© 2021 Le Xu

ELASTIC TECHNIQUES TO HANDLE DYNAMISM IN REAL-TIME DATA PROCESSING SYSTEMS

BY

LE XU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Indranil Gupta, Chair
Professor Klara Nahrstedt
Assistant Professor Jian Huang
Assistant Professor Shivaram Venkataraman, University of Wisconsin-Madison

ABSTRACT

Real-time data processing is a crucial component of cloud computing today. It is widely adopted to provide an up-to-date view of data for social networks, cloud management, web applications, edge, and IoT infrastructures. Real-time processing frameworks are designed for time-sensitive tasks such as event detection, real-time data analysis, and prediction. Compared to handling offline, batched data, real-time data processing applications tend to be long-running and are prone to performance issues caused by many unpredictable environmental variables, including (but not limited to) *job specification*, *user expectation*, and *available resources*.

In order to cope with this challenge, it is crucial for system designers to improve frameworks' ability to adjust their resource usage to adapt to changing environmental variables, defined as *system elasticity*. This thesis investigates how elastic resource provisioning helps cloud systems today process real-time data while maintaining predictable performance under workload influence in an *automated manner*. We explore new algorithms, framework design, and efficient system implementation to achieve this goal.

On the other hand, distributed systems today need to continuously handle various application specifications, hardware configurations, and workload characteristics. Maintaining stable performance requires systems to explicitly plan for resource allocation upon starting an application and tailor allocation dynamically during run time. In this thesis, we show how *achieving system elasticity can help systems provide tunable performance under the dynamism of many environmental variables without compromising resource efficiency.* Specifically, this thesis focuses on the two following aspects:

- *Elasticity-aware Scheduling:* Real-time data processing systems today are often designed in resource-, workload-agnostic fashion. As a result, most users are unable to perform resource planning before launching an application or adjust resource allocation (both within and across application boundaries) intelligently during the run. The first part of this thesis work (Stela [1], Henge [2], Getafix [3]) explores efficient mechanisms to conduct *performance analysis* while also enabling *elasticity-aware scheduling* in today's cloud frameworks.

- *Resource Efficient Cloud Stack:* The second line of work in this thesis aims to improve underlying *cloud stacks* to support self-adaptive, highly efficient resource provisioning. Today's cloud systems enforce full isolation that prevents resource sharing among

applications at a fine granularity over time. This work (Cameo [4], Dirigo) builds real-time data processing systems for emerging cloud infrastructures with high resource utilization through fine-grained resource sharing.

Given that the market for real-time data analysis is expected to increase by the annual rate of 28.2% and reach 35.5 billion by the year 2024 [5], improving system elasticity can introduce a significant reduction to deployment cost and increase in resource utilization. Our works improve the performances of real-time data analytics applications within resource constraints. We highlight some of the improvements as the following: i) Stela explores elastic techniques for single-tenant, on-demand dataflow scale-out and scale-in operations. It improves post-scale throughput by 45-120% during on-demand scale-out and post-scale throughput by 2-5× during on-demand scale-in. ii) Henge develops a mechanism to map application's performance into a unified scale of resource needs. It reduces resource consumption by 40-60% by maintaining the same level of SLO achievement throughout the cluster. iii) Getafix implements a strategy to analyze workload dynamically and proposes a solution that guides the systems to calculate the number of replicas to generate and the placement plan of these replicas adaptively. It achieves comparable query latency (both average and tail) by achieving 1.45-2.15× memory savings. iv) Cameo proposes a scheduler that supports data-driven, fine-grained operator execution guided by user expectations. It improves cluster utilization by 6× and reduces the performance violation by 72% while compacting more jobs into a shared cluster. v) Dirigo performs fully decentralized, function state-aware, global message scheduling for stateful functions. It is able to reduce tail latency by 60% compared to the local scheduling approach and reduce remote state accesses by 19× compared to the scheduling approach that is unaware of function states. These works can potentially lead to profound cost savings for both cloud providers and end-users.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Indranil Gupta. I have joined Indy's research group as a master's student, and he has been my first mentor in system research and introduced me to the field of distributed systems. Throughout the course of my Ph.D. journey, he has been encouraging and patient and helped me through many difficult times. His profound knowledge in research inspired me to continue building my research skills, and he also gave me the freedom to explore problems I'm interested in independently. Beyond research, he has also helped me tremendously with career advice as a researcher and an educator.

Next, I would like to extend my deepest appreciation to the rest of my thesis committee: Professor Klara Nahrstedt, Professor Jian Huang, and Professor Shivaram Venkataraman. Their insightful questions and comments helped to shape this thesis and inspire many future directions. I learned many things about systems from Klara and Jian's lectures. Klara encouraged me to switch to system research when I was admitted to a different program as a master's student. I would not be working on system research without her advice. Shivaram and I have collaborated on both Chi and Cameo projects starting from 2017. Not only did he provide invaluable feedback on the projects themself, but he has also been a great source of advice, both research and career -wise. He is one of the first people I would turn to for research feedbacks.

Throughout my Ph.D. journey, I am privileged to work with some of the best researchers through internships. I want to thank Jun Li, Lucy Chekasova, Mijung Kim, Harris Volos, Hernan Laffitte for hosting me at the internship in Hewlett Packard Enterprise (HPE) Lab. At the early stage of my Ph.D. journey, Lucy was the first researcher who taught me how to properly conduct a performance analysis on systems, while Jun and Mijung spent a significant amount of time teaching me how to analyze performance implications behind different configurations. Next, I would like to express my sincere gratitude to Rahul Potharaju and Kai Zeng, who mentored me at Microsoft. Rahul was my manager at the internship. From him, I learned how to distill interesting research ideas from in-production large-scale systems and present the research ideas to a broader audience. Kai was the mentor for two of my internships. Being the most hands-on mentor I've ever had, he taught me a full spectrum of skills, from how to develop an interesting story for my paper, to how to properly optimize every line of code. The success of this thesis would not be possible without them. Meanwhile, I would also like to thank members of Cloud and Information Services Lab (CISL) at

Microsoft, Carlo Curino, Matteo Interlandi, Alec Jindal, for their constructive feedback on my project and inspiring lunchtime conversation.

Next, I would like to thank all my peers, both at University of Illinois at Urbana-Champaign (UIUC) and at my internship, for supporting me throughout this journey. Firstly, I would like to thank Muntasir Raihan Rahman and Mainak Ghosh for sharing their wisdom as senior Ph.D. students and acting as my big brothers academically, and Wenting Wang, Luke Leslie, Shadi Noghabi, Faria Kalim, Shegufta Ahsan, Cong Xie, Rui Yang and Beomyeol Jeon for being geat friends through many of my ups and downs (mostly paper rejections). We share the same goal as and I have learned so much from them. I'll also extend this appreciation to other graduate and undergraduate students who have worked with Distributed Protocols Research Group (DPRG), for all the great times we spent together: Ala Alkhaldi, Hilfi Alkaff, Mayank Punidr, Jerry Peng, Son Nguyen, Guangxiang Du, Shiv Verma, Sharanya Bathay, Richa Meherwal, Akshun Gupta, Pallavi Srivastava, Ashwini Raina, Xiaoyao Qian, Jayasi Mehar, Srujun Gupta, Mayank Bhatt, and Harshit Agarwal. Specifically, I would like to thank Wenting, Jerry, Faria, Sharanya, Richa, Mainak, Ashwini, and Xiaoyao for collaborating on research projects included in this thesis. Additionally, I would like to thank all friends I made during my internships: Pradeep Fernando, Moriz Hoffman, Luo Mai, Joana Trindade, Yunseong Lee, Alberto Scolari, Chenggang Wu, Li Su, Xiaoming Qin, and Zichao Zhang. It has been a privilege to work with you and a great joy to spend summertime together. Specifically, I would like to thank Pradeep, Luo, Li, Xiaoming, and Zichao for collaborating on my past projects. I'm grateful to Microsoft for continuing to support my projects after my internship.

I began my undergraduate study at UIUC starting 2009, and I feel honored to be instructed by many professors and instructors from the Computer Science Department. I would like to thank Lawrance Angrave, who taught an excellent introductory CS course that made me decide to switch to study computer science. I would also like to thank Professor Brighten Godfrey, who introduced me to computer system research. His class was the first reason that I became a researcher in computer systems. I'd also like to show my appreciation to Professor Saurabh Sinha, Professor Roy Campbell, and Charles Blatti, who gave me the freedom to explore my research interest and encouraged me to switch my research area to distributed systems. During my study at UIUC, I'm grateful to receive tremendous help from the department advisors and staff: Viveka Kudaligama, Kara MacGregor, Mary Beth Kelly, Kathy Runk, and Maggie Chappell.

Finally, I would like to show my gratitude to my friends and family. I want to thank Silu Huang, Yunhui Long, Mengjia Yan, Yi Zhang, Liqi Xu for all the Friday dinner gatherings, weekend movie nights, and many road trips we had together. I'm indebted to my parents,

Li Zhou and Hui Xu, for giving me everything they have. Getting a doctoral degree in the United States was not something they had planned for me when I was little, but they chose to support my decision anyway. I dedicate this thesis to them.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION . . . 1
1.1 Background and Motivation . . . 1
1.2 Thesis Contributions . . . 3
1.3 Roadmap . . . 6

CHAPTER 2 STELA: ENABLING STREAM PROCESSING SYSTEMS TO SCALE-IN AND SCALE-OUT ON-DEMAND . . . 7
2.1 Introduction . . . 7
2.2 Stela Policy and the ETP Metric . . . 9
2.3 Implementation . . . 16
2.4 Evaluation . . . 20
2.5 Related Work . . . 24
2.6 Conclusion . . . 26

CHAPTER 3 HENGE: INTENT-DRIVEN MULTI-TENANT STREAM PROCESSING . . . 27
3.1 Introduction . . . 27
3.2 Overview of Henge . . . 28
3.3 Unifying User Requirements . . . 30
3.4 Juice as a Performance Indicator . . . 32
3.5 Evaluation . . . 33
3.6 Conclusion . . . 34

CHAPTER 4 POPULAR IS CHEAPER: CURTAILING MEMORY COSTS IN INTERACTIVE ANALYTICS ENGINES. . . . 36
4.1 Introduction . . . 36
4.2 The Replication Strategy . . . 37
4.3 Reducing Network Transfer through Matching . . . 39
4.4 Dynamic Replication Comparing To Prior Approach . . . 40
4.5 Conclusion . . . 42

CHAPTER 5 MOVE FAST AND MEET DEADLINES: FINE-GRAINED REAL-TIME STREAM PROCESSING WITH CAMEO. . . . 43
5.1 Introduction . . . 43
5.2 Background and Motivation . . . 45
5.3 Design Overview . . . 48
5.4 Scheduling Policies in Cameo . . . 49
5.5 Scheduling Mechanisms in Cameo . . . 54

5.6 Experimental Evaluation 60
5.7 Related Work 69
5.8 Conclusion 70

CHAPTER 6 SCHEDULING PERFORMANCE CRITICAL STATEFUL FUNCTIONS WITH DIRIGO 71
6.1 Introduction 71
6.2 Motivation 72
6.3 Related Works 74
6.4 Target Workloads 75
6.5 Scheduler Design 75
6.6 Scheduling Policies 81
6.7 Evaluations 87
6.8 Discussion 101
6.9 Conclusion 103

CHAPTER 7 CONCLUSION AND FUTURE WORK 104
7.1 Summary of Contributions 104
7.2 Future Work 105

REFERENCES 108

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

The past decade has seen the growth of real-time data processing [6], powered particularly by the development of *real-time data analytics systems*, including stream processing systems like [7, 8, 9, 10], and OLAP/real-time analytics systems like Druid [11, 12, 13, 14, 15]. These systems are widely adopted by companies to provide sub-second results based on real-time data input [16, 17, 18, 19]. Many of these frameworks handle a variety of real-time data processing workloads that can nowadays be characterized by 3Vs [20] — *Volume*, *Velocity*, and *Variety*. Provisioning resources under such variability leads to unprecedented challenges that have not been seen by their first-generation predecessors [21, 22, 23]. For each application, its deployment strategy cannot be determined statically before launch time and requires modification *dynamically*. Cloud platforms or services should elastically adjust resource allocation based on the needs of the applications it hosts. How to adaptively adjust resource allocation for various workloads becomes a major design decision for system designers. Herbst, Kounev, and Reussner [24] formally define *elasticity* as the ability for systems to *automatically* provision and de-provision resources to adapt to workload changes.

