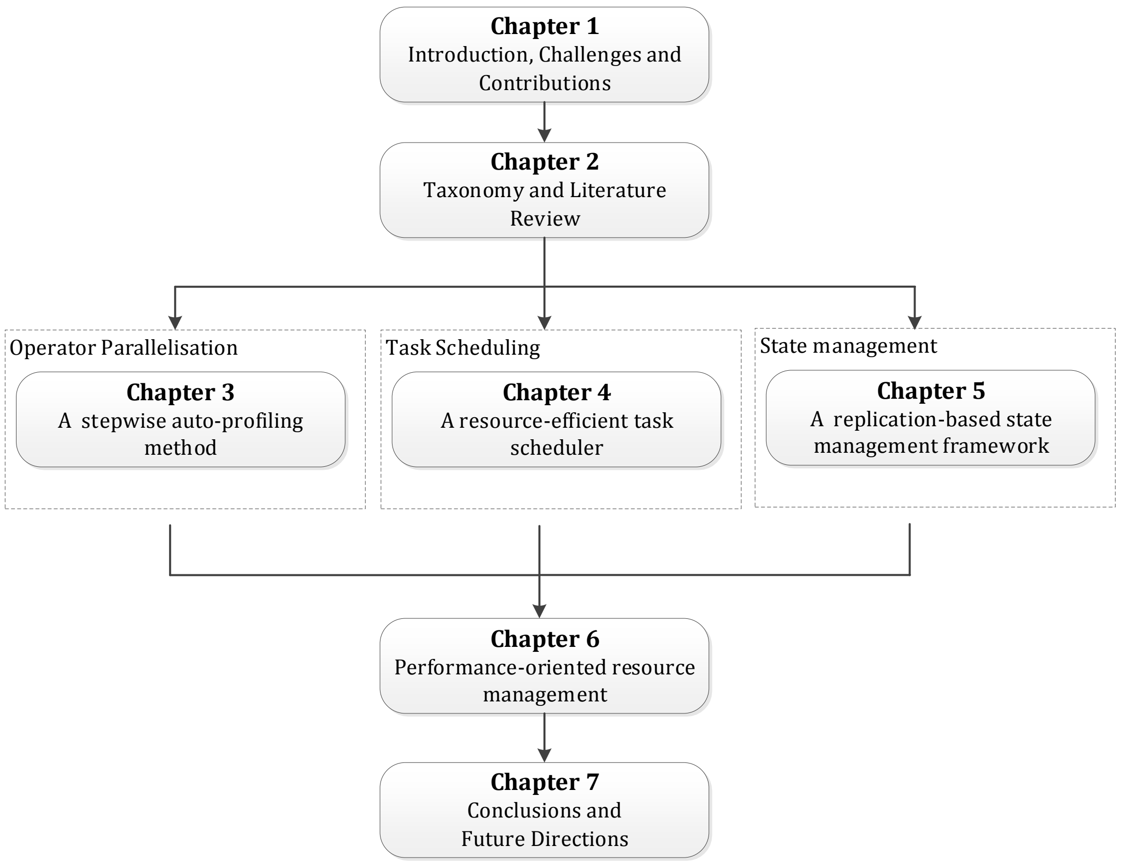

Chapter 1
Introduction, Challenges and Contributions
Chapter 2
Taxonomy and Literature Review
Operator Parallelisation
Chapter 3
A stepwise auto-profiling method
Task Scheduling
Chapter 4
A resource-efficient task scheduler
State management
Chapter 5
A replication-based state management framework
Chapter 6
Performance-oriented resource management
Chapter 7
Conclusions and Future Directions

Figure 1.3: The thesis organization

1.5 Thesis Organisation

The organisation of the chapters in this thesis is shown in Fig. 1.3. Chapter 2 provides a taxonomy and survey for the state-of-the-art resource management research in distributed stream processing systems. Chapter 3, 4, and 5 focus on a particular aspect of resource management, covering operator parallelisation, task scheduling and state management, respectively. Chapter 6 is a comprehensive resource management work that holistically optimises the deployment process to achieve a certain performance target. The core chapters of this thesis are mainly derived from the conference and journal publications completed during my PhD candidature, which are listed as follows.

- Chapter 2 presents a survey and taxonomy of resource management in distributed stream processing systems, which defines the scope of this thesis and positions its contribution in the area. This chapter is partially derived from:

- **Xunyun Liu** and Rajkumar Buyya, "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review and Future Directions," *ACM Computing Surveys*, ACM Press, 2018 (under review).
- **Xunyun Liu**, Amir Vahid Dastjerdi and Rajkumar Buyya, "Stream Processing in IoT: Foundations, State-of-the-Art, and Future Directions," *Internet of Things: Principles and Paradigms*, Pages: 145-161, Morgan Kaufmann, 2016.

• Chapter 3 proposes a stepwise auto-profiling method. This chapter is derived from:

- **Xunyun Liu**, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu and Rajkumar Buyya, "A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, Volume 12, Issue 4, Pages: 1-33, ACM Press, 2017.

• Chapter 4 proposes a resource-efficient task scheduler. This chapter is derived from:

- **Xunyun Liu** and Rajkumar Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," *in Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, China, Pages: 1-8, IEEE, 2017.
- **Xunyun Liu** and Rajkumar Buyya, "Dynamic Resource-Efficient Scheduling in Data Stream Management Systems Deployed on Computing Clouds," *ACM Transactions on Internet Technology (TOIT)*, ACM Press, 2017 (Under review).

• Chapter 5 proposes a replication-based state management framework. This chapter is derived from:

- **Xunyun Liu**, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein and Rajkumar Buyya, "E-Storm: Replication-based State Management in Distributed Stream Processing Systems," *in Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, Bristol, UK, Pages: 1-10, IEEE, 2017.

• Chapter 6 proposes a holistic performance-oriented resource management framework. It is derived from:

  - **Xunyun Liu** and Rajkumar Buyya, "Performance-Oriented Deployment of Streaming Applications on Cloud," *IEEE Transactions on Big Data (TBD)*, Accepted, In press, DOI:10.1109/TBDATA.2017.2720622 Pages: 1-14, IEEE, 2017.

- Chapter 7 concludes the thesis with a summary of the key findings and a discussion of future work directions.

Chapter 2
Literature Review

Stream processing is an emerging paradigm that handles continuous big data in memory on a process-once-arrival basis, powering latency-critical application such as fraud detection, algorithmic trading, and health surveillance. To achieve self-adaptive, SLA (Service Level Agreement) -aware, and resource-efficient deployment of stream processing systems, many research efforts have investigated a holistic framework for resource management and task scheduling. In this chapter, we introduce the hierarchical structure of a streaming system, define the scope of the resource management problem, and then present a comprehensive taxonomy regarding critical research topics such as resource provisioning, operator parallelisation, and task scheduling. We also review the existing works based on the proposed taxonomy, which helps in making a better comparison of the specific work properties and method features.

2.1 Introduction

As Internet started to connect everything, the number of intelligent devices used for monitoring, managing and servicing has rapidly increased. These interconnected data sources generate fresh data continuously, forming possible infinite data streams over the network that inevitably overwhelm the traditional data management systems. Meanwhile, the ever-growing data generation has been accompanied by the escalating

This chapter is partially derived from:
- **Xunyun Liu** and Rajkumar Buyya, "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review and Future Directions," *ACM Computing Surveys*, ACM Press, 2018 (under review).
- **Xunyun Liu**, Amir Vahid Dastjerdi and Rajkumar Buyya, "Stream Processing in IoT: Foundations, State-of-the-Art, and Future Directions," *Internet of Things: Principles and Paradigms*, Pages: 145-161, Morgan Kaufmann, 2016.

demands for real-time processing. Time-critical applications such as fraud detection, algorithmic trading and health surveillance are gaining increasing popularity, all of which rely heavily on the real-time guarantee to deliver meaningful results. The desire of fast analysis gives birth to the emergence of stream processing, a new in-memory processing paradigm that allows for the collection, integration, analysis, visualisation, and system integration of stream data in real time to deliver on-the-fly data insights with sub-second latencies.

Unlike the traditional store-first, process-later batch paradigm, stream processing adopts the process-once-arrival philosophy to exploit the volatile value of stream data. The incoming data are handled immediately upon arrival, with the results being incrementally updated as the data flow through the system. Equipped with only limited resources to handle possible infinite inputs, stream processing does not require random access to the whole stream. Instead, it installs continuous queries over a time- or buffer-based window, conducting lightweight and independent computations over the recent data. In this way, the strict latency requirement can be met by proper workload balancing and processing parallelisation on a host of distributed resources.

A stream processing system includes not only the application but also the services and resources needed to implement the application logic. Building a distributed streaming application from scratch is a tedious work and error-prone, so various Data Stream Management Systems (DSMS) have been proposed over the recent years to facilitate the development of streaming applications. From a structure perspective, DSMSs work as a middleware within the streaming system, offering unified stream management, imperative programming APIs, and a set of streaming primitives to simplify the application implementation. The state-of-the-art distributed DSMSs, such as Apache Storm [159] and Apache Flink [17], further provide transparent fault-tolerance, horizontal scalability and state management for the upper layer applications, while abstracting away the complexity of coordinating distributed resources. With such an operator-based DSMS, a streaming application is expressed as a set of interconnected operators that can run and scale on distributed resources, and a typical streaming system is thus a three-layer structure comprising user-applications, a DSMS and the underlying infrastructure.

Though the adoption of DSMSs has greatly eased the development of streaming applications, the resource management involved in the deployment process remain a challenging and labour-intensive work if one needs to set up a streaming system in a distributed environment satisfying certain Quality of Service (QoS) requirements with minimal resource cost.

Cloud computing has offered a scalable and elastic resource pool allowing for a new level of freedom in system deployment. Its customers can unilaterally provision computing capabilities as needed through an automatic-measured, subscription-oriented model, with the monetary cost calculated on a pay-as-you-go basis. However, the advent of cloud computing makes the resource management of streaming systems even more challenging due to a combination of influencing factors, such as the sensitive application requirements, dynamic workload characteristics, various cloud resource types, and diverse pricing models. The improper management of resource and scheduling directly affects the system performance on clouds. For example, over-provisioning and under-provisioning of resources lead to extra operational cost and Service Level Agreement (SLA) breaches, respectively. Acquiring resources from a suboptimal location causes additional communication latency and network traffic. The inappropriate parallelisation of operators results in either overload streaming tasks or excessive overhead of context switching. Last but not the least, misplacing streaming tasks to the underlying infrastructure leads to inefficient stream routing and resource contention that impair the system stability.

Although there are some surveys and taxonomies that are related to the resource management and scheduling contexts, each of them has a more specific focus in this area without holistically covering the resource management problem. Zhao et al. [181] surveyed various types of stream processing systems and discussed the default methods of managing resources in different DSMSs. Dias de Assunção et al. [41] surveyed the state-of-the-art stream processing engines with a focus on the enabling mechanisms of resource elasticity. Hummer et al. [80] also provided an overview of stream processing and explain the key concepts regarding the runtime adaptivity and cloud-based elasticity, but SLA-aware resource management is not included in their survey. There are also some surveys that have discussed the patterns and infrastructure to run stream processing systems elastically [65,66,73,129,138], but they emphasise more on the resource provisioning problem and lack the discussion on operator parallelisation and task scheduling.

As the research in this area advances, there is a long over-due effort to define of the scope of the resource management and scheduling problem, identify the main challenges lying in every aspect, and comprehensively analyse the developments in this field to improve the SLA-awareness and cost-efficiency of deployment. In this chapter, we aim to bridge this gap by **proposing** a taxonomy of literature on resource management and scheduling and surveying existing works following the taxonomy structure.

The rest of the chapter is organised as follows. We first introduce the hierarchy of a distributed stream processing system as background, with a system sketch to illustrate the research topics involved in the deployment process. Then we present a taxonomy of resource management and scheduling that details the problem scope and classifies the key properties of the proposed solutions. In light of the taxonomy, the surveyed works are mapped into different categories to better compare their strengths and weaknesses. Finally, we conclude the chapter with a summary.

2.2 Background

The resource management problem is part of the deployment process to ensure the pre-defined service level agreements (SLAs) or service level objectives (SLOs) are met and the resource cost is minimised.

To better understand the problem of study and define its scope, Fig. 2.1 presents the hierarchical structure of an example stream processing system. Sitting on the topmost level is the abstraction of the streaming logic, which consists of four operators in the example application. These inter-connected operators constitute a directed acyclic graph (DAG) called *topology*, which represents a continuous query that stands on the incoming data stream producing incremental results in real-time unless being explicitly terminated. Each operator encapsulates certain streaming logic such as data filtering, stream aggregation or function evaluation, while the edges denote the data paths as well as the sequence of operations conducted on the data streams. In most cases, the DAG of operators has been properly defined upon the completion of application development. So in the deployment phase, one has to decide where and how these streaming logic are executed in

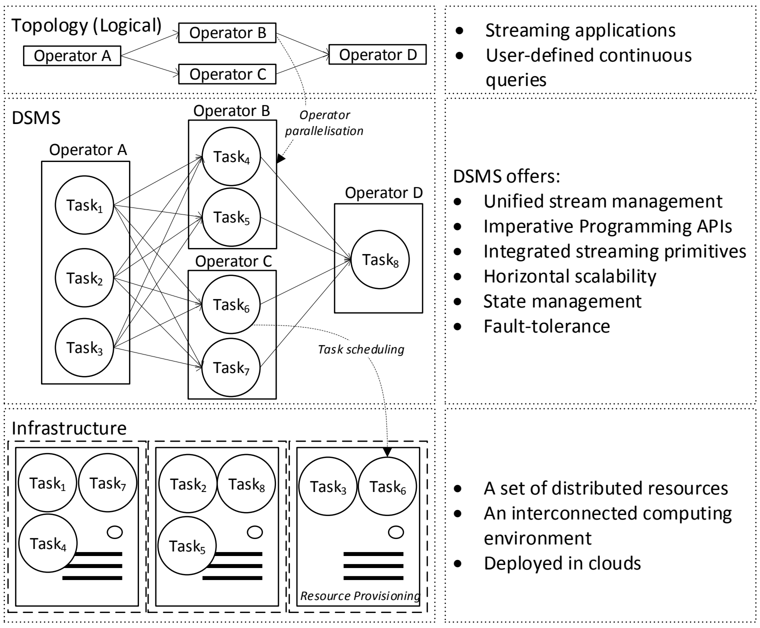


Figure 2.1: The sketch of the hierarchical structure of a stream processing system

a live distributed environment to cater to the continuous and possibly fluctuating workload.

A Data Stream Management System (DSMS) is positioned in the middle of the system structure, which serves a similar role as the Data Base Management Systems (DBMSs) in the conventional data processing context. In general, DSMSs expose a set of imperative programming APIs and streaming primitives to developers, encapsulating low-level implementation details such as stream routing, data serialisation and buffer management in a unified streaming model. Developers can thus focus on implementing the user-defined streaming logic without having to reinvent the wheel for routine data management. DSMSs also provide abstractions for parallel and distributed computing, allowing applications to enjoy horizontal scalability and fault-tolerance without code changes. During the deployment phase, the parallel operators in the topology can scale with a given parallelism degree, generating multiple replicas, known as *tasks*, to execute simultaneously on top of distributed resources. As demonstrated in Fig. 2.1, *Operator B* is parallelised into $Task_4$ and $Task_5$ as a result of *operator parallelisation*. After that, *task scheduling* is for dynamically mapping the streaming tasks to distributed resources, e.g. $Task_6$ of *Operator C* is mapped to the right end computing node in Fig. 2.1. Conveniently, the routine jobs such as stream splitting and tuple tracking that are required to maintain semantic correctness are automatically handled by the DSMS itself. Heinze et al. [66] classified the existing DSMSs into three generations, among which we mainly focus on the third generation that is highly distributed and even applicable to heterogeneous environments such as edge and fog clouds. Modern DSMSs falling into this generation include Apache Storm, Twitter Heron [96], Apache Flink, Samza [122] and Spark Streaming [178], etc.

The underlying infrastructure level represents the physical view of a stream processing system. *Resource provisioning* is a process to acquire a set of distributed resources from the cloud resource pool to constitute an interconnected computing environment. In this thesis, we consider only the Infrastructure-as-a-Service model for provisioning resources in clouds. This model visualises the physical infrastructure as separate service components such as computing, storage and network, so that users can deploy their applications with the finest control over the entire software stack, including operating systems, middleware and applications. There are also some streaming services available in the form of the Platform as a Service (PaaS) model or the Software as a Service (SaaS) model, including Silicus[1], Google Dataflow[2] and Microsoft Azure Stream Analytics[3]. However, the deployment of streaming applications on these services is usually managed by the service owner rather than the application provider, so it is impossible for the stakeholders to directly manage resources for performance improvement and cost-efficiency.

As a summary, the deployment of a streaming system can be regarded as a decision and configuration process to construct the hierarchical system structure in a distributed environment, where the higher layer needs to be mapped to and hosted on the lower layer to be concrete and runnable. During deployment, the main motivation of having a resource management and scheduling framework is to free the application providers from the burden of performing a cumbersome tuning and trial-and-error process. By applying a collection of profiling, modelling, and decisioning techniques, the framework can automatically ensure that the deployed system meet its SLA-requirements with minimal resource consumption.

In this chapter, we have identified three relatively independent research topics that

[1] https://www.silicus.com/iot/services/stream-processing-and-analytics.html
[2] https://cloud.google.com/dataflow/
[3] https://azure.microsoft.com/en-gb/services/stream-analytics/

fall within the scope of resource management and scheduling. We explain each topic by highlighting its peculiar problem domain, as well as discussing the issues and challenges it is facing to achieve the SLA-aware and cost-efficiency targets.

Resource Provisioning Resource provisioning describes the activities to estimate, select and allocate appropriate resources from the service provider to construct the interconnected infrastructure of the stream processing system.

- Resource estimation: estimating the type and amount of resources needed by the system to meet its performance and cost targets articulated in the SLA. The estimation can be derived from the analysis of historical data as well as the prediction of future workload, but its accuracy is often affected by the instantaneous, unexpected fluctuations of workload volume and system performance. Nevertheless, over-provisioning and under-provisioning resources both lead to undesirable consequences for both system administrators and users.
- Resource adaptation: it is common that the actual resource demands fluctuate along with the varying workload, or remain vague and unclear after the system has been put into runtime. Therefore, it is always challenging to find the right point in time to scale in/out, adapting the resource allocation to the fluctuating workload and system performance. In addition, the profitability of adaptation decisions is also affected by a number of factors such as the selected billing model and the geographical distribution of resource pools.

Operator Parallelisation Operator parallelisation divides a parallel operator into several functionally equivalent replicas, each handling a subset of the whole operator inputs to accelerate data processing.

- Parallelism calculation: calculation of operator parallelism requires accurate profiling of stream workload and probing the maximum processing capability of every single task. The composition of the infrastructure also plays an important role, as the number of cores/threads supported by the platform confines the maximum execution parallelism and the hardware implementation determines the costs of thread scheduling and context switching.

- Parallelism adjustment: performance bottleneck can surface at runtime caused by both over-parallelisation and under-parallelisation. Under-parallelisation results in overly-loaded streaming tasks that fail to catch up with the application inputs, while over-parallelisation increases the overhead of task management and leads to resource contention as reported by Chapter 6. Runtime task monitoring is thus required at the DSMS level to suggest possible operator congestions and tentative parallelism adjustments.

- Balancing data source/sinks: the parallelism degree of an operator reflects the degree of access it has to the distributed resources. While making parallelisation decisions, the balance between the data source and data sinks should be maintained due to the publisher-and-subscriber model adopted in the streaming system. An overly powerful data source may cause severe backlogs in data sinks, while an inefficient data source would starve the subsequent operators and encumber the overall throughput.

Task Scheduling Task scheduling decides the placement of streaming tasks across distributed resources, such that data streams are partitioned and processed at different locations simultaneously and independently.

- Minimising inter-node communication: inter-node communication is much more expensive than intra-node communication as former involves message serialisation and network transfer. Therefore, it is preferable to place communicating tasks on the same node as long as it does not cause contention. If the infrastructure consists of geographically distributed resources, it is also a challenge to reduce large data transmissions on remote and error-prone data links with limited bandwidth.

- Mitigating resource contention: one of the leading causes of performance deterioration is the competition of the computation and network resources among collocated tasks. There is a great interest to design a resource-aware scheduler that makes sure the accrued resource demands of collocated tasks do not exceed the node's capacity.

- Performance-oriented scheduling: the scheduling of tasks should be optimised towards the specific application performance targets defined in the SLA, regardless of the adverse impacts brought by workload fluctuations, VM performance variations, and the interference of the multi-tenancy mechanism enabled at the infrastructure and the DSMS layer.

The resource management and scheduling process is hardly a one-time effort. In order to satisfy the articulated SLA requirements within such a constantly changing environment, the current resource allocation and task placement need to be monitored, tuned, and adapted at runtime for the streaming system to cope with any internal and external changes.

2.3 Taxonomy

Figure 2.2 presents a taxonomy regarding the resource management and scheduling in distributed stream processing systems. It classifies the existing works based on the research topics and issues identified in Section 2.2. Furthermore, we extend each category with subdivisions to distinguish the specific work properties and classify the adopted methods based on their features. In particular, our taxonomy covers the following aspects:

- Resource Type: the various resource types involved in the resource management process to compose the infrastructure of the streaming system.
- Resource Estimation: how to estimate and model the resource cost for the streaming system to satisfy its SLA requirements.
- Resource Adaptation: how to adapt the resource allocation to the changes of workload volume and application performance.
- Parallelism Calculation: how to probe and calculate the parallelism degree for the parallel operators in the application topology.
- Parallelism Adjustment: how to adapt the operator parallelism to the workload variations and remain consistent with the processing requirements.
- Scheduling Objective: the various objectives of task scheduling and the rationale behind these objectives to achieve the overall deployment target.

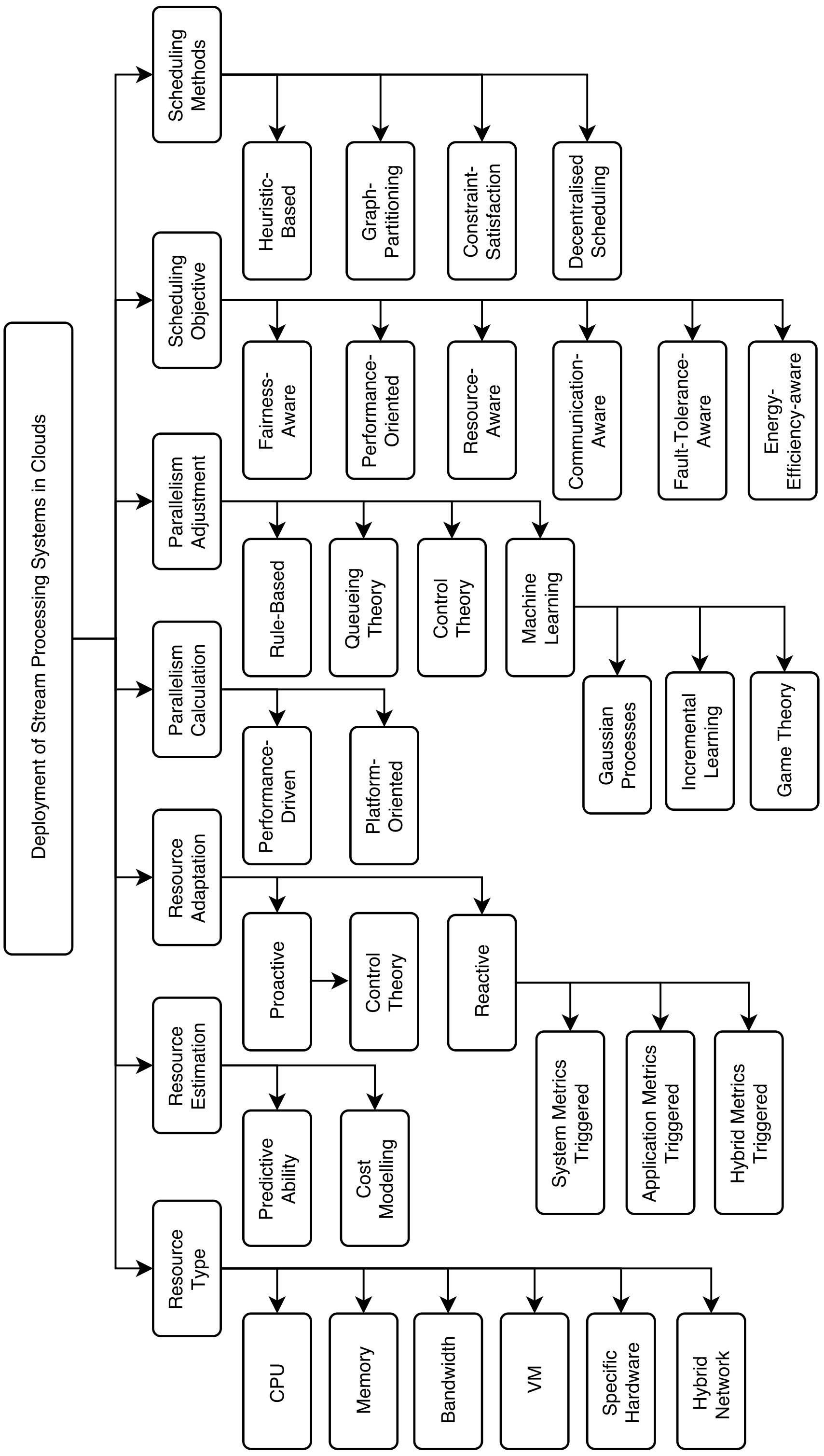


Figure 2.2: The taxonomy of resource management and scheduling in distributed stream processing systems

- Scheduling Methods: the various methods used for task scheduling.

Note that the above-mentioned aspects are only conceptually distinguished in this taxonomy, with no implication on the independence of research. In fact, the activities of resource management and scheduling are often tightly correlated and conducted in a bundle to fulfil a holistic deployment target. For example, a complete resource provisioning process consists of three steps — selecting particular resource types, estimating the amount of resource requirements, and adapting resource allocation for runtime changes, where the former step often works as a preparation for the latter. Since a surveyed research may stretch across multiple subcategories for completeness, the following sections (Section 2.4 ∽ Section 2.10) may cover the same work multiple times focusing on different aspects of the taxonomy.

2.4 Resource Type

Resource describes any physical or virtual component of limited availability within a computer system. However, depending on the actual context, the same term could contain diverse meanings and refer to various resource types at different levels of abstraction. For deployment in IaaS clouds, resource generally refers to the computing and network facilities that are available to rent through usage-based billing, such as Virtual Machines (VMs), IP addresses, and Virtual Local Area Networks (VLANs). However, for deployment of streaming systems in a more hybrid and geographically distributed environment, resources also include other infrastructural components such as specific hardware and hybrid networks.

In this section, we identify the various resource types involved in the deployment and resource management process. It is worth noting that the storage resources such as block storage, file or object storage are omitted in our classification due to the rare discussion in the previous work. This is credited to the fact that saving stream data to an off-site storage system is uncommon, which would block the dynamic data flow and cause unsustainable processing latency.

2.4.1 Resource Abstractions

Resource abstractions such as CPU, memory, and bandwidth quantify the resource consumption and requirements of a streaming system in a high-level and coarse-grained manner, regardless of the difference in hardware and the particular network that connects the streaming components. From the end-users' perspective, the measurement of resource abstractions is intuitive and straightforward. CPU resources can be counted by the number of used CPU cores, with loads measured by Million Instructions Per Second (MIPS) or percentage utilisations; memory usage is quantified by Megabytes (MB); and bandwidth consumption is gauged by Megabytes per second (MB/s) or Kilobytes per second (KB/s).

However, having ignored the particularity of the underlying infrastructure also means that resource provisioning can not be solely determined, or directly calculated on resource abstractions, the results of which would be susceptible to modelling errors and hardware discrepancies. Instead of being used directly to construct the infrastructure, resource abstractions are found more commonly used in rule-based approaches (Section 2.6.2) to approximate resource cost and adjust resource allocation, as they contain sufficient information to reflect the general system state and shed lights on the direction of adjustments in the future.

2.4.2 Virtual Machines

Virtual machine (VM) is an emulation of a computer system customisable to the specific user needs. In a cloud environment, virtual machine is the most common resource type that encapsulates the computing power and serves as the host of streaming tasks in a distributed environment.

Provisioning VMs from a particular cloud platform is a mixed problem of considering the VM price model, the location of data centres, and the network capacity of interconnections. The actual VM configurations and placement are determined by the specific computation and communication needs of the streaming system to meet its performance and cost SLA. For the generality of discussion, this survey also includes resource management techniques that originally apply to the on-premise cluster environment, as the proposed resource estimation and adaptation methods would also benefit the VM management in clouds to prevent resource leaks and contentions.

2.4.3 Specific Hardware

The infrastructure of streaming systems may require specific hardware to boost performance, improve manageability, and deal with particular streaming scenarios. Due to the scarcity of supply and the indispensability of functionality, provisioning of these critical resources is often prioritised over other common computing and network resources in clouds.

Chen et al. [31] proposed a GPU-enabled extension on Apache Storm, exploiting the massively parallel computing power of the Single Instruction Multiple Data (SIMD) architecture to accelerate the processing of stream data. Similarly, Espeland et al. [47] processed distributed real-time multimedia data on GPUs with support for transparent scaling and massive data- and task-parallelism.

FPGA is reconfigurable hardware designed to enable hardware-accelerated computations. The use of FPGA as central data processing elements allows exploiting low-level data and functional parallelism in streaming applications. To facilitate the application of FPGA for stream processing, Auerbach et al. [6] presented a Java-compatible language as well as the associated compiler and run-time system to integrate the streaming paradigm into a mainstream programming environment. Neuendorffer et al. [120] from Xilinx discussed the design tools required for the fast implementation of streaming systems on FPGAs, and Sadoghi et al. [137] investigated how to map multiple continuous queries to FPGA hardware using Hardware Description Language (HDL) code.

In some use cases, the deployment platform requires specific sensors to collect input data or monitor the current processing state such as network transmission and power consumption. For instance, data collection sensors are employed by Zhu and Vijayakumar [163,184] to aggregate stream data from the satellites and environmental monitoring facilities in real time. Kamburugamuve et al. [88] proposed a hybrid platform to connect smart sensors and cloud services, with the data processing logic deployed in the centralised cloud servers to enable new real-time robotics applications such as autonomous robot navigation. Traub et al. [160] optimised communication costs on sensor networks by sharing sensor reads among continuous queries, so that the amount of data transfer is reduced by employing a combination of data stream sampling and tailoring techniques. Also, power meters such as Watts Up are employed by Shen et al. [147] and Mashayekhy et al. [113] in their streaming systems to get real-time power readings from the host machines.

2.4.4 Hybrid Network

Traditionally, streaming systems are deployed in a single cluster or cloud environment as most of the data streams to be processed are collected from web analytic applications. However, there is an ongoing trend that the deployment platform migrates to a more heterogeneous and geographically distributed setting to process the huge data streams generated by the Internet of Things (IoT) applications. In this process, novel network elements and hybrid network structures have been employed to enhance the infrastructure connectivity and enable new application paradigms.

Collaborative Fog, Edge, and IoT networks are gaining popularity in stream processing for the ability to offload a substantial amount of control, computation and management workload to the network gateways close to data sources, thus reducing data transmission and bandwidth consumption. Papageorgiou et al. [123] identified that the low latency requirement is often challenged at the edge of the application topology because of the frequent communication to the external IoT entities, so they built new decision modules to place selected tasks on edge devices at runtime using resource descriptors. Hochreiner et al. [74] discussed the distributed deployment of streaming applications over a hybrid cloud, with a threshold-based resource elasticity mechanism to deal with the variation of IoT streams. Cardellini et al. [21] also investigated distributed deployment of streaming systems over a geographically distributed Fog infrastructure, in which they focused on the design and implementation of a QoS-aware and decentralised scheduling policy. Aggregation and processing of streaming data in smart city applications are tackled by a distributed IoT network developed by Puiu et al. [126], which is capable of enriching input streams with semantic annotations and utilising stream reasoning techniques to allow real-time intelligence with event detection.

Mobile devices have also taken part in the network infrastructure of a streaming system to move computation closer to the data sources. To deploy stream processing application directly on smartphones, Wang et al. [167] proposed a new check-pointing method to mask the simultaneous failure of mobile devices and employed a segmented, UDP-based data transmission method to reduce the cellular network overhead. Similarly, Morales et al. [118] relied on mobile devices to pre-process data streams, and they also proposed a new check-pointing method that is both connectivity-aware and energy-aware. Yang et al. [175] discussed how to enable mobile devices to work in partnership with VMs provisioned in clouds, with a focus on the dynamic partitioning of data streams between mobile devices and data centres to achieve higher throughput and scalability.

On the other hand, High-performance Computing (HPC) network has also been utilised in stream processing to enable advanced interconnectivity and better scalability than the conventional Ethernet connection. Recently, Kamburugamuve et al. [89] discussed the use of Infiniband and Intel Omni-Path to improve the performance of stream processing applications, where a new Storm extension is proposed utilising the native function of high performance interconnects to achieve significantly lower latencies and improved throughputs.

2.5 Resource Estimation

Based on the information retrieved, recorded or derived from the present and the past system states, resource estimation calculates the minimal amount of resources required by the streaming system to fulfil its SLA. The accuracy of resource estimation determines the cost-efficiency of resource provisioning, which plays a key role in a quick converge to optimal deployment and avoiding over- and under- resource utilisation.

Our taxonomy covers the following characteristics of a resource estimation method:

- Predictive Ability: whether the resource estimation method can predict the future application and system metrics, such as workload size, resource utilisation, and application performance.

- Resource cost modelling: how it models the resource costs based on the predicted or collected metrics, and what criteria in SLA determine the minimal amount of resource requirements.

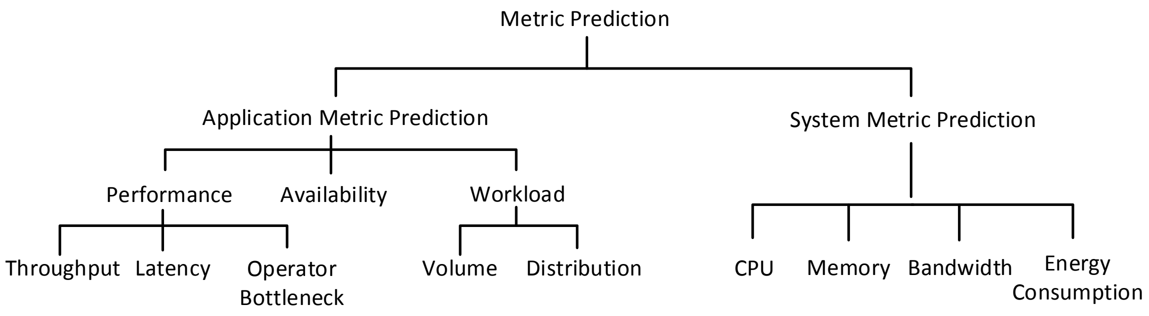


Figure 2.3: The classification of predicted metrics used for resource estimation

2.5.1 Predictive Ability

Prediction of future application and system metrics allows active speculation of future resource demands rather than assuming a constant resource consumption pattern.

Fig. 2.3 illustrates the classification of predicted metrics based on the level of which they are collected from the software stack. Metric of different granularities contain different information and thus contributing to resource estimation in different ways. The prediction of system metrics normally leads to a direct estimation of future resource requirements, which is consequently oblivious to the particularity of the hosted applications; whereas the prediction of application metrics leads to an indirect estimation of resource demands, which requires further resource cost modelling to suggest the minimal resource requirement without violating the SLA requirements.

From the methodology perspective, time series analysis and queueing theory are identified as the two prominent approaches for metric prediction.

Time Series Analysis A time series is a sequence of data records collected at successive points in time, and time series analysis is an umbrella term that describes a variety of models and methods on time series to find repeating patterns in the historical data. In the context of stream processing, time series analysis is applied to the past system resource usages and application metrics to forecast future values, leading to direct and indirect estimation of future resource requirements, respectively.

CloudScale [147] is an elastic resource scaling system built on Xen hypervisor[4] that directly predicts the short-term resource demands based on the recent history of sys-

[4]`https://www.xenproject.org/`

tem metrics. They adopted a hybrid time series analysis approach combining both Fast Fourier Transform (FFT) and a discrete-time Markov chain to balance between high estimation accuracy and low overhead. The light-weight FFT is tried first for fast identification of repeating patterns in the previous time series. If not found, the heavier Markov chain model performs multi-step analysis on the metric history to provide coarse-grained and long-term resource estimations.

The same approach is also seen in the group's previous work [182], with more details revealed on the prediction process. Fast Fourier Transform (FFT) identifies the dominant frequencies of variation in the observed resource-usage time series, followed by a discrete-time Markov chain model that unveils the deeper-hidden patterns through calculating the feature value distribution for the collected resource metrics. The combination of these two methods leads to a fast yet accurate estimation model, provided that there are patterns concealed in the resource usage history.

OrientStream [165] is a recent work on dynamic resource provisioning of stream processing systems. It features an online resource prediction module that employs an ensemble regression model on the past system metrics to suggest future resource usages. The prediction process is essentially a weighted vote of four independent regression models, reaping the benefit of reducing the overall Relative Absolute Error (RAE).

Dai et al. [35] presented VM provisioning as a multi-objective optimisation problem, which they solve with an auto-regressive model that learns and predicts the utilisation of each VM as well as the bandwidth consumption between routers. With further consideration on power management, Liu et al. [105] applied deep reinforcement learning over a linear combination of system metrics such as total power consumption, VM latency, and reliability metrics to synthetically predict future system states. Based on the forecast, a hierarchical resource provisioning model is proposed that saves energy consumption without significantly impacting the application performance and availability.

When applying to the historical data of application metrics, time series analysis can also indirectly suggest the future resource demands with the help of resource cost modelling.

Hidalgo et al. [72] applied a Markov chain model on the workload time series to predict whether an operator's future state would be overloaded, underloaded, or stable. Based on the state predictions, resource cost is modelled by checking the minimal amount of resources needed for the placement of tasks. Kombi et al. [95] adopted a similar method to forecast operator bottlenecks, which circumvents the rigorousness of queuing theory while still being able to estimate resource demands at the operator level.

Balkesen et al. [9] applied exponential smoothing on the periodic observations of input stream rate to forecast the volume of future workloads. Their rate-forecasting heuristic solves the bin packing problem formulation, suggesting the future resource usages based on the stream distribution and the placement of operators. Analogously, Ishii et al. [82] employed Sequentially Discounting AutoRegression (SDAR) to predict future input rates. They formulated an optimisation problem on resource provisioning and solved it with linear programming to find the minimal resource requirement without violating the application latency SLA. HoseinyFarahabady et al. [77] also predicted the changes in the input traffic with an Auto Regressive Integrated Moving Average (ARIMA) model, which lays the foundation for a resource provisioning algorithm that causes less QoS detriments over all available servers.

Mayer et al. [115] predicted the workload distribution and its parameters with a hybrid approach of distribution moments and maximum likelihood method. The predicted workload distribution feeds into the calculation of operator parallelism and then sheds light on the resource cost by counting the number of processor cores required for task execution. Imai et al. [81] trained a linear regression model on the performance data collected in an experimental environment, in order to predict the maximum sustainable throughput of the streaming application running on a larger number of VMs. Therefore, the cost model is built by directly linking the desired application performance with the number of VMs provisioned in the infrastructure.

Queueing Theory Queueing theory is a set of mathematical models studying waiting lines and queues to describe or predict the waiting time and queue lengths. In stream processing, queueing theory is often applied to application performance metrics — especially operator latency — to shed light on the possible data flow bottlenecks.

By modelling the operator as a $G/G/1$ queueing system, De Matteis et al. [37] regarded each operator task as a single server queue where both inter-arrival times and service times have a general distribution. The Kingman's formula is then used to approximate the mean waiting time at the operator level, which sheds light on the runtime adaptation of the number of used cores as well as the CPU frequency. The same modelling and solving technique are also found in a variety of literature [27, 38, 39, 98, 108], which proves that the Kingman's formula is widely accepted for latency modelling because of its accuracy and generality applying to arbitrary distributions of the inter-arrival time and service time.

Differently, HoseinyFarahabady et al. [77] modelled the operator as a $G/G/k$ queue (k is the number of processors for the target operator) and employed Allen-Cunneen approximation to give an upper-bound of the sojourn time experienced by each tuple. Since there is no exact formula known for the $G/G/k$-model, Allen-Cunneen approximation provides asymptotically exact results under heavy traffic and is particularly suitable for streaming applications with highly-utilised operators. There are also two papers [54, 149] that formulate the operator as a $M/M/k$ system, where M indicates Poisson distribution for arrival and Exponential distribution for service time. Accordingly, Erlang formula is applied to estimate the expected value of the total tuple sojourn time in the application. This is different to the $G/G/1$ and $G/G/k$ modelling at the operator level as the whole application topology is modelled as a Jackson open queueing network, which increases the rigorousness of the queueing model but is capable of providing more accurate latency estimations for the whole application if the model assumptions are met.

2.5.2 Resource Cost Modelling

With the domain knowledge of stream processing, a resource cost model summarises the various metrics collected at the runtime stack, suggesting an overall estimation of resource requirements for the streaming system to satisfy its particular SLA requirements.

Resource cost modelling is generally influenced by the application logic and the desired deployment target. For example, a filtering operator may show a resource usage pattern linear to its workload volume, while a window operator may exhibit periodical resource requirement peaks as the window slides or executes. For some applications that prefer higher reliability and availability, extra resources are required to provide fault-tolerance or confine the utilisation rate of each node in certain bounds. However, some applications are more sensitive to resource cost and communication overhead, so the deployment of these applications is consolidated to as fewer nodes as possible to reduce resource usages and inter-node communications. Proper resource cost modelling is the key to deal with these variations and suggest the overall resource requirements accordingly.

For the convenience of discussion, our taxonomy categorises different resource cost models based on the intended SLA optimisation, i.e. which SLA requirement is more critical to determine the system resource cost in general.

Minimal Cost Model This model intends to achieve the targeted performance requirement with minimal resource cost, so there is no additional resources provisioned to improve reliability and availability. Bin packing is the most common strategy to model the minimal resource cost based on the compact task placement. Setty et al. [144] used bin-packing formulation to determine the minimal number of VMs needed by the placement of topic-subscriber pairs, with a greedy heuristic to optimise cost while respecting the constraint that the application communication must not exceed the VM bandwidth capacity. Heinze et al. developed FUGU, an elastic data stream processing prototype, to evaluate different scaling policies [69] and optimise the scaling parameters [70]. In both works, the resource requirements are estimated using a bin-packing model solved by a First Fit and Best Fit heuristic. Balkesen et al. [9] applied bin packing to dynamically reassign data streams to different nodes, resulting in runtime adjustments to the previous round of stream assignment rather than re-optimisation from scratch to balance between result optimality and the overhead of stream redirections. Bin packing is also employed by Xu et al. [173], Nardelli et al. [119] and Ghaderi et al. [60] to suggest minimal resource cost under a certain SLA requirement.

Reliability-oriented Model This model provisions additional resources for state management and failure recovery. Madsen et al. [110, 111] proposed a Storm extension that replicates the same operator state across different nodes, allowing faster state migration to transparently scale and recover stateful operators. The resource cost is thus calculated by the needs of state management to maintain semantic correctness and fault-tolerance. Similarly, Castro Fernandez et al. [24] designed a set of state management primitives to expose internal operator states to the DSMS for transparent failure handling and scaling.

This leads to a resource cost model built on state management with considerations on the extra computation and communication overhead introduced by failure recovery and periodical state check-pointing.

Contention-aware Model Model of this type permits certain resource allowances to handle random workload bursts when needed. HoseinyFarahabady et al.[77] proposed a resource cost model that tracks and confines the CPU utilisation level of each node within an accepted range, and a similar approach is also found in Thamsen et al.'s work [157]. In order to limit the memory usage and CPU consumption within a certain bound, Cammert et al. [16] proposed a cost model to estimate resource utilisation of continuous queries based on the stream characteristics such as the average inter-arrival time and the average validity of tuples. The proposed fine-grained cost model is customised to a variety of operator types and streaming logic, making it possible to even quantify the impact of (re)optimisations on query plans.

Load-balancing-oriented Model This model focuses on the fair utilisation of available resources and is the opposite to the compact task placement which is commonly seen in the minimal cost model. Fischer et al. [50], Eskandari et al. [46] and Jiang et al. [86] regard the operator placement as a graph partitioning problem, so that they explicitly spread the streaming tasks across all available resources at the infrastructure for better load balancing. This cost model is also employed by the round-robin scheduler that is used as default by a variety of DSMSs, which favours even load distribution over participating computing nodes.

Distribution-based Model Also, depending on the nature of the cost model, the result of resource requirement may be a probability distribution rather than a definitive value. Khoshkbarforoushha et al. [92, 93] employed Mixture Density Networks (MDN), a statistical machine learning model combining Gaussian mixture models and feed-forward neural networks, to estimate the whole spectrum of resource usage as probability density functions. Modelling the resource usage as a distribution rather than a single point value captures the possible variances caused by resource contentions and interferences from parallel workloads.

2.6 Resource Adaptation

Resource adaptation in stream processing generally refers to horizontal scaling, i.e. adding or removing VMs within in the infrastructure to alter the scale of distributed computation. This is in contrast to vertical scaling that resizes the existing VMs and adjusts the capacity of existing hardware in terms of CPU, memory, and network resources. The main reason behind the rare use of vertical scaling is that its maximal scalability is limited by the size of the server, and that stop-free vertical scaling has received limited support from the mainstream cloud provider such as Amazon, Microsoft and Google. It is usually required to bring the whole streaming system offline for maintenance to make any configuration change to the provisioned virtual machines, the consequence of which is unacceptable in the presence of continuous inputs and strict latency SLA.

However, Dynamic Voltage and Frequency Scaling (DVFS) is an exception that vertical scaling is employed in streaming systems to optimise energy consumption. DVFS is a common power management technique that allows processors to dynamically change power states, lowering and raising CPU frequency and voltages on the fly according to the resource demands from virtual machines. It is used by Matteis et al. [37, 39] to explicitly regulate the CPU frequency, by Sun et al. [156] to model the power-to-frequency relationship, and by Shen et al. [147] to turn unused resources into energy savings without affecting application SLOs.

In the rest of this section, we categorise horizontal scaling techniques into two major categories based on how they select the proper scaling time: (1) proactive approaches that adjust resource provisioning according to the prediction of workload pressure and system behaviour in the future time horizon, and (2) reactive approaches that scale the infrastructure only when necessary as indicated by some threshold breaches or changes of system state.

The choice of proactive or reactive approaches much depends on the predictability of workload pattern and system behaviour. In some cases, the input stream exhibits gradual and repetitive variations in volume and composition, so it is preferable to learn from the history and apply the obtained knowledge to adjust resource provisioning proactively before the application requirement changes. In other cases, the arriving data stream contains random bursts and drastic workload changes with no clear pattern, leaving the prediction of future system state no longer a viable option. Hence, reactive approaches are required to deal with the bursty load on the best effort basis.

2.6.1 Proactive Adaptation

Proactive adaptation regards the infrastructure layer as a controllable system requiring certain corrective actions from time to time, e.g. acquiring more resources to tackle under-provisioning or relinquishing over-provisioned resources for cost-efficiency. Therefore, there are continuous controlling loops that monitor the various inputs and outputs of resource management and actively suggest optimal adjustments without delay or over-shoot.

A typical workflow of a controlling loop is as follows: (1) the resource estimation module predicts the future system state such as the workload arrival rate and the average input processing latency in the prediction horizon[5]. (2) The system model captures the relationship of various QoS variables, assessing the system's capability to maintain the articulated SLA. (3) the control algorithm solves an optimisation problem to find the best resource allocation for the next loop.

Based on how the optimisation problem is solved, we generally categorise the proactive adaptation methods into two groups. The first one is *loop-wise control*, which regards each prediction horizon as an independent control interval and derives proactive adjustments by applying the predefined scaling rules to the estimation of the next control loop. Methods falling this group are intuitive and straightforward to implement, but they may suffer from the problem of adjusting for short-term benefits while ignoring the long-term future.

To mitigate this, *Model Predictive Control* (MPC) optimises resource provisioning in a receding prediction horizon that consists of multiple control intervals. At each control interval, the controller solves an optimisation problem to obtain the optimal reconfiguration trajectory over the prediction horizon. However, when it comes to execution, only the first element of the optimal reconfiguration trajectory would be employed to steer the resource adaptation, while the whole trajectory is re-evaluated at the beginning of the next control interval to exploit the updated forecast in the shifted prediction horizon. De

[5]Prediction horizon: the period in which the future values of the interested metrics are predicted

Matteis et al. [37–39] employed MPC to achieve QoS-aware and energy-efficient resource adaptation, in which they model the optimisation problem as a minimisation of QoS cost, resource cost and adaptation cost. They proposed a tree structure to describe the search space and employed the Branch & Bound methods (B&B) to prune the search tree and reduce the runtime overhead of MPC in a real-time environment. Meanwhile, Hoseiny-Farahabady et al. [76, 77] employed MPC to proactively alleviate the resource contention between collocated applications, in which the optimisation problem is solved by Particle-Swarm Optimization (PSO) with its execution time capped to 1% of the control interval to limit its computational overhead.

2.6.2 Reactive Adaptation

Based on the metric classification shown in Fig. 2.3, we also categorise different reactive methods by the nature of the triggering metric.

System Metrics Triggered The system metrics such as CPU utilisation, memory usage, and bandwidth consumption contain raw information on system performance and resource utilisations, thus reflecting the need for adaptation when some metrics have breached certain thresholds. The common problem associated with this type of methods is that the system metrics may not faithfully reflect the application performance. For example, a higher CPU utilisation rate does not necessarily mean higher application throughput and lower processing latency. Instead, it may imply that the current resource provisioning is not sufficient for handling the incoming workload. On the other hand, methods falling into this category are versatile and easy to implement for being application-agnostic — the simplest example would be monitoring the CPU utilisation at each host, with an upper and lower bound defined to trigger scaling in and out actions [23, 24, 69, 162]. The memory threshold method is also found in Liu et al.'s work [103].

Application Metrics Triggered The application metrics include not only the application performance perceived by the end-user, but also internal metrics from the DSMS that describe the service time, the arrival rate, and the length of input/output queue for individual operators.

Lohrmann et al. [108] present a reactive scaling strategy that reacts to latency constraint violations with appropriate scaling actions, which minimises the total resource consumption under a varying load scenario. The same approach is also employed in their Nephele [109] implementation. Xu et al. [174] defined a metric named Effective Throughput Percentage (ETP) for each operator, which captures the state of congestion and estimates the impact of operator output towards the application throughput. The operator with the highest ETP will be given more parallelism and assigned to a new VM for scaling out.

By monitoring the input stream rates and the current processing rates within the DSMS, Cervino et al. [26] detect overload conditions in the operator buffer and then scale the number of used VMs accordingly to maintain the required throughput. Similarly, Vijayakumar et al. [163] defined a derived metric describing the difference between the processing time per data block and the average time interval of receiving one block, so that the adaptation is triggered by the calculated buffer-overflow. Kleiminger et al. [94] monitored the lengths of the input and output queues for stream processors, so that the computation can scale out from an on-premise cluster to clouds when needed. Satzger et al. [139] determined if an operator is overloaded by analysing the length of its incoming message queue, with thresholds hard-coded in the scaling logic to trigger adaptations on resource provisioning.

Hybrid Metrics Triggered Since relying on system metrics or application metrics alone may not faithfully reflect the actual application performance and resource utilisation, it is preferable to combine both to comprehensively trigger reactive resource adaptations. The most common combination is to monitor the operator throughput and the resource utilisation at each host node, so that it is possible to deduce the average processing cost per tuple at the operator granularity. Chapter 6 applied this method to trigger reactive resource adaptation, so that the overall application throughput can be maintained at a pre-defined level regardless of the initial allocation of resources. In Chapter 4, scale-in is performed when the input load decreases and so does the resource consumption of each operator. The scale of resource adaptation is derived from the monitored load difference and the comprehensive metric of per-tuple processing cost.

2.7 Parallelism Calculation

Parallelism calculation answers the question of how many streaming tasks are required for a particular operator to sustain its assigned workload without causing congestions to the whole application topology. We have identified two prominent approaches in the literature. The first approach is called performance-driven parallelisation — the calculation is a division of the operator input size and the anticipated capacity of each streaming task. The second approach is platform-oriented parallelisation — it first checks the maximum number of parallelism units supported by the provisioned platform, and then distributes them as resources among different operators to ensure that the platform is not over-utilised by an excessive amount of processes and threads.

2.7.1 Performance-driven Parallelisation

In this approach, the correct calculation of the parallelism degree for a particular operator hinges on the profiling of both operator inputs and the capacity of each streaming task, the latter of which is defined as the maximum number of tuples that a single task can sustainably handle per time unit (Chapter 6). There are direct and indirect methods to measure the volume of inputs for an individual operator. The direct methods install a metric collector at the task entrance that automatically gauges the flow traffic and regularly reports to the calculation logic [83], while the indirect method utilise the publisher-subscriber model adopted in the streaming system, inferring the input volume of a particular operator by examining the selectivity[6] of its upstream operators [141]. Note that nowadays light-weight stream monitoring and management have been supported by a variety of state-of-the-art DSMSs, so the hurdle of directly measuring operator inputs has been lowered with the abundance of built-in metrics.

In addition to measuring operator input, task profiling is another piece of the puzzle to achieve performance-driven parallelisation. There are a bunch of monitoring and sampling techniques available to profile the task performance from different perspectives. The most commonly profiled metrics include the average processing latency per tuple [29], the idleness of task execution [86,168], and the resource usages of a task entity

[6]Selectivity: an operator metric that describes the number of data tuples produced as outputs per tuple consumed in inputs.

(Chapters 4 and 6). The relationship between task capacity and the first two metrics is readily established — the capacity is reached when the task is occupied with tuple processing under the wall clock time; while using the last metric to estimate task capacity exploits the fact that a task is often hosted by a single thread or process, which means its peak performance is also limited by the maximum CPU utilisation of a single CPU core.

2.7.2 Platform-oriented Parallelisation

The rationale of platform-oriented parallelisation is twofold — to avoid over-utilising the available resources with excessive operator parallelism, and to help incorporate some rules of thumb suggested by the DSMS developers to make full use of the parallel processing capability of the deployment platform. Take Apache Storm as an example, it is suggested that the operator parallelism is a multiple of the number of machines deployed in the platform, and the parallelism of data source is a factor of the number of partitions of the message queue, as such configuration empirically facilitates load balancing between different hosts [154].

Platform-oriented parallelisation is commonly used in industrial deployment settings. Goetz et al. explain how their company decides on operator parallelism in a slide posted online[7]. Specifically, there is a concept of parallelism unit to describe the parallel processing capability of the platform, which essentially multiplies the number of nodes in the platform by the number of cores available on each node. For instance, there are 160 parallelism units available in a cluster consisting of 10 worker nodes with each incorporating 16 cores. The calculated parallelism units are then regarded as a special type of resources that can be distributed among parallel operators in the topology — the slower the task is in terms of the processing latency, the larger parallelism it gets from the resource pool of parallelism units. They also considered the fact that some tasks may exhibit a higher processing latency because of having intensive communications, so the number of parallelism units can be enlarged 10 to 100 times depending on the number of I/O bound operators present in the topology. This is to ensure that there are enough streaming tasks for each communication-intensive operator to split the workload and perform I/O operations.

[7]https://www.slideshare.net/ptgoetz/scaling-apache-storm-strata-hadoopworld-2014

2.8 Parallelism Adjustment

The direct calculation of operator parallelism may not be feasible in some user cases due to the lack of pilot run or monitoring facilities. Also, the results of calculation are prone to profiling errors that adversely affect the system performance. Therefore, an iterative adjustment process is needed to dynamically adapt the parallelism degree in response to the continuous variations of workload and system performance.

2.8.1 Rule-based Approaches

Rule-based approaches have attracted extensive research attentions due to the simplicity of implementation and effectiveness of adjustments. The core of the method is made of a collection of scaling rules that define the triggering thresholds as well as the corresponding scaling actions. In most cases, the scaling actions are greedy-based, which favour direct mitigation of the threshold violation and converging to suitable parallelism quickly at the expense of optimality. It also means that the resulting parallelism may be trapped in the local optimum and a proper backtrack mechanism is required to search for the global optimum [7].

Rule-based approaches can be generally classified as either static or dynamic in terms of execution.

Static Single Threshold A static threshold is pre-defined in the scaling logic to trigger parallelism adjustments in a single direction. For example, the threshold on processing latency is one-sided — when the monitored latency exceeds the SLA requirement, the operator parallelism is increased to amortise the processing workload by adding more streaming tasks to the fleet. Besides, Humayoo et al. [78] assessed the necessity of adjustment with a utility threshold to evaluate if the probability of obtaining positive gain outweighs that to incur a loss. Gulisano et al. [64] defined an upper imbalance threshold to ensure the standard deviation of load distribution is below a pre-defined limit. Though setting a single threshold statically makes it fairly easy to implement the adjustment logic, expert knowledge on application characteristics and the platform specification are still required to properly decide the threshold value and the corresponding scaling actions. Furthermore, methods falling into this category lack the ability to scale reversely nor being self-adaptive as the employed threshold is fixed during the complete runtime of the system.

Static Multiple Thresholds Multiple static thresholds are set in pairs to maintain the concerned parameters within certain upper and lower bounds. For instance, Fernandez et al. [24] defined two thresholds on the average CPU usages of each node to trigger parallelism adjustment from the perspective of local resource utilisation. This approach is also seen in Veen et al.'s work [162]. Kombi et al. [95] divided the estimated amount of operator input by the estimated capacity of a streaming task, where two performance thresholds are defined delimiting a low and a high activity level to trigger the corresponding scaling action. The major challenge for this type of methods is oscillation, where opposite scaling operations are conducted continuously due to the poorly configured thresholds or overreacting changes [58]. Therefore, a configuration of cooling time is set in practice to conservatively limit the frequency of adjustments and mitigate oscillation.

Dynamic Thresholds With the knowledge acquired from the evaluation of the previous adjustment results, dynamic thresholds improve the method adaptivity by updating the triggering thresholds and refining the adjustment behaviours at runtime. It also helps mitigate oscillation as the parameters of scaling are dynamically updated with regard to the previous run history. Heinze et al. [69] applied reinforcement learning to reward effective adjustments and punish unnecessary changes caused by inappropriate thresholds. Bilal et al. [12] examined whether a change of parameter value has an overall positive or negative impact on latency and throughput, where the dynamic thresholds are defined as the best performance monitored in the execution history.

2.8.2 Queueing Theory

The anticipation of operator congestion using queueing theory is not only useful for the estimation and adaptation of resource provisioning, but also for adjusting the relevant operator parallelism. Mayer et al. [114,115] built an adaptive data parallelisation middleware that deduces a stationary distribution of the queue length under a certain parallelisation degree, so that the operator parallelism is adjusted accordingly to make sure that the message buffer's limit is not exceeded with a high probability. Chapter 2 employed a queueing network to infer the throughput distribution among operators considering their selectivity and communication pattern, based on which the operator parallelism is scaled in batch in a way that the capability of the data source and data sink is balanced.

A predictive operator latency model is built on queueing theory and employed by Lohrmann et al. [108] to formulate a linear objective function on the minimisation of total parallelism. They applied a gradient descent search to find the optimal degree of parallelism for each operator that reduces resource footprints while enforcing the latency constraints. Similarly, Fu et al. [54] formulated a latency model based on queueing theory to determine the number of nodes that each operator needs to be placed on; however, their approach is dedicated to computationally intensive applications with no regards to the possible communication overhead and network delays. Cardellini et al. [20, 23] searched for the optimal parallelism by jointly considering operator replication and task placement within an integer linear programming formulation, and this process relies on modelling the underlying computing node as an M/M/1 queue to estimate the response time of a particular operator subject to its parallelism, service rate, and incoming load.

2.8.3 Control Theory

The versatile control theory also applies to the adjustment of operator parallelism. In Section 2.6.1, we have discussed various MPC-based algorithms that explore the optimal configuration of the target application under ever-changing operational conditions. The parallelism degree of each operator is part of configuration which is updated at the beginning of each control interval [37–39, 76, 77]. In addition, Gedik et al. [58, 59] investigated the profitability of parallelism adjustment with respect to the changes in workload volume and the availability of resources, where a control algorithm is proposed to manage the operator throughputs and congestion with appropriate parallelism. In Li et al.'s work [99], the operator parallelism is controlled by the comparison of congestion degrees[8] that are measured on the operator's receiving and sending queue, where the strength of intervention could be tweaked by an adjustment coefficient. Floratou et al. [53] presented a throughput-oriented policy that automatically configures the parallelism degree to ensure satisfactory throughput and alleviate backpressure. Similarly, Stela [174] also relies on monitoring throughput changes to make control decisions — the control algorithm increases the parallelism of the most congested and most influential operator to make full use of the newly added machines during scaling out. In Sun et al.'s work [153, 154], the parallelism degree of each operator is determined in proportion to its computational complexity, which is monitored and measured by the unit of MIPS, i.e. Millions of Instructions Per Second.

2.8.4 Machine Learning

The adjustment of operator parallelism can also resort to a variety of machine learning techniques. Gaussian processes (GP) is employed by Zacheilas et al. [177] to analyse historical data of workload volume and processing latency, so that the parallelism degree can be proactively adjusted to augment the system's performance. By applying incremental learning techniques to different query workloads as training sets, Wang et al. [166] predicted the operator resource usages under several manually supplied candidate configurations. The optimal parallelism is then selected to minimise resource usages while considering the current query requests and stream properties. Game theory is also explored to formulate the elastic parallelism scaling problem as a non-cooperative game, with each operator regarded as an independent agent performing a local control strategy. The operator parallelism is thus determined as the system reaches the agreement of Nash equilibrium [116].

2.9 Scheduling Objectives

Scheduling is of paramount importance to the successful deployment as it determines whether a streaming task can get enough resources to process inputs received from its

[8]The congestion degree for a particular operator queue refers to the ratio of the size of the queued messages to the overall queue buffer size.

predecessors. As we have discussed in Section 2.5.2, some scheduling targets cannot be achieved at the same time due to their competing nature, e.g. data locality and load balancing are two conflicting targets that require consolidated task assignment and scattered task distribution, respectively. In order to evaluate and compare the quality of different scheduling policies within the same scope, we have identified six major categories of scheduling objectives.

2.9.1 Fairness-aware Scheduling

The meaning of fairness is twofold when it comes to the scheduling of streaming tasks. Firstly, the amount of workload assigned to each node should be fair, so load-unbalance will not happen where part of the computing infrastructure is over-utilised while the other part is under-utilised. Secondly, the resources allocated to each streaming application should be fair, so that scheduling will not lead to application starvation and resource competition that are commonly caused by the multi-tenancy mechanism in the mainstream DSMSs. However, it should be noted that being fair in load distribution and resource allocation does not necessarily guarantee each streaming application can meet its SLA requirements.

2.9.2 Performance-oriented Scheduling

Throughput and latency are the two dominant metrics measuring the performance of a streaming application from the end-user's perspective. Maintaining throughput at the required level is of vital importance to the stability of a streaming system. In a streaming environment, the data sources usually work independently and asynchronously with respect to the other parts of the streaming system. So if the processing facility lags behind in sustaining the required throughputs, the message buffer between the data source and the deployment platform will be overwhelmed by the backlogs which eventually lead to the system crash (see Chapter 3). On the other hand, the importance of reducing processing latency stems from the fact that streaming applications are real-time in nature.

Performance-oriented scheduling used to be *platform-centric* in a cluster environment, which aims at producing better performance in a fixed deployment platform by optimising the resource utilisation or reducing the network communication of streaming tasks [5,29,46,50,85,124,153,173]. However, as cloud computing has enabled dynamic resource provisioning during runtime, performance-oriented scheduling has become *SLA-centric* that focuses on meeting the pre-defined performance targets with elastic scaling on resource and operator parallelism [54,81].

2.9.3 Resource-aware Scheduling

Resource-aware scheduling matches the resource demands of streaming tasks to the capacity of distributed nodes, so that over-utilisation and under-utilisation are mitigated preventively, and less computing and network resources are consumed to achieve the same performance target (Chapter 4). In practice, the resource demands and capacity are described as a multi-dimensional vector, with each element representing a particular resource type [124]. The scheduling process is thus finding a mapping of tasks to machines such that the overall cost of resource consumption is minimised and the resource constraints are satisfied, i.e. the accumulated vector of resource demands requested by the collocated tasks does not exceed the vector of resource availability on that node.

In addition, the need for resource-aware scheduling is driven by the ever-growing use of heterogeneous resources in the streaming infrastructure. The computing nodes could range from energy-constrained mobile devices to powerful virtual machines, which possess different computing powers. Hence, it is of crucial importance to ensure that the workload assignment does not exceed the node's capacity and the resulting task communications can be sustained by the network facilities connecting to it. Furthermore, the task scheduling on specific hardware such as GPU and FPGA should be optimised accordingly to unlock the potential of the heterogeneous hardware [135,136].

2.9.4 Communication-aware Scheduling

From the perspective of implementation, inter-node communication triggers a cumbersome process involving serialisation, message queueing and network transmission, while intra-node communication can be reduced to passing an object's pointer in memory or expedited by the use of a concurrent programming framework like Disruptor[9]. As inter-node data transmission incurs much higher resource consumption and significant network latency, it is preferable to place communicating task pairs on the same node as long as it does not lead to resource contention. This also implies that communication-aware scheduling is a special type of resource-aware scheduling with a focus on minimising inter-node communication [5,29,46,50,52,86,173].

To be communication-aware, the scheduler needs to monitor the task communication pattern as well as the resource usage at each computing node. The communication pattern can be represented by a weighted directed graph of streaming tasks, in which the weights associated with vertices denote the task resource requirement and the weights on edges represent the instantaneous throughput of internal streams or the accumulated volume of data transmission. On the other hand, the deployment infrastructure is also regarded as a weighted directed graph of computing nodes, where the weights on vertices denote the node's resource availability and the weights on edges represent the bandwidth capacity of network connection. Therefore, communication-aware scheduling is to find a proper mapping of these two graphs at runtime in order to minimise the number of messages sent between machines while respecting the constraints on computation and network resources.

2.9.5 Fault-Tolerant Scheduling

Due to the large size of deployment, faults in a stream processing system are not only considered as exceptions but rather normal events. This implies that fault-tolerance should be made a first-class citizen in the scheduling phase to allow fast and efficient error-handling. In a data streaming system, the consequences of faults can range from a single tuple failure to cascading node crashes [79]. A tuple failure affects the timely delivery of messages, which could be caused by the package discarding on overloaded networks. A node crash, on the other hand, impairs the proper functioning of stream operators that are allocated to this node. In general, we categorise various fault-tolerance techniques into two groups: (1) state management, which allows stateful operators to survive from possible node crashes, and (2) event tracking, which ensures that messages are delivered

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

with regard to the desired semantic. Schedulers that are fault-tolerance-aware can alleviate the overhead of state management, reduce the risk of event replay, and expedite the recovery process by taking the possible failures into consideration during the placement of streaming tasks [97,143,155,164,180]. For example, the frequency of state checkpointing can be reasonably decreased by being availability-aware [20]: stateful tasks can be scheduled on more reliable computing nodes while stateless tasks that are fail-fast and easy to recover can be assigned to nodes with relatively lower availability. Also, placing communicating tasks in the vicinity and making sure that the bandwidth of network link is not over-utilised can help reduce the risk of message delivery errors (Chapter 5).

2.9.6 Energy-Efficient Scheduling

Reducing the total energy consumption is of great interests to the scheduling process [153, 156]. The total energy consumption is unnecessarily increased by the under-utilised computing nodes, so it is preferable to perform workload consolidation periodically in order to put the low-load nodes into shut-down or low-power mode (Chapter 4). Another critical source of energy consumption is the continuous communication among different streaming tasks. Depending on the distance of data transfer as well as the implementation of the underlying network infrastructure, the actual energy consumption of conveying a tuple over a message channel can vary significantly. This implies that the scheduler should also be aware of energy consumptions when deciding the stream routing, putting a large volume of internal streams on wired and reliable network connections rather than channels that are susceptible to interferences to reduce the possibility of retransmission.

2.10 Scheduling Methods

The previous section covers the various objectives of scheduling but provides little explanation on how these targets are achieved. In this section, we categorise different scheduling methods into four groups and explain the design and implementation of associated schedulers in details.

2.10.1 Heuristic-based Scheduling

The scale of the scheduling problem increases exponentially along with the growing application and platform complexity. Since finding the optimal schedule in such a huge solution space is an NP-hard problem, heuristic methods are preferred over exact algorithms to trade off optimality, completeness, and accuracy for speed. Aniello et al. [5] pioneered the dynamically scheduling of streaming tasks to improve application performance at runtime, where a greedy heuristic is applied to minimise inter-node traffic and avoid load imbalances among all the nodes. T-Storm [173] extended their work by allowing hot-swapping of scheduling algorithms and fine-grained control over worker node consolidation. The proposed traffic-aware scheduling algorithm has a greedy-based heuristic in its kernel that keeps trying to assign streaming tasks to available nodes with minimum incremental traffic load. Chatzistergiou et al. [29] also proposed an improved heuristic that utilises the domain-specific group-wise communication pattern between streaming tasks to minimise the communication cost, which guarantees to produce a schedule in linear-time outperforming the existing quadratic-time solutions in practical cases. Similarly, Rizou et al. [132, 133] came up with a task placement heuristic to minimise the network load which is calculated as the bandwidth-delay product of data streams between operators. Sun et al. [156] proposed an energy-efficient heuristic that differentiates the scheduling of critical and non-critical operators to minimise the response time and system fluctuations. R-Storm modelled the scheduling problem as a multi-dimensional Knapsack problem, for which they proposed a heuristic algorithm to put communicating tasks in proximity while ensuring no resource constraints on CPU and memory are violated [124]. The list of heuristic-based schedulers goes on with works done by Cammert et al. [15], Sun et al. [155], Heinze et al. [67, 68] and Jiang et al. [86].

It is also worth mentioning that heuristic can play a complementary role alongside the exact algorithms for better execution efficiency. The SODA scheduler [170] for System S, a proprietary DSMS developed at IBM, uses a local search heuristic as a backup solution to the main approach of mixed-integer optimisation. The heuristic method steps in when the CPLEX-based solution fails or becomes too slow to converge. In addition, meta-heuristic has been employed in the scheduling process to improve method adaptivity. Smirnov et al. [150] investigated the use of genetic algorithms to yield throughput improvement as compared to the greedy heuristics, where the task placement is adapted as an evolutionary process utilising the performance statistics gathered at runtime.

2.10.2 Graph-Partitioning based Scheduling

As we have discussed in Section 2.9.4, the scheduling process can be regarded as a graph partitioning problem where the communication graph is divided into smaller components hosted on different computing nodes. The quality of partitioning is often measured by the total amount of inter-partition communications, the degree of load balance across the platform, and the execution time required to work out a partition plan. By assuming the streaming tasks cannot move after their initial placement, Xing et al. [172] employed a static partitioning method to select an operator placement plan that is resilient enough to withstand different input rate combinations. For dynamic scheduling, Fischer et al. [50] collected the communication behaviour of applications, built the communication graph at runtime, and then set a partitioning objective function in the METIS software to reduce network loads and balance the CPU usage and bandwidth consumption over the platform. Similarly, Khandekar et al. [91] proposed a minimum-ratio cut subroutine to achieve hierarchical partitioning of the operator graph in System S. Eskandari et al. [46] also discussed hierarchical scheduling of streaming tasks with METIS, proposing a two-phase approach that improves on the traditional k-way partitioning method by allowing to dynamically compute the number of computing nodes required in the platform. Ghaderi et al. [60] employed a randomised scheduling algorithm with a theoretically provable guarantee on low-complexity, which enables a smooth trade-off between the cost of approaching the optimal partitioning and the queueing performance. In Li et al.'s work [99], the streaming tasks are firstly partitioned based on the dependency graph of communication, while determining the actual task assignment further involves joint optimisation on the topology structure, inter-node traffic and worker node load-balancing.

The theoretical aspect of graph partitioning in the context of streaming task scheduling has been investigated by Eidenbenz et al. [45]. They proved that optimal partitioning is an NP-hard problem and proposed an approximation algorithm that deterministically achieves a constant-factor approximation under a few consumptions on resource provisioning and processing cost.

2.10.3 Constraint-Satisfaction based Scheduling

Constraint satisfaction problems (CSPs) study how to optimise the objective function while ensuring that the result satisfies a number of constraints or limitations. Since scheduling streaming tasks on a collection of computing nodes is subject to various resource and SLA constraints, it is natural to consider scheduling as one of the constraint satisfaction problems requiring efficient search methods to be solved in real-time. When comparing to the heuristic-based scheduling discussed in Section 2.10.1, constraint satisfaction based scheduling emphasises more on the result optimality and is willing to traverse through a large area of solution space to maximise the objective function.

Cardellini et al. [19, 22] formulated an optimal scheduling problem considering the application and resource heterogeneity. The objective function is to minimise migration costs, and the constraints are modelled as the satisfaction of the application SLA. The problem is then solved by CPLEX, a widely used integer programming toolkit. Jiang et al. [87] also formulated a mixed integer program on scheduling to achieve max-min fairness in resource allocation for multiple streaming applications, where the non-convex constraints are converted to several linear constraints using linearisation and reformulation techniques. Schneider et al. [142] proposed a scheduling algorithm for the ordered streaming runtime to minimise synchronisation, global data and access locks, which allows any thread to execute any operator while maintaining the constraints of tuple order in operator communication. Load-balancing is added as an implicit constraint by Zhang et al. [179] to ensure more task assignment will be assigned to the node with the lowest CPU and memory consumptions. For a similar purpose, Liu et al. [106] proposed a runtime-adaptive scheduler that assigns tasks loads in proportion to the processing capacity of nodes. By dynamically migrating tasks assignment from slow nodes to fast nodes, the latency difference between the fastest and slowest nodes is mitigated. Buddhika et al. [14] formulated a resource-constrained problem on scheduling to reduce interference that adversely impacts the performance of streaming computations. They proposed a proactive scheduling algorithm that accounts for the changes in the stream packet arrivals and cluster resource utilisations, which utilises a new data structure of prediction ring to track the amount of workload expected in a given time window.

Constraint satisfaction problems can also be solved by exhausted search. Li et al. [101]

trained a model with Support Vector Regression (SVR) on a collection of monitored features to predict metrics like the average latency of tuple processing and the average size of tuple transfer. The resulting scheduler algorithm is essentially an exhaust search algorithm that traverses the whole solution space to find the optimal schedule with the minimised end-to-end latency.

2.10.4 Decentralised Scheduling

A decentralised scheduler is not a tangible entity that collects global information from the deployment platform and makes holistic scheduling decisions for the whole streaming system. Instead, it offloads the scheduling logic to the individual streaming operator or computing node, regarding each as an independent agent that collaborates with each other to converge to a feasible scheduling plan. The first prominent benefit of decentralised scheduling is robustness, which eliminates the single point of failure and allows graceful degradation in the presence of computing node crashes — the nodes that are not actively cooperating will be excluded from the scheduling resource pool. The second merit of this design is that it can base the scheduling decision on the accurate prediction of communication latency between different hosts, which is of crucial importance for dealing with streaming systems that are geographically-distributed on Edge and Fog cloud.

Specifically, the Vivaldi algorithm [34] — a decentralised approach that has linear complexity with respect to the number of network locations — is often employed to calculate accurate coordinates of distributed nodes in a latency network. Pietzuch et al. [125] pioneered the use of the Vivaldi algorithm to make continuous optimisation in stream processing scheduling without the global knowledge of the system. In their work, a stream-based overlay network is proposed to map the upper streaming system and the underlying physical network, so that the task placement is determined by searching in a multi-dimensional cost space in a decentralised manner. Cardellini et al. [18] presented a distributed and self-adaptive QoS-aware scheduler based on the Vivaldi algorithm, which can deal with infrastructure with non-negligible latencies. Rizou et al. [134] employed the Vivaldi algorithm to form a continuous latency space, and the proposed scheduler ensures that the QoS guarantee on latency is fulfilled while the network load incurred is reduced.

Repantis et al. [131], on the other hand, provided a set of fully distributed algorithms to discover and evaluate the reusability of data streams and processing components, so that sharing-aware component composition is allowed while remaining consistent with QoS requirements. Zhou et al. [183] proposed a decentralised and asynchronous scheduling algorithm that improves load balancing by dynamically migrating operators from overloaded nodes to lightly loaded ones.

Unless otherwise stated, the schedulers surveyed in the other subsections are centralised designed, which are often collocated on the master node of the deployment platform for the convenience of metric collection and scheduling coordination.

At the end of our review, we present a tabular comparison of key works regarding resource management and scheduling in distributed streaming systems.