We focus on real-time data processing applications that are long-running and prone to the disturbances caused by environmental changes during the application's lifetime. For these applications, the ability to be elastic is especially critical for systems to provide *undisturbed* performance, correctness, and availability.

Thesis Statement: In this thesis, we show how achieving system elasticity can help systems provide tunable performance under the dynamism of many environmental variables without compromising resource efficiency. Specifically, we show that we could achieve elastic resource provisioning for real-time data processing by new techniques that work at multiple stages of resource provisioning at different layers of the cloud system stack.

We consider three environmental variables that affect job performance:

1. **Job Specification:** Job specification includes all characteristics describing the processing logic of a job (or an application) and the data it consumes. The variability in job specifications is one crucial factor that requires elastic adaptation from both applications and systems. In our work, we consider three varying job specifications: i) changing data volume over time, ii) various processing logic, and iii) changing number of users.

   Real-time data processing requires varying amounts of resources over time based on

the workload it is handling *at the moment*. For stream processing engines, the resource required for a job or an operator during each unit of time depends on the rate of data the application or the operator receives. This variability is further magnified by the varying number of users sharing the cluster. For interactive data analytics engines, the resource requirements vary with the number of queries it receives and the size of data range each query accesses.

2. **User Expectation:** The heterogeneity of user expectations is another factor that these systems should consider while performing elastic adaptation. User expectation indicates the performance expectation or target set by the user for a job (or application[1]). This factor is particularly dominant in resource provisioning for systems that handle multi-tenant scenarios (e.g., a streaming processing system may host applications that target users' latency expectations while others target throughput expectations).

   These expectations either become thresholds to trigger re-provisioning mechanisms or become performance targets of scheduling strategies during execution. Providing elastic resource provisioning towards each application introduces multiple challenges, e.g., How to map different types of performance expectations (e.g., expected latency, throughput, etc.) to resource needs? How to provide immediate resources to a running job that receives data that targets a strict user expectation without resource over-provisioning?

3. **Available Resource:** We also consider resources available to be another varying constraint that requires adaptation. Users can choose to modify the cluster setting due to changing workload expectations or budget changes. In our work, total resources might be static or elastic. We consider scenarios such as changing cluster size or clusters composed of machines with different configurations.

Steps of Elastic Resource Provisioning: Achieving resource provisioning elasticity for real-time data processing applications typically requires systems to take the following steps:

1. Detecting and modeling available resources;
2. Translating user expectations to resource needs;
3. Performing corresponding scheduling actions, including
   (a) Task placement

[1] We use job and application interchangeably.

(b) Task ordering

(c) Determining execution granularity

4. Interpreting scheduling feedback and determining future actions.

Each of these steps introduces many research challenges regarding both the *policies* and the *mechanisms* for systems to achieve elasticity. We introduce four problems that we focused on in our prior research.

1.2 THESIS CONTRIBUTIONS

Figure 1.1 illustrates the layer of the system stack where each work focuses. We list all contributions of this thesis in this section:

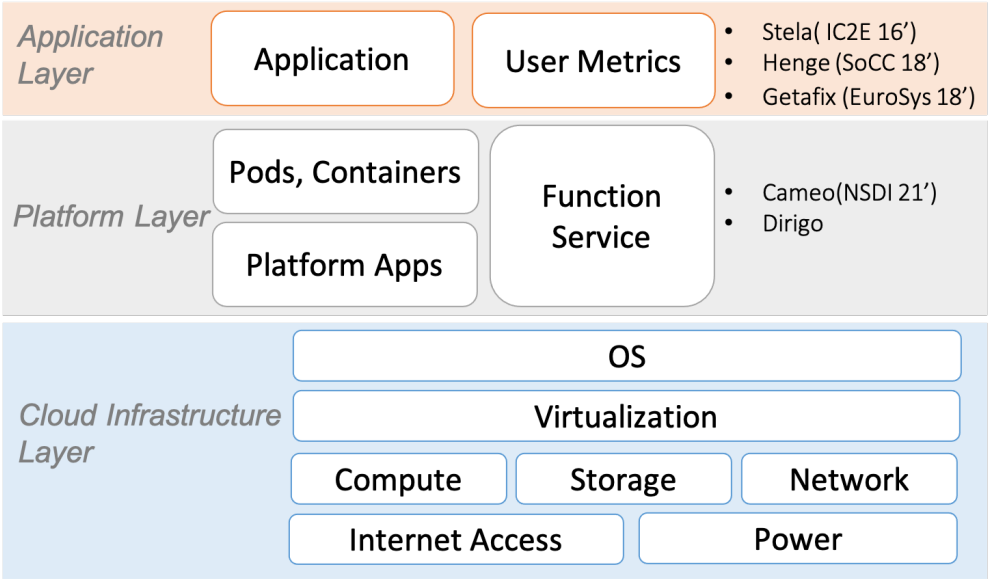


Figure 1.1: *Cloud stack layer each published work focuses on.*

On-demand Elasticity For Data Stream: Scaling out and scaling in streaming applications during execution time is important for long-running, streaming processing applications. As the data input to these applications is typically unpredictable in terms of speed and skewness, resource needs for these applications also change after it is first deployed. Therefore, these applications usually require constant monitoring and need to be scale-out to more machines *on-demand* when not enough resources are provisioned, or vice versa. However, existing distributed stream processing systems used in the industry largely lack the ability to *seamlessly* and *efficiently* scale the number of servers on-demand. And most of the prior

approaches we have seen focused on scale-out elastically (or -in) *automatically without* a restriction on the post-scale resource setting [25, 26, 27, 28, 29, 30] (e.g., not limiting the number of machines the applications can scale out to). In our work, Stela [1], built on Storm [7], we focus specifically on the *selection* and *placement* of scale-out (or scale-in) operators during an *on-demand* scale-out (or scale-in) to maximize post-scale throughput and minimize intrusion to the running application.

Workload Adaptation Under Various User Expectations: In Stela, we focused on elastic scale-in or scale-out towards a single data stream application. In the next work, Henge [2], we broaden our scenario to multi-tenant settings. Multi-tenant deployment in stream processing systems (i.e., multiple streaming applications share a single cluster) is a powerful technique to reduce resource consumption. However, naïve resource sharing in a streaming engine results in unpredictable behavior. This is because most real-time data processing engines today do not provide native performance isolation. In Henge, we provide a scheduling solution, built on top of Storm that supports cluster sharing among jobs with different *types* of user expectations. Henge defines *job intent* — a metric that measures the effectiveness of a certain application that satisfies its user-defined SLO. Henge uses job intent to identify jobs that require extra resources to achieve their expectations and jobs that can occupy under-utilized resources that can potentially scale in. Then it uses its reconfiguration mechanism to improve the overall success rate of achieving job intent across all applications sharing the cluster.

Adaptive Replication Under Skewed Popularity Access Pattern: In both Stela and Henge, we did not make any assumptions on input data's ingestion pattern (in terms of volume and locality). In this work, we leverage the skewed ingestion pattern observed from the production trace [31] and develop a data-popularity-aware replication strategy. Getafix [3] focuses on both shortening query makespan and memory consumption for interactive data analytics engines. It performs online data replication and placement based on a statically optimal strategy and adaptive to access pattern changes during run time.

Data-driven, Fine-grained Operator Scheduling: Today's most popular stream processing engines use a "slot-based" approach to achieve job-level and operator-level isolation and rely on users to manually configure the number of resources provisioned through resource managers [32, 33, 34]. Here slot-based approach means operators are assigned isolated resources. Without system elasticity, this deployment strategy results in severe over-provisioning as users typically estimate resource need based on workload peaks to avoid congestion. As we have discovered in Henge, existing solutions that aim to remedy this issue are largely *application-centric* — the platform generally assumes achieving performance expectations is part of applications or users' responsibility. Therefore, prior

solutions that have been proposed to solve this issue [35, 36, 37, 38, 39, 40] largely involve *online reconfiguration* for running pipelines. In these works, underlying engines are assumed to be black boxes that provide performance metrics, while users utilize these metrics to generate diagnoses [1, 36, 37, 38, 39, 41, 42, 43] that triggers pipeline reconfiguration [17, 44, 45, 46, 47, 48, 49, 50, 51]. In Cameo [4], we change our perspective and question the design of the *execution model* of existing stream processing engines. Cameo is a scheduler designed for stream processing engines that schedules each *operator execution* based on *data to be processed*.

Scheduling Real-time Dataflow Stateful Functions: We envision that we should build future real-time data processing engines with an event-driven, serverless architecture. All real-time data processing pipelines should be mapped to a chain of serverless functions. Building an event-driven service for real-time data processing applications requires fine-grained operator scheduling with the awareness of performance requirements and applications processing semantics. In Cameo [4], we explore the benefit of priority-based scheduling within a machine by building a scheduler for real-time dataflow operators. In Dirigo, we further explore this vision by building an event-driven scheduling framework that allows transparent reallocation of functions and runtime-managed function state storage. We explore several scheduling philosophies and how well they can support performance constraints and functions with various state accesses.

Achieving resource provisioning elasticity for real-time data processing introduces many research challenges regarding the *policies* and the *techniques* for systems to achieve elasticity. We argue that these research challenges occur during the following four steps:

- (1) Detecting and modeling available resources;
- (2) Translating user expectations to resource needs;
- (3) Performing corresponding scheduling actions, including
  - a. Task placement
  - b. Task ordering
  - c. Determining execution granularity
- (4) Interpreting scheduling feedback and determining future actions.

We map existing works into the types of environmental variability discussed by each, as well as the particular stages of resource provisioning each of these works contributed to in Table 1.2.