Table 2.1: A Review and comparison of key works regarding resource management and scheduling in distributed streaming systems

Work	Resource Type	Prediction Method	Cost Modelling	Resource Adaptation	Parallelism Calculation	Parallelism Adjustment	Scheduling Objective	Scheduling Methods
De Matteis et al. [39]	CPU	Time series analysis	Minimal cost	Proactive	—	MPC	—	—
De Matteis et al. [38]	CPU	Time series analysis	Minimal cost	Proactive	—	MPC	—	—
HoseinyFarahabady et al. [77]	CPU, Mem	Time series analysis	Contention-aware	Proactive	—	MPC	Resource-aware	Control theory
Imai et al. [81]	VM	Time series analysis	Minimal cost	Proactive	Platform-oriented	—	Performance-oriented	Heuristic
Cardellini et al. [23]	VM	—	Reliability-aware	Reactive	—	Queuing theory	Communication-aware	Heuristic
Xu et al. [174]	VM	—	Minimal cost	Reactive	—	Control theory	Performance-oriented	Heuristic
Khoshkbarforoushha et al. [93]	CPU	Time series analysis	Distribution-based	—	—	—	—	—
Wang et al. [165]	CPU, Mem	Ensemble regression	Minimal cost	Proactive	—	Rule-based	—	—
Thamsen et al. [157]	CPU, Mem	Time series analysis	Contention-aware	Proactive	—	Rule-based	—	—
De Matteis et al. [37]	CPU	Time series analysis	Minimal cost	Proactive	—	MPC	—	—
Cardellini et al. [20]	CPU, Mem	—	Reliability-aware	Reactive	—	Queuing theory	Communication-aware	Heuristic
Kombi et al. [95]	CPU, Mem	Time series analysis	Minimal cost	Proactive	—	Rule-based	Resource-aware	Heuristic
Hidalgo et al. [72]	CPU, Mem	Markov chain	Minimal cost	Proactive	—	MPC	Fairness-aware	Round-robin
Shieh et al. [148]	CPU	—	Minimal cost	Reactive	—	Rule-based	Fairness-aware	Round-robin
HoseinyFarahabady et al. [76]	CPU, Mem	ARIMA	Contention-aware	Proactive	—	MPC	Performance-oriented	MPC
Mencagli et al. [116]	VM	—	Minimal cost	Reactive	—	Machine learning	—	—
Smirnov et al. [150]	VM	—	Minimal cost	Reactive	—	—	Resource-aware	Heuristics
Jiang et al. [87]	VM	—	Load-balancing	Reactive	—	Rule-based	Fairness-aware	CSP-based
Sun et al. [155]	CPU, Mem	—	Reliability-aware	Reactive	Performance-oriented	Control theory	Fault-tolerant	Heuristics
Shukla et al. [149]	VM	Queueing theory	Minimal Cost	Proactive	—	Control theory	Resource-aware	Heuristics
Cardellini et al. [22]	CPU, Mem	—	Reliability-aware	Reactive	—	Queuing theory	Communication-aware	Heuristic
Buddhika et al. [14]	CPU, Mem, Bandwidth	Time series analysis	Contention-aware	Proactive	—	—	Performance-oriented	CSP-based
Li et al. [99]	CPU, Mem,	—	Minimal cost	Reactive	—	—	Fairness-aware	Graph-based
Schneider et al. [142]	CPU	—	Contention-aware	Reactive	—	Rule-based	Resource-aware	CSP-based
Liu et al. [106]	CPU,	—	Load-balancing	Reactive	—	Rule-based	Fairness-aware	CSP-based
Ghaderi et al. [60]	VM	—	Minimal cost	Reactive	—	—	Resource-aware	Graph-based
Zhang et al. [179]	CPU, Mem	—	Load-balancing	Reactive	—	—	Communication-aware	CSP-based
Li et al. [101]	VM	Support vector regression	Minimal cost	Proactive	—	—	Performance-oriented	CSP-based
Eskandari et al. [46]	VM	—	Load-balancing	Reactive	—	—	Performance-oriented	Graph-based
Sun et al. [153]	CPU, Mem	Time series analysis	Reliability-aware	Proactive	Performance-oriented	—	Fault-tolerant	Heuristics
Eidenbenz et al. [45]	VM	—	Cost minimal	Reactive	—	—	Communication-aware	Graph-based
Lohrmann et al. [108]	CPU	Queueing theory	Load-balancing	Reactive	—	Queueing theory	Fairness-aware	Round-robin
Heinze et al. [70]	CPU	Queueing theory	Minimal cost	Reactive	—	Queueing theory	—	—
Lin et al. [103]	BW	—	Minimal cost	Reactive	—	Rule-based	—	—

Continued on next page

Table 2.1 – continued from previous page

Work	Resource Type	Prediction Method	Cost Modelling	Resource Adaptation	Parallelism Calculation	Parallelism Adjustment	Scheduling Objective	Scheduling Methods
Veen et al. [162]	VM	—	Minimal cost	Reactive	—	Rule-based	Fairness-aware	Round-robin
Heinze et al. [69]	VM	—	Minimal cost	Reactive	—	Rule-based	Resource-aware	Heuristics
Madsen et al. [110]	VM	—	Reliability-aware	Reactive	—	Rule-based	Fault-tolerant	Heuristics
Setty et al. [144]	VM	—	Minimal cost	Reactive	—	—	Fairness-aware	Heuristics

2.11 Summary

It is of great interest to both academia and industry investigating resource management and scheduling of distributed streaming systems to satisfy the Quality of Service (QoS) requirements with minimal resource cost. This topic has received extensive attention in the literature — many have paved the way for SLA-aware, self-adaptive deployment by proposing enabling techniques such as elastic resource scaling, dynamic task scheduling, and runtime operator parallelisation.

In this chapter, we summarise the achievements made on this front by presenting a comprehensive taxonomy and survey regarding the resource management and scheduling in distributed stream processing systems. Our narrative starts with the definition of the problem scope, where a sketch of the hierarchical structure of a stream processing system is presented to enumerate the research topics of interest. We then identified the issues and challenges associated with each research topic and developed a taxonomy to classify the specific work properties and method features. Following the structure of the taxonomy, we discussed existing work in details and compared the strengths and weaknesses of different methods. In the following four chapters, we present our research contributions in this area.

Chapter 3

Stepwise Auto-Profiling for Performance Optimisation of Stream Processing

For parallel execution on a particular platform, the streaming operators need to be appropriately replicated into multiple instances that split and process the workload simultaneously. In this chapter, we propose a stepwise profiling approach to optimise application performance on a given execution platform. It profiles the application execution automatically, scales up distributed computations over streams based on application features and processing power of provisioned resources, and builds the relationship between provisioned resources and application performance metrics to evaluate the efficiency of the resulting configuration.

3.1 Introduction

CURRENTLY, most state-of-the-art Data Stream Management System (DSMS) such as Storm[1] and Samza[2] are data-driven and operator-based. In operator-based DSMSs, continuous operations on data are realised as logical operators standing on data streams, and the DSMS is responsible for the partition and distribution of resources among operators to achieve satisfactory performance [5].

This chapter is derived from:

- **Xunyun Liu**, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu and Rajkumar Buyya, "A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, Volume 12, Issue 4, Pages: 1-33, ACM Press, 2017.

[1] `https://storm.apache.org/`
[2] `http://samza.apache.org`

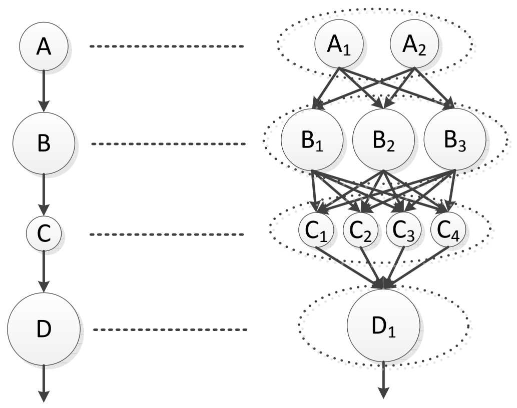


Figure 3.1: The logical view of a streaming application on an operator-based DSMS

For a streaming application, resource partitioning largely depends on how operators are built and organised. To better explain this process, Figure 3.1 illustrates the logical view of a typical streaming application. The left part of Figure 3.1 shows that all the queries[3] have been translated into a pipeline of *operators* that perform transformations on the data. The relative size of operators represents the relative time complexity, with edges indicating data flows within the application. These operators and edges constitute the *application topology*, which can be modelled as a directed acyclic graph (DAG) that wires the operations together and denotes the sequence by which a single datum traverses the system.

When it comes to the implementation, the topology of a streaming application is further subdivided. To enable parallel processing, each operator may have several tasks scattering out over the platform. Each task is an operator instance that ingests a portion of operator input and executes the whole operator logic simultaneously. As the right side of Figure 3.1 shows, tasks of a downstream operator in the topology take the results of its precedents as input and continuously feed the successors with its output stream. Clearly, it is important for an operator to secure a sufficient number of parallel tasks, so that it could timely process its inbound load and avoid being the bottleneck that throttles the overall throughput of the system.

However, the decision of the number of instances in each operator depends on the specific application deployment process, which involves provisioning resources from the underlying hardware infrastructure and determining how the logical representation is

[3]By query, we mean formal statements of information needs that apply on continuous streams and that demand some computational capacity to be processed.

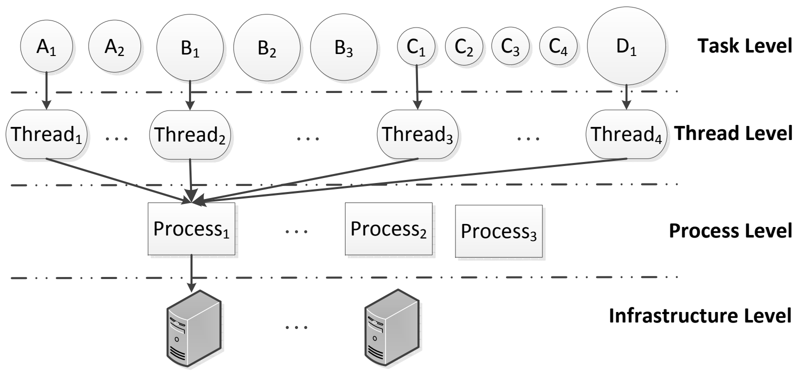


Figure 3.2: The physical view of an example streaming application on an operator-based DSMS. After being wrapped up by threads and processes, the tasks of operators are finally deployed on several physical or virtual machines.

mapped to a physical point of view for real execution. The latter is known as a critical *transition* from logic notation to real implementation. Figure 3.2 shows an example of this transition process. Tasks are wrapped up by threads, which are usually considered as the minimum units of execution in terms of resource scheduling, then threads affiliated to several processes are deployed on the particular execution environment. It is a non-trivial task to make optimal choices in such transition from logical to physical view because:

1. Different operators can have diverse requirements on different types of resources (CPU, memory, network bandwidth, etc.).

2. Changing the number of tasks for one operator may adversely affect the performance of other operators that are collocated in the same machine, causing unexpected bottleneck shift. Such kind of resource contention is hard to formally model.

3. The transition is largely platform-dependent. Thus, without field testing, it is difficult to guarantee the effectiveness and efficiency of the transition decision.

Due to the difficulties stated above, the most common approach used to determine operator parallelism is to gradually measure the execution capacity of each operator and adjust the number of tasks according to the expertise of the developer. Obviously, this method involves a huge number of man-hours and may result in a suboptimal configuration. As existing research mainly focuses on the other side of the problem, which is scheduling threads on processes or arranging processes on machines [5,11,15,117], the research question of automatically finding a proper and integral solution to this transition is largely overlooked.

Motivated by the goals of automation and enhanced developers' productivity, we design and implement a stepwise profiling framework that selectively evaluates several possible configurations, monitors feedbacks, and provides an entire solution to the transition. The objective of the proposed profiler is to determine the possible best performance[4] that the application can achieve in a particular execution environment.

To the best of our knowledge, this is the first work using profiling to holistically probe the best configuration for an arbitrary streaming application, which is capable of striking a balance between the data source and data ingestion subsystems for it to achieve sustainable high performance. Specifically, our main **contributions** are threefold:

- Our profiling system automatically scales up the streaming application on a given platform. Such processing parallelisation is achieved by profiling of both application features and processing power of provisioned resources. Therefore, developers are no longer required to provide parallel settings beforehand.
- The profiling strategy is designed as a feed-back control loop that allows for self-adaptivity, scalability, and general applicability to a wide range of streaming applications, which is demonstrated in our experiments.
- Based on the result of profiling, the relationship between resource provisioning and performance metrics of application is built, enabling further evaluation of the efficiencies of candidate topologies that are implemented for the same streaming application.

3.2 Motivation

The development cycle of a streaming application on an operator-based DSMS typically consists of two phases. The first phase consists in the logic development, where all the

[4]Though the meaning of performance may vary under various definitions of QoS (Quality of Service), we refer to it as the ability to steadily handle an input stream of throughput T within an acceptable processing latency L. In this sense, higher T means better performance as long as the latency constraint is met.

continuous queries or other data operations are implemented as logical operators working on data streams. The second phase consists in the application deployment, which mainly comprises a transition from logical to physical view. In this phase, the parallelism setting for each operator is determined and the decision on how tasks of operators are wrapped up and mapped to underlying resources is made, which are collectively referred to as a parallel *configuration*. Our primary motivation is to automate the transition and ensure that, in the resulting configuration, resources are properly partitioned among operators to enable better performance.

As mentioned above, optimisation of the application deployment is a non-trivial process. Here are three fundamental prerequisites that a streaming application should meet before it comes into service.

Application scaling: Scaling up[5] is a critical process for a streaming application to use distributed resources. As scaling is both resource specific and topology-dependent, there is no universal model able to provide a general solution. Therefore, the transition in the second phase has to be designed and performed by developers according to their own experiences, which causes additional development burden and may not yield efficient resource utilisation. It becomes even more problematic when the underlying resource structure is configurable. State-of-the-art DSMSs are integrating elasticity into their implementation to enable resource consumptions customisation with regard to fluctuating workloads. They support (1) dynamic resizing, e.g. DSMS can be scaled out by adding new machines, and (2) adjustable operator parallelisation, which allows stateful and stateless operators to choose their number of tasks in order to suit different sizes of execution environment. However, applications running on an elastic DSMS do not have the ability to adapt their configuration to infrastructure changes, meaning that they are unable to automatically take advantage of newly added compute resources when the DSMS is scaled out, and may face severe resource contention due to excessive parallelisation when the DSMS is scaled in. Our work fills in this gap by automating the scaling up process once the underlying system is updated, which complements efforts towards making DSMS scalable and elastic [24,58,140,141].

Besides, it is also desirable to quantitatively evaluate how efficient the transition is

[5]Scaling up refers to further parallelising the execution of logical operators to improve the resource utilisation of a streaming application, whereas scaling out/in stands for adding or removing machines.

and automatically probe whether there is still room for improvements. However, due to the labour-intensive task of manual deployment, it is a common practice for developers to stop scaling up the application when a transition that meets the requirement of performance is found. Nevertheless, it may result in suboptimal resource utilisation.

DAGs comparison: The topology of a streaming application is organised as a directed acyclic graph (DAG) of logical operators. However, the conversion of queries and operations on data streams into operators, which is performed in the first phase, can be conducted in multiple ways, resulting in different topologies that are logically equivalent. It means that, although different types of DAGs are formed by operators, they take the same input stream and all produce correct answers. It is difficult but still necessary to determine which one is better with respect to their performances in a particular platform.

Resource requirement analysis: It is essential to know how many resources are needed to meet time constraints to handle the inbound stream. The answer depends on the volume of the input stream and the application resource needs per input data element. In the context of stream processing, the input stream may vary significantly in volume and speed and so does the amount of resource needed per element. Usually, application developers do not have control over the input data [80], but tracking the latter could help them to guarantee real-time response with minimal resource consumption when the workload varies. Based on this, a rule-based auto-scaling approach could be proposed.

In this chapter, we choose application profiling as an empirical and adaptive approach to fulfil the above targets. Compared to analytical models based on abstract modelling, profiling excels as it provides more reliable results via real experiments. Furthermore, by taking advantage of profiling, our method is generic enough to support different execution environments, including variations in characteristics of underlying resources, load balancing of DSMS, and the type of streaming application running on it. Besides, a recalibration mechanism has been introduced to ensure that the decision on parallel configuration is up-to-date. Therefore, possible changes to application and DSMS, as well as data-dependent variation affecting the execution time of data elements, will not compromise the accuracy of profiled knowledge.

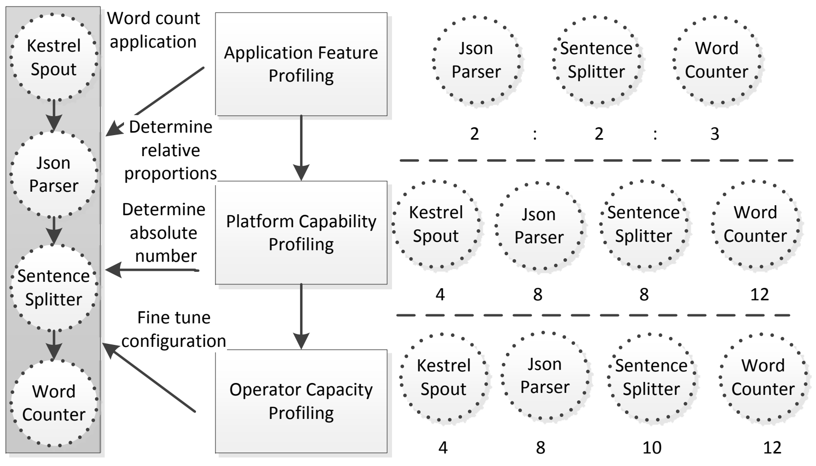


Figure 3.3: Flowchart of stepwise profiling and a working demonstration on a word count application.

3.3 Stepwise Profiling Overview

The profiling process works by selectively evaluating several possible configurations and finally choosing the one that shows the most promising performance potential, i.e. the one capable of processing more data streams per unit time while meeting the latency requirement.

Figure 3.3 describes the flowchart of our profiling approach and depicts how it applies to a word count application on Apache Storm. The topology of word count consists of four operators: the first operator, Kestrel Spout, pulls data from a 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. Next, the Sentence Splitter divides the main body of text into a collection of separate words, and finally the Word Counter is responsible for the final occurrence counting.

Regarding the profiling procedure, Application Feature Profiling (the first step) simulates the situation in which each task has adequate resources to conduct its data operation. It feeds the application with only a small size of input stream and aims to identify inherent application features that are not affected by the change of parallel configurations. On completion of this process, it determines the ratios of the numbers of tasks for the last three operators, which in this case is 2:2:3.

Platform Capability Profiling (the second step) stresses the platform with a high volume of input data to push it to its capability limit. At the end of this step, the actual

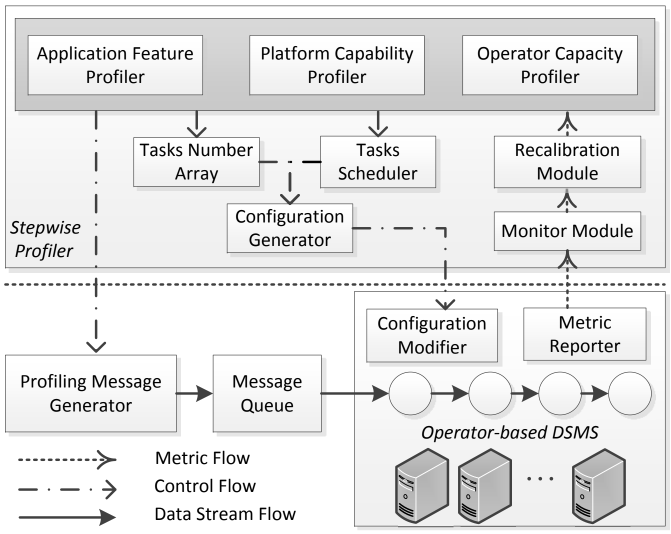


Figure 3.4: The framework of stepwise profiling system. Components that constitute the stepwise profiler are presented in the top of the figure and the profiling environment is depicted in the bottom of the figure.

number of tasks for each operator is determined.

Operator Capacity Profiling (the last step) makes necessary adjustment by monitoring the capacity of each operator. As our profiling model and measurement in the previous processes may have introduced some errors, this is the place where possible amendments are made.

The recalibration is essentially a repetition of the aforementioned profiling steps, triggered by performance degradation, detected via monitoring, when the resulted configuration is no longer suitable for the current system status.

3.4 Stepwise Profiling Design

Figure 3.4 illustrates the architecture of our stepwise profiler (top half of the figure) and how it interacts with the operator-based DSMS in the profiling environment (bottom half of the figure).

The profiling environment consists of a profiling message generator, a message queue, and an operator-based DSMS. All the profiling input originates from the message generator, where real data collected from the production environment is sent to the message queue at a controllable speed. In the meantime, the message queue works as a data buffer to store possible backlogs when the DSMS cannot cope with the speed of data generation. The operator-based DSMS contains the primitive streaming application logic as well as supporting hardware resources.

Each single round of profiling is a feedback control process that corresponds to a MAPE (Monitoring, Analysis, Planning, Execution) loop, the approach of which is widely adopted in the field of autonomic computing to enable self-adaptivity [90].

The MAPE loop starts with the metric reporter running alongside the DSMS, which constantly collects current performance metrics from the evaluated streaming application. This information is then acquired by the monitor module and being organised as a set of window-based performance histories. The analysis phase is conducted by the three control units of our stepwise profiler as shown in the grey box of Figure 3.4, which are referred to as Application Feature Profiler, Platform Capability Profiler and Operator Capacity Profiler. These modules check the collected performance metrics according to their designated profiling strategies and make decisions on whether another round of profiling is needed. The MAPE loop proceeds to the planning phase if the stopping condition is not met. In this phase, the three control units make necessary amendments on operator parallelism and rely on the configuration generator to compose a viable deployment plan, which includes determining the speed of data generation for profiling, the number of tasks for each operator, and how these tasks are deployed on DSMS. In the last execution phase, the configuration modifier is responsible for applying changes and facilitating automation of application deployment.

The recalibration module also works in the analysis phase to check if the previously profiled configuration still suits the current system state. If not, it will plan for the next round profiling without using any prior knowledge.

3.4.1 Application Feature Profiling

As illustrated in Figure 3.5, the logical view of a streaming application is divided into two parts: *data source*, the operator that forms the initial stream by continuously pulling data from external sources, and *data sinks*, where inbound data is buffered into a waiting queue before being processed by one of the parallel tasks.

The application feature profiling aims to identify two invariant properties that an

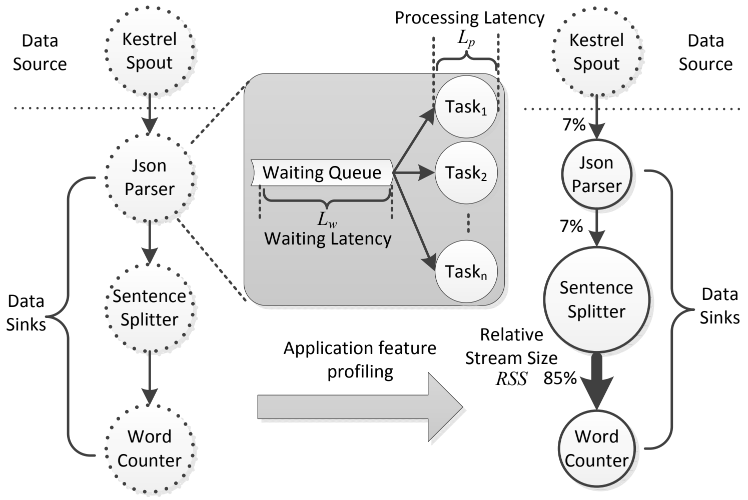


Figure 3.5: An example of application feature profiling in operation

operator should maintain regardless of its parallel degree through the analysis of a data stream of a relatively small size. The first property is the minimal processing latency $MinL_p$. As shown in Figure 3.5, the processing latency L_p is the time interval that a single datum would spend in a task for being processed, while $MinL_p$, illustrated by the size of operator on the right side of Figure 3.5, denotes the minimum value that L_p can reach in this particular platform. The second property is the **R**elative **S**tream **S**ize (RSS), which indicates the relative data transmission intensity for the operator. The word "relative" means that the amount of data transmitted between operators is normalised with regard to the total sum to show the proportion among operators. As shown in the right of Figure 3.5, the width of the lines between the operators represents the size of data flow relative to other visible streams.

These two properties remain constant regardless of the parallel configuration changes for two reasons. Firstly, for a given operator, $MinL_p$ only depends on its processing logic and how long it takes for this logic to be executed in the platform. To measure $MinL_p$ it is important to assert that the profiling stream load is sufficiently small, so that each task of this operator obtains enough resources as it requires to process the workload. On the other hand, changes in the parallelism for an operator, such as adding new tasks for it, influence the waiting latency L_w rather than the processing latency L_p, because a single datum in the stream cannot be executed by multiple tasks concurrently. However, it is still possible that L_p increases due to improper configurations, e.g. congested tasks may cause severe resource contention that causes high processing latency.

Secondly, the relative size of the data stream is a reflection of the operator type, which also makes it parallel-configuration irrelevant. More specifically, a given operator could be categorised into one of three types based on the correlation between its input stream and output stream, as presented in Table 3.1. S_i and S_o denote the relative size of input/output stream, respectively.

Table 3.1: Categorisation of operators based on its relative input/output stream size (selectivity)

Type of Correlation	**Expression**	**Example Operators**
Proportional	$S_o = C_{coef} * S_i$	Join, Function
Workload-Dependent	$S_o = f(workload) * S_i$	Split, Filter
Logic-Dependent	$S_o = g(logic)$	Top N, Aggregation

Proportional operators continuously work on one or more input streams and emit results based on each piece of input, which means that the size of the output stream is linear to the size of input stream, with C_{coef} as a constant that represents the linear coefficient. Examples for this category include stream joins and function operators. In the case of word count, the JSON Parser belongs to this category because its output size depends only on the particular input, and changes in the number of tasks do not affect the output size.

In the case of *workload-dependent operators*, the relative size of the output stream is not only decided by the size of input stream, but also it is influenced by the inherent property of the workload. For example, different tweets may have a different number of countable words, making the size of output stream fluctuate even when the size of the input stream is stable. But in the case of Figure 3.5, it is observed that, on average, the size of output stream for Sentence Splitter becomes 12 times larger than the input streams, which means that the application profiling helps in identifying what the value of $f(workload)$ would be when subject to a production input. Obviously, this correlation is also not affected by changes in the number of tasks available for the Sentence Splitter.

There are also *logic-dependent* operators whose output streams solely depend on the processing logic. Some common examples include the Top N operator, which compares and emits the most popular items based on their occurrences, and stream aggregation operator, which aggregates the input stream or regularly performs batch operations. The Word Counter operator in Figure 3.5 is used for aggregation and thus is an example of a logic-dependent operator.

Whenever an operator becomes a bottleneck, the DSMS has to throttle the upstream and downstream operators to maintain the system stability. This leads to the observation that the streaming application can be well-approximated with an intuitive queueing network of data flow, which runs on a computational system of unknown capability where contention affects all tasks in the same way. The latter assumption may not always hold true during the runtime, but it is reasonable for us to depict the relative parallelism requirement for each operator.

In light of this, more resources (in this context, more tasks) should be allocated to operators with relative larger input data streams and higher processing latency to prevent bottlenecks in the first place. After profiling the minimal processing latency and relative stream size for each operator, Algorithm 3.1 is proposed to provide an initial estimation on the number of tasks that an operator should incorporate considering its complexity and the amount of stream load it processes.

In summary, Algorithm 3.1 determines the parallelism ratio of each operator based on its calculated task load. The task load $TaskLoad_i$ of operator i is formulated as the product of its minimal processing latency $MinL_{p_i}$ multiplied by the sum of its input stream sizes $\sum\limits_k RSS_{k,i}$ (index k iterates over all the input streams of operator i). After the algorithm finishes traversing all the operators, each element in the resulted array $\overrightarrow{R}$ is updated with a parallelism ratio relative to the weight of its task load (line 23). Note that in both line 16 and 23, the index s of $\max\limits_{\forall s} TaskLoad_s$ iterates through all the operators in the topology.

However, there are two exceptions to this general rule. Some operators are logically *non-parallel*, which means that they can have only one task and thus are more likely to restrict the scalability of the whole system. Our algorithm takes these operators into consideration by fixing their parallelism degrees to 1 and quantitatively calculating the restriction they pose to the total size of stream flows ($TotalFlow$). $TotalFlow$ intuitively estimates the maximum sum of throughput generated by each operator. Based on this, Algorithm 3.1 decides the parallelism ratio or degree for other parallel operators to be able to handle the data stream of its share.

Secondly, some operators are *non-scalable* as they are dominated by logic-dependent operators, which means that there exists a logic-dependent operator in every path that

Algorithm 3.1: Calculate the relative ratio or number of tasks for each operator

```
Input: $MinL_{p_i}$: minimum processing latency of operator $i$
Input: $RSS_{i,j}$: relative stream size between consecutive operators $i$ and $j$
Output: $\overrightarrow{R}$: parallelism ratio array of parallel operators, in which $R_i$ corresponds
        to operator $i$
Output: $\overrightarrow{P}$: parallelism degree array of non-parallel and non-scalable operators,
        in which $P_j$ corresponds to operator $j$
1  Initialise each element of $\overrightarrow{R}$ to 1;
2  $TotalFlow \leftarrow \infty$;
3  Identify all the operators that are dominated by logic-dependent operator, label
     them as Non-Scalable;
4  foreach Operator i do
5      if i is Non-Parallel then
6          $P_i \leftarrow 1$ ;
7          $TotalFlow \leftarrow \min(TotalFlow, \frac{1}{Min_{Lp_i} * \sum_k RSS_{k,i}})$;
8      else
           /* Calculate TaskLoad for parallel operator i */
9          $TaskLoad_i \leftarrow MinL_{p_i} * \sum_k RSS_{k,i}$;
10 foreach Parallel Operator i do
11     if i is Non-Scalable then
12         if $TotalFlow = \infty$ then
13             $P_i \leftarrow \lceil \frac{TaskLoad_i}{\min_{\forall s} TaskLoad_s} \rceil$;
14         else
15             $P_i \leftarrow \lceil TaskLoad_i * TotalFlow \rceil$;
16     else
17         $R_i \leftarrow \frac{TaskLoad_i}{\max_{\forall s} TaskLoad_s}$;
18 return $\overrightarrow{R}$, $\overrightarrow{P}$;
```

connects this operator to the data source. Recalling that the size of output stream for a logic-operator is irrelevant to the input size, we can reasonably infer, in this case, that the size of input stream would be constant even if the system throughput increases. Therefore, instead of assigning parallelism ratios to them in $\overrightarrow{R}$, Algorithm 3.1 calculates the initial parallelism degrees for these operators in $\overrightarrow{P}$ based on $TotalFlow$ and task load, indicating that their parallelism degrees are not to be proportionally scaled in the next step.

The output $\overrightarrow{R}$ of Algorithm 3.1 only provides an array of decimals. However, the next step requires an array of integers, as this array represents the ratio of the number of tasks. In the decimal to integer conversion, precision is not the primary concern since the results may be subject to measurement errors introduced by the profiling process. Therefore, the chosen value in the resulted integer array is reduced if possible, in such a way that it still roughly depicts the basic proportions. For this purpose, a parameter of unit task load called *slice* is introduced to convert all the decimals in $\overrightarrow{R}$ into integers according to Equation 3.1.

$$R_i \leftarrow \lceil \frac{R_i}{slice} \rceil \quad slice \in (0,1] \tag{3.1}$$

The value of *slice* should be tailored to the specific streaming application. Our rule of thumb is to try small values (0.1, 0.2, etc.) and select the one that minimises the profiling effort in the next step. Section 3.6.4 will shed more light on the parameter selection with real experiments.

It is also worth mentioning that in line 3 we omit the process of identifying dominance relationship for the sake of simplicity. Actually, there are some breadth-first searches starting from each logic-dependent operator to examine which operators are affected logic-dependent successors. In summary, the algorithm sequentially evaluates the operator located at the head of the queue with regard to the status of its predecessors (each operator maintains a HashSet of all its status-undetermined predecessors for quick location and removal). If an operator has all its predecessors marked as either logic-dependent or already dominated, i.e. its HashSet of status-undetermined predecessors is emptied while this operator dequeues, it then should be identified as dominated and its successors are added to the tail of queue for further evaluation.

Algorithm 3.1 also has a computational complexity of $\mathcal{O}(n)$ with the worst case being $\mathcal{O}(n * (d^-_{avg} + 2))$, in which n is the number of operators and d^-_{avg} is the average vertex in-degree in the topology graph. The most time-consuming step lies in line 3 as each operator in the topology may be repeatedly visited, at most, its in-degree times to determine whether it has been dominated by logic-dependent operators or not. Besides line 3, the algorithm body traverses the topology graph only twice and all the required input can be collected with simply one round of profiling.

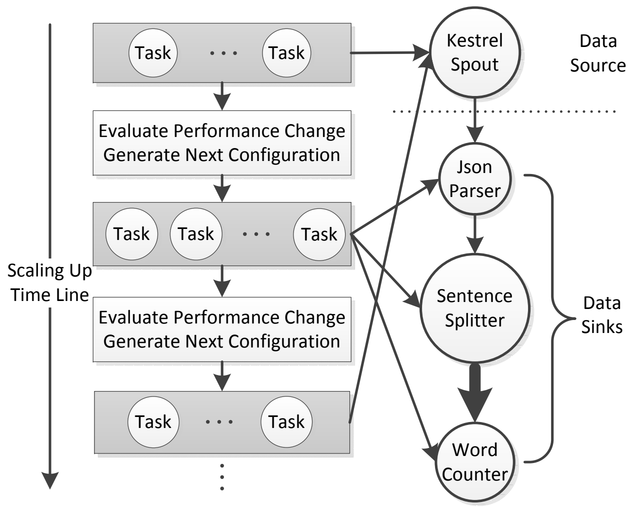


Figure 3.6: An example of platform capability profiling in operation

3.4.2 Platform Capability Profiling

Unlike the previous step which requires only a small data stream to probe application features, the platform capability profiling requires the message generator to produce a continuous data stream that is large enough to stress the streaming application. Given sufficient profiling data, the configuration of the application is changed through a trial-and-error process in order to determine the real capability of DSMS as well as its underlying infrastructure. The resulting configuration reveals a reasonable choice of resource partition in this platform where it is capable of handling a relatively large stream without violating the latency constraint[6].

As shown in Figure 3.6, each configuration trial is first evaluated in terms of system performance variation. Specifically, changes in throughput and latency are collected and reported to the monitor module, which can be used to identify if the new configuration improves the resource utilisation. Configuration changes that have a negative impact on the system performance are discarded in this phase.

Based on the result of performance evaluation, the profiler applies changes to the configuration according to Algorithm 3.2 and generates a new one for the next round of profiling. The new configuration not only targets throughput improvement, but also aims at maintaining the balance between data source and data sinks. If it failed to do

[6]Different applications may have different preferences with regard to their desirable performance. Though the final decision is left up to the application developer, as a default the profiler favours better throughput on the condition that the system still meets the pre-defined latency requirement.

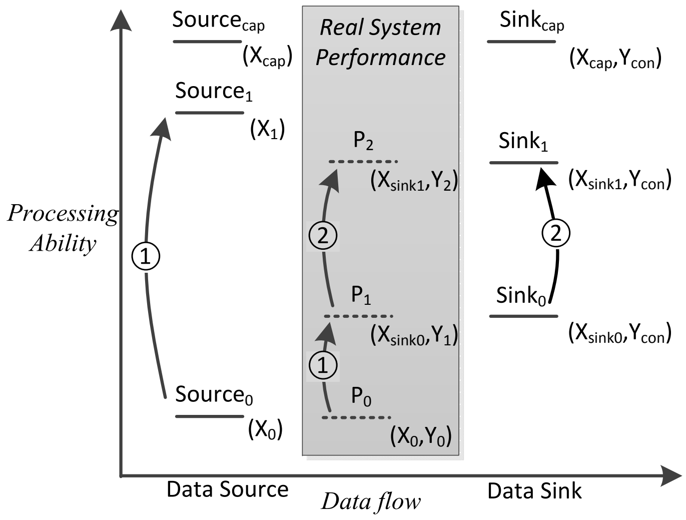


Figure 3.7: Balancing data source and sinks in platform capability profiling. The vertical axis represents the processing ability ordering by throughput. The horizontal axis denotes the system components we concern about.

so, an overly powerful data source may cause severe backlogs in data sinks and lead to a higher system latency, while an inefficient data source starves the following operators and encumbers the overall throughput. As a result, tasks are alternatively added to the data source or to data sinks to strengthen their processing abilities, and the search for the desired configuration leads the application to its performance limit where neither enhancing data source nor data sinks improves it.

Figure 3.7 illustrates the aforementioned scaling process with an example. At each edge of the figure, the solid short lines with labels indicate different configurations for data source and data sinks, while the dashed lines in the middle represent the overall system performance resulting from the configurations on each side. Different configurations lead to various processing potentials in terms of throughput and latency, which are shown in the right corner. Thus, short lines are all ranked by throughput with different heights in the figure, and the curves connecting them with numbers denote the potential throughput variances that resulted from different configuration changes by applying Algorithm 3.2. At first, the system is configured with $Source_0$ for data source and $Sink_0$ for data sinks, where $Source_0$ indicates that the data source is able to pull a data stream at a throughput of X_0, while $Sink_0$ with (X_{sink0}, Y_{con}) means that data sinks under this configuration are capable of dealing with a stream of size X_{sink0} within the user-specified latency constraint Y_{con}.

Given a particular platform, there should be a configuration that delivers the best performance in this profiling environment, indicating that the data source and data sinks have been properly coordinated. As denoted by the top two short lines in Figure 3.7, the data source $Source_{cap}$ and data sink $Sink_{cap}$ represent such ideal configuration that the profiler aims to achieve at the end of its operation. Initially, the data source $Source_0$ is less powerful than the data sink $Sink_0$. Thus, the system performance P_0 is confined at (X_0, Y_0), where X_0 is decided by $Source_0$ and $Y_0 < Y_{con}$ because the data sinks are underutilised. After detecting the latency margin, the profiler first scales up the data source to $Source_1$, causing bottleneck shifts to data sinks $Sink_0$. This time the performance P_1 is limited at (X_{sink0}, Y_1) where $Y_1 > Y_{con}$ since that some backlogs have already been accumulated. Afterwards, the profiler enhances the ability of data sinks from $Sink_0$ to $Sink_1$, improving the performance from P_1 to $P_2 = (X_{sink1}, Y_2)$. However, as indicated by $Y_2 > Y_{con}$, the last modification is still inadequate and another sink scaling is needed in the next round.

Scaling up data source and data sinks works by adding new tasks to operators that need to be further parallelised, but it also raises a question of how to map the updated task graph to the underlying machines in order to achieve better resource utilisation. This is also known as the task placement and scheduling problem. There are several policies available to decide the distribution of tasks across the platform, and certain applications may require a particular policy to suit a very specific need (e.g. assigning a particular task to a particular machine due to licence restrictions). We therefore design the platform capability profiler to enable scheduling policies to be plugged in so that it can be used in conjunction with various scheduling heuristics with different optimisation targets, such as minimising inter-node communication [5, 173], reducing the average tuple processing time [100], and being resource-aware to ensure the capability of each task to handle its task load [124]. Since the focus of this work does not lie in task placement and scheduling, we introduce our profiling approach in tandem with the widely adopted *round-robin policy* and apply it in a platform with homogeneous computational resources for ease of presentation. The round-robin policy is particularly suitable for homogeneous platforms as tasks are evenly distributed among available machines to enable fault-tolerance and load-balancing.

Algorithm 3.2: Generation of a new configuration under the round-robin policy

```
   Input: $T$: Topology throughput
   Input: $\alpha$: Threshold for triggering reconfiguration
   Input: $T_b$: Best throughput record
   Input: $\overrightarrow{R}$: Ratio of parallelism of each operator
1  if Last change increased $T$ then
2      $T_b \leftarrow T$;
3      if Latency constraint is not met then
4          foreach operator do Add tasks according to $\overrightarrow{R}$ and the operator's position
             in the topology;
5      else Increase number of tasks of source by 1;
6  else if Last change did not significantly change $T$ then
7      if Last change enhanced data sink then
8          Increase number of tasks of source by 1;
9      else if Last change enhanced data source then
10         foreach operator do Add tasks according to $\overrightarrow{R}$ and the operator's position
             in the topology;
11     else if Last change throttled the data source then
12         if latency requirement has been met then
13             Terminate the profiling;
14         else Increase throttle strength;
15 else if Last change decreased $T$ then
16     if $T < \alpha * T_b$ then
17         Return the system to the configuration where the best performance is
             observed;
18         Throttle the data source;
19     else if Last change enhanced data sink then
20         Increase number of tasks of source by 1;
21     else if Last change enhanced data source then
22         foreach operator do Add tasks according to $\overrightarrow{R}$ and the operator's position
             in the topology;
23     else if Last change throttled the data source then
24         Decrease throttle strength;
```

Algorithm 3.2 shows the interplay between performance evaluation and configuration generation carried out by the profiler under the round-robin policy[7]. Scaling data sources is a relatively lightweight operation: it only requires the number of tasks for the

[7]We adopt a Two Sample T-Test to determine whether a throughput change is significant or not, more details are given in Section 3.6.1 as a part of experiment setup.

data source to be increased by 1, so that the application has one extra task pulling data from the message queue and thus increasing the input rate. However, The decision of increasing the parallelism for a data sink operator depends on the type of operator and its position in the topology. For example, an operator should keep its number of tasks unchanged if it is a non-parallel operator, or if it is non-scalable dominated by logic-dependent operators as its input stream tends to be steady during the profiling process. As for other types of operators, $\overrightarrow{R}$ indicates the extent of enhancement for each operator.

Nevertheless, not every scaling effort, especially those applied for data sinks, can guarantee improvements. The reason why scaling data sinks is even more difficult than scaling data sources is that it has to exhaust current resources for additional computation and coordination. Therefore, to meet the latency constraint, our profiler performs a third operation on configuration, *source throttle*, which limits the size of input stream by controlling the amount of data that is allowed to sojourn in the system.

The complexity of computation required for configuration generation is constant. However, the profiling process that evaluates the effectiveness of a new configuration is relatively time-consuming since performance measurement must wait until the application is stabilised. To examine the number of profiling rounds required in the worst case, we regard Algorithm 3.2 as a search algorithm that explores a vectored value space, with each dimension confined by the actual parallelism degrees that can be seen in the ideal configuration. Given the fact that every three consecutive profiling efforts can increase the total number of used tasks at least by $\left\|\overrightarrow{R}\right\|_1$ through data sink enhancement (except for consecutive source throttles, which is rare), and that assigning excessive parallelism degree to an operator would harm the application performance, it is intuitive to deduce a conservative estimate that in the worst case there will be no more than $3 * \frac{nMax_p}{\left\|\overrightarrow{R}\right\|_1}$ rounds of profiling. In the expression, n is the number of operators in the topology, and Max_p represents the maximal parallelism degree among all the operators. However, Max_p is unknown before the actual profiling, but it can be approximated in practice by the number of threads able to run simultaneously in this particular platform (by multiplying the number of available cores by the number of thread(s) per core).

3.4.3 Operator Capacity Profiling

The previous step of profiling divided the streaming application into two parts (data sources and sinks), of which the parallel configurations of operators are collectively adjusted based on the overall performance of the system. Such coarse modifications may not be accurate enough to achieve the targeted configuration. Therefore, in the third step, profiling is carried out at operator level through the individual evaluation of performance of each operator. The goal of this step is to achieve finer granularity of performance tuning.

Operator capacity, which is formally defined in Equation 3.2, is used to quantitatively evaluate the degree of utilisation of operators in data sinks. In the equation, $Operator\_latency$ is the average time that a single datum would spend in this operator over a specific time period. The length of such time period is called $Window\_size$ and the amount of data processed in this period is denoted by $Executed\_load$. Thus, capacity represents the percentage of the time in the observation time window that the operator spent executing inputs. The closer to 1 is this value, the more likely the operator is the bottleneck in our topology.

$$Capacity = \frac{Operator\_latency * Executed\_load}{Window\_size} \tag{3.2}$$

This step utilises the same profiling environment used in the previous step. However, besides overall performance metrics such as throughput and latency, the profiler in this stage also collects the capacity information from each operator for fine-grained evaluation. The profiling strategy also resembles the previous one: the performance evaluation phase sheds light on the system status and the possible bottleneck, and the previous configuration change is revoked if it causes performance degradation. However, this process differs from the previous step in that it has only one operation, which is increasing the number of tasks by 1 for the operator that has the highest capacity and has not been enhanced nor revoked. If there is no performance improvement obtained from enhancing the operator with the highest capacity, the operator that has the second highest capacity is tested in the next round and so on.

There are two stopping conditions for the profiling. The first is when there are consecutive revocations observed indicating that recent scaling up efforts on candidate operators have failed. The second condition is when all the measured operator latencies approach the minimal processing latency $MinL_p$ by a factor k. We evaluate the effect of diverse values of k in the performance of the profiling later in Section 3.6.4.

3.4.4 Recalibration Mechanism

The application of the above three profiling steps yields a specific parallel configuration that builds a relation between provisioned resources and performance metrics. However, such relation is perceived to be volatile, since the performance under the same configuration may vary and the resulting configuration may need to be promptly modified due to the live changes that happen to the streaming application or platform. This section therefore discusses the recalibration mechanism, which repeats the profiling process when necessary to keep the configuration and operator profiling up-to-date with minimal adjustment cost.

In general, recalibration is triggered by any three types of changes: (i) resizing of DSMS, which leads to a new platform to be profiled after the infrastructure layer is dynamically scaled; (ii) re-deployment of the application, resulting from the alteration of application topology and the manipulation of some critical parameters that would greatly affect the application behaviour; and (iii) data-dependent variation, an uncontrollable factor related to the characteristic of workload, causing performance to vary even if the configuration remains unchanged.

For the first two causes, the recalibration decision is straightforward. If the platform or application turn into a state that has not been previously profiled, all the profiling steps are repeated. However, the process is more challenging when it comes to dealing with data-dependent variation, as all the changes are independent of the platform and application. We can safely assume that all data elements within the same stream are of the same type, but the time and space complexity of execution may vary along with the changing element size or the density of information contained. The Sentence Splitter, in the word count topology, is a typical example to show the effect of data-dependent variation: its process latency and relative size of output stream depend on the average length of incoming tweets.

To deal with such variation, the recalibration mechanism requires a monitoring system to oversee the degradation of performance during runtime. It continuously monitors the length of the message queue, which indicates the capability of application to handle a certain level of throughput that previously demonstrated in the profiling phase, and the system latency, which examines if the user-specified latency constraint is still satisfied. In order to reduce the frequency of adjustment, we adopt a threshold-based method which postpones any recalibration action until the monitored values have exceeded the predefined threshold for a specific period of time.

3.5 System Implementation

The architecture of the stepwise profiling system, as shown in Figure 3.4, consists of two main parts — the profiling environment and the stepwise profiler[8].

The setup of the profiling environment has been briefly introduced in Section 3.4. More specifically, the Profiling Message Generator is a Java program that reads the workload file on demand in order to emit a particular size of profiling stream. The Message Queue connecting the streaming application to the Profiling Message Generator is built with Twitter Kestrel[9], a distributed queueing system that enables controllable message buffering. Developers could make use of the Thrift interface provided by Kestrel to retrieve the length of the message queue and determine whether the streaming application has been overwhelmed by the profiling data.

As a specific DSMS was needed to enable the implementation and evaluation of the prototype, Apache Storm was chosen. This is because it is an open source software (and thus has all the source code available and detailed on-line documentation), and provides a built-in metric system and external configuration reader that facilitate the implementation of the stepwise profiler.

Figure 3.8 describes the integration of the profiler prototype into Apache Storm. The Stepwise Profiler module in the grey box is DSMS-independent, as it only interacts with other components of the architecture to make profiling decisions. Therefore, it is imple-

[8]The source code is available on https://github.com/xunyunliu/StepwiseProfiler
[9]https://github.com/twitter-archive/kestrel

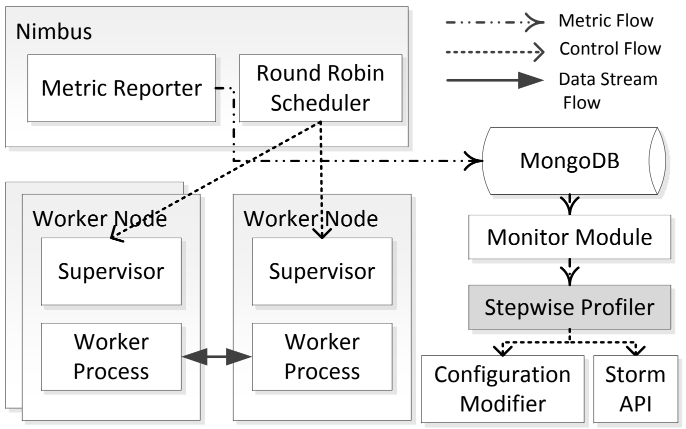


Figure 3.8: The integration of the profiler prototype into Apache Storm

mented as a stand-alone Java Program.

The other modules of the architecture interact directly with the DSMS to collect information or apply changes, thus the implementation of these modules are DSMS-dependent. The Metric Reporter component utilises Storm's built-in metric system and the associated RESTful interface to collect performance information and publish results. Such metrics are then periodically sent to MongoDB[10] to facilitate tracking of performance changes. The Monitor Module, implemented as a Java program, inquires the MongoDB for the latest system status and reports it to the Stepwise Profiler for decision-making. In this process, some performance metrics, like complete latency (average time taken by a tuple and all its offspring to be completely processed by the topology), number of data emitted, and operator capacity can be directly used in the stepwise profiler. Some metrics, however, require certain post-processing in the Monitor Module. For example, there is no default definition for throughput among the built-in metrics. Thus, to avoid ambiguity, the Monitor Module calculates the overall throughput of a streaming application based on the observed number of acknowledgements or emitted data per unit of time, depending on whether the application adopts reliable message processing or not.

We also utilise some useful features of Storm in the process of generating and applying new configurations. Specifically, Storm not only supports reading parallelism setting of operators from an external configuration file, but also provides a command line tool (Storm API) to manage the topology with additional operational parameters. The stepwise profiler thus makes use of the Configuration Modifier component, which is

[10]https://www.mongodb.com/

implemented as a script file, to pack up all the profiling decisions in a deployment configuration file, and then invokes the command line tool to submit the application with the updated deployment scheme for the next round of profiling. The round-robin scheduler guarantees that tasks are evenly distributed across Worker Nodes and that load is equally distributed among machines.

Another aspect relating to implementation is the management of operator states during the scaling up process. We do not address dynamic stream rerouting and live state migration since the Configuration Modifier relies on the rebalance command to apply any deployment changes. This command, as a built-in Storm functionality, essentially pauses the application during the redeployment and then restarts it from scratch with the new configuration, following the so-called Pause and Resume protocol [66]. As our current prototype treats stateful operators the same ways as stateless operators in terms of scaling, the management of operator states is not transparently handled by the profiling framework. Therefore, it is required that stateful operators preserve their states at the application level when the rebalance command is triggered, and these operators should also be initialised with the previous states when the application is restarted. However, there are some advanced mechanisms proposed in the literature that enable application-agnostic state management and interruption-free operator scaling, which is discussed in Section 3.7.

3.6 Performance Evaluation

We have conducted three different sets of experiments to validate the effectiveness of our prototype.

1. The first experiment presented in Section 3.6.2 evaluates whether the stepwise profiling effectively applies to a variety of streaming applications, and if it fulfils the other goals discussed in Section 3.2.

2. The second one in Section 3.6.3 assesses the scalability of our prototype and showcases its runtime overhead under relative large test cases.

3. The last experiment in Section 3.6.4 investigates the effect of different parameters

on the profiler performance, based on which we suggest default preferences.

3.6.1 Experiment Setup

The experiment environment is set up on a private cloud running OpenStack. 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 control machine, a ZooKeeper node and several processing nodes. The first two nodes are "m1.large" (4 VCPU and 8 GB RAM), while the rest of the processing nodes are "m1.medium" (2 VCPU and 4GB RAM per machine). The control machine hosts the Stepwise Profiler, Profiling Message Generator, and the Message Queue components of the architecture to avoid possible interference to the profiling result.

Test Applications

We adapt six streaming topologies as our evaluation applications[11]. These include three synthetic topologies (collectively referred to as Micro-benchmark) and three real-world streaming applications: Word Count (WC), Synthetic Word Count (SWC), and Twitter Sentiment Analysis (TSA). All applications are configured with acknowledgements enabled in order to track the complete latency, and they process the same type of workload to calculate comparable throughput. The profiling stream used for performance test is recursively generated from a single workload file, which contains 159,620 tweets in JSON format collected from 24/03/2014 to 14/04/2014. In addition, these applications are carefully tuned to avoid the out-of-memory crash and other failures due to insufficient resource allocation, so that the only potential consequence of improper configuration is suboptimal performance, rather than abrupt termination of the application.

Micro-benchmark: the micro-benchmark topology is synthetically designed to evaluate how the stepwise profiler generalises to different topology structures. As shown in Figure 3.9, it covers three common structure patterns: *Linear*, *Diamond*, and *Star*, where an

[11]In the following section, we use application and topology interchangeably to refer to the streaming logic developed on Apache Storm.

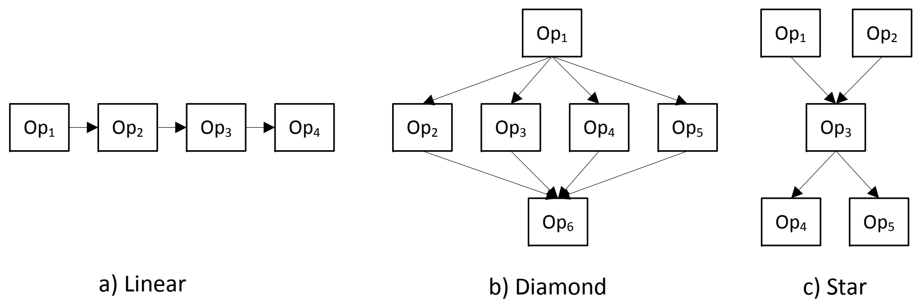


Figure 3.9: Structure of the synthetic Micro-benchmark topologies

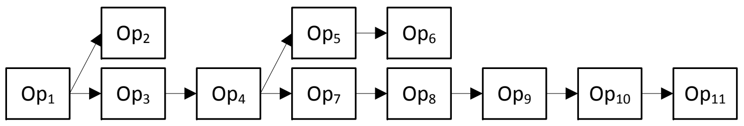


Figure 3.10: Structure of the Twitter Sentiment Analysis (TSA) topology

operator has (1) one-input-one-output, (2) multiple-outputs or multiple-inputs, and (3) multiple-inputs-multiple-outputs, respectively.

In addition, the *execute* method of each operator is implemented in three different patterns in order to reflect diverse time-space complexities. Some operators are (1) *CPU bound*, as they invoke a random number generation method Math.random() 10000 times for each tuple received. Some are (2) *I/O bound* with only a JSON parse operation applied on the incoming tuple, so that they spend more time on waiting for I/O operations rather than actually processing the current data. The rest of the operators are (3) *Sojourn time-bond*, which sleep for 5 *ms* upon any tuple receipt. These operators are introduced to mimic the cases where an external service is requested to complete the tuple transaction. Consequently, they demand almost no CPU and memory usages on the execution platform, but still consume a substantial sojourn time for each incoming tuple to be processed.

All these operators have a function implemented to read the operator selectivity[12] from the external configuration file. Higher selectivity can be specified to produce saturated network usages, so that I/O bound operators could be overwhelmed by large internal streams.

Word Count and Synthetic Word Count: the Word Count topology is illustrated in Figure

[12]The selectivity is defined as the ratio between the number of output tuples produced and the number of tuples consumed by this operator.

3.3. The Synthetic Word Count topology adds a Waiting operator (a bolt[13] in Storm's terminology) between the Kestrel Spout and the JSON Parser, where each incoming tuple is kept for 1 *ms* before being sent to the next operator. Therefore, WC and SWC are actually two different implementations for the same streaming application.

Twitter Sentiment Analysis: we adapted this topology from a mature open-source project hosted on Github[14] with the structure shown in Figure 3.10. It has 11 bolts constituting a tree-style topology that has 8 stages in depth. The processing logic of this application is straightforward: once a new tweet is pulled into the system through Kestrel Spout (Op_1), it is firstly stored 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 bolt (Op_4) that splits the sentence and calculates the sentimental score for the whole content using AFINN[15], which contains a list of words with their pre-computed sentiment valence from minus five (negative) to plus five (positive). There are also several bolts to count the average sentiment result (Op_5, Op_6) and to rank the most frequent hashtags occurring over a specific time window ($Op_7 \sim Op_{11}$).

Evaluation Methodology

We use throughput and complete latency to quantitatively evaluate the performance of streaming applications. Higher monitored throughput indicates higher performance potential, as long as the complete latency satisfies the desired target. In other words, if a streaming application has demonstrated throughput T in the profiling environment, we can confidently assume that it has the ability to process any throughput $T^{'} < T$ without violating the latency constraint, unless the profiling knowledge needs to be recalibrated. Therefore, to probe the maximum sustainable throughput, the profiling environment feeds the applications with large inputs, until the performance hits its highest stable point before recording it as the observed value.

The measurement of performance metrics first requires the test application to be deployed on the execution platform. Apart from complying with the generated configuration, we also set the number of workers to one per machine and the number of tasks to

[13]Operators in Storm are called *spouts*—if they are data sources—or *bolts* otherwise.
[14]https://github.com/kantega/storm-twitter-workshop
[15]http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010

be the same as the number of executors, which conforms to the recommendation of the Storm community[16]. All the topologies run for 10 minutes to enable sufficient stabilisation, and then performance data are collected every 30 seconds for 10 minutes, forming an array of 20 observations on throughput and latency. These settings were chosen because we observed that the fluctuation among the average results of repeat experiments did not exceed 3%, and the Lilliefors Test does not reject the null hypothesis that the observations on throughput are normally distributed (at the 5% significance level). However, other applications may require longer time to reach a stable state, or a larger monitoring interval to avoid drastic but periodic throughput variation.

As we have collected an array of throughput metrics in each profiling round, the significant change mentioned in Algorithm 3.2 can be determined by a Two Sample T-Test (at the 5% significance level) to determine if there is a statistically significant difference between the performance of previous and new configuration.

For completeness, Table 3.2 summarises the parameter settings used for setting up the stepwise profiler in our evaluation.

Table 3.2: The parameter settings used by the stepwise profiler in evaluations

Parameters	**Values**
Latency constraint (Y_{con})	500 ms
Task load unit ($slice$)	0.3
Stopping coefficient (k)	2
Threshold for triggering reconfiguration (α)	0.9

Comparable Methods

We compare the stepwise profiling prototype with two existing scaling up approaches: the threshold-based method and Stela [174].

The threshold-based method adjusts the parallelism hint of each operator based on its monitored capacity as formulated in Equation 3.2, in contrast to those in the literature that set up thresholds over the CPU utilisation of worker nodes [64, 69]. The scaling up threshold in our experiment is set to be 0.8 and we reduce the capacity of congested operators by gradually increasing their parallelism. In this sense, it may take several rounds

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

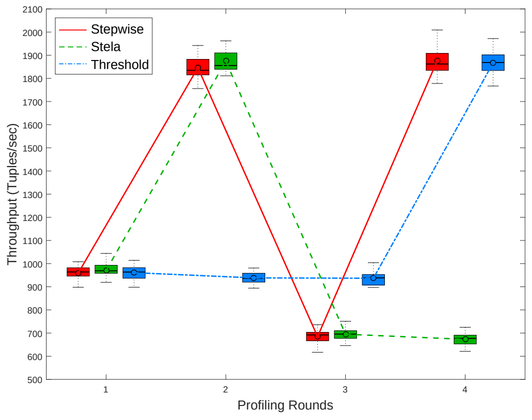
to complete the scaling up process: the application is deployed with no parallelism configured[17] at the beginning. In the following rounds, the most overloaded operator will be provided with an extra task in an attempt to rectify the congestion and optimise performance.

Stela scales up the streaming applications with the same goal of optimising post-scaling throughput. In contrast to the threshold-hold method that examines only the operator capacity for bottleneck detection, Stela prioritizes those congested yet influential operators in the scaling up process by calculating the ETP (Effective Throughput Percentage) metric [174]. Furthermore, it allows the parallelism degree of multiple operators to be adjusted in a single monitoring round, thus greatly reducing the time span of scaling up process. However, Stela is initially designed for on-demand elasticity, hence some changes are required to make it comparable with our approach:

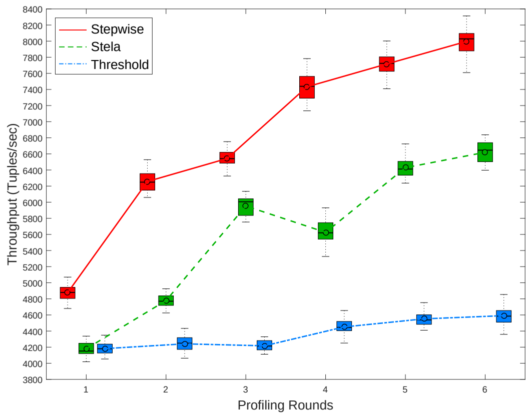
1. The scaling out process is omitted as we intend to optimise the application performance on a pre-configured cluster. All infrastructural resources are made available to Stela from the beginning of the scaling up process.

2. A single monitoring round of Stela corresponds to an on-demand scaling request in its original form, which may involve multiple scaling up iterations. During each iteration, Stela calculates the ETP for all operators and assigns a new task to the operator with the highest ETP. Before proceeding to the next iteration, the table of ETP is updated with projected values that estimate the consequence of scaling, such as the projected input rate and the processing rate of the targeted operator.

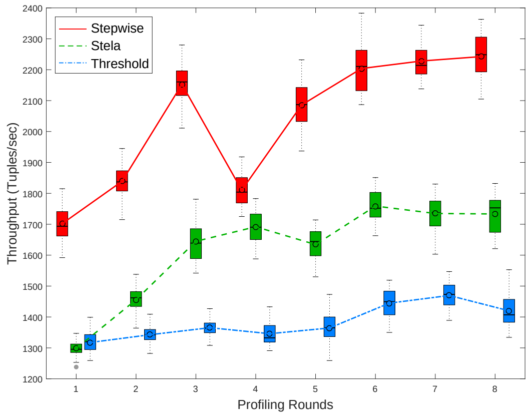
3. Since the estimation of ETP is prone to error propagation, we limit the maximum number of scaling up attempts in a monitoring round to m, which is the number of worker nodes available at the infrastructure level. In this way, the efficacy of the scaling algorithm is assured as the table of ETP is revised with monitored data every m iterations; and the risk of over-scaling is controlled since each machine will be assigned with no more than one new task in a single monitoring round.



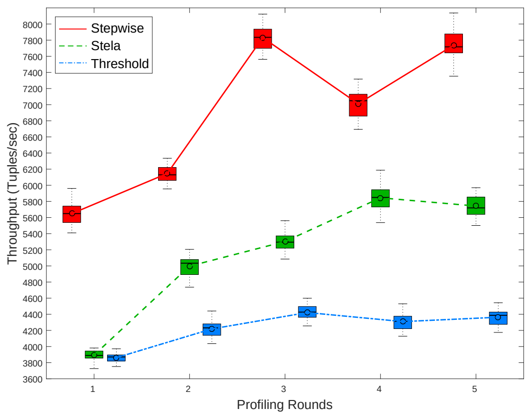
(a) Linear Topology (Sythetic, CPU-bound)



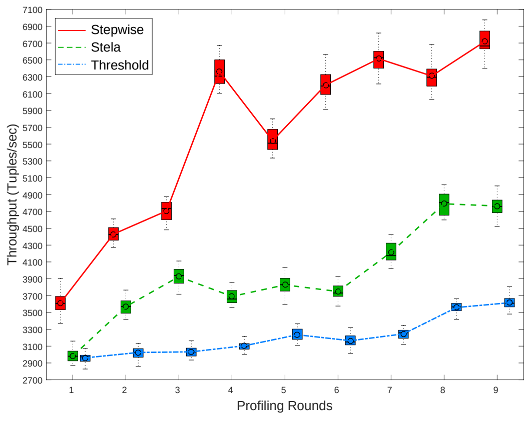
(b) Star Topology (Sythetic, I/O-bound)



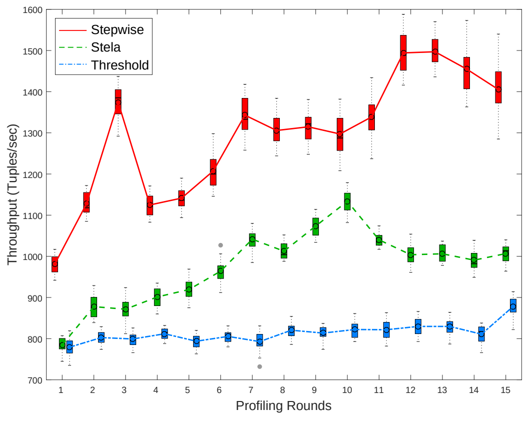
(c) Diamond Topology (Sythetic, Hybrid)



(d) Word Count (WC)



(e) Sythetic Word Count (SWC)



(f) Twitter Sentiment Analysis (TSA)

Figure 3.11: Scaling up testing applications on 6 processing nodes, the X axis represents a series of profiling rounds and the Y axis compares the throughput resulting from different configurations. In each profiling round, we use 3 boxplots that each contains 20 readings of throughput to denote the observation variances.

3.6.2 Applicability Evaluation

In the applicability experiment, all the topologies are executed in 6 worker nodes. We configured the micro-benchmark topologies with different resource complexities in order to examine how application diversity affects the performance optimisation process. Specifically, the Linear topology incorporates only CPU-bound operators so that the whole application is bound by available CPU resources; while the Star topology consists of only I/O bound operators, causing its performance to be bound by communication capability[18]. The Diamond topology, on the other hand, is a hybrid streaming application that includes all sorts of operators (1 CPU bound, 1 I/O bound, and 2 Sojourn time-bound) in the intermediate tier, making its bottleneck more difficult to identify and resolve in the scaling up process.

The results in Figure 3.11 show that the stepwise profiler successfully scales up the targeted topologies. In particular, the Linear topology reaches its maximum throughput at 1876 with the parallelism set as $(1, 2, 2, 2)$[19], which is 95.7% higher than its initial throughput performance yielded by $(1, 1, 1, 1)$. It took 4 rounds for the scaling up process to converge: the stepwise profiler tried the configuration of $(1, 3, 3, 3)$ at round 3, but it then rejected such configuration change due to the observed performance degradation. Note that the operator capacity profiling is entirely omitted in this scaling up process, as the measured operator latencies have all fallen into the vicinity of the monitored $MinL_p$ by a factor of 2.

Being I/O intensive in nature, the Star topology requires much higher parallelism settings to enable satisfactory performance, which consequently leads to a longer scaling up process. In our evaluation, the scaling up process took 6 rounds to finish, with the parallelism finally set as $(3, 3, 48, 24, 24)$ delivering 64% higher throughput than the first round. Thank to the homogeneity of operator implementation, there is no need to fine tune the operator capacities as the stopping condition on latency has been met.

The Diamond topology, in contrast, spent 3 rounds in the third step to further scale up the I/O bound operator (Op_3). During the process of platform capability profiling, stepwise profiler successfully determines the right parallelism for CPU-bound and So-

[17]By default, Apache Storm initialises each operator in the topology with one task for execution.
[18]For I/O bound topologies (e.g. Star), we set Υ_{con} to 100 ms to reflect stricter timeliness requirement.
[19]From left to right, each number corresponds to the number of tasks of each operator in the Linear Topology.

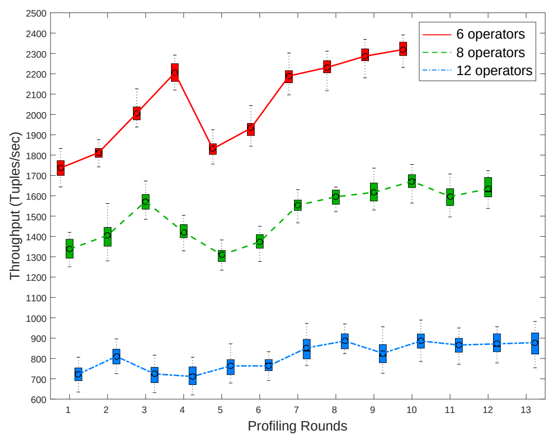
journ time-bound operators; however, it underestimates the number of tasks for Op_3 and causes it to be the throughput bottleneck. The reason of insufficient scaling is that Equation 3.1 made a conservative decimal conversion by using *slice* of 0.3, which prevents Op_3 from scaling more than 4 times faster than the other operators. We will shed more light on the effect of parameter selection in Section 3.6.4.

In addition, by interpreting the scaling up process of real-world streaming applications, we conclude that our method is consistently better than the other two scaling approaches in the following three aspects. Firstly, stepwise profiling exploits the inherent feature of a streaming application and thus has a more reasonable starting point of profiling comparing to the other two baseline methods, which by contrast determine the initial configuration only based on the topology structure. Figure 3.11 illustrates that the application feature profiling for WC, SWC, and TSA improves the performance by 45%, 21.1% and 25% at the beginning, respectively.

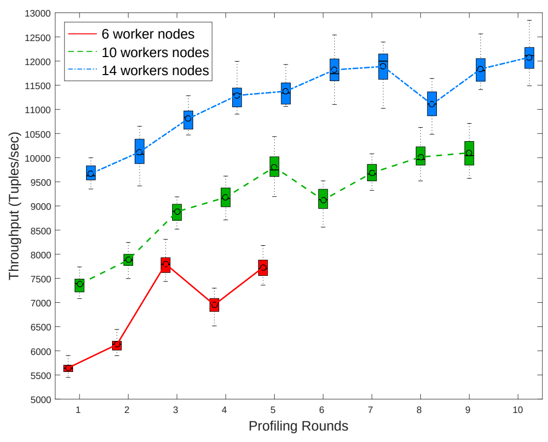
Secondly, as platform capability profiling collectively adjusts the parallelism hints for a set of operators, it significantly enhances the performance gains obtained from the first few profiling rounds. On average, the relative performance improvement observed from the first four rounds in our method is 2.48 times as large as that of Stela, and 11.63 times compared to that of the threshold-based method. Besides, despite having the ability to tune multiple parallelism hints in a single round, Stela's estimation-based algorithm could lead to incorrect scaling decision, e.g. it added new tasks to logic-dependent operators and caused performance degradation at round 10 in Figure 3.11f. To make things worse, there is no reversal mechanism to rollback the wrong move.

Finally, the stopping condition introduced in Section 3.4.3 greatly limited the number of profiling rounds. Specifically, stepwise profiling stops trying new configuration in TSA because there are successive revocations that show increasing parallelism hint no longer benefits the performance. In WC and SWC, the profiler execution terminates when the latencies for each bolt dropped into a range of $(0, 2{*}MinL_p]$, which indicates that the application has been sufficiently scaled up. In the end, our approach is 34.1%, 40.1%, 31.9% better than the best alternative in terms of the throughput resulted from the final configuration, respectively.

With the performance information profiled, the quality of different topology imple-



(a) Linear Topology (Sythetic, Hybrid)



(b) Word Count (WC)

Figure 3.12: Scalability evaluation of the stepwise profiler. The X axis represents a series of profiling rounds and the Y axis compares the throughput resulting from different configurations.

mentations in terms of their performance potentials can be easily observed. In this case, SWC is consistently worse than WC as the former implementation only reaches 86.8% throughput of the latter and it takes more effort (9 rounds vs 5 rounds) to probe a reasonable configuration.

3.6.3 Scalability Evaluation

We explore the scalability of our stepwise profiling prototype in two dimensions. The first dimension is topology complexity, which examines how the increasing number of operators in the topology affects the scaling up process. The other dimension is platform size, which checks if the prototype is able to deliver a reasonably higher post-scaling performance using more resources. Meanwhile, we also compare stepwise profiling with Stela in terms of the minimal resources needed to reach a specific performance target.

In the first experiment, we run the Linear topology with various types of operators on 6 worker nodes. The topology depth is further extended to 6, 8 and 12 in order to construct a more complex structure. Results in Figure 3.12a show that increasing the topology chain leads to a longer operator capacity profiling process, but the overall profiling effort does not scale linearly with the number of operators. This is because the monitored $MinL_p$ also increases along with the topology complexity and contributes to the timely termination of the profiling process. In fact, we observed that the stopping condition on latency is satisfied by most operators at the end of the platform capability profiling, and only I/O bound operators demand further adjustment of capacity as their $MinL_p$ are relatively small and hard to approach. This observation enables the conclusion that the parameter selection process is application-dependent and a higher k should be set for I/O bound topologies.

Additionally, this experiment showcases that the increasing complexity of target application compromises the performance gain from profiling, with the monitored improvement being 33.5%, 24%, 20.5% in the three test cases, respectively. Therefore, a larger platform is required for complex streaming topologies to obtain satisfactory performance.

In the second experiment, we run the Word Count topology on 6, 10 and 14 worker nodes, respectively. Note that by using 14 worker nodes we can still guarantee that one virtual CPU corresponds to a physical core so as to avoid the interference of CPU overbooking. The results in Figure 3.12b demonstrate that the application features profiled in the first step, i.e. $MinL_p$ and RSS, are nicely maintained on larger platforms, therefore the process of platform capability profiling is accordingly extended to provide higher parallelism for different operators. However, the stepwise profiler is not able to achieve linear growth of performance using more resources. This is because other factors, such as task location and concurrency settings, also influence the throughput outcome, but they are not fine-tuned by the profiler due to the hardness of modelling.

Using WC as the test topology, we also applied Stela and stepwise profiling on an increasing number of nodes, from 2 to 14, to determine the performance limits of the topology given different resources. Both methods were executed with the same iterations to ensure fairness, and the results of scaling shed light on the minimal resource provision needed for the test application to reach a specific throughput target.

Figure 3.13 shows that the stepwise profiling is able to reduce resource usage by up to 57.1%. For example, stepwise profiling can achieve a target of 6000 tweets processed per second using only 4 nodes. On the other hand, Stela suggests 8 nodes to process such stream without breaking SLA, which results in a significant resource wastage.

Additionally, it took 200 minutes for the stepwise profiler to complete the scaling up process of WC on 16 machines (2 for control and coordination and 14 for execution).

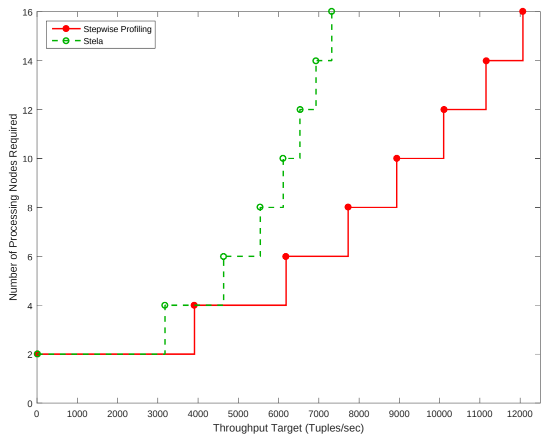


Figure 3.13: Relationship between the throughput target and required resources to handle it without latency violation.

Given that a configuration trial in each round runs for 20 minutes, and that 10 configurations are evaluated, the overhead incurred by the profiling algorithm in the whole process is negligible.

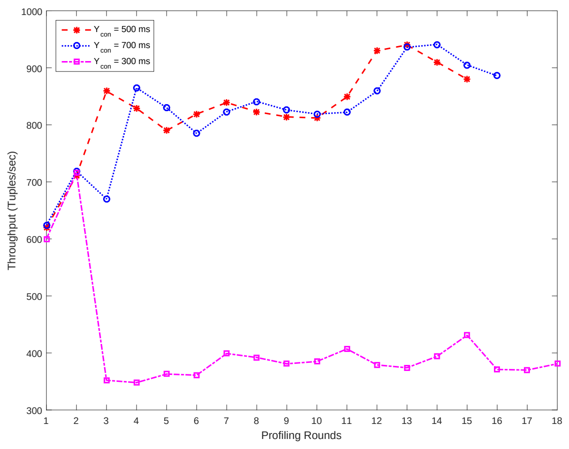
3.6.4 System Parameters Evaluation

In this experiment, we evaluate the influences of three parameters in the performance of stepwise profiler. These include the user-specified latency constraint Y_{con}, the task load unit *slice* and the stopping coefficient k. In particular, TSA is executed on 4 processing nodes. When a particular parameter is being examined, the others were set to their default values. Table 3.3 describes the evaluated and default values for each parameter.

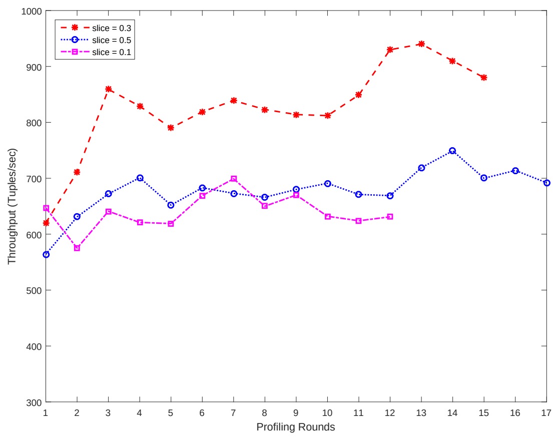
Table 3.3: Evaluated parameters and their values. Default values are showed in bold

Parameters	**Values**
Latency constraint (Y_{con})	300 ms, **500 ms**, 700 ms
Task load unit (*slice*)	0.1, **0.3**, 0.5
Stopping coefficient (k)	1.5, **2**, 3

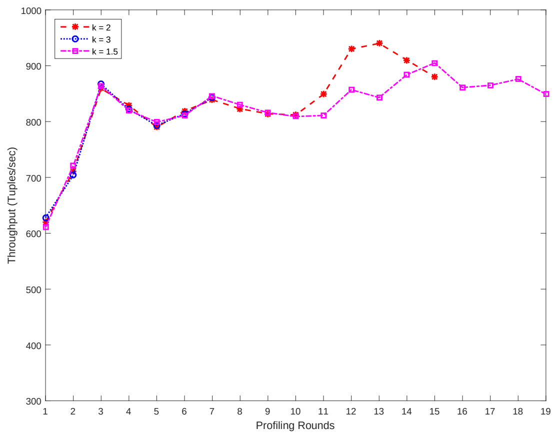
Results show that relaxing Y_{con} does not necessarily increase the throughput. In fact, it encourages the profiler to try further data source scaling operations in the second profiling step, checking if the bottleneck lies in insufficient data supply. As shown in Figure 3.14a, the third data point marked with a circle denotes the operation that scales up the data source, but it is revoked because the overall throughput is impaired by this change.



(a) Varying Y_{con}



(b) Varying *slice*



(c) Varying k

Figure 3.14: Influence of different parameters on the performance of stepwise profiling. For better readability, we only plot the average of throughput in each profiling round.

On the other hand, the throughput would be significantly affected if Y_{con} were set to an overly small value. As indicated by the third data point marked with a square, our method has to throttle the data source at the end of the second profiling step to meet the latency requirement, and the following rounds in the third step do not compensate the performance degradation due to the strict latency constraint.