Name	Environmental Variability	Stages of Resource Provisioning
Stela	Resource Available, Workload Specification	1, 3 a, 3 b, 4
Henge	User Expectation, Workload Specification	2, 3 c, 4
Getafix	Resource Available, Workload Specification	1, 3 a, 4
Cameo	User Expectation, Workload Specification	2, 3 b, 3 c
Dirigo	User Expectation, Resource Available	1, 3 a, 3 b, 3 c

Figure 1.2: *List of past works, each mapped to the types of environmental variability discussed by the work, and stages of provisioning process targeted by each solution.*

1.3 ROADMAP

Chapter 2 describes Stela and ETP metrics and their on-demand scale-out and scale-in mechanisms in detail. Chapter 3 describes Henge and its Juice metrics and techniques to unify user requirements by translating topology utility. Chapter 4 introduces Getafix and its ModifiedBestFit algorithm to bin-pack and replicate data segments. Chapter 5 describes the Cameo framework and its technique to perform priority-based scheduling based on user-specified requirements. Chapter 6 presents the architecture and scheduling policies we explore through Dirigo. We conclude this thesis and introduce future directions in Chapter 7.

CHAPTER 2: STELA: ENABLING STREAM PROCESSING SYSTEMS TO SCALE-IN AND SCALE-OUT ON-DEMAND

2.1 INTRODUCTION

As our society enters an age dominated by digital data, we have seen unprecedented levels of data in terms of volume, velocity, and variety. Processing huge volumes of high-velocity data in a timely fashion has become a major demand. According to a recent article by BBC News [52], in the year 2012, 2.5 Exabytes of data was generated everyday, and 75% of this data is unstructured. The volume of data is projected to grow rapidly over the next few years with the continued penetration of new devices such as smartphones, tablets, virtual reality sets, wearable devices, etc.

In the past decade, distributed batch computation systems like Hadoop [53] and others [54][55][56][57] have been widely used and deployed to handle big data. Customers want to use a framework that can process large dynamic streams of data on the fly and serve results with high throughput. For instance, Yahoo! uses a stream processing engine to perform for its advertisement pipeline processing, so that it can monitor ad campaigns in real-time. Twitter uses a similar engine to compute trending topics [58] in real time.

To meet this demand, several new stream processing engines have been developed recently, and are widely in use in industry, e.g., Storm [58], System S [29], Spark Streaming [10], and others [59][27][60]. Apache Storm is the most popular among these. A Storm application uses a directed graph (dataflow) of operators (called "bolts") that runs user-defined code to process the streaming data.

Unfortunately, these new stream processing systems used in industry largely lack an ability to *seamlessly* and *efficiently* scale the number of servers in an *on-demand* manner. On-demand means that the scaling is performed when the user (or some adaptive program) requests to increase or decrease the number of servers in the application Today, Storm supports an on-demand scaling request by simply unassigning all processing operators and then reassigning them in a round robin fashion to the new set of machines. This is not seamless as it interrupts the ongoing computation for a long duration. It is not efficient either as it results in sub-optimal throughput after the scaling is completed (as our experiments show later).

Scaling-out and -in are critical tools for customers. For instance, a user might start running a stream processing application with a given number of servers, but if the incoming data rate rises or if there is a need to increase the processing throughput, the user may wish to add a few more servers (scale-out) to the stream processing application. On the

other hand, if the application is currently under-utilizing servers, then the user may want to remove some servers (scale-in) in order to reduce dollar cost (e.g., if the servers are VMs in AWS [61]). Supporting on-demand scale-out is preferable compared to over-provisioning which uses more resources (and money in AWS deployments), while on-demand scale-in is preferable to under-provisioning.

On-demand scaling operations should meet two goals: 1) the post-scaling throughput (tuples per sec) should be optimized and, 2) the interruption to the ongoing computation (while the scaling operation is being carried out) should be minimized. We present a new system, named Stela (STream processing ELAsticity), that meets these two goals. For scale-out, Stela carefully selects which operators (inside the application) are given more resources, and does so with minimal intrusion. Similarly, for scale-in, Stela carefully selects which machine(s) to remove in a way that minimizes the overall detriment to the application's performance.

To select the best operators to give more resources when scaling-out, Stela uses a new metric called *ETP (Effective Throughput Percentage)*. The key intuition behind ETP is to capture those operators (e.g., bolts and spouts in Storm) that are both: i) congested, i.e., are being overburdened with incoming tuples, and ii) affect throughput the most because they reach a large number of sink operators. For scale-in, we also use an ETP-based approach to decide which machine(s) to remove and where to migrate operator(s).

The ETP metric is both hardware- and application- agnostic. Thus Stela neither needs hardware profiling (which can be intrusive and inaccurate) nor knowledge of application code.

Existing work on elasticity in System S [25][26], StreamCloud (elasticity in Borealis) [49], Stormy [60] and [37] propose the use of metrics such as the congestion index, throughput, CPU, latency or network usage, etc. These metrics are used in a closed feedback loop, e.g., under congestion, System S determines *when* the parallelism (number of instances of an operator) should increase, and then does so for *all* congested operators. This is realistic only when infinite resources are available. Stela assumes finite resources (fixed number of added machines or removed machines, as specified by the user), and thus has to solve not only the "when" problem, but also the harder problem of deciding *which* operators need to get/lose resources. We compare Stela against the closest-related elasticity techniques from literature, i.e., [62].

The design of Stela is generic to any data flow system (Section 2.2.1). For concreteness, we integrated Stela into Apache Storm. We present experimental results using micro-benchmark Storm applications, as well as production applications from industry (Yahoo! Inc. and IBM [25]). Our experiments show that Stela's scale-out operation reduces interruption time to

a fraction as low as 12.5% that of Storm and achieves throughput that is about 21-120% higher than Storm's. Stela's scale-in operation performs 2X-5X better than Storm's default strategy. We believe our metric can be applied to other systems as well.

The contributions of our work are: 1) development of the novel metric, ETP, that captures the "importance" of an operator, 2) to the best of knowledge, this is the first work to describe and implement on-demand elasticity within Storm, and 3) evaluation of our system on both micro-benchmark applications and on applications used in production.

2.2 STELA POLICY AND THE ETP METRIC

In this section, we first define our data stream processing model. Then, we focus on Stela scale-out and how it uses the ETP metric. Finally we discuss scale-in.

2.2.1 Data Stream Processing Model and Assumptions

In this paper, we target distributed data stream processing systems that represent each application as a directed acyclic graph (DAG) of *operators*. An operator is a user-defined logical processing unit that receives one or more streams of *tuples*, processes each tuple, and outputs one or more streams of tuples. We assume operators are stateless. We assume that tuple sizes and processing rates follow an ergodic distribution. These assumptions hold true for most Storm topologies used in industry. An example of this model is shown in Figure 2.1. Operators that have no parents are *sources* of data injection, e.g., 1. They may read from a Web crawler. Operators with no children are *sinks*, e.g., 6. The intermediate operators (e.g., 2-5) perform processing of tuples. Each sink outputs data (e.g., to a GUI or database), and the application throughput is the sum of throughputs of all sinks in the application. An application may have multiple sources and sinks.

An *instance* (of an operator) is an instantiation of the operator's processing logic and is the physical entity that executes the operator's logic. The number of instances is correlated with the operator's parallelism level. For example, in Storm, these instances are called "executors" (Section 2.3.1).

2.2.2 Stela: Scale-Out Overview

In this section, we give an overview of how Stela supports scale-out. When the user requests a scale-out with a given number of new machines Stela needs to decide which operators to give more resources to, by increasing their parallelism.

Stela first identifies operators that are congested based on their input and output rates. Then it calculates a per-operator metric called *Expected Throughput Percentage (ETP)*. ETP takes the topology into account: it captures the percentage of total [1] application throughput (across all sinks) that the operator has direct impact on, but ignores all down-stream paths in the topology that are already congested. This ensures that giving more resources to a congested operator with higher ETP will improve the effect on overall application throughput. Thus Stela increases the parallelism of that operator with the highest ETP (from among those congested). Finally Stela recalculates the updated execution speed and *Projected ETP* (given the latest scale-out) for all operators and selects the next operator to increase its parallelism, and iterates this process. To ensure load balance, the total number of such iterations equals the number of new machines added times average number of instances per machine pre-scale. We determine the number of instances to allocate a new machine as: $N_{instances} = (Total\ \#\ of instances)/(\#\ of\ machines)$, in other words $N_{instances}$ is the average number of instances per machine prior to scale-out. This ensures load balance post-scale-out. The schedule of operators on existing machines is left unchanged.

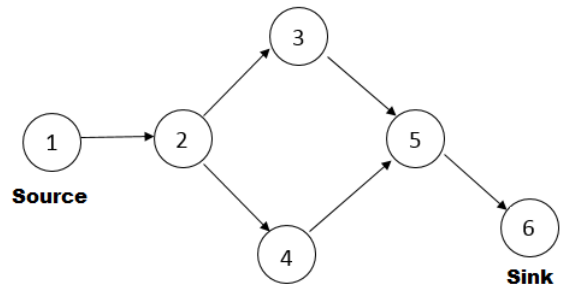


Figure 2.1: *An Example Of Data Stream Processing Application.*

The ETP approach is essentially a greedy approach because it assigns resources to the highest ETP operator in each iteration. Other complex approaches to elasticity may be possible, including graph theory and max-flow techniques–however these do not exist in literature yet and would be non-trivial to design. While we consider these to be interesting directions, they are beyond the scope of this paper.

2.2.3 Congested Operators

Before calculating ETP for each operator, Stela determines all congested operators in the graph by calling a CONGESTIONDETECTION procedure. This procedure considers an operator to be congested if the combined speed of its input streams is much higher than the speed at which the input streams are being processed within the operator. Stela measures the input rate, processing rate and output rate of an operator as the sum of input rates,

[1] This can also be generalized to a weighted sum of throughput across sinks.

processing rates and output rates, respectively, across all instances of that operator. An application may have multiple congested operators. In order to determine the best operators that should be migrated during a cluster scaling operation, Stela quantifies the impact of scaling an operator towards the application overall throughput by using the ETP metric.

Stela continuously samples the input rate, emit rate and processing rate of each operator in the processing the topology respectively. The input rate of an operator is calculated as the sum of emit rate towards this operator from all its parents. Stela uses periodic collection every 10 seconds and calculates these rates in a sliding window of recent tuples (of size 20 tuples). These values are chosen based on Storm's default and suggested values, e.g., the Storm scheduler by default runs every 10s.

When the ratio of input to processing exceeds a threshold `CongestionRate`, we consider that operator to be congested. An operator may be congested because it's overloaded by too many tuples, or has inefficient resources, etc. When the operator's input rate equals its processing rate, it is not considered to be congested. Note that we only compare input rates and processing rates (not emit rates) – thus this applies to operators like filter, etc., which may output a different rate than the input rate. The `CongestionRate` parameter can be tuned as needed and it controls the sensitivity of the algorithm: lower `CongestionRate` values result in more congested operators being captured. For Stela experiments, we set `CongestionRate` to be 1.2.

2.2.4 Effective Throughput Percentage (ETP)

Effective Throughput Percentage (ETP): To estimate the impact of each operator towards the application throughput, Stela introduces a new metric called Effective Throughput Percentage (ETP). An operator's ETP is defined as the percentage of the final throughput that would be affected if the operator's processing speed were changed.

The ETP of an operator o is computed as in Equation 2.1:

$$ETP_o = \frac{Throughput_{EffectiveReachableSinks}}{Throughput_{workflow}} \tag{2.1}$$

Here, $Throughput_{EffectiveReachableSinks}$ denotes the sum of throughput of all sinks reachable from o by at least one un-congested path, i.e., a path consisting only of operators that are not classified as congested. $Throughput_{workflow}$ denotes the sum throughput of all sink operators of the entire application. The algorithm to calculate an operator's ETP is shown in Algorithm 2.1. This algorithm does a depth first search throughout the application DAG, and calculates ETPs via a post-order traversal. ProcessingRateMap stores processing rates of all operators. Note that if an operator o has multiple parents, then the effect of o's ETP

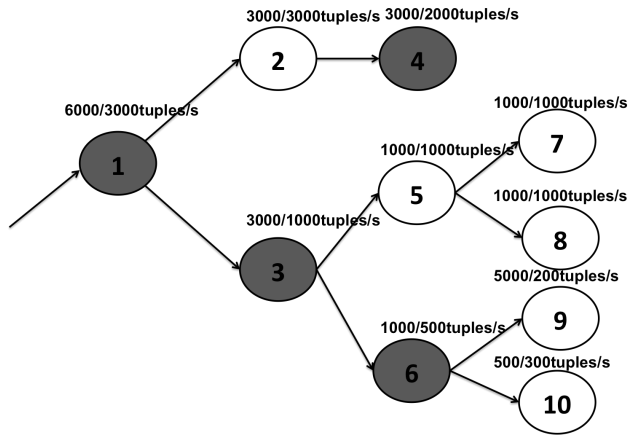


Figure 2.2: *A sliver of a stream processing application. Each operator is denoted by its input/execution speed. Shaded operators are congested.* `CongestionRate`=1.

is the same at each of its parents (i.e. it is replicated, not split).

While ETP is not a perfect measurement of post-scaling performance, it provides a good estimate. Our results in Section 4 show that using ETP is a reasonable and practical approach.

Algorithm 2.1 Find ETP of an operator o of the application

```
1: function FindETP(ProcessingRateMap)
2:     if o.child = null then return ProcessingRateMap.get(o)/ThroughputSum  ▷ o is a sink
3:     SubtreeSum ← 0;
4:     for each descendant child ∈ o do
5:         if child.congested = true then
6:             continue;  ▷ if the child is congested, give up the subtree rooted at that child
7:         else
8:             SubtreeSum+ = FindETP(child);
       return SubtreeSum
```

ETP Calculation Example and Intuition: We illustrate the ETP calculation using the example application in Figure 2.2. The processing rate of each operator is shown. In Figure 2.2, the operators congested are shown as shaded, i.e. operators 1, 3, 4 and 6. The total throughput of the workflow is calculated as the sum of throughput of sink operators 4, 7, 8, 9 and 10 as $Throughput_{workflow}$=4500 tuples/s.

Let us calculate the ETP of operator 3. Its reachable sink operators are 7, 8, 9 and 10.

Of these only 7 and 8 are considered to be the "effectively" reachable sink operators, as they are both reachable via an un-congested path. Thus, increasing the speed of operator 3 will improve the throughput of operators 7 and 8. However, operator 6 is a non-effective reachable for operator 3, because operator 6 is already congested – thus increasing operator 3's resources will only increase operator 6's input rate and make operator 6 further congested, without improving its processing rate. Thus, we ignore the subtree of operator 6 when calculating 3's ETP. The ETP of operator 3 is: $ETP_3 = (1000 + 1000)/4500 = 44\%$.

Similarly, for operator 1, the sink operators 4, 7, 8, 9 and 10 are reachable, but none of them are reachable via a non-congested path. Thus the ETP of operator 1 is 0. Likewise, we can calculate the ETP of operator 4 as 44% and the ETP of operator 6 as 11%. Thus, the priority order for Stela to assign resources to these operators is: 3, 4, 6, 1.

2.2.5 Iterative Assignment and Intuition

During each iteration, Stela calculates the ETP for all congested operators. Stela targets the operator with the highest ETP and it increases the parallelism of the operator by assigning a new instance of that operator at the newly added machine. If multiple machines are being added, then the target machine is chosen in round-robin manner. Overall this algorithm runs $N_{instances}$ iterations to select $N_{instances}$ target operators (Section 2.2.1 showed how to calculate $N_{instances}$).

Algorithm 2.2 depicts the pseudocode for scale-out. In each iteration, Stela constructs a $CongestedMap$, as explained earlier in Section 2.2.3. If there are no congested operators in the application, Stela chooses a source operator as a target – this is done to increase the input rate of the entire application. If congested operators do exist, for each congested operator, Stela finds its ETP using the algorithm discussed in Section 2.2.4. The result is sorted into $ETPMap$. Stela chooses the operator that has the highest ETP value from $ETPMap$ as a target for the current iteration. It increases the parallelism of this operator by assigning one additional random instance to it, on one of the new machines in a round robin manner.

For the next iteration, Stela estimates the processing rate of the previously targeted operator o proportionally, i.e., if the o previously had an output rate E and k instances, then o's new *projected* processing rate is $E \cdot \frac{k+1}{k}$. This is a reasonable approach since all machines have the same number of instances and thus proportionality holds. Note that even though this may not be accurate, we find that it works in practice. Then Stela uses this to update the output rate for o, and the input rates for o's children (o's children's processing rates do not need updates as their resources remain unchanged. The same applies to o's grand-

descendants.). Stela updates emit rate of target operator in the same manner to ensure estimated operator submission rate can be applied.

Once this is done, Stela re-calculates the ETP of all operators by again using Algorithm 2.1 – we call these new ETPs as *projected* ETPs, or PETPs, because they are based on estimates. The PETPs are used as ETPs for the next iteration. These iterations are repeated until all available instance slots at the new machines are filled. Once this procedure is completed, the schedule is committed by starting the appropriate executors on new instances.

Algorithm 2.2 Stela: Scale-out

```
1:  function SCALE-OUT
2:      slot ← 0;
3:      while slot < N_instances do
4:          CongestedMap ← CONGESTIONDETECTION;
5:          if CongestedMap.empty == true then
             return source;                          ▷ none of the operators are congested
6:          for each operator o ∈ workflow do
7:              ETPMap ← FINDETP(Operator o);
8:          target ← ETPMap.max;
9:          ProcessingRateMap.update(target);
10:         EmitRateMap.update(target);             ▷ update the target execution rate
11:         slot++;
```

In Algorithm 2.2, procedure FindETP involves searching for all reachable sinks for every congested operator – as a result each iteration of Stela has a running time complexity of $O(n^2)$ where n is the number of operators in the workflow. The entire algorithm has a running time complexity of $O(m \cdot n^2)$, where m is the number of new instance slots at the new workers.

2.2.6 Stela: Scale-In

For scale-in, we assume the user only specifies the number of machines to be removed and Stela picks the "best" machines from the cluster to remove (if the user specifies the exact machines to remove, the problem is no longer challenging). We describe how techniques used for scale-out can also be for scale-in, particularly, the ETP metric. For scale-in we will not be merely calculating the ETP per operator but instead *per machine* in the cluster. That is, we first, calculate the *ETPSum* for each machine as in Equation 2.2:

$$ETPSum(machine_k) = \sum_{i=1}^{n} FindETP(FindComp(\tau_i)) \tag{2.2}$$

ETPSum for a machine is the sum of all ETP of instances of all operators that currently reside on the machine. Thus, for every instance, τ_i, we first find the operator that instance τ_i is an instance of (e.g., $operator_o$) and then find the ETP of that $operator_o$. Then, we sum all of these ETPs. ETPSum of a machine is thus an indication of how much the instances executing on that machine contribute to the overall throughput. The intuition is that a machine with lower ETPSum is a better candidate to be removed in a scale-in operation than a machine with higher ETPSum since the former influences less the application in both throughput and downtime.

Algorithm 2.3 Stela: Scale-in

```
 1: function SCALE-IN
 2:     for each Machine n ∈ cluster do
 3:         ETPMachineMap ← ETPMACHINESUM(n)
 4:     ETPMachineMap.sort()                    ▷ sort ETPSums by increasing order
 5:     REMOVEMACHINE(ETPMachineMap.first())
 6: function REMOVEMACHINE(Machine n, ETPMachineMap map)
 7:     for each instance τ_i on n do
 8:         if i > map.size then
 9:             i ← 0
10:         Machine x ← map.get(i)
11:         ASSIGN(τ_i, x)                      ▷ assigns instances to a round robin fashion
12:         i++
```

The SCALE-IN procedure of Algorithm 2.3 is called iteratively, as many times as the number of machines requested to be removed. The procedure calculates the ETPSum for every machine in the cluster and puts the machine and its corresponding ETPSum into the ETPMachineMap. The ETPMachineMap is sorted in increasing order of ETPSum values. The machine with the lowest ETPSum will be the target machine to be removed in this round of scale-in. Operators from the machine that is chosen to be removed are re-assigned to the remaining machines in the cluster, in a round robin fashion in increasing order of their ETPSum.

Performing operator migration to machines with lower ETPSum will have less of an effect on the overall performance since machines with lower ETPSum contribute less to the overall performance. This also helps shorten the amount of downtime the application experiences due to the rescheduling. This is because while adding new instances to a machine, existing computation may need to be paused for a certain duration. Instances on a machine with lower ETPSum contribute less to the overall performance and thus this approach causes lower overall downtime.

After this schedule is created, Stela commits it by migrating operators from the selected machines, and then releases these machines. Algorithm 2.3 involves sorting *ETPSum*, which results in a running time complexity of $O(nlog(n))$.

2.3 IMPLEMENTATION

We have implemented Stela as a custom scheduler inside Apache Storm [58].

2.3.1 Overview of Apache Storm

Storm Application: Apache Storm is a distributed real time processing framework that can process incoming live data [58]. In Storm, a programmer codes an application as a Storm *topology*, which is a graph, typically a DAG (While Storm topologies allow cycles, they are rare and we do not consider them in this paper.), of operators, sources (called spouts), and sinks. The operators in Storm are called bolts. The streams of data that flow between two adjacent bolts are composed of tuples. A Storm task is an instantiation of a spout or bolt.

Users can specify a *parallelism hint* to say how many executors each bolt or spout should be parallelized into. Users can also specify the number of tasks for the bolt or spout – if they don't, Storm assumes one task per executor. Once fixed, Storm does not allow the number of tasks for a bolt to be changed, though the number of executors is allowed to change, to allow multiplexing – in fact Stela leverages this multiplexing by varying the number of executors for congested bolts.

Storm uses *worker processes*. In our experiments, a machine may contain up to 4 worker processes. Worker processes in turn contain *executors* (equivalent to an "instance" in our model in Section 2.2.1). An executor is a thread that is spawned in a worker process. An executor may execute one or more tasks. If an executor has more than one task, the tasks are executed in a sequential manner inside. The number of tasks cannot be changed in Storm, but the number of executors executing the tasks can increase and decrease.