The behaviour of the platform capability profiling mainly depends on the value of $slice$. When $slice$ increases from 0.3 to 0.5, both the starting point and the performance gain from scaling are worse than the default case, reaching only 90.9% and 79.7% of the default case performance, respectively. This is because, in this case, $\overrightarrow{R}$ is no longer able to describe the proportion of different operators, which in turn causes heavy bottlenecks in bolts of the whole topology. By contrast, a value of $slice$ that is too small exaggerates this proportion and makes each scaling attempt more extreme. Data points marked with squares in Figure 3.14b show that, even though the profiler managed to improve the performance of the starting point against the normal case, the following scaling trials in the second step all failed because of over-scaling — too many tasks were being added each time.

Variation of the parameter k mainly affects the number of rounds in the operator capacity profiling. Figure 3.14c shows that, when k changes from 2 to 3, the whole third step of profiling is omitted at round 7 because each operator satisfies the stopping condition, though only a suboptimal configuration is obtained in this case. On the contrary, when k is decreased to 1.5, more operators are involved in the third step, which causes a longer series of performance fluctuation. Note that more rounds of profiling in the third step do not guarantee a better throughput due to the nature of greedy heuristics.

3.7 Related Work

In summary, our work introduces a controlled profiling environment allowing evaluation of different configurations, with the objective of finding and employing a tailored deployment plan to capture relevant characteristics of both the application and target execution platform. Since our research goal is to achieve performance oriented deployment for applications on operator-based DSMS, and the adopted method falls into the broad

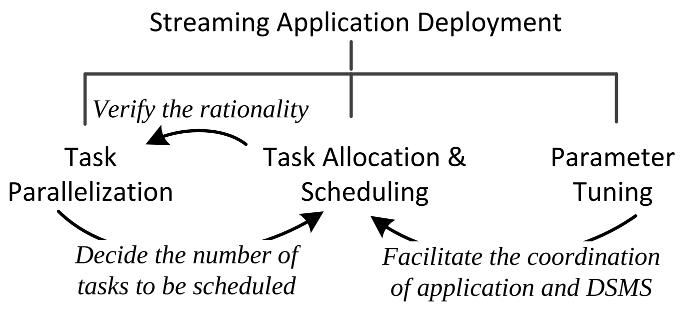

Streaming Application Deployment
Verify the rationality
Task Parallelization
Task Allocation & Scheduling
Parameter Tuning
Decide the number of tasks to be scheduled
Facilitate the coordination of application and DSMS

Figure 3.15: Three processes of deploying a streaming application on the operator-based DSMS running in a cloud and cluster environment. Text in italic describes the interrelation between them.

scope of application profiling, this section reports relevant works in these two fields—performance oriented deployment and application profiling. There is also a line of work applicable to the previous generation of DSMS that focused on tuning performance without changing the semantic of a streaming application. Therefore, we succinctly review them and summarise how performance optimisation is achieved on other types of DSMS.

3.7.1 Performance Oriented Deployment

As shown in Figure 3.15, there are three tightly coupled processes involved in streaming application deployment once the target cluster or cloud environment has been provisioned: (i) *task parallelisation*, which involves the decision of the parallelism degree for the logic DAG, such that each abstract operator is translated into a certain number of tasks to conduct real data operations; (ii) *task allocation & scheduling*, which involves allocation and scheduling of tasks among participating compute nodes; and (iii) *parameter tuning*, which concerns fine-grained adjustment of available parameters for better coordination of the application and the platform.

Only a handful of works investigated the task parallelisation problem. Researchers in IBM [58,140,141] tried to automate this process using a compiler and runtime system that is capable of identifying and levering potential data-parallel regions for applications on System S [84]. But instead of altering the parallelism to improve the application performance, their work mainly focused on addressing the safety challenge related to parallelisation, which has already been handled by the implementation of state-of-the-art DSMSs. Fischer et al., who abstract the streaming application as a black box with an unknown performance function, proposed another similar work that regards the task parallelisation as only a part of parameter tuning [51]. Though the adopted Bayesian optimisation method has demonstrated its effectiveness through extensive evaluation, it would lead to an inherent lengthy convergence process compared to our stepwise profiling approach in which operators are heuristically parallelised with insights obtained from the queuing model.

Elasticity in DSMSs has received increasing research attention as it enables cost-efficient handling of workload variations. Some works explored dynamically scale out/in streaming applications through the adjustment of parallelism settings as well as tuning relevant parameters. DRS is a resource scheduler that dynamically decides the parallelism hint for each operator based on queueing theory, with the goal of minimising the total sojourn time of an average input [54]. However, it targets only computation-intensive applications. Lohrmann et al. [108] continuously rebalance the topology with new configurations according to a proposed latency model, and they double the parallelism of any operator found to be a bottleneck. Nevertheless, the proposed bottleneck resolving method is coarse-grained and may lead to resources wastage. Heinze et al. compared three different scaling techniques in terms of the quality of the produced scaling decisions, and the results demonstrated that reinforcement learning is more adaptive and robust than the threshold-based alternatives [69]. Hidalgo et al. combined the threshold-based method with the Markov chain model to dynamically change the operator parallelism, so that the short-term and mid-term workload variations can be handled with reactive and predictive approaches, respectively [72]. Besides, realising elasticity for stateful operators requires non-trivial efforts to handle issues such as stream rerouting and state migration. While the adopted pause-and-resume strategy is commonly seen in the literature [23,24,111], there are also advanced protocols for operator movement and state management that allow for interruption-free elasticity [39,130,171]. In future work, these techniques can be integrated in our prototype to improve its responsiveness against workload burst. As for parameter tuning, Das et al. [36] proposed a control algorithm to automatically determine the most suitable batch for a given state, while online parameter optimisation has been investigated by Heinze et al. to deal with the situation where the application needs to be dynamically scaled as a reaction to workload changes [70].

Our target is different to all those above as we try to determine the configuration for any streaming application given the platform that maximises the throughput under latency constraint.

In contrast, the task allocation and scheduling problem has received much more attention from the research community. Aniello et al. pioneered this area with two scheduling algorithms on Apache Storm: the off-line version makes all the scheduling decisions through a static analysis of the logic DAG, while the on-line version regularly collects runtime information to sort all the communicating pairs of tasks, with an attempt to sequentially co-locate them in the same node to reduce communication cost [5]. Inspired by this idea, many works extended the on-line algorithm by adding some other aspects into consideration, such as scheduling overhead, resource awareness and energy efficiency. Chatzistergiou et al. proposed a linear time task allocation algorithm to adaptively reconfigure task locations in the presence of environmental changes, resulting in a significant improvement from the existing quadratic time solutions [29]. Fischer et al. presented an application agnostic algorithm that supports scheduling of large-scale task graphs with regard to the communication pattern, the problem of minimising inter-node messages is thus translated into a graph partitioning problem which can be solved by the use of METIS algorithm [50].

On the other hand, there are also some papers that explored the area of resource-aware scheduling and put an emphasis on worker node consolidation. The algorithm used in T-Storm [173] tries to minimise both inter-node and inter-process traffic while avoiding overloading the dwindled worker nodes. Similarly, Peng et al. [124] considered the task scheduling as a variation of the Knapsack problem with several hard/soft resource constraints, so that it can be solved by the application of linear programming given that the user has provided the resource demand and availability information. Apart from the common target of reducing communication cost, Re-Stream, an energy-efficient resource scheduling mechanism by Sun et al., proposed the minimisation of energy consumption as long as the response latency meets SLA requirements. This is achieved by an analytic model that depicts the relationship among energy consumption, response time, and the resource utilisation [152, 156]. Our work can be used along with these methods above as none of them addresses the issue of task parallelisation.

Besides cluster and cloud environments that are the target of our approach, the problem of deploying streaming applications on multi-core systems and distributed networks has also been discussed by several works. Hormati et al. [75] proposed a framework that dynamically adapts applications to the changing characteristics of the multi-core resources in order to maximise the throughput, using a hybrid approach of static compilation and dynamic configurations adjustments. Similarly, Suleman et al. [151] introduced a framework to tune the parallelism for each stage in a processing pipeline using a hill-climbing algorithm that can both save time and reduce the number of used cores. As for network deployment, Cardellini et al. [18] extended Apache Storm with a self-adaptive distributed scheduling mechanism, which allows execution of streaming applications on a geographically distributed environment with a certain level of QoS guarantee.

3.7.2 Application Profiling

Application profiling is a technique that actively extracts and evaluates the characteristics of applications, for example, the space or time complexity, to facilitate the use of computing resources. The profiled data sets can be either low-level usage traces of CPU, memory, and network bandwidth, or high level metrics that are part of application SLA, such as throughput, latency and fault-tolerance ability [169]. In order to make sure that the application profile would accurately reflect resource needs, the profiling process is normally conducted in a dedicated profiling environment following the MAPE-K autonomic loop (Monitor, Analyze, Plan, Execute - Knowledge) [90], which enables controllable organisation of input data and eliminates variation factors that would affect the result collection and analysis.

Most of the state-of-the-art programming IDEs, such as Microsoft Visual Studio and Eclipse, provide tools to aid in determining bottlenecks in the code that affect the overall performance of a program. However, the research community has gone way beyond code-level performance profiling. Urgaonkar et al. [161] investigated the overbooking problem by the use of application profiling, which helps to deliver an accurate estimate of resource needs for application components co-located on shared hosts. Do et al. [43] achieved better virtual machine placement with a performance prediction model derived from the application profile. To obtain higher profiling accuracy, they identify background load, which is the interference of other applications into consideration. Shen et al. [146] used profiling to automate the detection of performance bottleneck for web applications with a large set of input parameters. Similar to our work, the proposed profiling method is able to heuristically search the best configuration that maximises the objective performance function. Qian et al. [127] developed a tool that profiles the cross-layer interaction within mobile applications, aiming to better reveal the performance and energy bottlenecks hidden in the inefficient resource usages. Still, our work is different to them in that we adopt the profiling method to guide the deployment process of streaming application, while the above-mentioned models mostly target batch-oriented (MapReduce) or interactive-oriented (web and mobile) applications and thus cannot be directly applied in streaming applications.

It is also worth mentioning that we have carefully designed the stepwise profiler to avoid DSMS lock-in. Besides Apache Storm, there are many operator-based DSMSs that support general purpose stream processing, including Microsoft TimeStream [128], Apache Samza, Apache Flink [109], and Twitter Heron [96]. None of them has a built-in feature to automatically decide the parallel configuration for a particular application, and thus all of them can benefit from the proposed profiler.

3.7.3 Other Performance Optimisation Techniques

It has been more than a decade since the first generation DSMSs, including Aurora [2], Niagara [30] and Telegraph [28], were introduced to facilitate the development and deployment of streaming applications. Along with the increasing adoption of DSMS, various optimisation techniques have been developed to improve the performance of applications without changing their topology or semantics.

Operator placement optimisation, for example, is a process of assigning operators to specific hosts and cores to reach a trade-off between communication cost and resource contention. Though it is still a kind of adjustment in application layout rather than spreading and scheduling tasks (as discussed in Section 3.7.1), operator placement in previous generation DSMS regards each operator as an indivisible entity that can only appear in one place at a time. In this context, Gordon et al. designed a software-programmable substrate capable of generating custom communication code to reduce message hops when placing operator on multi-core systems [62]. Auerbach et al. proposed a placement mechanism to guarantee that operators compiled for an FPGA will always be placed on hosts with FPGAs [6]. In addition to resource matching, Wolf et al. [170] considered other constraints during the placement process, such as licensing and security requirement.

Load balancing is another commonly used optimisation technique to evenly distribute workload across available resources. This requires either a balanced operator placement plan or a runtime mechanism to dynamically assign stream tuples to operators. As examples of these two approaches, Xing et al. migrated conflicting operators that experience load spikes at the same time to separate locations to avoid resource contention and thus improving load balance [176], while Amini et al. [4] discussed the use of back-pressure in System S to compensate skews found in runtime.

However, these optimisation techniques are no longer applicable to state-of-the-art streaming applications built on top of operator-based DSMS, as the implementation of DSMS has greatly evolved towards scalability and robustness, causing operator placement and load balancing to rely heavily on the parallelisation and scheduling of tasks that constitute the operator.

3.8 Summary

In this chapter, we proposed a streaming application profiler that consists of three steps, namely (i) application feature profiling, which aims to identify the complexity and task load for each operator; (ii) platform capability profiling, which endeavours to scale up the application with the knowledge learned from the previous step; and (iii) operator capacity profiling, which makes necessary amendments on fine-grained operator level to further improve performance of the application. Our profiler can be used to scale up the streaming application, build the relationship between the underlying resources and the performance metrics, and further evaluate the choice of resource provisioning. An evaluation of a profiler prototype applied to three real world applications showed that our approach is able to automatically improve the throughput up to 40.1% compared to Stela, a state-of-the-art alternative scaling approach.

However, this chapter employed the static round-robin method for task placement and scheduling, which could lead to over- and under- utilisation at runtime as the volume of workload fluctuates. It could also result in an immense amount of inter-node communication with significant network overhead as the scheduling is not communication-aware. In the next chapter, we propose a dynamic resource-efficient scheduler that automatically matches the resource requirements of task execution with the resource availability of the infrastructure, achieving better performance and cost efficiency through a bin-packing formation of compact task placement.

Chapter 4
Dynamic Resource-Efficient Scheduling in Stream Processing Systems

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 chapter, 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 that validates the efficacy and efficiency of our scheduling algorithm. The evaluation proves that D-Storm outperforms the existing schedulers 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.

4.1 Introduction

SCHEDULING of streaming applications is one of the many tasks that should be transparently handled by the Data Stream Management Systems (DSMS). 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

This chapter is derived from:

- **Xunyun Liu** and Rajkumar Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," *in Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, China, Pages: 1-8, IEEE, 2017.
- **Xunyun Liu** and Rajkumar Buyya, "Dynamic Resource-Efficient Scheduling in Data Stream Management Systems Deployed on Computing Clouds," *ACM Transactions on Internet Technology (TOIT)*, ACM Press, 2017 (Under review).

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 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 [5, 29, 32, 50, 99, 153, 173]. 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

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

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. [124] 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 chapter, 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.

The main **contributions** reported in this chapter 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 chapter is organised as follows: we introduce Apache Storm as a background system in Section 4.2 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 4.3 and 4.4. The performance evaluation is presented in Section 4.5, followed by the related work and conclusions in Sections 4.6 and 4.7, respectively.

4.2 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

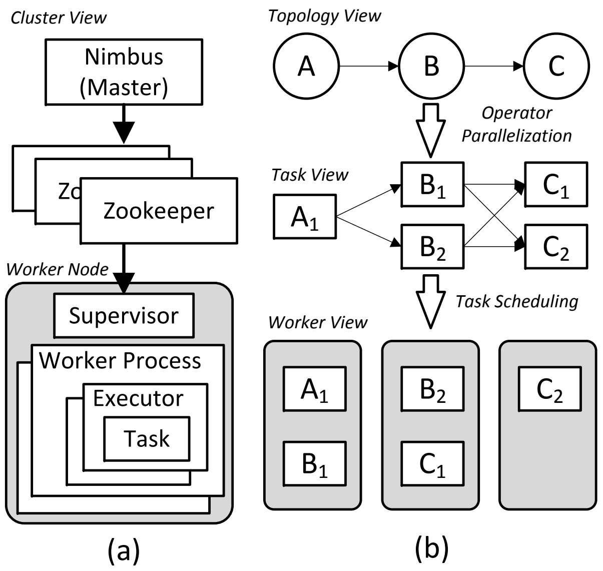


Figure 4.1: The structural view and logical view of a Storm cluster, in which the process of task scheduling is illustrated with a three-operator application.

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. 4.1a 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 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. 4.1b 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. 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 4.1, such a simple scheduling policy may lead to over/under resource utilisation.

On the other hand, the scheduler proposed by Peng et al. [124] 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 4.5, we conduct a thorough comparison of our

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

approach and Peng et al.'s work with performance evaluation in a real environment.

4.3 Dynamic Resource-Aware Scheduling

The dynamic resource-aware scheduling in D-Storm exhibits the following characteristics: (1) each task has a set of resource requirements that are constantly changing 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.

4.3.1 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 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 4.1.

Table 4.1: 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 [124], 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 chapter, 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.

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. (4.1).

$$
\begin{aligned}
\omega_c^{\tau_i} &= \Big(\sum_{\tau_j \in \Theta_{\tau_i}} \xi_{\tau_j,\tau_i}\Big) * \rho_c^{\tau_i} \\
\omega_m^{\tau_i} &= \Big(\sum_{\tau_j \in \Theta_{\tau_i}} \xi_{\tau_j,\tau_i}\Big) * \rho_m^{\tau_i}
\end{aligned}
\tag{4.1}
$$

Note that i and j in Eq. (4.1) are just two generic subscripts that represent certain values within a range defined in Table 4.1. 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 $W_c^{\nu_i}$

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

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:

$$\begin{aligned}
\text{minimise} \quad & \varkappa(\boldsymbol{\xi}, \boldsymbol{x}) = \sum_{i,j \in \{1,..,n\}} \xi_{\tau_i,\tau_j}(1 - \sum_{k \in \{1,..,m\}} x_{i,k} * x_{j,k}) \\
\text{subject to} \quad & \sum_{k=1}^{m} x_{i,k} = 1, \qquad i = 1, ..., n, \\
& \sum_{i=1}^{n} \omega_c^{\tau_i} x_{i,k} \leq W_c^{\nu_k} \quad k = 1, ..., m, \\
& \sum_{i=1}^{n} \omega_m^{\tau_i} x_{i,k} \leq W_m^{\nu_k} \quad k = 1, ..., m,
\end{aligned} \tag{4.2}$$

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. 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. (4.2): (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. (4.3) shows that $\boldsymbol{x}$ can be used to reason the number of used worker nodes as the result of scheduling:

$$\begin{aligned}
V_{\text{used}} &= \{\nu_j \mid \sum_{i \in \{1,...,n\}} x_{i,j} > 0, j \in \{1, ..., m\}\} \\
m_{\text{used}} &= |V_{\text{used}}|
\end{aligned} \tag{4.3}$$

Algorithm 4.1: The multidimensional FFD heuristic scheduling algorithm

```
Input: A task set $\vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be assigned
Output: A machine set $\vec{\nu} = \{\nu_1, \nu_2, \ldots, \nu_{m_{\text{used}}}\}$ with each machine hosting a
        disjoint subset of $\vec{\tau}$, where $m_{\text{used}}$ is the number of used machines
1  Sort available nodes in descending order by their resource availability as defined
     in Eq. (4.4)
2  $m_{\text{used}} \leftarrow 0$
3  while there are tasks remaining in $\vec{\tau}$ to be placed do
4      Start a new machine $\nu_m$ from the sorted list;
5      if there are no avaiable nodes then
6          return Failure
7      Increase $m_{\text{used}}$ by 1
8      while there are tasks that fit into machine $\nu_m$ do
9          foreach $\tau \in \vec{\tau}$ do
10             Calculate $\varrho(\tau_i, \nu_m)$ according to Eq. (4.5)
11         Sort all viable tasks based on their priority
12         Place the task with the highest $\varrho(\tau_i, \nu_m)$ into machine $\nu_m$
13         Remove the assigned task from $\vec{\tau}$
14         Update the remaining capacity of machine $\nu_m$
15 return $\vec{\nu}$
```

4.3.2 Heuristic-based Scheduling Algorithm

The classical bin-packing problem has proved to be NP-Hard [33], and so does the scheduling of streaming applications [124]. 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 [158], 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 [55] and branch-and-price [40], 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 4.1.

Firstly, all the available machines are arranged in descending order by their resource availability so that the more powerful ones get utilised first for task placement. This step is to ensure that the FFD heuristic has a better chance to convey more task communications within the same machine, thus reducing the cumbersome serialisation and de-serialisation procedures 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. (4.4).

$$\wp(\nu_i) = \min\{\frac{nW_c^{\nu_i}}{\sum\limits_{j\in\{1,\ldots,n\}} \omega_c^{\tau_j}}, \frac{nW_m^{\nu_i}}{\sum\limits_{j\in\{1,\ldots,n\}} \omega_m^{\tau_j}}\} \tag{4.4}$$

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 [61], 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:

$$
\begin{aligned}
\varrho_1(\tau_i, \nu_m) &= \sum_{j \in \{1,\ldots,n\}} x_{j,\nu_m} (\xi_{\tau_i,\tau_j} + \xi_{\tau_j,\tau_i}) \\
\varrho_2(\tau_i, \nu_m) &= \sum_{j \in \Phi_{\tau_i}} (1 - \sum_{k \in \{1,\ldots,m\}} x_{j,k}) \xi_{\tau_i,\tau_j} \\
&\quad + \sum_{j \in \Theta_{\tau_i}} (1 - \sum_{k \in \{1,\ldots,m\}} x_{j,k}) \xi_{\tau_j,\tau_i} \\
\Re_{\nu_m} &= \max \{ \frac{\sum_{j \in \{1,\ldots,n\}} \omega_c^{\tau_j} x_{j,\nu_m}}{W_c^{\nu_m}}, \frac{\sum_{j \in \{1,\ldots,n\}} \omega_m^{\tau_j} x_{j,\nu_m}}{W_m^{\nu_m}} \} \\
\varrho_3(\tau_i, \nu_m) &= \Re_{\nu_m}^{x_{\tau_i,\nu_m}=1} - \Re_{\nu_m}^{x_{\tau_i,\nu_m}=0} \\
\varrho(\tau_i, \nu_m) &= \begin{cases} \dfrac{\alpha \varrho_1(\tau_i, \nu_m) + \beta \varrho_2(\tau_i, \nu_m)}{\varrho_3(\tau_i, \nu_m)} & \Re_{\nu_m} \leq D_{\text{Threshold}} \\ \dfrac{\alpha \varrho_1(\tau_i, \nu_m)}{\varrho_3(\tau_i, \nu_m)} & \text{otherwise;} \end{cases}
\end{aligned}
\tag{4.5}
$$

In Eq. (4.5), $\varrho_1(\tau_i, \nu_m)$ 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. (4.5), 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. (4.2).

4.3.3 Complexity Analysis

We analyse the work-flow of Algorithm 4.1 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 8 to Line 14 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 10 consumes linear time of n, and the sorting at Line 11 takes at most $O(nlog(n))$ time to complete. Therefore, the whole algorithm has the worst case complexity of $O(mlog(m) + n^2log(n))$.

4.4 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.

- 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,

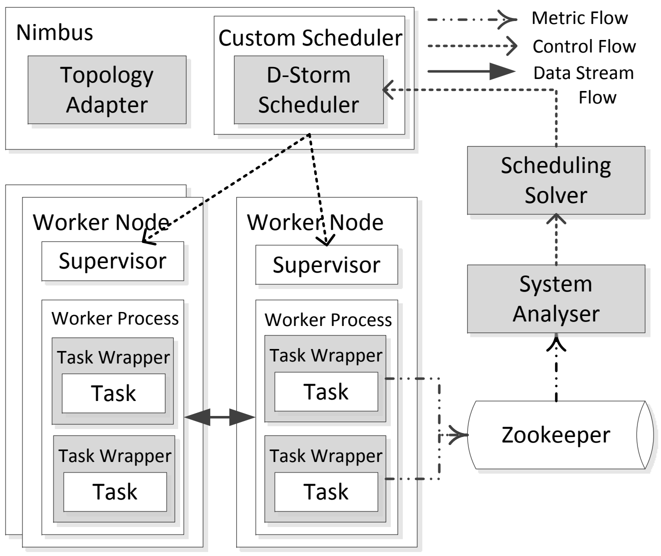


Figure 4.2: The extended D-Storm architecture on top of the standard Storm release, where the newly introduced modules are highlighted in grey.

Planning, Execution, and Knowledge) framework as shown in Fig. 4.2. This architectural concept was first introduced by IBM to design autonomic systems with self-* capabilities, such as self-managing, self-healing [25], and self-adapting [90]. 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

[5]Complete latency: the average time a tuple tree takes to be completely processed by the topology.

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 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

[6]https://collectd.org/

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 4.3. 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. 4.2, 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.

4.5 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 [124], 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 4.5.2)
- How much resource cost is incurred by D-Storm to handle various volumes of workload? (Section 4.5.3)
- How long does it take for D-Storm to schedule relatively large streaming applications? (Section 4.5.4)

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 NCI availability zone 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.

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

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

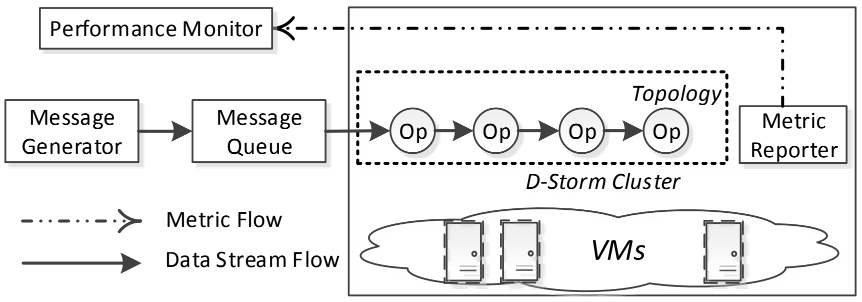


Figure 4.3: The profiling environment used for controlling the input load. The solid lines denote the generated data stream flow, and the dashed lines represent the flow of performance metrics.

stream with finer-grained control. Fig. 4.3 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.

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 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. 4.4a, 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 4.2.

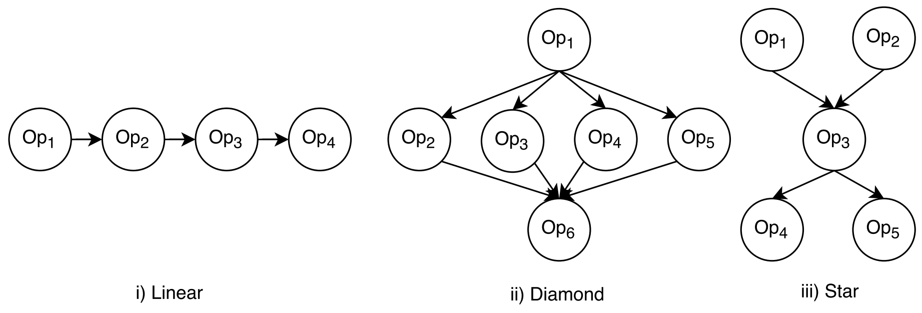
Table 4.2: The configurations of synthetic operators in the micro-benchmark

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.

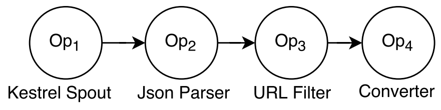
From the implementation point of view, the configuration items listed in Table 4.2 have a significant impact on the operator execution. Specifically, C_s determines how many times the method of random number generation *Math.random()* is invoked by the operator upon any tuple receipt (or tuple sending 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.

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

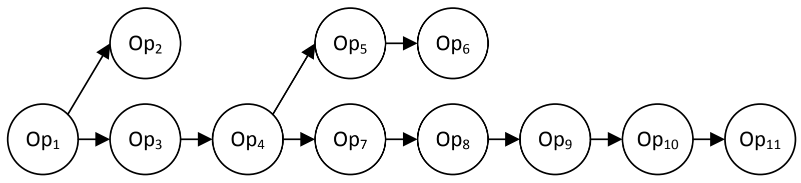
[10]Selectivity is the number of tuples emitted per tuple consumed; e.g., selectivity = 2 means the operator emits 2 tuples for every 1 consumed.



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



(b) The topology of the URL-converter application



(c) The topology of the twitter sentiment anaysis application

Figure 4.4: The topologies of test applications. Fig. 4.4a is synthetically designed while the rest two are drawn from realistic use cases.

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. 4.4b, 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. 4.4c 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 [121] for more details of this analysis process and the application implementation.

Parameter Selection and Evaluation Methodology

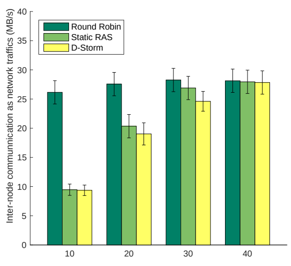
In our evaluation, the metric collection window is set to 1 minute and the scheduling interval is set to 10 minutes. These values are empirically determined for D-Storm to avoid overshooting and mitigate the fluctuation of metric observations on the selected test applications. As for the heuristic parameters, 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.

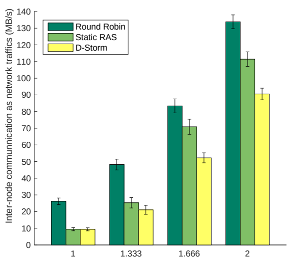
Also, the performance of the static resource-aware scheduler largely depends on the accuracy of resource profile. As the scheduler required that users submit the static resource profile at compile time [124], we conducted pilot run on test applications, probing their up-to-date resource profile and leading 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 ap-

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

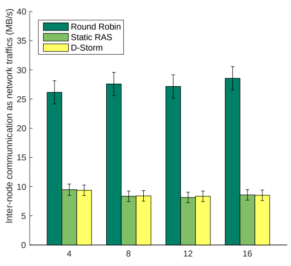
[12] https://storm.apache.org/releases/1.0.2/javadocs/org/apache/storm/metric/LoggingMetricsConsumer.html



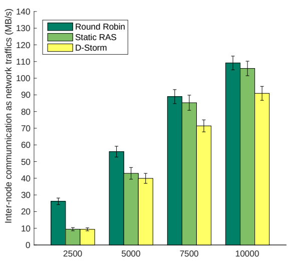
(a) Varying C_s (Synthetic Linear)



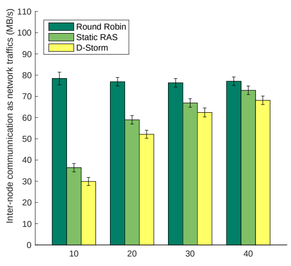
(b) Varying S_s (Synthetic Linear)



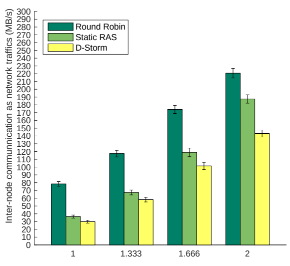
(c) Varying T_s (Synthetic Linear)



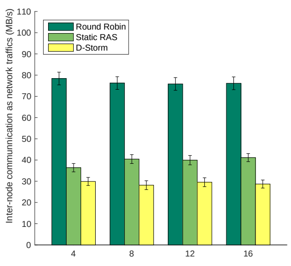
(d) Varying P_s (Synthetic Linear)



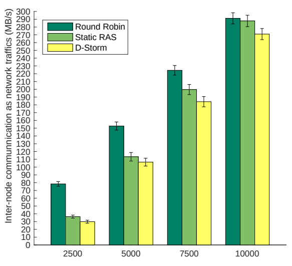
(e) Varying C_s (Synthetic Diamond)



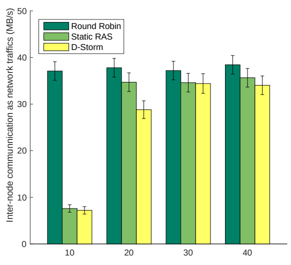
(f) Varying S_s (Synthetic Diamond)



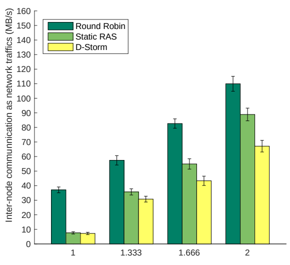
(g) Varying T_s (Synthetic Diamond)



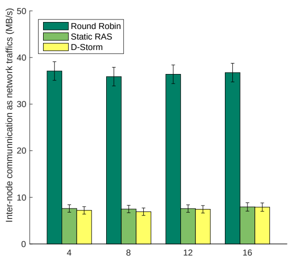
(h) Varying P_s (Synthetic Diamond)



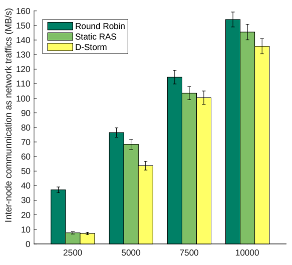
(i) Varying C_s (Synthetic Star)



(j) Varying S_s (Synthetic Star)



(k) Varying T_s (Synthetic Star)



(l) Varying P_s (Synthetic Star)

Figure 4.5: 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.

plication configuration is changed.

4.5.2 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.

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 4.3 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 4.3: 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

Fig. 4.5 presents the changes of inter-node communication while altering the micro-benchmark configurations listed in Table 4.3. 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.

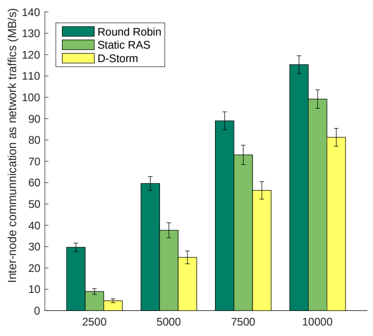
Specifically, a study of Figs. 4.5a, 4.5e and 4.5i reveals that D-Storm performs slightly better than, or at least similarly to the static Resource-Aware Scheduler (RAS) when applied to CPU-intensive use cases. In most instances, D-Storm achieves the similar communication reduction as the static RAS, 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.

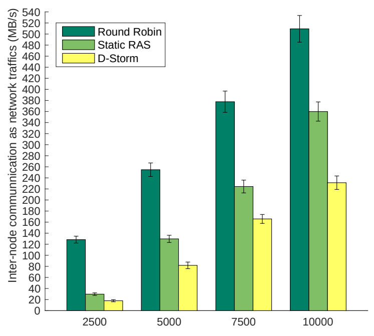
However, it is worth noting that the communication reduction brought by the static RAS is based on the correct resource profile provided by the pilot run. If this information were not specified correctly, the static resource-aware scheduler would lead to undesirable scheduling results, causing over-utilisation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

Figs. 4.5b, 4.5f and 4.5j, 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, 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. 4.5c, 4.5g and 4.5k 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. 4.5g 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



(a) Varying P_s (URL-Conveter)



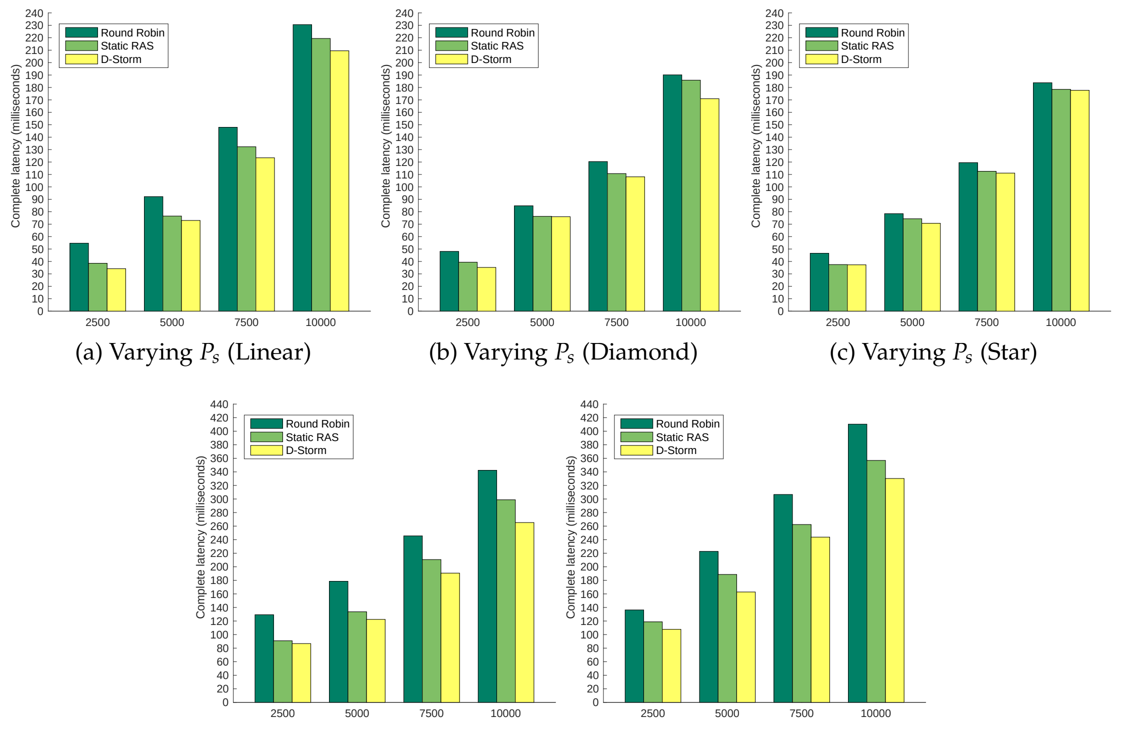
(b) Varying P_s (TSA)

Figure 4.6: 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.

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. 4.5d, 4.5h and 4.5l, while the results of the realistic applications are presented in Fig. 4.6. 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. 4.6 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.

We also collected the metric of complete latency to examine the application responsiveness while using different schedulers. As observed in Fig. 4.7, the complete latency



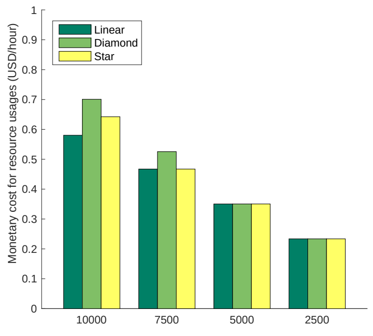
(a) Varying P_s (Linear) (b) Varying P_s (Diamond) (c) Varying P_s (Star)

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

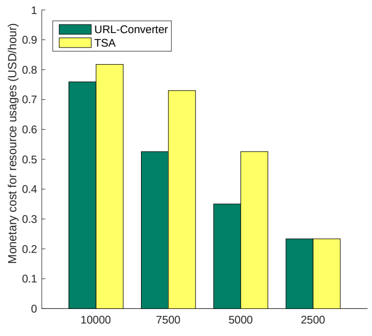
Figure 4.7: The application complete latency under different volumes of profiling streams. Each result is an average of statistics collected in a time window of 10 minutes, so the error bar is omitted as the standard deviation of latency is negligible for stabilised applications.

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. 4.7d and 4.7e, 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-



(a) Decreasing P_s (Micro-benchmark)



(b) Decreasing P_s (Realistic apps)

Figure 4.8: 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.

Converter and Twitter Sentiment Analysis, respectively. These results confirm the fact that conducting communication on network 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.

4.5.3 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 4.5.1 are included in this evaluation, in which the synthetic topologies have configured their settings to the default values. As shown in Fig. 4.8, the cost of resources used by the D-Storm scheduler steadily reduces when the input load decreases. Specifically, the diamond topology is the most resource-

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

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.

4.5.4 Evaluation of Scheduling Overhead

We also examine the time required for D-Storm to calculate a viable scheduling plan using Algorithm 4.1, 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 4.4.

Studying Table 4.4, 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

Table 4.4: 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		

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 4.3.3 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.

4.6 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 4.5.

Table 4.5: Related work comparison

Aspects	Related Works								Our
	[5]	[124]	[50]	[173]	[156]	[155]	[101]	[99]	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 [5]. 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. [50] 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 [5]. 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 [173]. 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 [156]. They also studied reliability-oriented scheduling to trade-off between competing objectives like better fault tolerance and lower response time [155]. 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 [99], 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. [21] 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. [123] 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 [124] has been introduced in Sections 4.1 and 4.2. The main limitation of their work, as well as [101, 150], is that the runtime changes to the resource consumptions and availability are not taken into consideration during the scheduling process.

4.7 Summary

In this chapter, 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.