The user specifies in a *topology* (equivalent to a DAG) the bolts, the connections, and how many worker processes to use. The basic Storm operator, namely the bolt, consumes input streams from its parent spouts and bolts, performs processing on received data, and emits new streams to be received and processed downstream. Bolts may filter tuples, perform aggregations, carry out joins, query databases, and in general any user defined functions. Multiple bolts can work together to compute complex stream transformations that may require multiple steps, like computing a stream of trending topics in tweets from Twitter [58]. Bolts in Storm are stateless. Vertices 2-6 in Figure 2.1 are examples of Bolts.

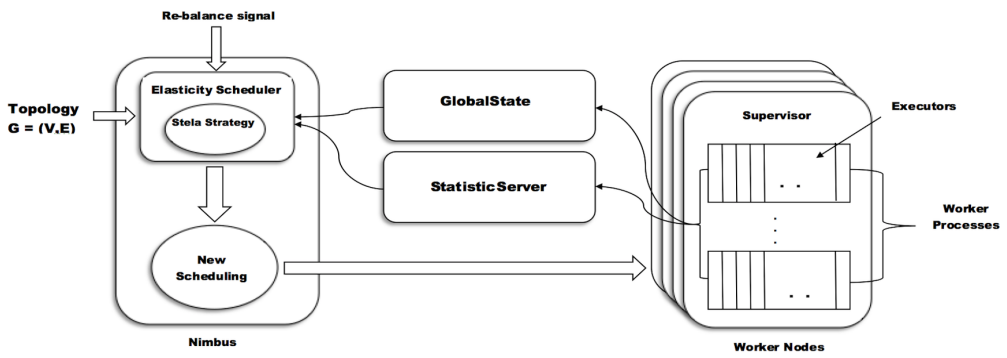


Figure 2.3: *Stela Architecture.*

Storm Infrastructure: A typical Storm Cluster has two types of machines: the master node, and multiple workers nodes. The master node is responsible for scheduling tasks among worker nodes. The master node runs a daemon called *Nimbus*. Nimbus communicates and coordinates with Zookeeper[63] to maintain a consistent list of active worker nodes and to detect failure in the membership.

Each server runs a worker node, which in turn runs a daemon called the *supervisor*. The supervisor continually listens for the master node to assign it tasks to execute. Each worker machine contains many worker processes which are the actual containers for tasks to be executed. Nimbus can assign any task to any worker process on a worker node. Each source (*spout*) and operator (bolt) in a Storm topology can be parallelized to potentially improve throughput. The user specifies the parallelization hint for each bolt and spout.

Storm Scheduler: Storm's default Storm scheduler (inside Nimbus) places tasks of all bolts and spouts on worker processes. Storm uses a round robin allocation in order to balance out load. However, this may result in tasks of one spout or bolt being placed at different workers. Currently, the only method for vanilla Storm to do any sort of scale-in or -out operation, is for the user to execute a *re-balance* operation. This re-balance operation simply deletes the current scheduling and re-schedules all tasks from scratch in a round robin fashion to the modified cluster. This is inefficient, as our experiments later show.

2.3.2 Core Architecture

Stela runs as a custom scheduler in a Java class that implements a predefined IScheduler interface in Storm. A user can specify which scheduler to use in a YAML formated configuration file call *storm.yaml*. Our scheduler runs as part of the Storm Nimbus daemon. The architecture of Stela's implementation in Storm is visually presented in Figure 2.3. It consists of three modules:

1. StatisticServer - This module is responsible for collecting statistics in the Storm cluster, e.g., throughput at each task, bolt, and for the topology. This data is used as input to congestion detection in Sections.

2. GlobalState - This module stores important state information regarding the scheduling and performance of a Storm Cluster. It holds information about where each task is placed in the cluster. It also stores statistics like sampled throughputs of incoming and outgoing traffic for each bolt for a specific duration, and this is used to determine congested operators as mentioned in Section 2.2.3.

3. Strategy - This module provides an interface for scale-out strategies to implement so that different strategies (e.g., Algorithm 2.2) can be easily swapped in and out for evaluation purposes. This module calculates a new schedule based on the scale-in or scale-out strategy in use and uses information from the Statistics and GlobalState modules. The core Stela policy (Section 2.2.1) and alternative strategies (Section 2.3.3) are implemented here.

4. ElasticityScheduler - This module is the custom scheduler that implements IScheduler interface. This class starts the StatisticServer and GlobalState modules, and invokes the Strategy module when needed.

When a scale-in or -out signal is sent by the user to the ElasticityScheduler, a procedure is invoked that detects newly joined machines based on previous membership. The ElasticityScheduler invokes the Strategy module, which calculates the entire new scheduling, e.g., for scale-out, it decides all newly created executors that need to be assigned to newly joined machines. The new scheduling is then returned to the ElasticityScheduler which atomically (at the commit point) changes the current scheduling in the cluster. Computation is thereafter resumed.

Fault-tolerance: When no scaling is occurring, failures are handled the same way as in Storm, i.e., Stela inherits Storm's fault-tolerance. If a failure occurs during a scaling operation, Stela's scaling will need to be aborted and restarted. If the scaling is already committed, failures are handled as in Storm.

2.3.3 Alternative Strategies

Initially, before we settled on the ETP design in Section 2.2, we attempted to design several alternative topology-aware strategies for scaling out. We describe these below, and

we will compare the ETP-based approach against the best of these in our experiments in Section 2.4. One of these strategies also captures existing work [62].

Topology-aware strategies migrate existing executors instead of creating more of them (as Stela does). These strategies aim to find "important" components/operators in a Storm Topology. Operators, in scale-out, deemed "important" are given priority for migration to the new worker nodes. These strategies are:

- Distance from spout(s) - In this strategy, we prioritize operators that are closer to the source of information as defined in Section 2.2.1. The rationale for this method is that if an operator that is more upstream is a bottleneck, it will affect the performance of all bolts that are further downstream.

- Distance to output bolt(s) - Here, higher priority for use of new resources are given to bolts that are closer to the sink or output bolt. The logic here is that bolts connected near the sink affect the throughput the most.

- Number of descendants - Here, importance is given to bolts with many descendants (children, children's children, and so on) because such bolts with more descendants potentially have a larger effect on the overall application throughput.

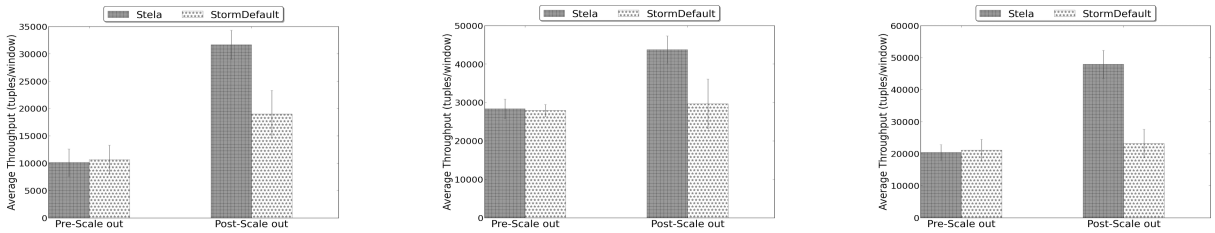
- Centrality - Here, higher priority is given to bolts that have a higher number of in- and out-edges adjacent to them (summed up). The intuition is that "well-connected" bolts have a bigger impact on the performance of the system then less well-connected bolts.

- Link Load - In this strategy, higher priority is given to bolts based on the load of incoming and outgoing traffic. This strategy examines the load of links between bolts and migrates bolts based on the status of the load on those links. Similar strategies have been used in [62] to improve Storm's performance. We implemented two strategies for evaluation: Least Link Load and Most Link Load. Least Link Load strategy sorts executors by the load of the links that it is connected to and starts migrating tasks to new workers by attempting to maximize the number of adjacent bolts that are on the same server. The Most Link Load strategy does the reverse.

During our experiments, we found that all the above strategies performed comparably for most topologies, however the Link Load based strategy was the only one that improved performance for the Linear topology. Thus, the Least Link Load based strategy is representative of strategies that attempt to minimize the network flow volume in the topology

Toplogy Type	# of tasks per Component	Initial # of Executors per Component	# of Worker Processes	Initial Cluster Size	Cluster Size after Scaling	Machine Type
Star	4	2	12	4	5	1
Linear	12	6	24	6	7	1
Diamond	8	4	24	6	7	1
Page Load	8	4	28	7	8	1
Processing	8	4	32	8	9	1
Network	8	4	32	8	9	2
Page Load Scale-in	15	15	32	8	4	1

Table 2.1: *Experiment Settings and Configurations.*



(a) Star Topology. (b) Linear Topology. (c) Diamond Topology.

Figure 2.4: *Scale-out: Throughput Behavior for Micro-benchmark Topologies. (Window size 10 seconds)*

schedule. Hence in the next section, we will compare the performance of the Least Link Load based strategy with Stela. This is thus a comparison of Stela against [62].

2.4 EVALUATION

Our evaluation is two-pronged, and consists of both microbenchmark topologies and real topologies (including two from Yahoo!). We adopt this approach due to the absence of standard benchmark suites (like TPC-H or YCSB) for stream processing systems. Our microbenchmarks include small topologies such as star, linear and diamond, because we believe that most realistic topologies will be a combination of these. We also use two topologies from Yahoo! Inc., which we call PageLoad topology and Processing topology, as well as a Network Monitoring topology [25] . We also present a comparison among Stela, the Link Load Strategy (Section 2.3.3 and [62]), and Storm's default scheduler (which is state of the art).

2.4.1 Experimental Setup

For our evaluation, we used two types of machines from Emulab [64] testbed to perform our experiments. Our typical Emulab setup consists of a number of machines running Ubuntu 12.04 LTS images, connected via a 100Mpbs VLAN. A type 1 machine has one 3 GHz processor, 2 GB of memory, and 10,000 RPM 146 GB SCSI disks. A type 2 machine has one 2.4 GHz quad core processor, 12 GB of memory, 750 GB SATA disks. The settings for all topologies tested are listed in Table 2.1. For each topology, the same scaling operations were applied to all strategies.

2.4.2 Micro-benchmark Experiments

Storm topologies can be arbitrary. To capture this, we created three micro-topologies that commonly appear as building blocks for larger topologies. They are:

- Linear - This topology has 1 source and 1 sink with a sequence of intermediate bolts.
- Diamond - This topology has 1 source and 1 sink, connected parallel via several intermediate bolts (Thus tuples will be processed via the path of "source - bolt - sink".
- Star - This topology has multiple sources connected to multiple sinks via a single unique intermediate bolt.