However, our D-Storm prototype has not considered the management of operator states during the scheduling process. It also lacks the ability to deal with possible node crashes and JVM failures that impair the availability and integrity of intermediate results cached in the state. In the next chapter, we present a replication-based state management framework to support on-demand state migration among streaming tasks, which helps in maintaining the semantic correctness of stateful operations despite the runtime changes of the infrastructure.

Chapter 5
Replication-based State Management in Stream Processing Systems

The exiting checkpointing framework involves a remote data store for state preservation and access, resulting in significant overheads to the performance of error-free execution. We propose E-Storm, a replication-based state management system that actively maintains multiple state backups on different worker nodes. We build a prototype on top of Storm by extending it with monitoring and recovery modules to support inter-task state transfer whenever needed. The experiments carried out on synthetic and real-world streaming applications confirm that E-Storm outperforms the checkpointing method in terms of the resulting application performance, obtaining as much as 9.44 times throughput improvement while reducing the application latency down to 9.8%

5.1 Introduction

THERE are three fault-tolerance mechanisms built in Storm that enable reliable stream processing, namely: (1) Supervised and stateless daemon execution, which allows the failed Storm daemons to be restarted, resuming their stateless execution under the supervision of an external process monitoring tool; (2) Message delivery guarantee, which ensures the consistency of processing semantics by using a subtle anchoring and acknowledgement algorithm; and (3) State persistence, which persists the computation states to somewhere in order to mask the loss of states caused by JVM or node crashes.

This chapter is derived from:

- **Xunyun Liu**, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein and Rajkumar Buyya, "E-Storm: Replication-based State Management in Distributed Stream Processing Systems," *in Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, Bristol, UK, Pages: 1-10, IEEE, 2017.

This chapter proposes a novel state management framework to better achieve this goal.

Since version 1.0.0, Storm's core has abstractions for stream operators to save and retrieve states against a persistent state store. However, the current state persistence technique introduces significant overhead to error-free execution. From the implementation's perspective, state persistence is now achieved through checkpointing, where a remote data store is constantly involved in all state accesses. Specifically, there is an internal data source that initialises a checkpoint transaction by sending signals across the streaming application. Upon receiving the checkpoint signal, stateful operators prepare and preserve their intermediate states to a Redis[1] store, and then empty the in-memory cache to commit the transaction. The frequency of checkpointing is defaulted to every second, which brings significant state synchronization overhead; while setting the checkpoint interval too large would risk losing state between checkpointing and being unable to replay failed messages. Secondly, the use of any committed state resorts to the remote data store, which imposes non-trivial data access delay for latency-sensitive streaming applications. Lastly, there could be a massive amount of operators accessing the remote data store simultaneously for state retrieval or check-pointing, which makes the store a potential performance bottleneck to application throughput.

In this chapter, we propose E-Storm, a light-weight, replication-based state management framework in Storm that eliminates the use of remote data store during error-free execution. To ensure state persistence in the case of failures, our framework automatically maintains live state replicas on different nodes of Storm and transfers state when needed. The number of replicas can be customised with regard to the user's needs, but in general a stateful operator with k replicas is able to tolerate the failure of any $k-1$ worker nodes.

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

- We propose a replication-based state management framework for achieving state persistence in the case of failures, which exposes a concise fluent-style interface and works transparently to the upper-level logic.
- We design a failure recovery protocol that guarantees application integrity when failover occurs. The recovery operates at the lowest thread level and is seamlessly

[1] https://redis.io/

integrated to Storm's execution flow. The replication of state is also autonomous and high-performance, which allows multiple transfers to occur concurrently.

- We implement the framework and conduct extensive experiments to demonstrate the superiority of our approach compared to the existing check-pointing method, which reaches as much as 9.44 times throughput boosts and 90.2% latency reduction.

Our implementation of E-Storm is loosely coupled with the existing Storm modules, and externally configurable to provide different levels of state resilience in different use cases. Such implementation design makes it viable to be generalised to other operator-based stream processing systems.

5.2 Background

In recent years, Apache Storm emerged as a new generation of data stream management system for tackling many real-time use cases such as on-line machine learning, continuous computation and Distributed Remote Procedure Call (DRPC). The scalable, fault-tolerant, and language-agnostic design of Storm offers seamless integration with the mainstream queueing and database technologies, making it much easier to process unbounded fast data on a set of distributed resources.

From the structure point of view, a Storm cluster much resembles Hadoop — its counterpart in batch processing. It also includes a master node and several worker nodes: the master node has a *nimbus* daemon that is responsible for monitoring the cluster and distributing workload; while the worker node hosts the *worker processes* to carry out the streaming logic in JVMs. Additionally, there is a *supervisor* daemon that communicates with the nimbus and constantly governs the worker processes during runtime. The whole Storm cluster relies on *Zookeeper* — a distributed hierarchical key-value store to coordinate and failover.

In Storm's terminology, a *tuple* is an ordered list of key-value pairs (each pair is referred to as a *field*) and a *stream* is an unbounded sequence of tuples. From the logical perspective, the workflow of a streaming application is represented by the *topology* — a Directed Acyclic Graph (DAG) of *operators* standing on streams. Among operators, the sources of streams are called *spouts* that pull stream data to the topology, while the others are referred to as *bolts* that can either generate new streams based on inputs or simply consume data without emission. Different bolts may contain various user-defined processing logics such as functions, filtering, and aggregations.

From the viewpoint of execution, an operator is distributed across the Storm cluster as one or more tasks, the process of which is called *operator parallelisation*. Each task is an operator instance that handles a portion of the operator input with the same streaming logic, so Storm makes full use of distributed resources by distributing tasks to different worker nodes. Also, as a consequence of parallelisation, each incoming stream is accompanied by a *grouping policy* that determines how tuples are routed among the receipt tasks. When a streaming application is submitted to the cluster, the worker process will spawn *executors* — the minimal schedulable entity of Storm — to wrap the execution of tasks. Note that each executor is a thread that may run one or more tasks for the same component (spout or bolt), Therefore, internally tasks have to run in sequence.

Most of the streaming applications involve stateful operators that accumulate states such as window-grouped tuples or aggregation results during runtime. Therefore, it is crucial to ensure the integrity of operator states in the case of failures. From the execution point of view, the parallel execution of stateful operator requires each task to maintain a unique partition of the internal state. All the internal states are temporally stored in memory and are thus subject to JVM failure. Currently, the only way to achieve state persistence is through regularly checkpointing them to a remote Redis store. Storm has a built-in checkpoint mechanism which implements a three-phase commit protocol on top of the existing message delivery system, ensuring that the states of different tasks would be saved in a consistent and atomic manner. However, as we have explained in Section 5.1, such implementation introduces non-trivial overhead to the error-free execution of streaming applications.

5.3 Framework Overview

In order to maintain multiple state backups independently, our state management framework duplicates the execution of stateful tasks on different worker nodes. Fig. 5.1 uses

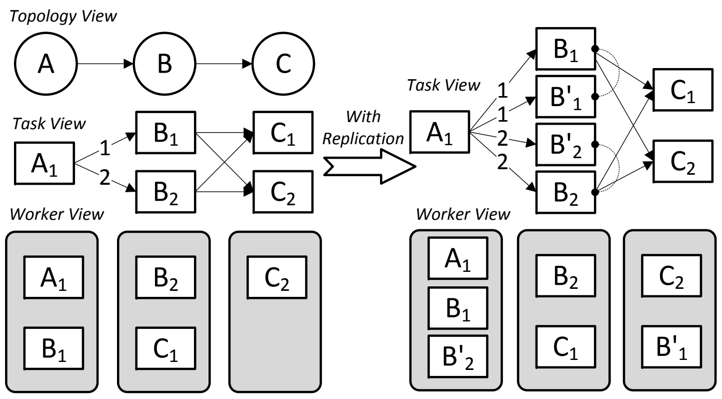


Figure 5.1: The execution of an example streaming application on Storm, with or without state replication.

an example application to illustrate the changes we have made to the Storm execution model. There are three linearly connected operators in the example application: Op_A is the spout, Op_B is a stateful operator, and Op_C is a stateless operator. Both Op_B and Op_C are parallelised into two tasks for distributed execution:

1. Having set the number of replicas for Op_B to 2, the framework spawns two *shadow tasks* $T_{B'_1}$ and $T_{B'_2}$ to mirror the execution of the *primary task* T_{B_1} and T_{B_2}, respectively. Tasks sharing the same state make up a *task fleet*, which are exclusively placed on different worker nodes for independent execution.

2. Any input stream sent to the primary task is copied to its shadow counterparts. However, shadow tasks have no output stream as they only serve as state containers.

3. In the case of failures, the restarted tasks recover their lost states from the alive partners of the same fleet.

We have extended Storm with several modules to implement these changes. These include the Topology Adapter, the State Monitor, the Task Wrapper, the Recovery Manager and the State Transit Station, which are highlighted in grey in Fig. 5.2.

The *Topology Adapter* is written in Storm core to help alleviate the adaptation effort on the application level. Developers can define the number of replicas using a fluent-style replication API, just like how they specify the number of tasks for operators. The adapter is also in charge of re-grouping streams for stateful operators and initialising

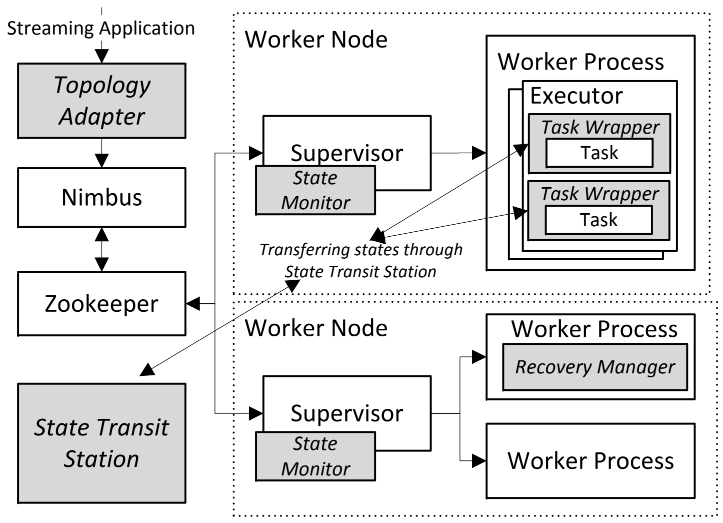


Figure 5.2: The extended Storm architecture with the state management framework, where the newly introduced modules are highlighted in grey.

other modules for state management. When the application is submitted to Nimbus, this module ensures that the shadow tasks are transparently set up across the Storm cluster.

The *State Monitor*, located alongside the supervisor daemon, is responsible for monitoring the health of states residing in this worker node. Once a state issue is detected, it will send a recovery request to the Recovery Manager through Zookeeper. The State Monitor itself is stateless and fail-fast, with execution placed under constant supervision.

The *Recovery Manager* is an internal operator that initialises, oversees and finalises the recovery process. It implements the Zookeeper watcher interface to monitor recovery requests, then exploiting the Storm's acknowledgement system to ensure the consistency of recovery. Being a stateless operator, its fault-tolerance is guaranteed by Storm to survive from node and JVM crashes.

The *Task Wrapper* encapsulates the task execution with the logic to handle state transfer and recovery. There is also the *State Transit Station* that decouples the senders and receivers during the state transferring process. By directing all the state transfers to the station, task wrappers perform state management without synchronization and leader selection, which would have been necessary in a peer-to-peer style recovery and introduce non-negligible overhead.

The state management framework has two different working modes, namely error-free execution and failure recovery. In the following sections, we discuss in detail how

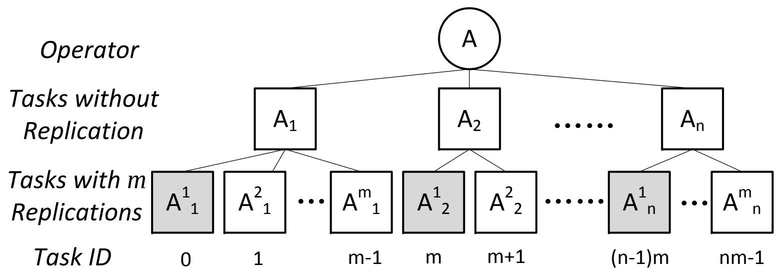


Figure 5.3: The role of a task is decided based on the index of its task ID. The primary tasks are coloured in grey

these extended modules are implemented to achieve state persistence against JVM and node crashes.

5.4 Error-free Execution

When a streaming application is submitted, the topology adapter is responsible for deciding the task roles in execution (primary or shadow). It is also in charge of rewiring the task communication for message replication and placing the tasks of the same fleet on different machines for failure-independence.

The role of a task is statically decided based on its task ID — the primary task is the one with the lowest ID in a fleet. As shown in Fig. 5.3, Op_A is a stateful operator that is initially parallelised as n tasks. After users set the number of replicas to m, the topology adapter transparently multiplies the number of tasks to n times m and composes each m tasks as a task fleet. Therefore, $T_{A_1^1}, T_{A_2^1} ... T_{A_n^1}$ are set as primary tasks and each one of them is accompanied by $m-1$ shadow tasks that are adjacent in ID. Note that the role of these tasks will not change throughout the application lifecycle.

In order to replicate states across the task fleet, the contained tasks must receive the same inputs for processing. To this end, the topology adapter replaces the original grouping that connected to the stateful bolt with a custom, replication-aware stream grouping method, which replicates the tuples in transmission transparently at the message channel.

Take the *fields grouping* — the most common grouping type in stateful computation — as an example. It routes a particular tuple $Tuple$ to its target task T_{target} according to the following equation: $T_{target} = hash(Tuple.fields)\%n$, where n is the number of tasks and *hash* is a concatenating function on the hash codes of the selected grouping fields. When the fields grouping is replaced with the replication-aware fields grouping, the message channel computes a list of m target tasks rather than a single one, which is formulated as $T_{targets} = \{hash(Tuple.fields)\%n * m + i | (i = 0, .., m-1)\}$.

5.4.1 Replication-aware Task Placement

Essentially, the task placement problem is a bin-packing variant that takes tasks as items and worker nodes as bins, while the optimization target is to reduce the number of inter-node communication pairs for improving application performance. Besides, our problem has a hard constraint that tasks from the same fleet are not to be put on the same worker node.

The task placement problem itself is NP-Hard since it can be reduced to the PARTITION problem [33]. However, it is feasible to find a sub-optimal solution by using efficient heuristic methods. We therefore propose a replication-aware task placement algorithm based on the greedy heuristic, with the following desirable features in its design:

- It is only responsible for placing shadow tasks to worker nodes; while the placement of other tasks are left for the user-given task scheduling algorithm to decide. Such a design allows for the use of various existing scheduling algorithms that optimise towards different targets, such as throughput, latency, resource-awareness, etc.

- The shadow tasks are spread as far as possible across the cluster, so the overhead of replication is balanced and the effort of state recovery is minimised in the case of failures.

- The algorithm makes use of the topology structure to place communicating tasks as close as possible.

Algorithm 5.1 depicts the pseudo-code for the replication-aware task placement. It first calculates A_{m_i}, the capacity of each node, by enforcing the shadow tasks to spread out across the cluster. Then the standard Breadth-first traversal procedure is applied to the topology structure, yielding an operator queue Q_{op} which is a partial ordering of operators with the communicating pairs placed in succession.

Algorithm 5.1: The replication-aware task placement algorithm

```
Input: A Storm cluster with $n_m$ nodes and a topology $\Gamma$
Input: A task set $\vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ to be assigned
Output: A node set $\vec{m} = \{m_1, m_2, \ldots, m_{n_m}\}$ with each node hosting a disjoint
        subset of $\vec{\tau}$
1  $A_{m_i} \leftarrow \lceil \frac{n}{n_m} \rceil \ (i = 1, 2, ...n_m)$
2  $Q_{op} \leftarrow BFSTraversal(\Gamma, spout)$
3  $\vec{\tau}_{ordered} \leftarrow \varnothing$
4  while $\vec{\tau}_{ordered}$ does not contain all the tasks in $\vec{\tau}$ do
5      foreach Operator $op \in Q_{op}$ do
6          if $op$ has an unvisited shadow task $\tau_i$ then
7              $\vec{\tau}_{ordered}.append(\tau_i)$
8              $op.remove(\tau_i)$
9  foreach Task $\tau_i \in \vec{\tau}_{ordered}$ do
10     foreach node $m_j \in \vec{m}$ that has $A_{m_j} > 0$ do
11         if $m_j$ has no conflicting tasks to $\tau_i$ then
12             $I_{m_j} \leftarrow$ the increase of the intra-node communication pairs if $\tau_i$ were put
                onto $m_j$
13     Place $\tau_i$ to node $m_j$ with the largested $I_{m_j}$
14     $A_{m_j} \leftarrow A_{m_j} - 1$
15 return $\vec{m}$
```

Lines 3-8 of the algorithm describe the procedure to generate an ordered list of tasks $\vec{\tau}_{ordered}$ based on Q_{op}. For each operator being traversed, the algorithm takes out one shadow task at a time and appends it to the ordering list. This process continues until all the tasks to be placed are ordered, which ensures that, in the later placement phase, the communicating tasks have better chance to be placed in close vicinity.

The rest of the algorithm determines the exact node where a particular task would be placed. As shown in line 13, the greedy heuristic chooses the one that is capable of turning more communications into intra-node message passing. If there is a tie between multiple alternatives, the one with the highest remaining capacity will be selected.

5.5 Failure Recovery

The recovery phase is triggered when any worker crashes during runtime. Fig. 5.4 briefly illustrates the work flow of recovery after the failure occurs.

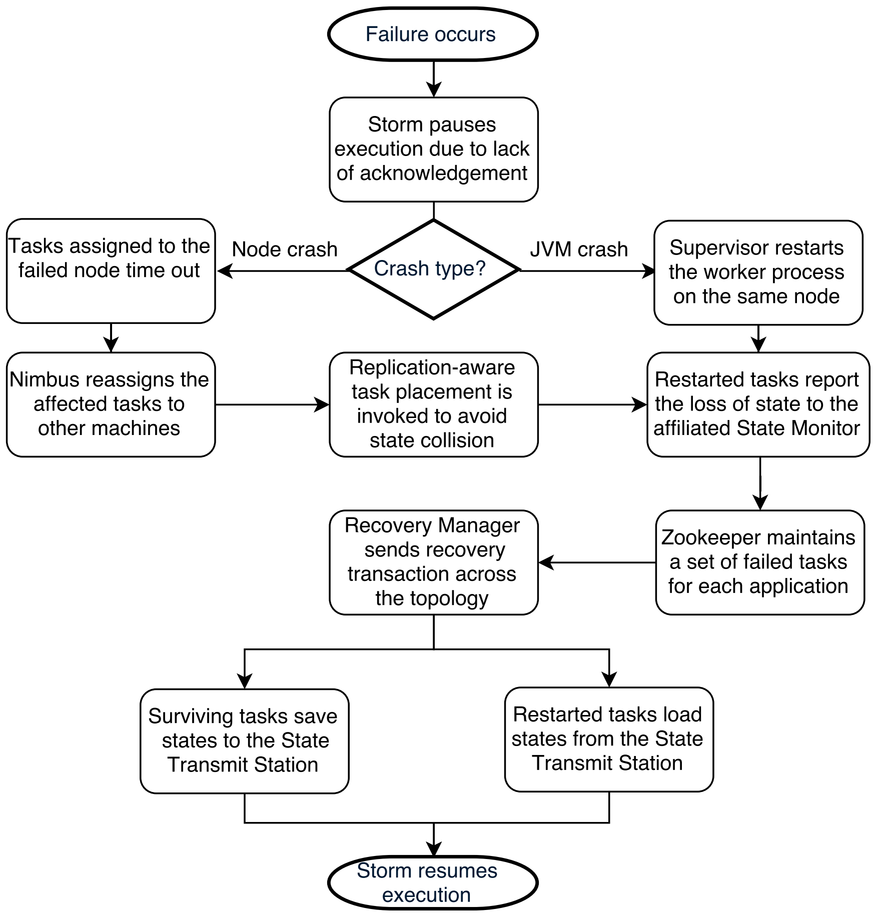

Failure occurs
Storm pauses execution due to lack of acknowledgement
Crash type?
Node crash
JVM crash
Tasks assigned to the failed node time out
Supervisor restarts the worker process on the same node
Nimbus reassigns the affected tasks to other machines
Replication-aware task placement is invoked to avoid state collision
Restarted tasks report the loss of state to the affiliated State Monitor
Zookeeper maintains a set of failed tasks for each application
Recovery Manager sends recovery transaction across the topology
Surviving tasks save states to the State Transmit Station
Restarted tasks load states from the State Transmit Station
Storm resumes execution

Figure 5.4: The flowchart of the recovery process, which is seamlessly integrated with the Storm's error-handling logic and leverages the acknowledgement system to pause and resume the execution flow.

In general, Storm automatically pauses the application execution due to the lack of tuple acknowledgement. Through the heartbeat mechanism between worker processes and Storm daemons, the failed tasks will be transparently restarted with the same task ID, but possibly placed on different worker nodes depending on the type of failure. Therefore, it is required that the replication-aware task placement is invoked to avoid two tasks from the same fleet being collocated. During the preparation process, these restarted tasks report the loss of state to the state monitor, which initialises a recovery transaction on a dedicated Zookeeper node, recording a transaction ID as well as the set of tasks being affected. The recovery manager that constantly monitors the Zookeeper would make sure that all the affected tasks get initialised with its previous states through the failure recovery process.

The recovery manager is implemented as an internal spout, which is automatically

Algorithm 5.2: The state operation logic encapsulated in the StateManipulator

```
Input: A recovery transaction $tx$ with ID $tx.id$ and the set of tasks that have lost
       their state $tx.set$
Input: An initilisation flag $isInit$ that indicates if $s$, the state of the wrapped task,
       has been initilised
Input: A CuratorFramework client $c_f$ that opeartes on the Zookeeper
Input: The task fleet $t_f$ that the wrapped task belongs to
1  if $isInit == True$ then
2      $c_f$ initilises a shared inter-process lock on $t_f$
3      if $c_f.acqureLock(t_f)$ and $s \in tx.set$ then
4          if s is not on the State Transition Station then
5              Save $s$ to the State Transition Station
6          $c_f.releaseLock(t_f)$
7  else
8      while s is not on the State Transition Station do
9          Sleep a while, recheck until recovery times out
10     if s exist in the State Transition Station then
11         Read $s$ from the State Transition Station and assign it to the wrapped task
12         Process the pending tuples that received before $s$ is initilised
13         $isInit = True$
14     else
15         Return with a recovery failure flag
16 Emit the recovery transaction $tx$ to downstream
17 Acknowledge the recovery transaction $tx$
18 return
```

added by the topology adapter if there is at least one stateful bolt and the state replication is turned on. The adapter also connects the recovery manager with other operators through a separate internal stream, allowing it to send recovery signal across the topology for starting and supervising the failure recovery process. Once the recovery manager receives acknowledgement from all the downstream operators, the state recovery is complete and the streaming application can resume execution from the point it left off.

As mentioned in Section 5.3, the task wrapper encapsulates the state transfer and recovery logic, making the state management mechanism autonomous and transparent to its wrapped task. There are two different types of wrappers in our framework, encompassing stateless and stateful tasks, respectively. The wrapper for stateless tasks is called *SignalForwarder*, whose only duty is to forward the signal tuple to all its downstream tasks; while the *StateManipulator* for stateful tasks not only handles the state management on receiving the recovery transaction, but also relays the received signal for it to be broadcast across the topology DAG.

Specifically, Algorithm 5.2 illustrates the pseudo-code of state operations in the StateManipulator. Lines 1-6 of the algorithm are executed by the statful tasks that are not affected by the failure. Considering that there could be multiple tasks alive in the same fleet and they all attempt to preserve states without prior-synchronization, our algorithm takes advantage of the inter-process lock on Zookeeper, ensuring that there is only one task in each crash-affected fleet communicating to the state transmit station, which greatly reduces the network flow during the state transfer process.

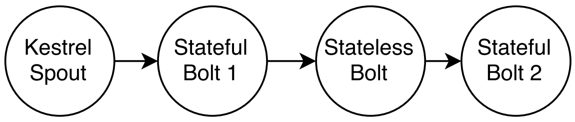
Lines 7-15 of the algorithm describe the state recovery logic for restarted tasks. Once the recovery signal is received, tasks that are initialised from scratch start querying the state transmit station for accessing their lost state. However, the corresponding state preservation process may not be complete by the time they restarted, so these tasks need to repeat the retrieval attempts until the recovery times out. Besides, any tuple that is received before the state initialisation is added to a pending list to delay its execution.

Due to the limitation of the acknowledgement system in Storm's core, our failure recovery logic cannot eliminate duplicate tuple evaluation and provide only at-least-once message processing guarantee. However, it is possible to achieve exactly-once semantics with the Trident abstraction, where the idea of replication still applies for state persistence.

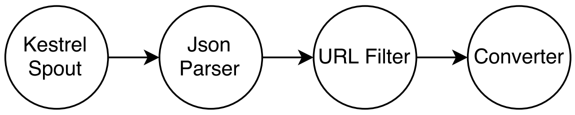
5.6 Performance Evaluation

In this section, we explore in detail the performance of our prototype (E-Storm) compared to the existing checkpointing method, by applying them to both synthetic and real-world streaming applications. The design of evaluation answers the following questions:

- What are the runtime overheads for enabling state persistence and how do these overheads vary in different use cases? (Section 5.6.2)
- How does the resilience level, i.e. the number of state replications, affect the performance of E-Storm? (Section 5.6.2)



(a) The topology of the synthetic test application



(b) The topology of the real-world test application

Figure 5.5: The illustration of the test application topologies. In Fig. 5.5a, Stateful Bolt 1 and State Bolt 2 have the same implementation.

- How long does it take for E-Storm to recover a streaming application from JVM crashes. (Section 5.6.3)

5.6.1 Experiment Setup

Our experiments are conducted on Storm v1.0.2 using the Nectar IaaS Cloud[2]. The Storm cluster consists of 10 worker nodes and 2 administrative nodes for Nimbus and Zookeeper, respectively. Apart from that, there is also (1) a Kestrel[3] node that caches inputs for streaming applications when the processing capability of the cluster cannot catch up with the speed of data generation; and (2) a Redis node that works as the remote data store in the checkpointing method and the state transmit station in E-Storm. The above-mentioned nodes are all of "m2.medium" size, provisioned from the same NCI availability zone and equipped with 2 VCPUs, 6GB RAM and 30GB root disk.

Test Applications

Our first test application is synthetically designed to mimic different intensities of state usages. As shown in Fig. 5.5a, it consists of four operators ($Op_1, ..., Op_4$): Op_1 is the KestrelSpout that pulls input data from the Kestrel queue server, while Op_2 and Op_4 are two stateful operators that are connected through the stateless operator Op_3. Table 5.1 illustrates the application configurations that adjust the size of internal states and the way of accessing them.

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

Table 5.1: The configuration of the synthetic test application

Symbol	Configuration Description
N_s	The number of stateful tasks in this topology
E_s	The size of states being kept in each stateful task
F_s	The number of state access in the execute method

In particular, N_s denotes the number of stateful tasks in total, where Op_2 and Op_4 equally get $N_s/2$ tasks for parallel execution; whereas the parallelism degree of Op_3 is fixed at 10, as it is sufficiently large to ensure the stateless operator will not be the bottleneck of the topology. In terms of the streaming logic, each stateful task maintains a key-value map and continuously fills it with the recently received data. We externally cap the maximum number of map entries at E_s, which essentially determines the size of the internal state to be kept in this particular task. Lastly, F_s denotes the number of state access operations encapsulated in the *execute* method, which effectively determines the frequency of state access for processing a single tuple.

The second application is drawn from a real-world use case — extracting short Uniform Resource Locators (URLs) from incoming tweets and replacing them with complete URL addresses. As depicted in Fig. 5.5b, the whole application also consists of 4 operators: the KestrelSpout as used in the synthetic application, the JsonParser bolt that parses the tweet string and extracts the main body from the JSON content, the URLFilter that isolates and filters short URLs from the message body, and the Converter that actually performs the conversion. Among them, the Converter is a stateful operator caching the map of short and complete URLs in its memory, so trending pages can be identified from the map statistics and it does not need to check the remote database whenever an input tuple is received. As for configuration, this application has only one parameter to be set, i.e. the number of tasks that are evenly distributed among all these four operators.

Evaluation Methodology

We examine the application performance in two major metrics, namely throughput and complete latency. The application throughput is obtained externally by observing the number of acknowledgements per unit of time, while the complete latency is a built-in

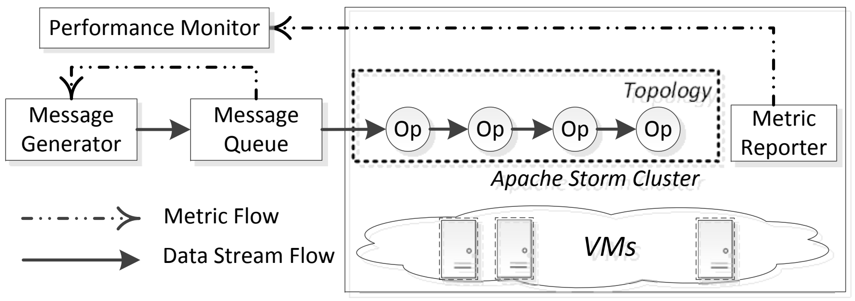


Figure 5.6: The profiling environment set for the performance evaluation. Solid connectors represent the generated data stream flow, while the dashed connectors denote the flow of performance metrics.

Storm metric, which calculates the average time taken by a tuple and all its offspring to be completely processed by the topology.

In order to evaluate the performance overhead brought by different approaches of state persistence, we have set up a profiling environment that feeds the streaming application with sufficient inputs and continuously monitors the resulting performance. The components of the profiling environment are briefly depicted in Fig. 5.6. The Message Generator is a Java program that reads the workload file on-demand to emit a particular size of profiling stream, and the workload file contains 899,560 tweets in JSON format that collected from 24/03/2014 to 14/04/2014. Running on the kestrel node, the Message Queue module is built with Twitter Kestrel, which exposes a Thrift interface for the message generator to retrieve the length of the message queue and further determine whether the streaming application has been overwhelmed by the profiling data. The application metrics, such as throughput and latency, are externally collected by the performance monitor which is implemented as a RESTful client. With ample profiling inputs, the Storm cluster will be pushed to its performance limit, i.e. exhibiting the highest throughput, after the application is stabilised. For all the test applications, we also set the Storm configuration *MaxSpoutPending* to 10000, which is the maximum number of unacknowledged tuples that can be pending on a spout task at any given time. Therefore, such environment setting makes it possible to compare the performance across different test applications.

5.6.2 Performance of Error-free Execution

Overhead of State Persistence

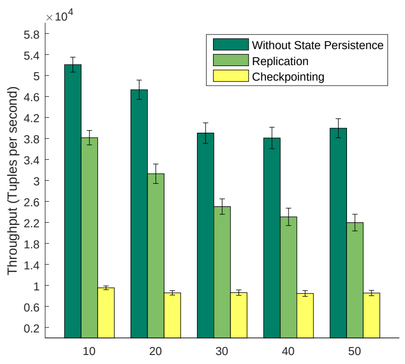
In this section, we first examine the performance overhead brought by state persistence, as well as how it varies under different application behaviours when the configurations of state have been altered. Table III describes the evaluated and default values for each application parameter. When a particular parameter is being examined, the others were set to their default values.

Table 5.2: Evaluated parameters and their values (Default values are showed in bold).

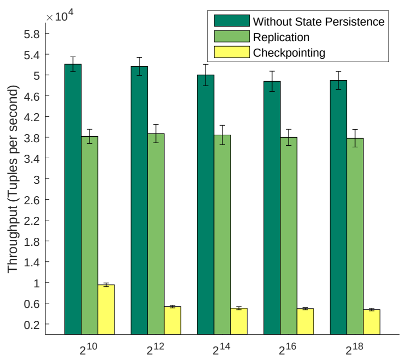
Parameters	**Values**
N_s (synthetic application)	**10**, 20, 30, 40, 50
E_s (synthetic application)	$\mathbf{2^{10}}$, 2^{12}, 2^{14}, 2^{16}, 2^{18}
F_s (synthetic application)	**4**, 6, 8, 10, 12
Number of tasks (real application)	**8**, 16, 24, 32, 40
Number of state replications	**2**, 4, 6, 8, 10

As shown in Fig. 5.7 and Fig. 5.8, the results obtained from the synthetic application clearly demonstrate that enabling checkpoint for state persistence leads to significant performance degradation. Under the default configuration, checkpointing yields 18.3% throughput and 5.38 times complete latency, compared to the baseline case with no state management. As a matter of fact, the acknowledgement of processed tuples have to be delayed until the internal state has been committed to the remote data store, therefore, it is not possible for the checkpointing method to reduce the complete latency below the pre-designated checkpoint interval, which is default to 1 second for performance consideration.

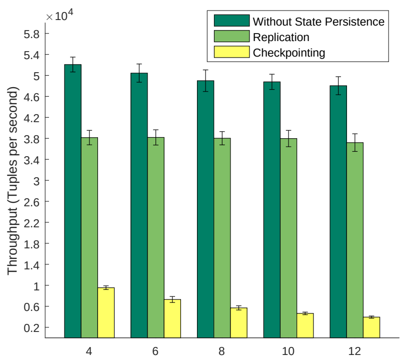
Furthermore, by altering the application configuration, we can identify and measure the factors that contribute to the checkpointing overhead, namely periodic synchronization and state access. When the size of state is increased from 2^{10} to 2^{18}, the application throughput drops to about 49.9% while the complete latency soars to 196.7%, indicating that larger state involves more state updates and thus imposing significant overhead for the remote data store to synchronize. However, this overhead does not increase linearly along with the size of state as the checkpointing method actually adopts the strategy of



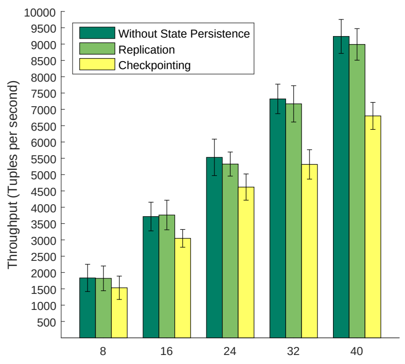
(a) Varying N_s (Sythetic App)



(b) Varying E_s (Sythetic App)

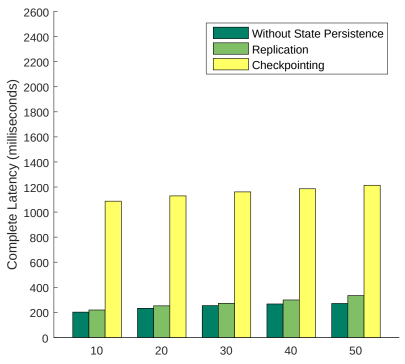


(c) Varying F_s (Sythetic App)

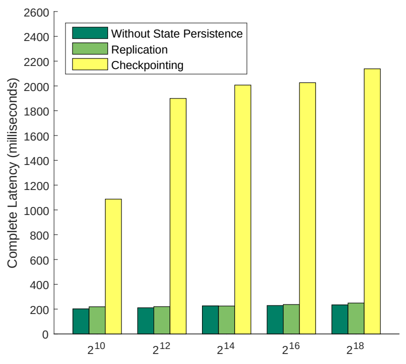


(d) Varying number of tasks (Real App)

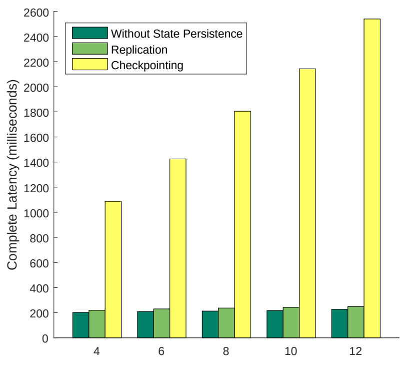
Figure 5.7: The application throughput under different state persistence methods. Each result bar is an average of 10 consecutive throughput readings collected every 60 seconds, with the standard deviation plotted in the error bar.



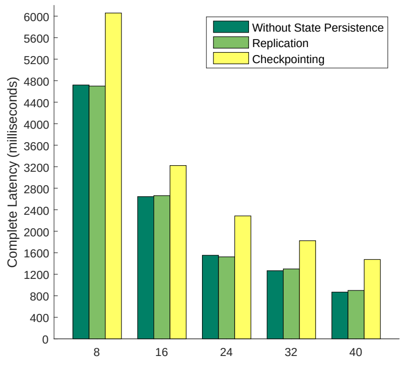
(a) Varying N_s (Sythetic App)



(b) Varying E_s (Sythetic App)



(c) Varying F_s (Sythetic App)



(d) Varying number of tasks (Real App)

Figure 5.8: The application latency under different state persistence methods. Each result is an average of statistics collected in a time window of 10 minutes, and the error bar is omitted as the standard deviation of latency is negligible for stabilised applications.

incremental update for synchronization.

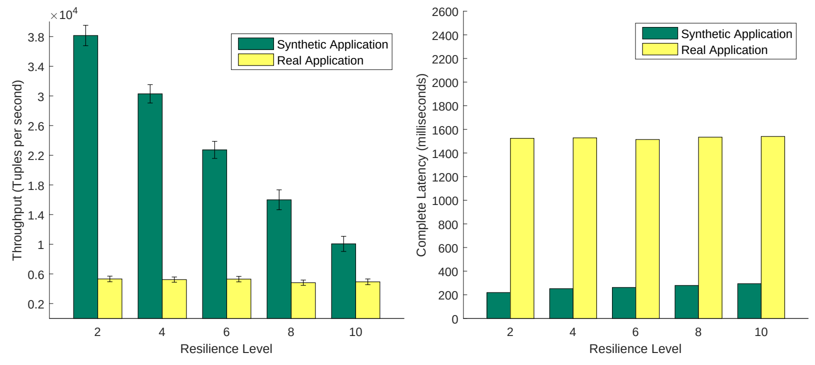
However, such update strategy also brings non-negligible network delay for state access. After F_s varies from 4 to 12, the checkpointing method suffered from 59% throughput loss and 233.7% latency increase, and the performance degradation almost changes linearly in regard to the variation of F_s.

The replication-based state persistence, by contrast, shows promising performance against checkpointing. To start with, the replication method exhibited steady throughput and latency when varying E_s and F_s, i.e. the size of state and the frequency of state access. In the worst case, it accounts for 74.9% of throughput and introduces only 11.5% of latency compared to the non-persistent baseline. The rationale behind these results is that our approach manages the internal states in memory resembling the way how baseline works, and the performance is unlikely to be bottlenecked by the memory access speed.

However, as expected, it has been identified that the overhead of state replication climbs as the number of stateful tasks increases. To put it quantitatively, after adjusting N_s from 10 to 50, the throughput of replication reduces from 73.3% to 55% and the complete latency rises from 108.4% to 123.2%, with all figures obtained from the comparison to the baseline in which no state persistence is provided. Our analysis deems such performance degradation as the result of bandwidth contention. As there are more shadow tasks to be spawned at different nodes and their inputs to be replicated at the message channel, our approach causes additional bandwidth consumption and impairs the maximum performance. However, this overhead does not increase super-linearly with the number of stateful tasks, so we argue that the proposed method is still applicable to production-scale applications for state persistence.