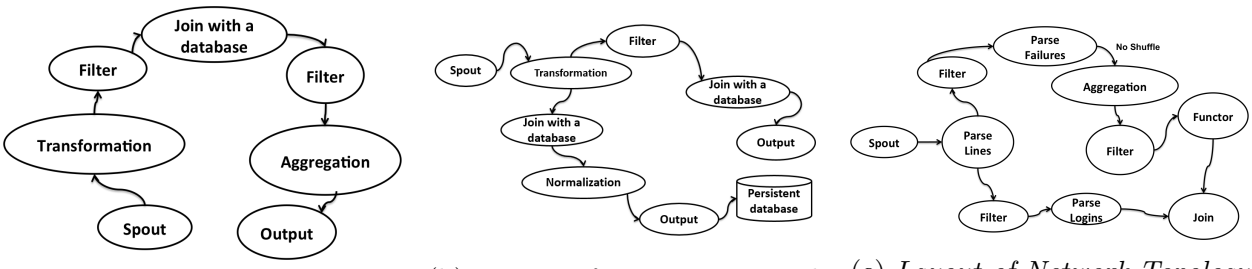
Figure 2.4a, 2.4b, 2.4c present the throughput results for these topologies. For the Star, Linear, and Diamond topologies we observe that Stela's post scale-out throughput is around 65%, 45%, 120% better than that of Storm's default scheduler, respectively. This indicates that Stela correctly identifies the congested bolts and paths and prioritizes the right set of bolts to scale out. The lowest improvement is for the Linear topology (45%) – this lower improvement is due to the limited diversity of paths, where even a single congested bolt can bottleneck the sink.

In fact, for Linear and Diamond topologies, Storm's default scheduler does not improve throughput after scale-out. This is because Storm's default scheduler does not increase the number of executors, but attempts to migrate executors to a new machine. When an executor is not resource-constrained and it is executing at maximum performance, migration doesn't resolve the bottleneck.

2.4.3 Yahoo Storm Topologies and Network Monitoring Topology

We obtained the layouts of two topologies in use at Yahoo! Inc. We refer to these two topologies as the Page Load topology and Processing topology (these are not the original

names of these topologies). The layout of the Page Load Topology is displayed in Figure 2.5a, the layout of the Processing topology is displayed in Figure 2.5b and the layout of the Network Monitoring topology is displayed in Figure 2.5c.

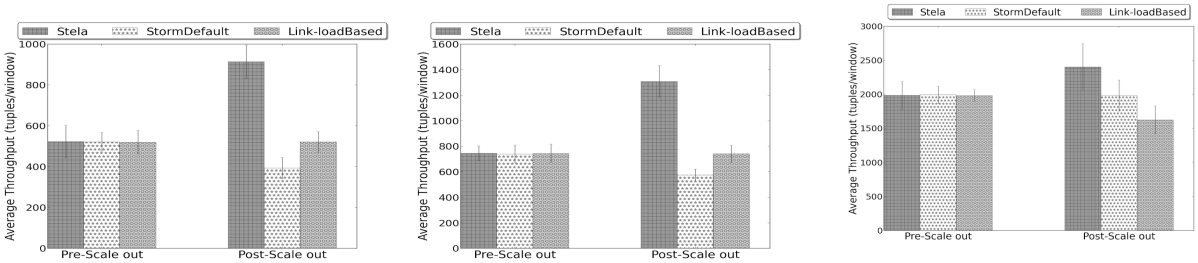


(a) *Layout of Page Load Topology.*

(b) *Layout of Processing Topology.*

(c) *Layout of Network Topology [25].*

Figure 2.5: *Two Yahoo! Topologies and a Network Monitoring Topology derived from [25]*



(a) *Page Load topology.*

(b) *Processing topology.*

(c) *Network Monitoring topology.*

Figure 2.6: *Scale-out: Throughput Behavior for Yahoo! Topologies and Network Monitoring Topology. (Window size 10 seconds)*

We examine the performance of three scale-out strategies: default, Link based (Section 2.3.3 and [62]), and Stela. The throughput results are shown in Figure 2.6. Recall that link load based strategies reduce the network latency of the workflow by co-locating communicating tasks to the same machine.

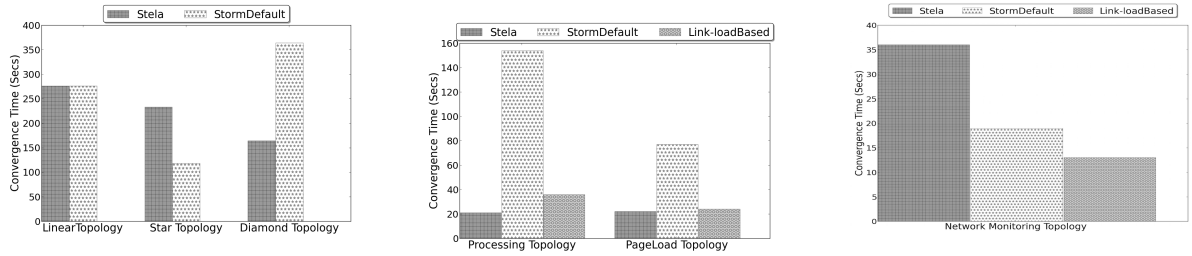
From Figure 2.6, we observe that Stela improves the throughput by 80% after a scale-out for both Yahoo topologies. In comparison, Least Link Load strategy barely improves the throughput after a scale-out because migrating tasks that are not resource-constrained will not significantly improve performance. The default scheduler actually decreases the throughput after the scale-out, since it simply unassigns all executors and reassigns all the executors in a round robin fashion to all machines including the new ones. This may cause machines with "heavier" bolts to be overloaded thus creating newer bottlenecks that are damaging to performance especially for topologies with a linear structure. In comparison, Stela's post-scaling throughput is about 125% better than Storm's post-scaling throughput

for both Page Load and Processing topologies – this indicates that Stela is able to find the most congested bolts and paths and give them more resources.

In addition to the above two topologies, we also looked at a published application from IBM [25], and we wrote from scratch a similar Storm topology (shown in Figure 2.5c). By increasing cluster size from 8 to 9, our experiment (Figure 2.6c) shows that Stela improves the throughput by 21% by choosing to parallelize the congested operator closest to the sink. In the meantime Storm default scheduler does not improve post scale throughput and Least Link Load strategy decreases system throughput.

2.4.4 Convergence Time

We measure interruption to ongoing computation by measuring the *convergence time*. The convergence time is the duration of time between when the scale-out operation starts and when the overall throughput of the Storm Topology stabilizes. Concretely, the convergence time duration stopping criteria are: 1) the throughput oscillates twice above and twice below the average of post scale-out throughput, and 2) the oscillation is within a small standard deviation of 5%. Thus a lower convergence time means that the system is less intrusive during the scale out operation, and it can resume meaningful work earlier.



(a) *Throughput Convergence Time for Micro-benchmark Topologies.*

(b) *Throughput Convergence Time for Yahoo! Topologies.*

(c) *Throughput Timeline for Network Monitoring Topologies.*

Figure 2.7: *Scale-out: Convergence Time Comparison (in seconds).*

Figure 2.7a and Figure 2.7b show the convergence time for both Micro-benchmark Topologies and Yahoo Topologies. We observe that Stela is far less intrusive than Storm when scaling out in the Diamond topology (92% lower) and about as intrusive as Storm in the Linear topology. Stela takes longer to converge than Storm in some cases like the Star topology, primarily because a large number of bolts are affected all at once by the Stela's scaling. Nevertheless the post-throughput scaling is worth the longer wait (Figure 2.4a). Further, for the Yahoo Topologies, Stela's convergence time is 88% and 75% lower than that of Storm's default scheduler.

The main reason why Stela has a better convergence time than both Storm's default scheduler and Least Link Load strategy [62] is that Stela does not change the current scheduling at existing machines (unlike Storm's default strategy and Least Link Load strategy), instead choosing to schedule operators at the new machines only.

In Network Monitoring topology, Stela experiences longer convergence time than Storm's default scheduler and Least Link Load strategy due to re-parallelization during the scale-out operation (Figure 2.7c). However, the benefit, as shown in Figure 2.6c, is the higher post-scale throughput provided by Stela.

2.4.5 Scale-In Experiments

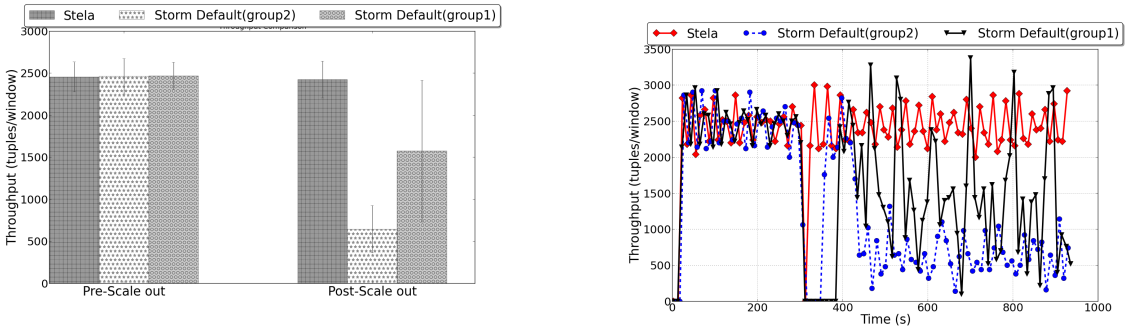
We examine the performance of Stela scale-in by running Yahoo's PageLoad topology. The initial cluster size is 8 and Figure 2.8a shows the throughput change after shrinking cluster size to 4 machines. (We initialize the operator allocation so that each machine can be occupied by tasks from fewer than 2 operators (bolts and spouts)). We compare against the performance of a round robin scheduler (same as Storm's default scheduler), using two alternative groups of randomly selected machines.

We observe Stela preserves throughput after scale-in while the Storm groups experience 80% and 40% throughput decrease respectively. Thus, Stela's post scale-in throughput is 2X - 5X higher than randomly choosing machines to remove. Stela also achieves 87.5% and 75% less down time (time duration when throughput is zero) than group 1 and group 2, respectively – see Figure 2.8b. This is primarily because in Stela migrating operators with low ETP will intrude less on the application, which will allow downstream congested components to digest tuples in their queues and continue producing output. In the PageLoad Topology, the two machines with lowest ETPs are chosen to be redistributed by Stela, which generates less intrusion for the application thus significantly better performance than Storm's default scheduler.

Thus, Stela is intelligent at picking the best machines to remove (via ETPSum). In comparison, Storm has to be lucky. In the above scenario, 2 out of the 8 machines were the "best". The probability that Storm would have been lucky to pick both (when it picks 4 at random) $= \binom{6}{2}/\binom{8}{4} = 0.21$, which is low.

2.5 RELATED WORK

Based on its predecessor Aurora [65], Borealis [66] is a stream processing engine that enables queries to be modified on the fly. Borealis focuses on load balancing on individual



(a) *Throughput Convergence Time for Yahoo! Topologies.*

(b) *Post Scale-in Throughput Timeline*

Figure 2.8: *Scale-in Experiments (Window size 10 seconds).*

machines and distributes load shedding in a static environment. Borealis also uses ROD (resilient operator distribution) to determine the best operator distribution plan that is closest to an "ideal" feasible set: a maximum set of machines that are underloaded. Borealis does not explicitly support elasticity.

Stormy [60] uses a logical ring and consistent hashing to place new nodes upon a scale out. It does not take congestion into account, which Stela does. StreamCloud [49] builds elasticity into the Borealis Stream Processing Engine [59]. StreamCloud modifies the parallelism level by splitting queries into sub queries and uses rebalancing to adjust resource usage. Stela does not change running topologies because we consider it intrusive to the applications.