Overhead of Maintaining More Replicas

To investigate how the number of replicas affects the application performance, we deployed the two test applications to the testbed: the synthetic application uses its default configuration, while the real application sets the number of tasks to 24 for better demonstration. The evaluation results are illustrated in Fig. 5.9. For the synthetic application whose performance is bounded by the inter-node bandwidth, the resulting throughput



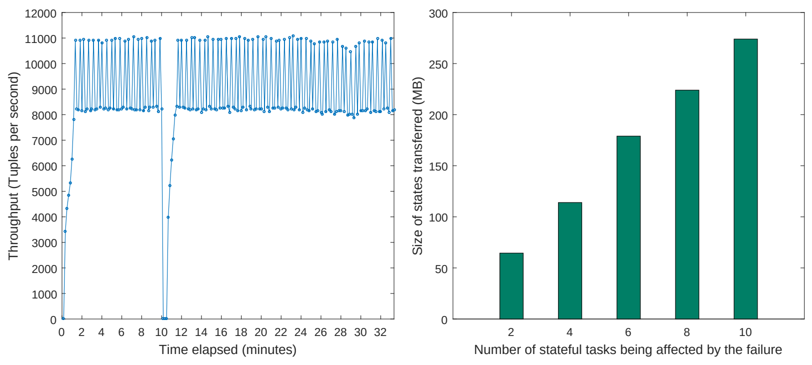
(a) Throughput changes (b) Latency changes

Figure 5.9: The application performance under different resilience levels, i.e. number of replicas for stateful tasks. The throughput and latency notations have the same meaning as explained in Fig. 5.7 and Fig. 5.8.

dramatically decreases to roughly 26.4% of the highest point as the number of replicas increases to 10, while the complete latency experiences a slight increase from 219 ms to 294 ms during this variation. On the other hand, introducing more state replicas to the real application has not produced noticeable performance degradation, which can be explained by the fact that the whole application is actually bounded by the lack of tasks to process the incoming stream in parallel, rather than the duplication of messages at the communication channel. We also observed that tasks of the real application spend most of their time executing tuples, contrasting to the tasks of the synthetic topology having a relative small capacity with more time spent on waiting I/O operation to complete. Through this experiment, we reach the conclusion that our method is better suited for applications with mild or medium bandwidth consumptions. If an operator is known to be bandwidth bound, E-Storm has the ability to individually set the number of replicas for this operator to a relatively small value, in order to avoid significant performance penalty.

5.6.3 Performance of Recovery

To inject failures and invoke the recovery process, we send *SIGKILL* signals to the designated worker processes, forcing them to terminate without proper clean-up. The real-world application (url-mapping) is selected as the test application, with number of tasks set to 40 and each stateful task having one replica running on another worker node. The



(a) Monitored application throughput (b) Size of transferred states

Figure 5.10: The recovery test performed on the real-world streaming application

application outputs and metrics are collected throughout the test to validate the efficacy and efficiency of E-Storm.

Recovery from a Single Error

In the first experiment, a JVM crash was injected to the test application on the 10^{th} minute, causing two stateful tasks to lose their states. Fig. 5.10a depicts the application throughput obtained through the Storm's RESTFul API. Note that they are calculated as an average statistics of a short time period (10 seconds), as the instantaneous throughput can better reflect the consequence of failure on the application performance. The results demonstrated that the application was paused for 30 seconds, and then gradually increased its throughputs for 60 seconds until stabilisation. The recovery time includes: (1) the time used for the supervisor daemon to restart the failed worker process (5.6 s); (2) the time taken for the failed tasks to be prepared (1.8 s); (3) the time taken by the alive tasks to write to Redis (6.2 s) and (4) the time taken by the restarted tasks to retrieve their states from Redis (7.9 s). This break-down data is obtained from the analysis of the Storm log file and the status of Redis server. However, it is also worth mentioning that by default, the Kestrel Spout waits for 30 seconds before replaying the failed tuples, so the application did not produce throughputs immediately after completing the recovery process.

Recovery from Multiple Errors

The second experiment is to evaluate the performance of recovery triggered by multiple errors. We injected multiple *SIGKILL* signals to the target worker processes at the same time, with care taken not to bring down all the states backups for a stateful task. As shown in Fig. 5.10b, the size of states being cached in Redis almost rises proportionally along with the number of tasks being affected. This result can be explained by two factors: firstly, each stateful tasks maintains roughly the same size of states as the hash-code based streaming grouping balances the load well among them; secondly, only one alive task in each affected task fleet got to preserve its state to Redis, while the others were staying idle during the whole recovery process.

Table 5.3: The comparison of recovery time under multiple errors (unit: seconds)

No. of stateful tasks affected	W_r	T_p	W_s	R_s	Total
2	5.6	1.8	6.2	7.9	90
4	5.6	1.9	11.2	13.2	120
6	5.6	2	16.7	18.9	120
8	5.6	1.9	20.2	22.5	120
10	5.7	2.1	25.8	28.7	140

In Table 5.3, we also compare the recovery time needed for the application to resume execution under multiple errors. In this table, W_r is the time used for the failed worker processes to be restarted; T_p is the time for restated tasks to be prepared; W_s is the time used for writing states to Redis and R_s is the time for loading states to the restated tasks. Note that each failed node executes the recovery protocol asynchronously, so they may take different times to complete each recovery stage. Therefore, we report the results by averaging the readings collected on the failed nodes, except the *Total* column which is the time taken for the topology to restore its normal performance (reaching 90% of the average throughput observed before failure).

The comparison results indicate that the recovery time for Storm daemons are independent from the scale of failures, but the time used for writing and reading states increases along with the state sizes, which are shown in Fig. 5.10b. However, we can reasonably envision that the use of multiple Redis instances can reduce these time, as different task fleets are mapped to their corresponding Redis instance for concurrent state transfer.

5.7 Related Work

State management is one of the major research topics in distributed stream processing. In this section we review the approaches that manage the transient operator states with particular goals in data stream management systems.

Some works manage states for application integrity in the event of failure or operator scaling. Fernandez et al. proposed a set of state management primitives to expose operator states explicitly to the middle-ware system, so that the DSMS is able to periodically checkpoint them to the upstream VMs with partitions in order to enable state recovery and scaling [24]. Similarly, ChronoStream [171] provides elastic support for stateful operators by dividing the application states into a set of computation slices, which are checkpointed to specified nodes exploiting locality-affinity and lineage-free progress tracking to ensure deterministic semantics. StreamScope is a recent effort to provide declarative interface for users to express complex streaming logic, which also offers the *snapshots* abstraction that periodically checkpoints the operator states without user intervention [104]. Also, MillWheel checkpoints its work in progress at fine granularity so that the states are persisted against failure and message senders are relieved from buffering the pending data for a long period [3]. There are even more works that fall into this area [49, 109, 128]. However, regardless the level of which the checkpoint is performed or the place where the checkpoint data is stored, periodic state manipulation still introduces non-negligible runtime overhead.

Some works, on the other hand, focus on migrating states for dynamical application scaling. Cardellini et al. realise dynamic horizontal scaling for stateful operators in Storm by allowing the states to be migrated between existing and newly added tasks [23]. Gedik et al. explores the profitability of auto-parallelisation by providing a state management API, a run-time migration protocol, and compile-time topology optimization techniques [58, 141]. Ding et al. further investigate the trade-off between synchronization overhead and result delay during state migration, so that the selection of migrated tasks can be optimised to lower the latency spike [42]. However, these methods cannot be used to provide state persistence against failures.

The strategy of replication in stream processing has also been discussed in the literature. Stormy uses replications for high availability, so there is no failure recovery mechanism provided to transfer states between different replicas [107]. For fault-tolerance, Balazinska et al. incorporate a replication-based approach in the Borealis system [1], which duplicates the execution of the same query network on multiple worker nodes [8]. To ensure all replicas processing data in the same order, they also introduce a data-serialising operator that sorts the multiple streams as input and produces a single output stream with a deterministic order. By contrast, E-Storm performs replication at the fine-grained operator level with the flexibility to adjust the resilience guarantee individually, and it does not duplicate the execution of stateless operations. Also, we achieve replica consistency through a lightweight acknowledgement mechanism, thus avoiding the heavy messaging sorting overhead. To reduce the burden of active replication, Martin et al. present an approach that first conducts state partitioning and then distributes state slices across the participating worker nodes [112]. However, this method only profits in MapReduce-like event processing systems as the state partitioning is a side effect of execution during runtime, whereas in the state-of-the-art data stream systems, this method would incur significant state transfer cost when the execution is error-free.

With the similar goal of reducing the replication overhead, Henize et al. combine active replication with upstream backup, allowing for the adaptive selection of replication mechanism for individual operators based on the characteristics of the current workload [71]. However, the placement of operator and replicas to hosts is not discussed in the paper. Flux is an opaque operator implemented in TelegraphCQ [28] that composes duplicated dataflows to enable online-recovery and mask load imbalances [145]. Nevertheless, replicating the whole data flow and adding an Exchange layer [63] between each Producer-Consumer pair would incur more overhead than our approach, which requires only replicating the stateful operations.

5.8 Summary

In this chapter, we designed and implemented E-Storm, a replication-based state management system that masks the loss of operator states in the case of JVM and node crashes. By using E-Storm, the stateful tasks are replicated on different worker nodes under a replication-aware placement strategy, and the restarted tasks are able to retrieve their previous states from the alive partners through an asynchronous recovery protocol. During the state transfer process, the implementation of E-Storm takes advantage of Redis to decouple the coordination of state senders and receivers, and makes use of Zookeeper to reduce the size of states being transmitted. Therefore, it achieves concurrent and high performance system recovery in the presence of failures.

Through a comprehensive performance evaluation, the results confirm that our approach greatly outperforms the existing checkpointing method in terms of throughput and latency overhead. Specifically, E-Storm can bring up to 9.44 times throughput improvement while reducing the application latency down to 9.8% compared to that of the checkpointing method (witnessed in the synthetic test application when F_s, the number of state access in the *execute* method, is set to 12). We also identified that the overhead of checkpointing is attributable to the frequent state access and remote synchronization, which cannot be mitigated by enlarging the checkpointing interval as it would incur unacceptable latency penalty for real-time streaming applications.

After discussing individual resource management topics such as operator parallelisation, task scheduling, and state management, the next chapter presents a comprehensive framework to deploy the streaming applications in clouds towards a pre-defined performance target. We aim to optimise resource provisioning using an iterative feedback and control process, so that the performance target can be achieved with minimal costs regardless of the initial resource allocation.

Chapter 6

Performance-Oriented Deployment of Streaming Applications on Cloud

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 be used in clouds with an on-demand resource pool. In this chapter, 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 parallelisation, 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.

6.1 Introduction

FOR better programmability and manageability, streaming applications are developed on top of a specialised 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

This chapter is derived from:

- **Xunyun Liu** and Rajkumar Buyya, "Performance-Oriented Deployment of Streaming Applications on Cloud," *IEEE Transactions on Big Data (TBD)*, Accepted, In press, DOI:10.1109/TBDATA.2017.2720622 Pages: 1-14, IEEE, 2017.

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 [5, 10, 46, 100, 106]. As developers have assumed a static environment and only tune the application configuration towards higher utilisation 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 virtualisation, 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 customises 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 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

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

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 pre-defined performance target. Specifically, there are three optimization problems involved: (1) operator parallelisation, 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 parallelisation and task mapping have been separately investigated in the existing literature for various optimization targets [5, 46, 50, 54, 173], 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 utilisation 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 pre-defined

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.

6.2 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. 6.1, there are three layers in the streaming application structure. the *topology* lying in the topmost *logical layer* 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 layer* 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 realisation 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 customised with regard to the specific situation, such as

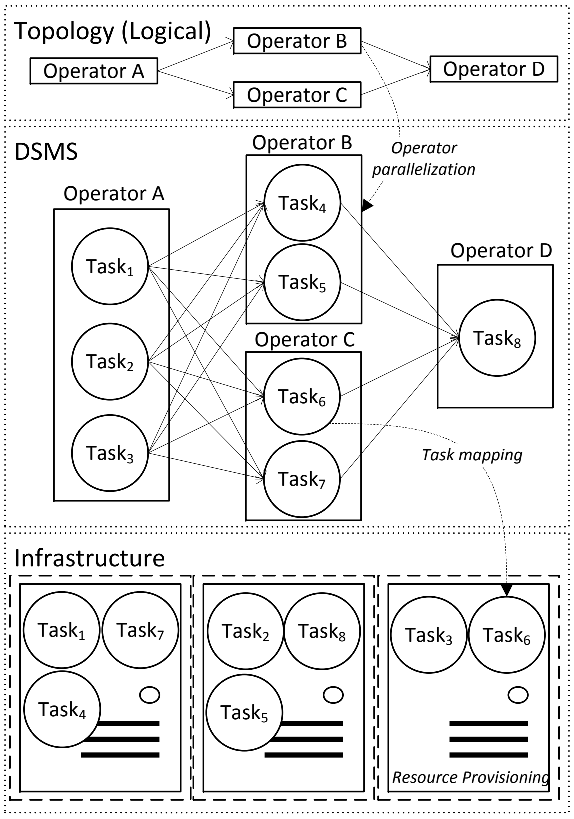


Figure 6.1: Layered structure of an example streaming application

the workload characteristics, performance target, and the capacity of underlying infrastructure.

The first deployment choice incurred in this layer is (1) *operator parallelisation*. Specifically, DSMS treats each operator as a dividable logical entity and parallelises 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. 6.1 using a dash arrow labelled as "operator parallelisation", which shows that *Operator B* is parallelised into two tasks: $Task_4$ and $Task_5$.

The underlying *infrastructure layer* is the place where the parallel execution of tasks actually happens. The construction of this layer 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. 6.1, 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 pre-defined throughput target, maximizing distributed resource utilisation, minimizing the overall data processing latency, and etc. In Fig. 6.1, $Task_6$ 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 optimise them to achieve automatic, performance-oriented, and cost-efficient streaming application deployment.

6.3 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. 6.2 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

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

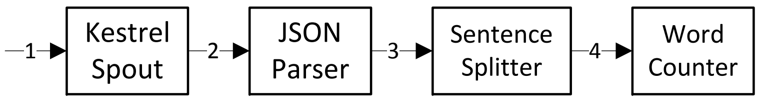


Figure 6.2: The topology of word count streaming application, where the solid arrows represent data streams.

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 after the application is stabilised. 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 [124], 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.

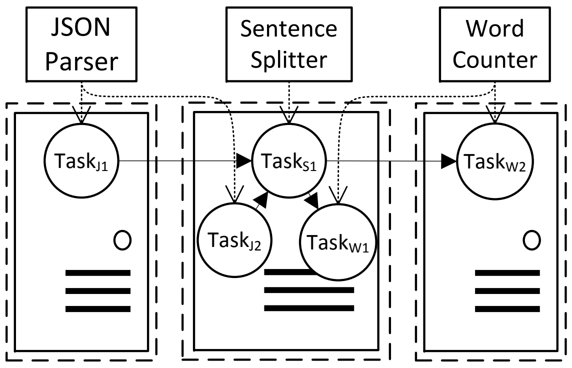


Figure 6.3: Partial deployment sketch that shows how to designedly assign the inter-node communication ratio of $Task_{S1}$ to 50%. Solid arrows represent data streams while dash arrows denote the "operator-task" containment relationship.

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) and constantly vary its inter-node communication ratio by parallelising 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. 6.3 illustrates how we can deploy the word count application to achieve this goal. Specifically, we parallelise 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 parallelising 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

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

parallelisation 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 initialises 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.

6.3.1 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 [5,29,124,173].

3. The same type of machines are selected on the IaaS cloud to form a homogeneous infrastructure.

6.4 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

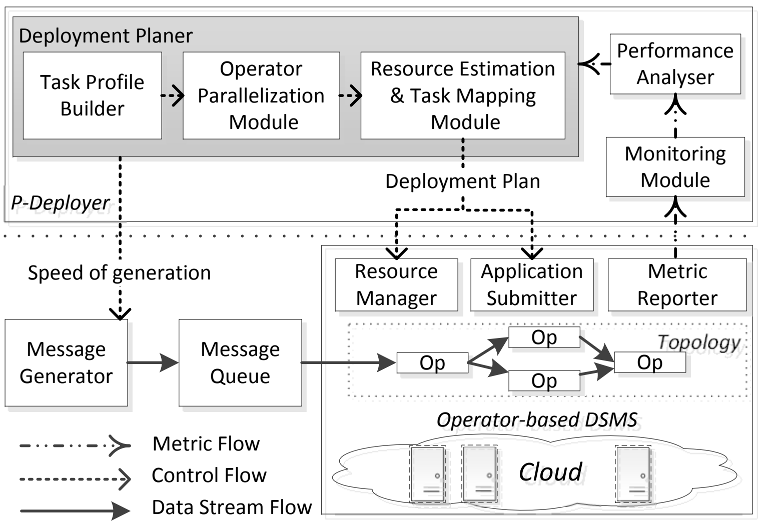


Figure 6.4: The Monitor-Analyze-Plan-Execute (MAPE) architecture of P-Deployer and its working environment

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. 6.4, 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 parallelisation*: 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.

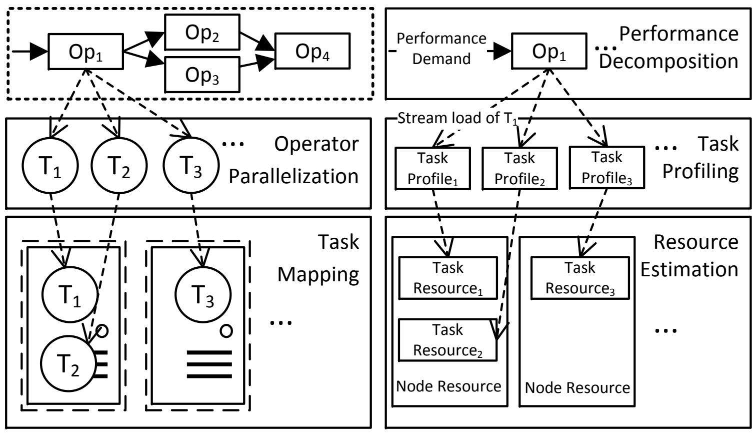


Figure 6.5: Translating the performance demand of a streaming application into the estimated resource consumption through its task profiles

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 minimise the number of used machines while satisfying the resource needs of collocating tasks.

6.5 Deployment Plan Generation

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

6.5.1 Building Task Profile

As introduced in Sec. 6.3, 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.

The intuition behind task profile is illustrated in Fig. 6.5. 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 parallelisation, 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 6.1: 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. 6.1 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 serialisation/deserialisation, message buffering and performing the actual send/receive operation. Note that we 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.

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. 6.1:

$$\begin{aligned} C_\tau &= \sum_{k=1}^{n_i} S_k c_p + (\sum_{k\in\Theta} S_k + \sum_{k\in\Theta} S'_k) c_t \\ C_{\tau,\min} &= \sum_{k=1}^{n_i} S_k c_p \\ C_{\tau,\max} &= \sum_{k=1}^{n_i} S_k c_p + (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S'_k) c_t \end{aligned} \tag{6.1}$$

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. 6.2:

$$M_\tau = (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S'_k) m_{pt} \tag{6.2}$$

6.5.2 Operator Parallelisation

In light of the observation that over-parallelisation hurts the operator performance (see Sec. 6.3), 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 parallelise 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. 6.2:

Table 6.2: Symbols used for operator parallelisation.

Symbol	**Description**
S_{op}	Monitored stream load of operator op
P_{op}	Parallelism degree (number of tasks) of op
$\begin{Bmatrix} c_p, c_t \\ m_{pt}, t_{soj} \end{Bmatrix}$	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 utilise 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 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 parallelisation 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

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

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 Parallelisation 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.

6.5.3 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 minimised. 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. 6.3.1. The optimization target of this problem is to minimise the number of used machines. Therefore, this is a variant of bin-packing problem that can be formalised in the linear programming form.

Table. 6.3 summarizes the newly introduced symbols in this subsection.

Table 6.3: 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. 6.3, 6.4 and shown in Fig. 6.6
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

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:

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

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. 6.3, which is derived from Eq. 6.1 but making use of the monitored stream loads and the result of operator parallelisation to deduce the sizes of input/output streams for task τ:

$$\begin{aligned}
C_{\tau_i} = & \sum_{k=1}^{n_i} \frac{S_k}{P_{op_k} * P_{op}} c_p \quad + \\
& (\sum_{k=1}^{n_i} \frac{S_k}{P_{op_k} * P_{op}} (P_{op_k} - Adj_k) + \sum_{k=1}^{n_o} \frac{S_k^{'}}{P_{op_k}^{'} * P_{op}} (P_{op_k}^{'} - Adj_k^{'})) c_t
\end{aligned} \tag{6.3}$$

The variables used in Eq. 6.3 are illustrated in Fig. 6.6: 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

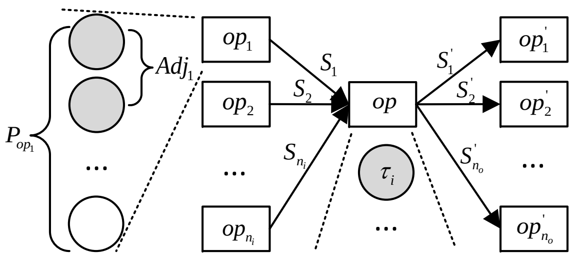


Figure 6.6: The illustration of variables used in the calculation of resource consumption for task τ_i, where tasks in grey indicate that they are collocated on the same machine.

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 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. 6.6, we also present the formulation of memory consumption for task τ as below:

$$M_{\tau_i} = (\sum_{k=1}^{n_i} S_k + \sum_{k=1}^{n_o} S_k^{'})\frac{m_{pt}}{P_{op}} \tag{6.4}$$

In order to minimise 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 [57]. 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.

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) [55] and branch-and-price (BCP) [40]. 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 generalised in three aspects which is shown in Algorithm 6.1.

As CPU and memory are measured in different metrics, Algorithm 6.1 first normalises 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.

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 maximise the potential resource savings.

Lastly, since the calculation of $s(\tau, m)$ actually depends on what have already been