SEEP [44] uses an approach to elasticity that mainly focuses on operator's state management. It proposes mechanisms to backup, restore and partition operators' states in order to achieve short recovery time. There have been several other papers focusing on elasticity for stateful stream processing systems. [25, 26] from IBM both enable elasticity for IBM System S [27, 28, 29] and SPADE [30], by increasing the parallelism of processing operators. These papers apply networking concepts such as congestion control to expand and contract the parallelism of a processing operator by constantly monitoring the throughput of its links. These works do not assume fixed number of machines provided (or taken away) by the users. Our system aims at intelligently prioritizing target operators to further parallelize to (or migrate from) user-determined number of machines joining in (or taken away from) the cluster, with a mechanism to optimize throughput.

Twitter's Heron [67] improves Storm's congestion handling mechanism by using back pressure – however elasticity is not explicitly addressed. Recent work [68] proposes an elasticity model that provides latency guarantee by tuning task-wise parallelism level in a fixed size cluster. Meanwhile, another recent work [37] implemented stream processing system elasticity. However, [37] focused on latency (not throughput) and on policy (not mechanism).

Nevertheless, Stela's mechanisms can be used as a black box inside of [37].

Some of these works have looked at *policies* for adaptivity [37], or [25, 26, 49, 60] focus on the *mechanisms* for elasticity. These are important building blocks for adaptivity. To the best of our knowledge, [62] is the only existing mechanism for elasticity in stream processing systems – Section 2.4 compared Stela against it.

2.6 CONCLUSION

In this chapter, we presented novel scale-out and scale-in techniques for stream processing systems. We have created a novel metric, ETP (Effective Throughput Percentage), that accurately captures the importance of operators based on congestion and contribution to overall throughput. For scale-out, Stela first selects congested processing operators to re-parallelize based on ETP. Afterwards, Stela assigns extra resources to the selected operators to reduce the effect of the bottleneck. For scale-in, we also use a ETP-based approach that decides which machine to remove and where to migrate affected operators. Our experiments on both micro-benchmarks Topologies and Yahoo Topologies showed significantly higher post-scale out throughput than default Storm and Link-based approach, while also achieving faster convergence. Compared to Apache Storm's default scheduler, Stela's scale-out operation reduces interruption time to a fraction as low as 12.5% and achieves throughput that is 45-120% higher than Storm's. Stela's scale-in operation chooses the right set of servers to remove and performs 2X-5X better than Storm's default strategy.

CHAPTER 3: HENGE: INTENT-DRIVEN MULTI-TENANT STREAM PROCESSING

3.1 INTRODUCTION

While stream processing systems for clusters have been around for decades [30, 59], neither classical nor modern distributed stream processing systems support *intent-driven multi-tenancy*. Multi-tenancy is attractive as it reduces acquisition costs and allows sysadmins to only manage a single consolidated cluster. In industry terms, multi-tenancy reduces capital and operational expenses (Capex & Opex), lowers total cost of ownership (TCO), increases resource utilization, and allows jobs to elastically scale based on needs. We believe the deployer of each job should be able to clearly specify their performance expectations as an intent to the system, and it is the underlying engine's responsibility to meet this intent. This alleviates the developer's burden of monitoring and adjusting their job. Modern distributed stream processing systems are very primitive and do not admit intents.

In Henge, we allow each job in a multi-tenant environment to specify its own intent as a Service Level Objective (SLO) [69]. It is critical that the metrics in an SLO be *user-facing* and thus does not involve internal metrics like queue lengths or CPU utilization as these can vary depending on the software, cluster, and job mix[1]. We believe lay users should not have to grapple with such complex metrics. To address this issue, we define a new *input rate-independent* metric for throughput SLOs called *juice*. We show how Henge calculates juice for arbitrary topologies.

Our latency SLOs and throughput SLOs are immediately useful. Time-sensitive jobs (e.g., those related to an ongoing ad campaign) are latency-sensitive and will specify latency SLOs, while longer running jobs (e.g., sentiment analysis of trending topics) will have throughput SLOs. Table 3.1 summarizes several real use cases spanning different SLO requirements.

Table 3.2 compares Henge with multi-tenant schedulers that are generic (Mesos, Yarn), as well as those that are stream processing-specific (Aurora, Borealis and R-Storm). Generic schedulers largely use reservation-based approaches to specify intents. A reservation is an explicit request to hold a specified amount of cluster resources for a given duration [75]. Besides not being user-facing, reservations are known to be hard to estimate even for a job with a static workload [76], let alone the dynamic workloads prevalent in streaming applications. Classical stream processing systems are either limited to a single node environment (Aurora), or lack multi-tenant implementations (e.g., Borealis has a multi-tenant proposal,

[1]However, these latter metrics can be monitored and used internally by the scheduler for self-adaptation.

Business	Use Case	SLO Type & Value
Bloomberg	High Frequency Trading	Latency < Tens of ms
	Updating top-k recent news articles on website	Latency < 1 min.
	Combining updates into one email sent per subscriber	Throughput > 40K messages/s [70]
Uber	Determining price of a ride on the fly, identifying surge periods	Latency < 5 s
	Analyzing earnings over time	Throughput > 10K rides/hr. [71]
The Weather Channel	Monitoring natural disasters in real-time	Latency < 30 s
	Processing collected data for forecasts	Throughput > 100K messages/min. [72]
WebMD	Monitoring blogs to provide real-time updates	Latency < 10 min.
	Search indexing related websites	Throughput: index new sites at the rate found
E-Commerce Websites	Counting ad-clicks	Latency: update click counts every second
	Processing logs at Alipay	Throughput > 6 TB/day [73]

Table 3.1: *Stream Processing: Use Cases and Possible SLO Types.*

but no associated implementation). R-Storm [74] is resource-aware Storm that adapts jobs based on CPU, memory, and bandwidth, but does not support user-facing SLOs.

3.2 OVERVIEW OF HENGE

Juice: As input rates vary over time, specifying a throughput SLO as an absolute value is impractical.

Juice lies in the interval [0, 1] and captures the ratio of processing rate to input rate: a value of 1.0 is ideal and implies that the rate of incoming tuples equals rate of tuples processed by the job. Conversely, a value less than 1 indicates that tuples are building up in queues, waiting to be processed. Throughput SLOs contain a minimum threshold for juice, making the SLO independent of input rate. We consider processing rate instead of output rate as this generalizes to cases where input tuples may be filtered or modified: thus, they affect results but are never outputted.

SLOs: A job's SLO can capture either latency or juice (or a combination of both). The SLO has: a) a threshold (min-juice or max-latency), and b) a job priority. Henge combines these via a user-specifiable *utility function*, inspired by soft real-time systems [77]. The utility function maps current achieved performance (latency or juice) to a value that represents the current benefit to the job. Thus, the function captures the developer intent that a job attains

Schedulers	Jobs	Multi-Tenant?	User-Facing SLOs?
MESOS [33]	General	✓	✗ Uses Reservations: CPU, Mem, Disk, Ports
YARN [32]	General	✓	✗ Uses Reservations: CPU, Mem, Disk
AURORA [22]	Streaming	✓	✓ For Single Node Environment Only
BOREALIS [23]	Streaming	✗	✓ Latency, Throughput, Others
R-STORM [74]	Streaming	✗	✗ Schedules based on: CPU, Mem, Bandwidth
Henge	**Streaming**	✓	✓ **Latency, Throughput, Hybrid**

Table 3.2: *Henge vs. Existing Multi-Tenant Schedulers.*

full "utility" if its SLO threshold is met and partial utility if not. Our utility functions are monotonic: the closer the job is to its SLO threshold, the higher its achieved maximum possible utility.

State Space Exploration: Moving resources in a live cluster is challenging. It entails a state space exploration where every step has both: 1) a significant realization cost, as moving resources takes time and affects jobs, and 2) a convergence cost, since the system needs time to converge to steady state after a step. Henge adopts a *conservatively online* approach where the next step is planned, executed in the system, then the system is allowed to converge, and the step's effect is evaluated. Then, the cycle repeats. This conservative exploration is a good match for modern stream processing clusters because they are unpredictable and dynamic. Offline exploration (e.g. simulated annealing) is time consuming and may make decisions on a cluster using stale information (as the cluster has moved on). Conversely, an aggressive online approach will over-react to changes, and cause more problems than it solves.

The primary actions in our state machine are: 1) Reconfiguration (give resources to jobs missing SLO), 2) Reduction (take resources away from overprovisioned jobs satisfying SLO), and 3) Reversion (give up an exploration path and revert to past good configuration). Henge gives jobs additional resources proportionate to how congested they are. Highly intrusive actions like reduction are kept small in number and frequency.

Maximizing System Utility: Via these actions, Henge attempts to continually improve each individual job and converge it to its maximum achievable utility. Henge is amenable to different goals for the cluster: either maximizing the minimum utility across jobs, or

maximizing the total achieved utility across all jobs. While the former focuses on fairness, the latter allows more cost-effective use of the cluster, which is especially useful since revenue is associated with total utility of all jobs. Thus, Henge adopts the goal of maximizing total achieved utility summed across all jobs. Our approach creates a weak form of Pareto efficiency [78]; in a system where jobs compete for resources, Henge transfers resources among jobs only if this will cause the total cluster's utility to rise.

Preventing Resource Hogging: Topologies with stringent SLOs may try to take over all the resources of the cluster. To mitigate this, Henge prefers giving resources to topologies that: a) are farthest from their SLOs, and b) continue to show utility improvements due to recent Henge actions. This spreads resources across all wanting jobs and mitigates starvation and resource hogging.

3.3 UNIFYING USER REQUIREMENTS

Topology Utility: A *Service Level Objective (SLO)* [79] in Henge is user-facing and can be set without knowledge of internal cluster details. An SLO specifies: a) an SLO *threshold* (min-throughput or max-latency); and b) a job priority. Henge girds these requirements together into a *utility function*. Intuitively, utility function is a *monotonic* function that measures users' satisfaction towards job performance.

Currently, Henge supports both latency SLOs and throughput SLOs (and hybrids thereof). For a job with minimum latency requirement, the job' utility function should be defined such as the utility is non-increasing as job's latency is above its latency requirement and increases (and vice versa for a job with maximum latency SLO). Once the job reaches its performance target, the utility function should returns constant value.

Given these requirements, Henge allows a variety of utility functions: linear, piece-wise linear, step functions, lognormal, etc. Utility functions may not be continuous.

Users can pick any utility functions that are monotonic. For concreteness, our Henge implementation uses a piece-wise linear utility function called a *knee* function. A knee function has two parts: a plateau after the SLO threshold, and a sub-SLO for when the job does not meet the threshold. Concretely, the achieved utility for jobs with throughput and latency SLOs respectively, are:

$$\frac{\texttt{Current Utility}}{Job\ Max\ Utility} = min(1, \frac{Current\ Throughput\ Metric}{SLO\ Throughput\ Threshold}) \tag{3.1}$$

$$\frac{\texttt{Current Utility}}{Job\ Max\ Utility} = min(1, \frac{SLO\ Latency\ Threshold}{Current\ Latency}) \tag{3.2}$$

Utility function provides a unified transformation for jobs with different performance requirements. Using utility function as universal metrics across jobs, Henge performs resource re-balancings among jobs dynamically using its adaptation state machine [2].