Algorithm 6.1: 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
 1 Normalise resource demands for $\vec{\tau}$
 2 $n_m \leftarrow 0$
 3 while there are tasks remaining in $\vec{\tau}$ to be placed do
 4     Start a new machine $m$
 5     Insrease $n_m$ by 1
 6     while there are tasks that fit in machine $m$ do
 7         foreach $\tau \in \vec{\tau}$ do
 8             Calculate $s(\tau, m)$ (Eq. 6.5)
 9             Calculate $p(\tau, m)$ (Eq. 6.6)
10         Place the task with the largest $p(\tau, m)$ into machine $m$
11         Remove the task from $\vec{\tau}$
12         Update the remaining capacity of machine $m$
13 return $\vec{m}$
```

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. 6.3. Note that $\vec{Adj}$ is initialised to all zeros for every task, implying that all unpacked tasks are considered to be external by default when calculating Eq. 6.3. 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:

$$s(\tau, m) = C_{\tau,\max} - C_{\tau} + \sum_{\tau_k \in \wp(m)} (C_{\tau_k}^{\vec{Adj}} - C_{\tau_k}^{\vec{Adj}+1}) \tag{6.5}$$

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:

$$p(\tau, m) = a_r \cdot s(\tau, m) - a_c(C_{\tau} - r_m^{cpu})^2 - a_m(M_{\tau} - r_m^{mem})^2 \tag{6.6}$$

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: $a_c = \frac{1}{n}\sum_{k=1}^{n} C_{\tau_k}, a_c = \frac{1}{n}\sum_{k=1}^{n} M_{\tau_k}$.

6.6 Implementation

The setup of the profiling environment has been briefly introduced in Sec. 6.4. More specifically, the Message Generator in Fig. 6.4 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

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

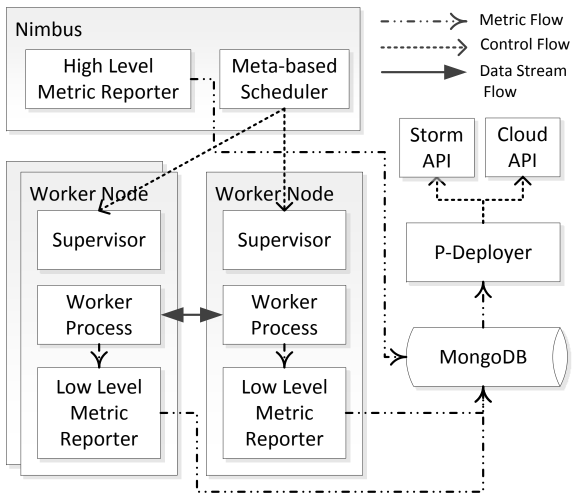


Figure 6.7: The integration of P-Deployer into Apache Storm

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. 6.7 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. 6.4 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 6.1. 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 utilisation 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 *Thread-*

[7]https://www.mongodb.com/
[8]https://collectd.org/

MXBean 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.

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 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 initialised 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

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

guarantees that each task is assigned to its designated Supervisor and Work Node.

6.7 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.

6.7.1 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.

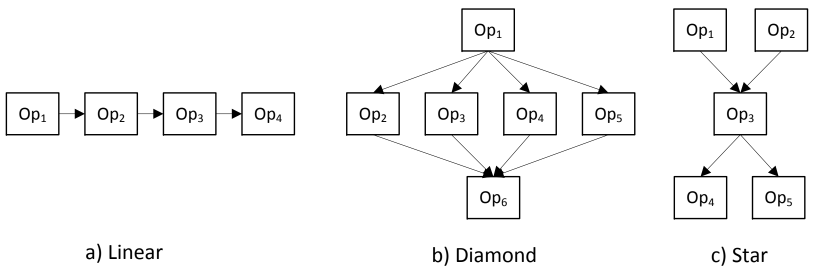


Figure 6.8: The synthetic Micro-benchmark topologies, the structures of which are referenced from R-Storm [124].

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 generalises to different topology structures. As shown in Fig. 6.8, 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.

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. 6.3 for its description.

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

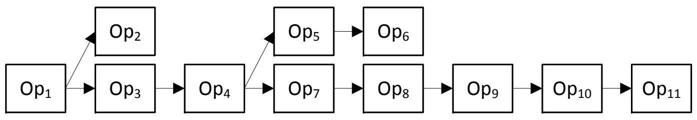


Figure 6.9: The Twitter Sentiment Analysis topology

Twitter Sentiment Analysis: This topology, as shown in Fig. 6.9, 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 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.

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

[11] https://github.com/kantega/storm-twitter-workshop
[12] http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010

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 stabilise 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 6.4.

Table 6.4: 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. 6.6)	1

6.7.2 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 microbenchmark topology with different time and resource complexities. In particular, the Linear topology includes only CPU-bound operators so that the whole topology is bound

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

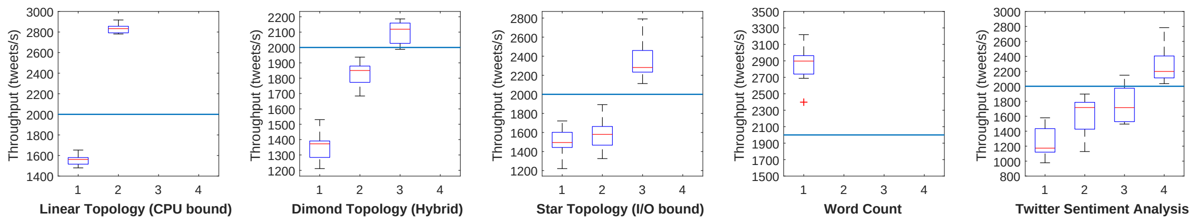


Figure 6.10: The monitored throughputs of topologies in the applicability evaluation. Each MAPE iteration corresponds to a plotted box that contains 10 readings of throughputs. P-Deployer finalises the deployment once the performance target denoted by the horizontal line is reached.

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.

Fig. 6.10 demonstrates that P-Deployer has been able to steadily improve the topology throughputs and finally realise 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 utilised 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

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

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 finalised. 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].

6.7.3 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, respec-

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

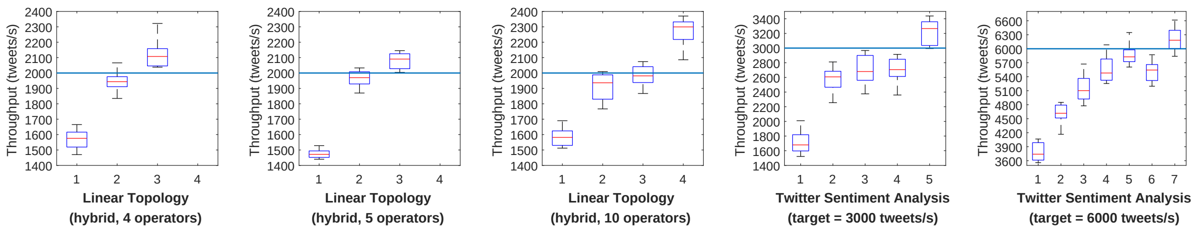


Figure 6.11: The monitored throughputs of topologies in the scalability evaluation. Each MAPE iteration corresponds to a plotted box that contains 10 readings of throughputs. P-Deployer finalises the deployment once the performance target denoted by the horizontal line is reached.

tively.

Fig. 6.11 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 realise 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 [51], but they have not been considered in P-Deployer due to the hardness of generalisation. 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. 6.5, which demonstrates that our profiling mecha-

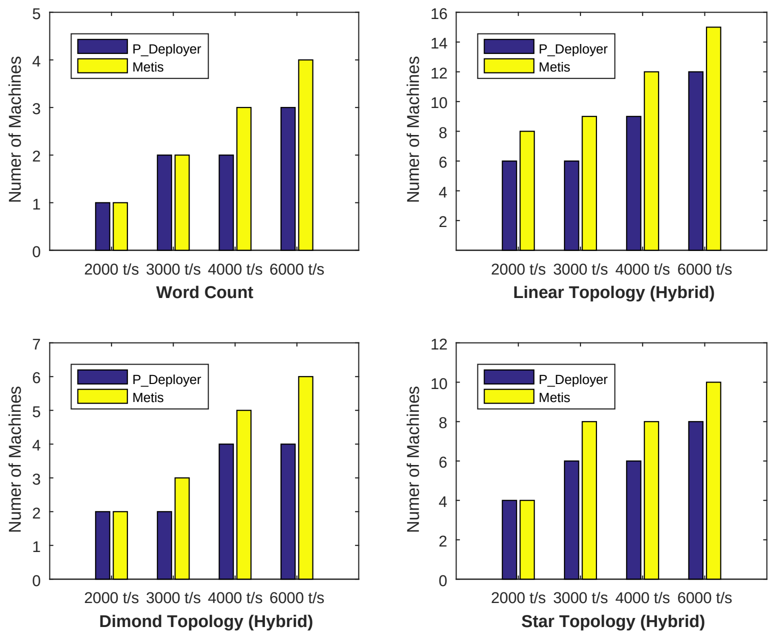


Figure 6.12: Comparison of P-Deployer and the Metis scheduler against the resource costs required to reach the same performance target

nism is low-overhead and the FFD heuristic is sufficiently fast for solving the overlapping bin-packing problem.

Table 6.5: The profiling cost (P_c) and the running time of Algorithm 6.1 (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

6.7.4 Cost Efficiency Evaluation

Comparable Method

We choose a state-of-the-art platform-oriented approach — Metis scheduler as our comparable method, which outperforms the greedy scheduler [5] in major metrics such as throughput and resource utilisation [50]. 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.

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. 6.12 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 utilisation 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 parallelisation would in turn impede the high utilisation 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.

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

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

6.8 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 utilisation. Fisher et al. employed Bayesian optimization to tune the operator parallelisation and other configurations for achieving higher throughputs [51]. 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 [50]. Nevertheless, as shown in our experiments, the resource utilisation 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 optimise workload distribution and inter-node communication [124]. 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 utilisation, Vanderveen et al. [162] 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. [54] 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. [177] 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. [100] presented a predictive scheduling framework, utilising Support Vector Regression to predict the complete latency under certain deployment arrangement. But they left out operator parallelisation, 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 minimise the negative impacts of dynamic workload adaptation. With the knowledge of predicted workload distribution, Mayer et al. [115] investigated dynamic stream partitioning using queueing theory models and time series analysis. Their work can be integrated with our operator parallelisation 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. [66] 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 [48, 56] and machine learning based frameworks[100, 185] have been investigated in the literature.

There are also several papers on reducing inter-node communication [5, 19, 29, 173] and improving system manageability and energy-efficiency [10, 37]. 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.

6.9 Summary

In this chapter, 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.

Chapter 7
Conclusions and Future Directions

This chapter provides a summary of thesis contributions towards realising robust resource management in distributed stream processing systems. The summary of research findings and working experiences of extending the state-of-the-art data stream management systems also leads to a discussion that identifies promising research directions to be explored in the future.

7.1 Summary and Conclusions

STREAM processing is an emerging in-memory computing paradigm that deals with the velocity characteristic of big data. By applying on-the-fly analytic as standing continuous queries, input streams are processed upon arrival to yield real-time insights with sub-second latencies.

Resource management in a stream processing system refers to the continuous adjustment of the deployment stack over a set of distributed resources. It applies a collection of profiling, modelling, and decision-making techniques to ensure that the system meets its functional and non-functional requirements with minimal resource costs.

Particularly, robust resource management is concerned with guaranteeing a certain level of performance and reliability with minimum additional resources. The importance of maintaining a satisfactory system performance stems from the fact that dynamic data streams are transient and volatile in nature. Failing to meet the latency constraints or handle the required throughput is not considered as a deterioration of Quality of Experience

This chapter is partially derived from:
- **Xunyun Liu** and Rajkumar Buyya, "Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review and Future Directions," *ACM Computing Surveys*, ACM Press, 2018 (under review).

(QoE), but rather a stability issue that eventually affects the correctness and sustainability of the processing logic. On the other hand, reliability becomes a prominent concern as the infrastructure scales out horizontally with commodity hardware, which requires proper resource management to mask possible partial failures that impair the execution of streaming entities and the availability of intermediate states.

Robust resource management also tackles the ongoing trend of migration where the deployment platform shifts from on-premise clusters to computing clouds. As a further step towards utility computing, cloud computing is committed to provide seemingly endless computing resources through a subscription-based service. The pay-as-you-go billing model allows users to rent computing power on-demand, paying bills on a usage basis similar to other utilities such as water, gas and electricity. Specifically, the adoption of the cloud computing model is beneficial to the practice of resource management in the following aspects: (1) it provides an elastic resource pool for the stream processing system to scale out/in dynamically under the fluctuating workload. The remote resource pool eliminates the upfront investments in hardware and licences, as well as the need for routine operation and maintenance of the infrastructure. (2) It offers a consistent entry point for conducting autonomous resource management. With a self-service user portal and a set of programming APIs, the application developers can manage resources unilaterally to cope with the continuously shifting stream processing requirements. (3) It provides various types of resources that are customisable in specification and location to suit the requirements and the particularity of the continuous queries. So developers have more flexibility to decide where and how the long-standing stream logic is executed in a distributed environment.

However, there are many challenges present to realise robust resource management in clouds for the distributed stream processing systems to exhibit required performance and reliability. In this thesis, we have conducted a thorough literature review, proposed novel resource management mechanisms, and implemented prototype systems to improve the current state-of-the-art in these fields. Particularly, Chapter 1 identifies the challenges of robust resource management and breaks them into five major categories — application profiling, operator parallelisation, task scheduling, state management, and resource provisioning. The thesis objectives are then defined accordingly with a list of formulated research questions in these areas, achieving an overall target of maintaining the SLA in performance and reliability with minimal amount of resources. Afterwards, it explains the evaluation methodology used throughout the thesis and summarises the contribution of the conducted research. A graph illustrating thesis organisation is also included at the end to help users navigate the thesis structure.

Chapter 2 presented a literature review of existing work falling into the scope of robust resource management. This chapter first defined the resource management problem by introducing the hierarchical structure of stream processing systems, and then identified the specific research topics involved in the deployment process to address the performance and reliability concerns. The storyline of the chapter is built following a comprehensive taxonomy of resource management and scheduling, which covers various topics such as resource type/estimation/adaptation, parallelism calculation/adjustment, and scheduling objective/method. The rest of the chapter is a complete survey that enriches the taxonomy with thorough discussions and comparison of related work, which concludes with a tabular review of key works as a quick summary of the state-of-the-art.

Chapters 3 to 5 focused on specific subdivisions of robust resource management. Particularly, Chapter 3 presented a fine-grained profiling mechanism for deciding operator parallelism, Chapter 4 discussed resource-efficient task scheduling, and Chapter 5 investigated replication-based state management. In these chapters, not only novel algorithms and frameworks are introduced, but also prototype systems are implemented and evaluated to demonstrate improvements in performance and reliability aspects.

Chapter 3 proposed a stepwise profiling approach to optimise application performance at runtime in an IaaS cloud platform. It automatically scales up the distributed computations over streams by increasing the parallelism degrees of streaming operators, which takes into consideration the application features and the processing power of provisioned resources. Designed and implemented as a Monitor-Analyse-Plan-Execute (MAPE) architecture, the proposed profiler also models the relationship between the amount of provisioned resources and the monitored application performance. This relationship model is revised iteratively to ensure the efficiency of scaling and the balance between data source and data sinks is achieved through proper resource allocation.

Chapter 4 modeled the scheduling problem as a bin-packing variant and proposed a heuristic-based algorithm to solve it with minimised inter-node communication. The scheduling algorithm is built on top of the fine-grained profiling information provided by the profiler discussed in Chapter 3, aiming to reduce the resource consumption through proper task consolidation without causing resource contention. A prototype scheduler named D-Storm is also extended on Apache Storm implementing a self-adaptive MAPE architecture, which validates the efficacy and efficiency of the proposed scheduling algorithm through comparison to a state-of-the-art resource-aware scheduler [124]

Chapter 5 proposed a replication-based state management system that actively maintains multiple state backups on different worker nodes to mask possible node crash and JVM failures. The exiting checkpointing framework involves a remote data store for state preservation and access, resulting in significant overheads to the performance of error-free execution. In contrast, the proposed state management system harnesses unused memory resources to maintain multiple state replicas and eliminate the needs for cumbersome state synchronisation. A prototype system, E-Storm, is built on top of Apache Storm with extensions of monitoring and recovery modules, which speeds up the recovery process through concurrent inter-task state transfer and avoids unnecessary bandwidth consumption during recovery by making use of inter-process locks.

Chapter 6 is a comprehensive work aiming to incorporate performance awareness into the resource management process. The current deployment practices are mostly platform-oriented, meaning that the deployment configuration is tuned to a static resource-set environment and does not fit the on-demand resource pool in clouds. This chapter proposed a deployment framework that enables streaming applications to run on IaaS clouds with satisfactory performance and minimal resource consumption, regardless of the initial resource allocation that the system has prior to the application submission. It achieves performance-oriented, cost-efficient and automated deployment by holistically optimising the decisions of operator parallelisation, resource provisioning, and task mapping. Using a Monitor-Analyze-Plan-Execute (MAPE) architecture, the prototype resource management system iteratively builds the relationship between performance outcome and resource consumption through fine-grained task profiling. Extensive experiments using both synthetic and real-world streaming applications have demonstrated the correctness and scalability of the proposed approach, as well as its superiority compared

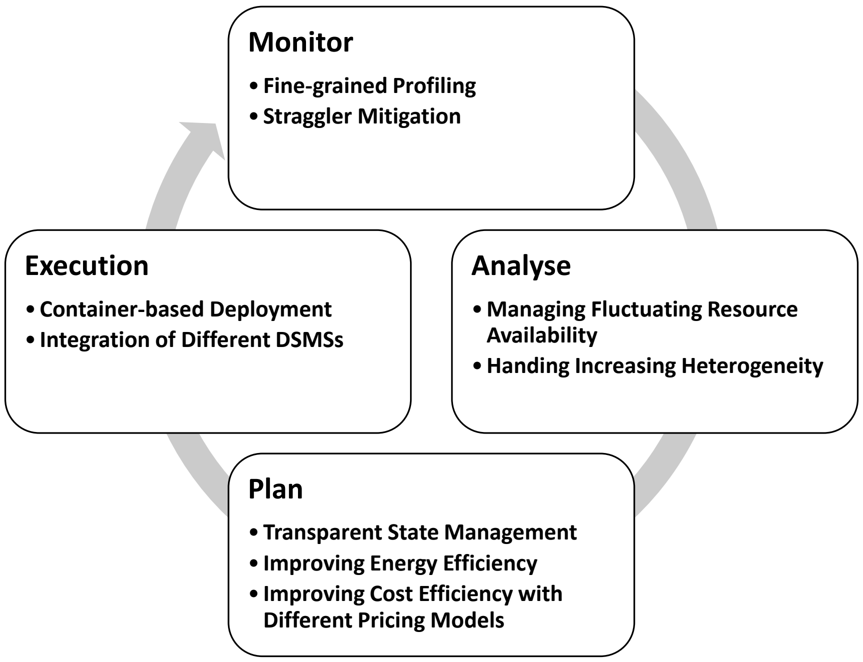

Monitor
• Fine-grained Profiling
• Straggler Mitigation
Analyse
• Managing Fluctuating Resource Availability
• Handing Increasing Heterogeneity
Plan
• Transparent State Management
• Improving Energy Efficiency
• Improving Cost Efficiency with Different Pricing Models
Execution
• Container-based Deployment
• Integration of Different DSMSs

Figure 7.1: Future research directions

to platform-oriented methods in terms of cost efficiency.

7.2 Future Directions

Although many research efforts have investigated the resource management and scheduling problem in distributed streaming systems, there are still several gaps and challenges to be explored in the future adapting to unpredictable workload fluctuations while hiding the intrinsic complexity to developers and operators. This section gives some insights into promising research topics and fields, which are orgainsed following the well-known Monitor-Analyse-Plan-Execute (MAPE) architecture in Fig. 7.1 to achieve the self-managing characteristics of distributed computing resources.

7.2.1 Fine-Grained Profiling

Accurate profiling of application and system metrics plays an important role in the decision-making process as they reflect the current state of the streaming system and indicate whether the desired SLA requirements have been satisfied. However, most of the existing works have based their deployment decisions on coarse-grained metrics such as application throughput, end-to-end latency, operator capacity and the volume of internal streams. These metrics collected at the operator or application level are too general to reveal the actual bottleneck of the data stream, so that the amendments can only be made on a best-effort basis with little guarantee on the adjustment effects. In order to capture the real culprit that throttles the application performance, a fine-grained profiling mechanism is required to fulfil the following expectations. (1) It should be installed at the task level to obtain fine-grained information such as the lengths of input/output queue, the task capacity on different infrastructure, and the average resource cost for processing a single tuple. (2) the application metric collected from the DSMS layer should be cross-validated with the system metrics to identify the probable cause and the severity of the processing bottleneck, allowing accurate amendments to be made in the next adjustment cycle. (3) proper sampling and quantisation techniques should be employed to reduce the profiling overhead while providing strong enough guarantee on result accuracy.

7.2.2 Straggler Mitigation

A straggler is a slow running entity that adversely impacts the performance of the whole streaming system. It could be a streaming task enduring severe resource contention or data skew, or it could be a computing node that is over-utilised or subjected to the performance variation of the host cloud instance. In either case, the local performance degradation caused by the straggler will soon propagate throughout the topology structure due to the publisher-and-subscriber streaming model — the upstream operator will be throttled because of the accumulated backlogs, and the downstream operators will stagnate without receiving sufficient inputs. Worse yet, a single task that becomes straggler will result in the whole operator being throttled. This is because the particular mechanism of stream routing has led to performance correlation between sibling tasks of the same operator. If one of these tasks becomes a straggler and performs significantly worse than the others, the logic of tuple emitting will also reduce the volume of stream sent to the other sibling tasks to not overwhelm the straggler. Note that this could lead to under-utilisation on other nodes as the healthy sibling tasks could have been placed in different places waiting for more inputs.

A straggler mitigation mechanism needs to quickly identify the root cause of the performance deterioration and cuts the chain of propagation with active intervention. A straggling computing node can be detected by its soaring resource usages and slow response time, while a straggler streaming task is revealed by the extended average tuple processing time or the sudden rise in resource consumption. In the context of resource management and scheduling, the measures that can be taken to mitigate stragglers include provisioning new resources, adding more parallelism, or rescheduling the placement of the straggler on a different infrastructure, while a comprehensive solution may involve all of these to avoid causing new stragglers during the adjustment process.

7.2.3 Managing Fluctuating Resource Availability

The existing work have often falsely assumed that, once provisioned, the same amount of resources will be made available to the streaming system throughout its standing lifecycle. Therefore, few of them has considered the consequences of fluctuation in node resource availability.

In fact, there are several reasons as to why it is common and inevitable to experience fluctuating resource availability in a distributed cloud environment. (1) Multitenancy: multiple tenants of a shared platform may experience performance interference as they compete for limited resources, despite separation mechanisms such as virtualisation and cgroups have provided a certain level of isolation for resource allocation; (2) Background activity: unexpected background events, such as scheduled system backup, security update, and initialisation of another collocated application could take a portion of resources that were occupied by the streaming system previously.

Resource-availability-aware management is particularly useful when there are no spare resources for the system to scale out due to the limitation of budget or other performance constraints. In that case, we are interested in changing the mapping of tasks to underlying resources in order to amortise the local resource shortage over the whole platform. The basic idea is that, if tasks of different operators in the topology process less workload accordingly, their resource consumption is expected to be reduced proportionally. Therefore, there is a possibility that a new task mapping can be found that satisfies the updated resource allocation constraints affected by the fluctuation of availability.

Such informed scheduling decision can allow the application performance to degrade gracefully without causing straggler problems discussed in Section 7.2.2.

7.2.4 Handling Increasing Heterogeneity

To improve application performance and provide new functions, an increasing number of heterogeneous hardware elements and network facilities have been employed in the deployment platform. However, the coordination of different types of resources also imposes new challenges to the design and implementation of a heterogeneity-aware resource provisioner and task scheduler. The potential challenge is that it would be impossible to describe the computing capability and the network capacity of heterogeneous resources using a unified measure. For example, we can hardly assert that a VM provisioned in clouds is always ten times faster than a mobile device regardless of the workload characteristics, either can we be certain about the exact bandwidth difference between a LAN and a wireless connection across regions when routing internal streams. Therefore, the anticipated heterogeneity-aware management framework should incorporate a customised resource model for each component to track its current resource usage and availability individually. This means that the profiling method should be customised for each device and the results are not necessarily shared across the whole deployment platform. As a consequence, a gossip-like negotiation protocol is also required for different components to converge to a scheduling plan that best suits their current situation.

7.2.5 Transparent State Management

An integrated state management system consists of two parts: (1) State elasticity, which allows dynamically scaling up and down the operator parallelism with a state repartitioning and migration mechanism, supporting the relocation of the operator internal state and providing a guarantee on the semantic correctness during the scaling process. (2) State persistence, which backups the computational states to persistent storage or a different node in order to mask the loss of states caused by JVM or node crashes. There are some preliminary efforts from both academia and industry towards achieving transparent state management [23,24,110], however, significant gaps still present in the following aspects. First, there is limited support for the diverse representation of operator state. In most existing state management frameworks, the abstraction and presentation of operator states are limited to key-value mapping for the ease of implementation. However, computational states can be organised in other forms such as graphs, hashes and trees that can hardly be indexed by certain keys. One promising research direction would be supporting arbitrary data structure for operator state representation while keeping the repartitioning and migration process entirely transparent to the end-users. The second gap is to reduce the excessive overhead of state migration, which could be overwhelming if the adaptation of resource provisioning, operator parallelism, and task scheduling have not considered the current state placement. Particularly, there is little research on gradual, stepwise task scheduling that eventually converges to the state satisfying the SLA requirements without incurring too much state migration overhead over a short adjustment period. In contrast, most scheduling algorithms in existence determine a new task mapping from scratch by re-applying the scheduling heuristic, re-invoking a graph partitioning algorithm, or re-conducting an exhausting search in the solution space.

7.2.6 Improving Energy Efficiency

Apart from reducing the total energy consumption through active workload consolidation, it is also of great interests to reduce the proportion of brown energy consumption through a proper resource management process. In the recent years, the energy supply of the infrastructure of streaming systems has been enriched by the green power generated from renewable sources such as sun, wind, water and biomass waste. Energy-efficient resource management intends to reduce the carbon emission and other negative impacts on our environment by scheduling computational-intensive tasks on nodes driven by green power, and assigning a large chunk of communication on links powered by green energy. In some cases, these two goals may conflict with each other when deciding the placement of particular streaming tasks, so a theoretical or empirical model on energy consumption is required to evaluate and compare different scheduling and resource provisioning plans and achieve the overall optimal in reducing the use of brown energy.

7.2.7 Improving Cost Efficiency with Different Pricing Models

The monetary cost of resource usages in clouds largely depends on the actual pricing and billing model selected by the users. Apart from the on-demand pricing model that has been intensively studied in the literature, a variety of alternative pricing models are also offered by mainstream cloud service provider like Amazon, Google and Microsoft to help users tailor their choices on resource provisioning and reduce the operational cost. To start with, reserved instances requiring a fixed-term contract are much cheaper than the on-demand instances, which makes them a good fit to host the baseline workload while leaving the on-demand instances for scaling out when needed. Also, the bidding price model can lower the cost of resource usage significantly as the instances are often hosted on the spare compute capacity in the cloud. However, with the price-bidding instances the streaming system needs to be prepared for interruptions under a fairly short notice, which imposes great challenges for the real-time system to adapt task placement and migrate the associated computational state accordingly. A comprehensive resource provisioning and task scheduling model combining the use of on-demand, reserved, and price-bidding resources is a promising research topic that would be welcomed by industry users.

7.2.8 Container-based Deployment

Containerisation of clouds allows the services and applications to adapt efficiently and operate at an unprecedented scale. Containers offer a logical packaging mechanism that decouples the applications from the environment in which they actually run, so there is a clean separation of concerns by clearly differentiating the procedures of application development and deployment. The ability of containers to run virtually anywhere and the isolation of the CPU, memory, storage, and network resources at the OS-level make it profitable to host streaming applications that are dynamic in nature [13].

However, resource management and scheduling in streaming systems over containers would require an overhaul in the design and implementation of existing DSMSs. The most prominent challenge is that transparent state management is hard to achieve over the container cloud that is initially designed to host state-less micro-services. The stateful streaming tasks may have to store their computational state externally which could raise new concerns about the performance of state access. Also, the flexibility of arbitrary placement and dynamic scaling of containers makes it hard to keep track of the destination of each internal stream, so that the tuple emitting logic needs to be refined to make sure that the provisioned containers are coordinated properly in sending and receiving real-time messages.

7.2.9 Integration of Different DSMSs

The diverse user requirements may require different DSMSs to be deployed at the same time to tackle different use cases. It then raises the questions of how to avoid performance interference between collocated DSMSs and how to select the appropriate middleware that best improves the user experience. There are some preliminary efforts to enable federated execution on top of different streaming engines [44, 102], however, they all lack the ability to theoretically formulate an engine selection problem for the arriving continuous queries, where the objective function and the resource and performance constraints that are caused by DSMS collocation are clearly defined. It is also interesting to investigate how to concatenate different DSMSs together to host a single streaming application, where each DSMS can handle the part of workload or streaming logic that it excels at processing.

7.3 Final Remarks

The wide adoption of cloud computing model has called for novel resource management methods to provide performance and reliability guarantees. In this thesis, we explored problems of robust resource management in distributed stream processing systems to ensure the articulated SLA with minimal resource consumption. It proposed a series of frameworks and prototypes to tackle challenges in application profiling, task scheduling, state management, and performance-oriented deployment. The research outcome of this thesis is instrumental in building the next generation of Data Stream Management Systems, which enables self-managing, SLA-aware, and robust resource management for stream processing systems deployed in computing clouds and Edge/Fog environments.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, and Others, "The Design of the Borealis Stream Processing Engine," in *Proceedings of the Conference on Innovative Data Systems Research*, vol. 5, 2005, pp. 277–289.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a New Model and Architecture for Data Stream Management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[3] T. Akidau, S. Whittle, A. Balikov, K. Bekirolu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, and P. Nordstrom, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[4] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive Control of Extreme-scale Stream Processing Systems," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE, 2006, pp. 71–77.

[5] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive Online Scheduling in Storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. ACM Press, 2013, pp. 207–218.

[6] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A Java-Compatible and Synthesizable Language for Heterogeneous Architectures," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 89–108, 2010.

[7] N. Backman, R. Fonseca, and U. Çetintemel, "Managing Parallelism for Stream Processing in the Cloud," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, ser. HotCDP '12. ACM Press, 2012, pp. 1–5.

[8] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in The Borealis Distributed Stream Processing System," *ACM Transactions on Database Systems*, vol. 33, no. 1, pp. 1–44, 2008.

[9] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive Input Admission and Management for Parallel Stream Processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. ACM Press, 2013, pp. 15–26.

[10] 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.

[11] P. Bellavista, A. Corradi, A. Reale, and N. Ticca, "Priority-Based Resource Scheduling in Distributed Stream Processing Systems for Big Data Applications," in *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 2014, pp. 363–370.

[12] M. Bilal and M. Canini, "Towards Automatic Parameter Tuning of Stream Processing Systems," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM Press, 2017, pp. 189–200.

[13] A. Brogi, G. Mencagli, D. Neri, J. Soldani, and M. Torquati, "Container-Based Support for Autonomic Data Stream Processing Through the Fog," in *Proceedings of the 23rd European Conference on Parallel Processing*, vol. 8374, 2018, pp. 17–28.

[14] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online Scheduling and Interference Alleviation for Low-Latency, High-Throughput Processing of Data Streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, 2017.

[15] M. Cammert, C. Heinz, J. Kramer, B. Seeger, S. Vaupel, and U. Wolske, "Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams," in

Proceedings of the 23rd IEEE International Conference on Data Engineering Workshop. IEEE, 2007, pp. 624–633.

[16] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel, "A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 230–245, 2008.

[17] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Unified Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, pp. 28–38, 2015.

[18] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed QoS-Aware Scheduling in Storm," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. ACM Press, 2015, pp. 344–347.

[19] ——, "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] ——, "Optimal Operator Replication and Placement for Distributed Stream Processing Systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 4, pp. 11–22, 2017.

[21] V. Cardellini, V. Grassi, F. L. Presti, and 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.

[22] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing," *Concurrency and Computation: Practice and Experience*, no. August, p. e4334, 2017.

[23] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic Stateful Stream Processing in Storm," in *Proceedings of the International Conference on High Performance Computing & Simulation*. IEEE, 2016, pp. 583–590.

[24] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. ACM Press, 2013, pp. 725–736.

[25] S. Caton and O. Rana, "Towards Autonomic Management for Cloud Services Based upon Volunteered Resources," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 9, pp. 992–1014, 2012.

[26] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch, "Adaptive Provisioning of Stream Processing Systems in the Cloud," in *Proceedings of the 28th IEEE International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 295–301.

[27] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate Latency Estimation in a Distributed Event Processing System," in *Proceedings of the 27th IEEE International Conference on Data Engineering*, ser. ICDE '11. IEEE, 2011, pp. 255–266.

[28] S. Chandrasekaran, M. A. Shah, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, and F. Reiss, "TelegraphCQ: Continuous Dataflow Processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. ACM Press, 2003, pp. 668–668.

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

[30] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. ACM Press, 2000, pp. 379–390.

[31] Z. Chen, J. Xu, J. Tang, K. Kwiat, C. Kamhoua, and C. Wang, "GPU-accelerated High-throughput Online Stream Data Processing," *IEEE Transactions on Big Data*, pp. 1–12, 2016.

[32] L. Cheng and 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*. IEEE, 2016, pp. 91–100.

[33] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, "Bin Packing Approximation Algorithms: Survey and Classification," in *Handbook of Combinatorial Optimization*, P. M. Pardalos, D.-Z. Du, and R. L. Graham, Eds. Springer New York, 2013, pp. 455–531.

[34] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A Decentralized Network Coordinate System," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 15–26, 2004.

[35] W. Dai, L. Qiu, A. Wu, and M. Qiu, "Cloud Infrastructure Resource Allocation for Big Data Applications," *IEEE Transactions on Big Data*, vol. 7790, no. c, pp. 1–11, 2016.

[36] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing using Dynamic Batch Sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–13.

[37] T. De 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2016, pp. 1–12.

[38] ——, "Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators," in *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2017, pp. 61–68.

[39] ——, "Proactive Elasticity and Energy Awareness in Data Stream Processing," *Journal of Systems and Software*, vol. 127, pp. 302–319, 2017.

[40] J. Desrosiers and 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.

[41] M. Dias de Assunção, A. da Silva Veith, and R. Buyya, "Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions," *Journal of Network and Computer Applications*, vol. 103, no. 7, pp. 1–17, 2018.

[42] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang, Z. Zhang, and H. Chao, "Optimal Operator State Migration for Elastic Data Stream Processing," *arxiv:1501.03619 [cs]*, pp. 1–15, 2015.

[43] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou, "Profiling Applications for Virtual Machine Placement in Clouds," in *Proceedings of the 4th IEEE International Conference on Cloud Computing*. IEEE, 2011, pp. 660–667.

[44] M. Duller, J. S. Rellermeyer, G. Alonso, and N. Tatbul, "Virtualizing Stream Processing," in *Proceedings of the 12th International on Middleware Conference*, vol. 7049 LNCS. Springer, 2011, pp. 269–288.

[45] R. Eidenbenz and T. Locher, "Task Allocation for Distributed Stream Processing," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[46] L. Eskandari, Z. Huang, and D. Eyers, "P-Scheduler: Adaptive Hierarchical Scheduling in Apache Storm," in *Proceedings of the Australasian Computer Science Week Multiconference*, ser. ACSW '16. ACM Press, 2016, pp. 1–10.

[47] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen, "P2G: A Framework for Distributed Real-Time Processing of Multimedia Data," in *Proceedings of the 40th International Conference on Parallel Processing Workshops*. IEEE, 2011, pp. 416–426.

[48] E. Feller, L. Rilling, and C. Morin, "Energy-Aware Ant Colony Based Workload Placement in Clouds," in *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing*. IEEE, 2011, pp. 26–33.

[49] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal, "Scalable Stateful Stream Processing for Smart Grids," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM Press, 2014, pp. 276–281.

[50] 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.

[51] 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.

[52] L. Fischer, T. Scharrenbach, and A. Bernstein, "Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling." in *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems*. CEUR-WS.org, 2013, pp. 81–96.

[53] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-Regulating Stream Processing in Heron," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, 2017.

[54] 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 35th IEEE International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 411–420.

[55] A. S. Fukunaga and R. E. Korf, "Bin-Completion Algorithms for Multicontainer Packing and Covering Problems," *Journal of Artificial Intelligence Research*, vol. 28, pp. 117–124, 2005.

[56] 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.

[57] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[58] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic Scaling for Data Stream Processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[59] B. Gedik, "Partitioning Functions for Stateful Data Parallelism in Stream Processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014.

[60] J. Ghaderi, S. Shakkottai, and R. Srikant, "Scheduling Storms and Streams in the Cloud," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 4, pp. 1–28, 2016.

[61] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness : Fair Allocation of Multiple Resource Types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 323–336.

[62] M. I. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, and H. Hoffmann, "A Stream Compiler for Communication-Exposed Architectures," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 291–303, 2002.

[63] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *ACM SIGMOD Record*, vol. 19, no. 2, pp. 102–111, 1990.

[64] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[65] T. Heinze, "Elastic Complex Event Processing," in *Proceedings of the 8th Doctoral Symposium on Middleware*, ser. MDS '11. ACM Press, 2011, pp. 1–6.

[66] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-Based Data Stream Processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. ACM Press, 2014, pp. 238–245.

[67] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-Aware Elastic Scaling for Distributed Data Stream Processing Systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. ACM Press, 2014, pp. 13–22.

[68] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer, "Elastic Complex Event Processing under Varying Query Load," in *Proceedings of the First International Workshop on Big Dynamic Distributed Data*, 2013, pp. 25–30.

[69] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-Scaling Techniques for Elastic Data Stream Processing," in *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 296–302.

[70] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online Parameter Optimization for Elastic Data Stream Processing," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. ACM Press, 2015, pp. 276–287.

[71] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer, "An Adaptive Replication Scheme for Elastic Data Stream Processing Systems," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. ACM Press, 2015, pp. 150–161.

[72] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-Adaptive Processing Graph with Operator Fission for Elastic Stream Processing," *Journal of Systems and Software*, vol. 127, pp. 205–216, 2017.

[73] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lecue, "Elastic Stream Processing for Distributed Environments," *IEEE Internet Computing*, vol. 19, no. 6, pp. 54–59, 2015.

[74] C. Hochreiner, M. Vogler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," in *Proceedings of the 9th IEEE International Conference on Cloud Computing*. IEEE, 2016, pp. 100–107.

[75] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. IEEE, 2009, pp. 214–223.

[76] M. R. Hoseiny Farahabady, H. R. Dehghani Samani, Y. Wang, A. Y. Zomaya, and Z. Tari, "A QoS-Aware Controller for Apache Storm," in *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*. IEEE, 2016, pp. 334–342.

[77] M. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari, "QoS- and Contention- Aware Resource Provisioning in a Stream Processing Engine," in *Proceedings of the IEEE International Conference on Cluster Computing*, no. Llc. IEEE, 2017, pp. 137–146.

[78] M. Humayoo, Y. Zhai, Y. He, B. Xu, and C. Wang, "Operator Scale Out Using Time Utility Function in Big Data Stream Processing," in *Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications*, ser. Lecture Notes in Computer Science, Z. Cai, C. Wang, S. Cheng, H. Wang, and H. Gao, Eds. Springer International Publishing, 2014, pp. 54–65.

[79] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, and S. Dustdar, "Deriving a Unified Fault Taxonomy for Event-Based Systems," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12. ACM Press, 2012, pp. 167–178.

[80] W. Hummer, B. Satzger, and S. Dustdar, "Elastic Stream Processing in the Cloud," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 5, pp. 333–345, 2013.

[81] S. Imai, S. Patterson, and C. A. Varela, "Maximum Sustainable Throughput Prediction for Data Stream Processing over Public Clouds," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017, pp. 1–10.

[82] A. Ishii and T. Suzumura, "Elastic Stream Computing with Clouds," in *Proceedings of the 4th IEEE International Conference on Cloud Computing*. IEEE, 2011, pp. 195–202.

[83] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar, "Building User-Defined Runtime Adaptation Routines for Stream Processing Applications," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1826–1837, 2012.

[84] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core," in *Proceedings of the ACM SIGMOD International conference on Management of Data*, ser. SIGMOD '06. ACM Press, 2006, pp. 431–442.

[85] Jiahua Fan, Haopeng Chen, and Fei Hu, "Adaptive Task Scheduling in Storm," in *Proceedings of the 4th International Conference on Computer Science and Network Technology*. IEEE, 2015, pp. 309–314.

[86] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu, "StroMAX: Partitioning-Based Scheduler for Real-Time Stream Processing System," in *Proceedings of the International Conference on Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, M. Li Lee, K.-L. Tan, and V. Wuwongse, Eds., vol. 3882. Springer Berlin Heidelberg, 2017, pp. 269–288.

[87] Y. Jiang, Z. Huang, and D. H. K. Tsang, "Towards Max-Min Fair Resource Allocation for Stream Big Data Analytics in Shared Clouds," *IEEE Transactions on Big Data*, vol. XX, no. XX, pp. 1–1, 2017.

[88] S. Kamburugamuve, L. Christiansen, and G. Fox, "A Framework for Real Time Processing of Sensor Data in the Cloud," *Journal of Sensors*, vol. 2015, pp. 1–11, 2015.

[89] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox, "Low Latency Stream processing: Apache Heron with Infiniband & Intel Omni-Path," in *Proceedings of the 10th International Conference on Utility and Cloud Computing*. ACM Press, 2017, pp. 101–110.

[90] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[91] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik, "COLA: Optimizing Stream Processing Applications via Graph Partitioning," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer, 2009, pp. 308–327.

[92] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, P. P. Jayaraman, J. Hosking, and E. Abbasnejad, "Resource Usage Estimation of Data Stream Processing Workloads in Datacenter Clouds," *arXiv:1501.07020 [cs]*, 2015.

[93] A. Khoshkbarforoushha, R. Ranjan, and P. Strazdins, "Resource Distribution Estimation for Data-Intensive Workloads: Give Me My Share and No One Gets Hurt!"

in *Communications in Computer and Information Science*. Springer, 2016, vol. 393, no. 90, pp. 228–237.

[94] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, "Balancing Load in Stream Processing with the Cloud," in *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops*. IEEE, 2011, pp. 16–21.

[95] R. K. Kombi, N. Lumineau, and P. Lamarre, "A Preventive Auto-Parallelization Approach for Elastic Stream Processing," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. IEEE, 2017, pp. 1532–1542.

[96] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM Press, 2015, pp. 239–250.

[97] M. Lee, M. Lee, S. J. Hur, and I. Kim, "Load Adaptive Distributed Stream Processing System for Explosive Stream Data," in *Proceedings of the 17th International Conference on Advanced Communication Technology*, vol. 5, no. 1. IEEE, 2015, pp. 753–757.

[98] B. Li, Y. Diao, and P. Shenoy, "Supporting Scalable Analytics with Latency Constraints," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1166–1177, 2015.

[99] C. Li, J. Zhang, and Y. Luo, "Real-Time Scheduling Based on Optimized Topology and Communication Traffic in Distributed Real-Time Computation Platform of Storm," *Journal of Network and Computer Applications*, vol. 87, no. 10, pp. 100–115, 2017.

[100] 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.

[101] ——, "Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing," *IEEE Transactions on Big Data*, vol. 7790, no. 99, pp. 1–12, 2016.

[102] H. Lim and S. Babu, "Execution and Optimization of Continuous Queries with Cyclops," in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD '13. ACM Press, 2013, pp. 1069–1072.

[103] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, "Scalable Distributed Stream Join Processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM Press, 2015, pp. 811–825.

[104] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "StreamScope: Continuous Reliable Distributed Processing of Big Data Streams," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 439–453.

[105] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. IEEE, 2017, pp. 372–382.

[106] Y. Liu, X. Shi, and H. Jin, "Runtime-Aware Adaptive Scheduling in Stream Processing," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 14, pp. 3830–3843, 2016.

[107] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An Elastic and Highly Available Streaming Service in the Cloud," in *Proceedings of the Joint Workshops on Extending Database Technology and Database Theory*, ser. EDBT-ICDT '12. ACM Press, 2012, pp. 55–60.

[108] B. Lohrmann, P. Janacik, and O. Kao, "Elastic Stream Processing with Latency Guarantees," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 399–410.

[109] B. Lohrmann, D. Warneke, and O. Kao, "Nephele Streaming: Stream Processing Under QoS Constraints at Scale," *Cluster Computing*, vol. 17, no. 1, pp. 61–78, 2014.

[110] K. G. S. Madsen, P. Thyssen, and Y. Zhou, "Integrating Fault-Tolerance and Elasticity in a Distributed Data Stream Processing System," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '14. ACM Press, 2014, pp. 1–4.

[111] K. G. S. Madsen, Y. Zhou, and L. Su, "Enorm: Efficient Window-Based Computation in Large-Scale Distributed Stream Processing Systems," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM Press, 2016, pp. 37–48.

[112] A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, and A. Brito, "Low-Overhead Fault Tolerance for High-Throughput Data Processing Systems," in *Proceedings of the 31st International Conference on Distributed Computing Systems*. IEEE, 2011, pp. 689–699.

[113] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-Aware Scheduling of MapReduce Jobs for Big Data Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10, pp. 2720–2733, 2015.

[114] R. Mayer, B. Koldehofe, and K. Rothermel, "Meeting Predictable Buffer Limits in the Parallel Execution of Event Processing Operators," in *Proceedings of the IEEE International Conference on Big Data*. IEEE, 2014, pp. 402–411.

[115] ——, "Predictable Low-Latency Event Detection with Parallel Complex Event Processing," *IEEE Internet of Things Journal*, vol. 2, no. 4, pp. 274–286, 2015.

[116] G. Mencagli, "A Game-Theoretic Approach for Elastic Distributed Data Stream Processing," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 11, no. 2, pp. 1–34, 2016.

[117] L. A. Moakar, A. Labrinidis, and P. K. Chrysanthis, "Adaptive Class-Based Scheduling of Continuous Queries," in *Proceedings of the 28th IEEE International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 289–294.

[118] J. Morales, E. Rosas, and N. Hidalgo, "Symbiosis: Sharing Mobile Resources for Stream Processing," in *Proceedings of the IEEE Symposium on Computers and Communications*, vol. Workshops. IEEE, 2014, pp. 1–6.

[119] M. Nardelli, "QoS-Aware Deployment of Data Streaming Applications over Distributed Infrastructures," in *Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 2016, pp. 736–741.

[120] S. Neuendorffer and K. Vissers, “Streaming Systems in FPGAs,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer Berlin Heidelberg, 2008, vol. 5114 LNCS, pp. 147–156.

[121] F. A. Nielsen, “A New Anew: Evaluation of a Word List for Sentiment Analysis in Microblogs,” *arXiv:1103.2903 [cs]*, vol. 718, pp. 93–98, 2011.

[122] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, “Samza: Stateful Scalable Stream Processing at LinkedIn,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[123] A. Papageorgiou, E. Poormohammady, and 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.

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

[125] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-Aware Operator Placement for Stream-Processing Systems,” in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE ’06. IEEE, 2006, pp. 49–49.

[126] D. Puiu, P. Barnaghi, R. Tnjes, D. Kmper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar, F. Gao, T. Iggena, T. L. Pham, C. S. Nechifor, D. Puschmann, and J. Fernandes, “Citypulse: Large scale data analytics framework for smart cities,” *IEEE Access*, vol. 4, no. 1, pp. 1086–1108, 2016.

[127] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, “Profiling Resource Usage for Mobile Applications: A Cross-layer Approach,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’11. ACM Press, 2011, pp. 321–334.

[128] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable Stream Computation in the Cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM Press, 2013, pp. 1–14.

[129] R. Ranjan, "Streaming Big Data Processing in Datacenter Clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, 2014.

[130] S. Ravindra, M. Dayarathna, and S. Jayasena, "Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams," in *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering*. ACM Press, 2017, pp. 91–102.

[131] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems," in *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, vol. 4290. Springer, 2006, pp. 322–341.

[132] S. Rizou, F. Diirr, and K. Rothermel, "Fulfilling End-To-End Latency Constraints in Large-Scale Streaming Environments," in *Proceedings of the IEEE International Performance Computing and Communications Conference*. IEEE, 2011, pp. 1–8.

[133] S. Rizou, F. Durr, and K. Rothermel, "Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks," in *Proceedings of the 19th International Conference on Computer Communications and Networks*. IEEE, 2010, pp. 1–6.

[134] S. Rizou, F. Durr, K. Rothermel, F. Durr, and K. Rothermel, "Providing QoS Guarantees in Large-Scale Operator Networks," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2010, pp. 337–345.

[135] M. Rychly, P. Koda, and P. Mr, "Scheduling Decisions in Stream Processing on Heterogeneous Clusters," in *Proceedings of the Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 2014, pp. 614–619.

[136] M. Rychlý, P. Škoda, and P. Smrž, "Heterogeneity-Aware Scheduler for Stream Processing Frameworks," *International Journal of Big Data Intelligence*, vol. 2, no. 2, pp. 70–80, 2015.

[137] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen, "Multi-query Stream Processing on FPGAs," in *Proceedings of the 28th IEEE International Conference on Data Engineering*. IEEE, 2012, pp. 1229–1232.

[138] K.-U. Sattler and F. Beier, "Towards Elastic Stream Processing: Patterns and Infrastructure," in *Proceedings of the First International Workshop on Big Dynamic Distributed Data*, 2013, pp. 49–54.

[139] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an Elastic Stream Computing Platform for the Cloud," in *Proceedings of the 4th IEEE International Conference on Cloud Computing*. IEEE, 2011, pp. 348–355.

[140] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic Scaling of Data Parallel Operators in Stream Processing," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[141] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-Parallelizing Stateful Distributed Streaming Applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. ACM Press, 2012, pp. 53–63.

[142] S. Schneider and K.-L. Wu, "Low-Synchronization, Mostly Lock-Free, Elastic Scheduling for Streaming Runtimes," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2017, pp. 648–661.

[143] Z. Sebepou and K. Magoutis, "CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators," in *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2011, pp. 145–156.

[144] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. V. Steen, "Cost-Effective Resource Allocation for Deploying Pub/Sub on Cloud," in *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 555–566.

[145] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly Available, Fault-Tolerant, Parallel Dataflows," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 2004, pp. 827–838.

[146] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating Performance Bottleneck Detection Using Search-Based Application Profiling," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. ACM Press, 2015, pp. 270–281.

[147] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. ACM Press, 2011, pp. 1–14.

[148] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti, "A Topology-Based Scaling Mechanism for Apache Storm," *International Journal of Network Management*, vol. 27, no. 3, p. e1933, 2017.

[149] A. Shukla and Y. Simmhan, "Model-driven Scheduling for Distributed Stream Processing Systems," *arXiv:1702.01785 [cs]*, 2017.

[150] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-Aware Scheduling of Streaming Applications Using Genetic Algorithm," *Procedia Computer Science*, vol. 108, no. 6, pp. 2240–2249, 2017.

[151] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-Directed Pipeline Parallelism," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. ACM Press, 2010, pp. 147–156.

[152] D. Sun, G. Fu, X. Liu, and H. Zhang, "Optimizing Data Stream Graph for Big Data Stream Computing in Cloud Datacenter Environments," *International Journal of Advancements in Computing Technology*, vol. 6, no. 5, pp. 53–65, 2014.

[153] D. Sun and R. Huang, "A Stable Online Scheduling Strategy for Real-Time Stream Computing Over Fluctuating Big Data Streams," *IEEE Access*, vol. 4, pp. 8593–8607, 2016.

[154] D. Sun, H. Yan, S. Gao, X. Liu, and R. Buyya, "Rethinking Elastic Online Scheduling of Big Data Streaming Applications over High-Velocity Continuous Data Streams," *The Journal of Supercomputing*, pp. 615–636, 2017.

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

[156] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, and K. Li, "Re-Stream: Real-Time and Energy-Efficient Resource Scheduling in Big Data Stream Computing Environments," *Information Sciences*, vol. 319, pp. 92–112, 2015.

[157] L. Thamsen, T. Renner, and O. Kao, "Continuously Improving the Resource Utilization of Iterative Parallel Dataflows," in *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 2016, pp. 1–6.

[158] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, "Resource Management for Bursty Streams on Multi-Tenancy Cloud Environments," *Future Generation Computer Systems*, vol. 55, pp. 444–459, 2016.

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

[160] J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl, "Optimized On-Demand Data Streaming from Sensor Nodes," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM Press, 2017, pp. 586–597.

[161] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosting Platforms," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 239–254, 2002.

[162] J. S. van der Veen, B. van der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically Scaling Apache Storm for the Analysis of Streaming Data," in *Proceedings of the First IEEE International Conference on Big Data Computing Service and Applications*. IEEE, 2015, pp. 154–161.

[163] S. Vijayakumar, Q. Zhu, and G. Agrawal, "Dynamic Resource Provisioning for Data Streaming Applications in a Cloud Environment," in *Proceedings of the Second IEEE*

International Conference on Cloud Computing Technology and Science. IEEE, 2010, pp. 441–448.

[164] R. Wagle, H. Andrade, K. Hildrum, C. Venkatramani, and M. Spicer, "Distributed Middleware Reliability and Fault Tolerance Support in System S," in *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*. ACM Press, 2011, pp. 335–346.

[165] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang, "OrientStream: A Framework for Dynamic Resource Allocation in Distributed Data Stream Management Systems," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM Press, 2016, pp. 2281–2286.

[166] ——, "Automating Characterization Deployment in Distributed Data Stream Management Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2669–2681, 2017.

[167] H. Wang and L.-S. Peh, "MobiStreams: A Reliable Distributed Stream Processing System for Mobile Devices," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 51–60.

[168] D. Warneke and O. Kao, "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 985–997, 2011.

[169] R. Weingärtner, G. B. Bräscher, and C. B. Westphall, "Cloud Resource Management: A Survey on Forecasting and Profiling Models," *Journal of Network and Computer Applications*, vol. 47, pp. 99–106, 2015.

[170] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer, "SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '08. Springer, 2008, pp. 306–325.

[171] Y. Wu and K.-L. Tan, "ChronoStream: Elastic Stateful Stream Computation in the Cloud," in *Proceedings of the 31st IEEE International Conference on Data Engineering*. IEEE, 2015, pp. 723–734.

[172] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. B. Zdonik, "Providing Resiliency to Load Variations in Distributed Stream Processing," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006, pp. 775–786.

[173] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-Aware Online Scheduling in Storm," in *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 535–544.

[174] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand," in *Proceedings of the IEEE International Conference on Cloud Engineering*. IEEE, 2016, pp. 22–31.

[175] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[176] Ying Xing, S. Zdonik, and Jeong-Hyon Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," in *Proceedings of the 21st International Conference on Data Engineering*. IEEE, 2005, pp. 791–802.

[177] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic Complex Event Processing Exploiting Prediction," in *Proceedings of the IEEE International Conference on Big Data*. IEEE, 2015, pp. 213–222.

[178] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM Press, 2013, pp. 423–438.

[179] J. Zhang, C. Li, L. Zhu, and Y. Liu, "The Real-Time Scheduling Strategy Based on Traffic and Load Balancing in Storm," in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2016, pp. 372–379.

[180] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A Hybrid Approach to High Availability in Stream Processing Systems," in *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*. IEEE, 2010, pp. 138–148.

[181] X. Zhao, S. Garg, C. Queiroz, and R. Buyya, "A Taxonomy and Survey of Stream Processing Systems," in *Software Architecture for Big Data and the Cloud*, 1st ed. Elsevier, 2017, pp. 183–206.

[182] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for Cloud Systems," in *Proceedings of the International Conference on Network and Service Management*. IEEE, 2010, pp. 9–16.

[183] Y. Zhou, B. C. Ooi, K.-l. Tan, and J. Wu, "Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System," in *On the Move to Meaningful Internet Systems*. Springer Berlin Heidelberg, 2006, pp. 54–71.

[184] Q. Zhu and G. Agrawal, "Resource Allocation for Distributed Streaming Applications," in *Proceedings of the 37th International Conference on Parallel Processing*. IEEE, 2008, pp. 414–421.

[185] 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.