The Juice Metric: For applications that have throughput SLOs, juice defines the fraction of input data that are effectively processed per unit of time. Juice lies in the interval [0, 1] and captures the ratio of processing rate to input rate: a value of 1.0 is ideal and implies that the rate of incoming tuples equals rate of tuples processed by the job. Conversely, a value less than 1 indicates that tuples are building up in queues, waiting to be processed. Throughput SLOs contain a minimum threshold for juice, making the SLO independent of input rate. We consider processing rate instead of output rate as this generalizes to cases where input tuples may be filtered or modified: thus, they affect results but are never outputted.

Juice is formulated to reflect the *global* processing efficiency of a topology. An operator's contribution to juice is the proportion of input passed in originally *from the source (i.e., from all spouts)* that it processed in a given time window. This is the impact of that operator *and* its upstream operators on this input. The juice of a topology is then the normalized sum of juice values of all its sinks.

Essentially, juice captures what fraction of the input data is being processed by the topology. It allows us to: 1) abstract away throughput SLO requirements in a way that is independent of absolute input rate, and 2) track whether congestion is occurring inside a topology (job). Our design of the juice metric is based on three principles:

- **Code Independent:** It should be independent of the operators' code, and should be calculate-able by only considering the number of tuples generated by operators.
- **Rate Independent:** It should be input-rate independent.
- **Topology Independent:** It should be independent of the shape and structure of the topology. It should be correct in spite of duplication, merging, and splitting of tuples.

Henge calculates juice in configurable windows of time (unit time). *Source input* tuples are those that arrive at a spout in unit time. For each operator o in a topology that has n parents, we define T_o^i as the sum of tuples sent out from its ith parent per time unit, and E_o^i as the number of tuples that operator o executed (per time unit), from those received from parent i. The per-operator contribution to juice, J_o^s, is the proportion of source input *sent* from spout s that operator o received and processed. Given that J_i^s is the juice of o's i^{th} parent, then J_o^s is:

$$J_o^s = \sum_{i=1}^{n} \left(J_i^s \times \frac{E_o^i}{T_o^i} \right) \tag{3.3}$$

A spout s has no parents, and its juice: $J_s = \frac{E_s}{T_s} = 1.0$.

In Equation. 3.3, the fraction $\frac{E_o^i}{T_o^i}$ reflects the proportion of tuples an operator received from its parents, and processed successfully. If no tuples are waiting in queues, this fraction is equal to 1.0. By multiplying this value with the parent's juice we accumulate through the topology the effect of all upstream operators.

We make two important observations. In the term $\frac{E_o^i}{T_o^i}$, it is critical to take the denominator as the number of tuples *sent* by a parent rather than received at the operator. This allows juice: a) to account for data splitting at the parent (fork in the DAG), and b) to be reduced by tuples dropped by the network. The numerator is the number of *processed* tuples rather than the number of output tuples – this allows juice to generalize to operator types whose processing may drop tuples (e.g., filter).

Given all operator juice values, a topology's juice can be calculated by normalizing w.r.t. number of spouts:

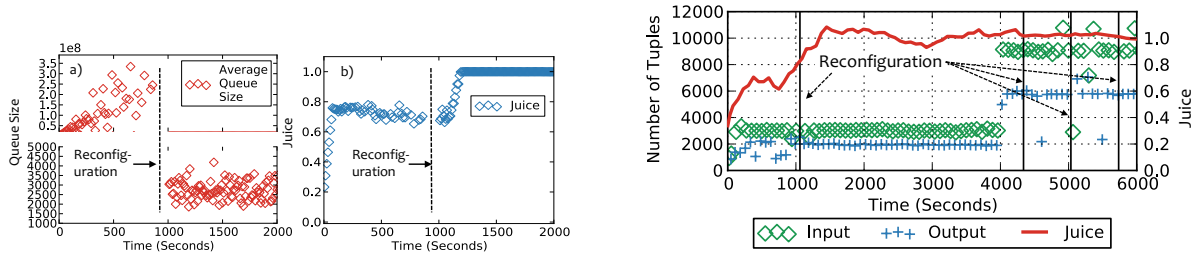
$$\frac{\sum_{Sinks\ s_i,\ Spouts\ s_j} (J_{s_i}^{s_j})}{Total\ Number\ of\ Spouts} \tag{3.4}$$

If no tuples are lost in the system, the numerator equals the number of spouts. To ensure that juice stays below 1.0, we normalize the sum with the number of spouts.

3.4 JUICE AS A PERFORMANCE INDICATOR

Juice is an indicator of queue size: Fig. 3.1a shows the inverse correlation between topology juice and queue size at the most congested operator of a PageLoad topology. Queues buffer incoming data for operator executors, and longer queues imply slower execution rate and higher latencies. Initially queue lengths are high and erratic–juice captures this by staying well below 1. At the reconfiguration point (910 s) the operator is given more executors, and juice converges to 1 as queue lengths fall, stabilizing by 1000 s.

Juice is independent of operations and input rate: In Fig. 3.1b, we run 5 PageLoad topologies on one cluster, and show data for one of them. Initially juice stabilizes to around 1.0, near t=1000 s (values above 1 are due to synchronization errors, but they don't affect our logic). PageLoad filters tuples, thus output rate is < input rate–however, juice is 1.0 as



(a) **Juice vs. Queue Size:** *Inverse Relationship.*

(b) **Juice is Rate-Independent:** *Input rate is increased by 3 × at 4000 s, but juice does not change. When juice falls to 0.992 at 4338 s, Henge stabilizes it to 1.0 by 5734 s.*

Figure 3.1: *Evaluating Juice Effectiveness.*

it shows that all input tuples are being processed.

Then at 4000 s, we triple the input rate to all tenant topologies. Notice that juice stays 1.0. Due to natural fluctuations, at 4338 s, PageLoad's juice drops to 0.992. This triggers reconfigurations (vertical lines) from Henge, stabilizing the system by 5734 s, maximizing cluster utility.

3.5 EVALUATION

Henge uses topology utilities to make dynamic recofiguration choices. Here we simulate two types of workloads that exhibit a *diurnal* pattern [80, 81]: SDSC-HTTP [82] and EPA-HTTP traces [83]. We inject these workloads into PageLoad topologies with 5 jobs run with the SDSC-HTTP trace and concurrently and 5 other jobs run with the EPA-HTTP trace. All jobs have max-utility=35, and a latency SLO of 60 ms.

Fig. 3.2 shows the result of running 48 hours of the trace (each hour is mapped to 10 mins). In Fig. 3.2a, workloads increase from hour 7 of day 1, reach their peak by hour $13\frac{1}{3}$, and then fall. Henge reconfigures all 10 jobs, reaching 89% of max cluster utility by hour 15.

Fig. 3.2b shows a topology running the EPA workload (other topologies exhibited similar behavior). Observe how Henge reconfigurations from hour 8 to 16 adapt to the fast changing workload. This results in fewer SLO violations during the second peak (hours 32 to 40). Thus, Henge tackles diurnal workloads without extra resources.

Fig. 3.2c shows the CDF of SLO satisfation for three systems. Default Storm gives 0.0006% SLO satisfaction at the median, and 30.9% at the 90th percentile (meaning that 90% of the time, default Storm provided at most 30.9% of the cluster's max achievable utility.). Henge

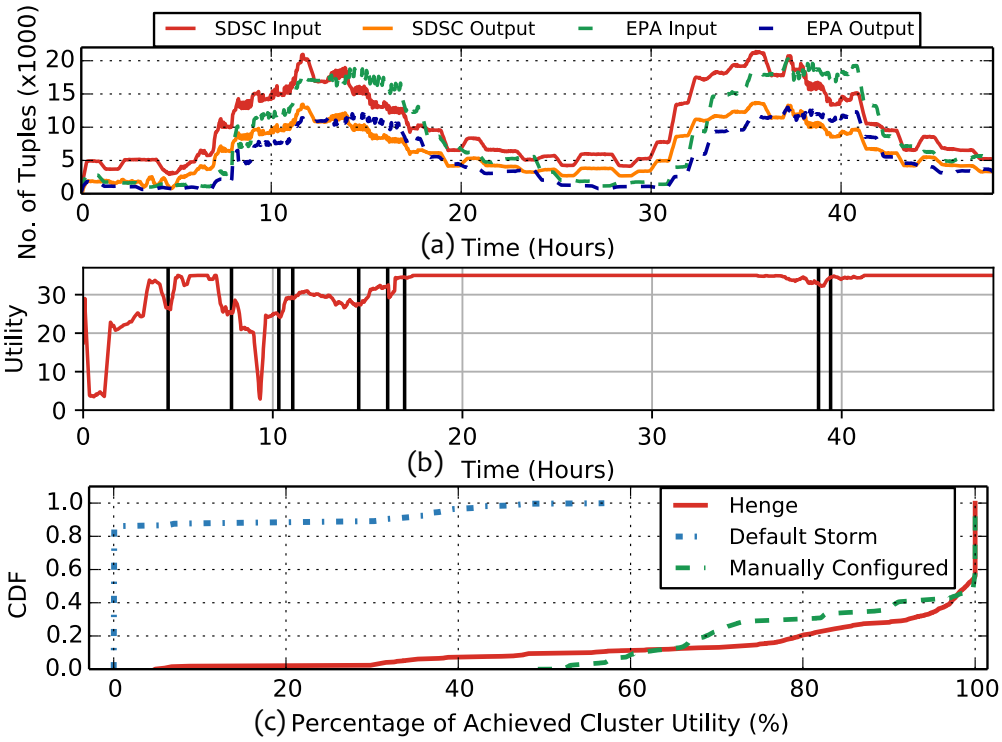


Figure 3.2: **Diurnal Workloads:** *a) Input and output rates vs time, for two diurnal workloads. b) Utility of job (reconfigured by Henge) with EPA workload, c) CDF of SLO satisfaction for Henge, default Storm, & manual configuration. Henge adapts during first cycle and fewer reconfigurations are needed later.*

yields 74.9%, 99.1%, and 100% SLO satisfaction at the 15th, 50th, and 90th percentiles respectively.

Henge is preferable over manual configurations. We manually configured all topologies to meet their SLOs at median load. They provide 66.5%, 99.8% and 100% SLO satisfaction at the 15th, 50th and 90th percentiles respectively. Henge is better than manual configurations from the 15th to 45th percentile, and comparable later.

Henge has an average of 88.12% SLO satisfaction rate, while default Storm and manually configured topologies provide an average of 4.56% and 87.77% respectively. Thus, Henge gives 19.3× better SLO satisfaction than default Storm and does better than manual configuration. The *total* time taken by Henge's actions, summed across all topologies composed of only 0.49% of the total runtime per topology. The longest convergence time for any topology was takes only 0.56% of the total runtime.

3.6 CONCLUSION

We presented Henge as a system for intent-driven, SLO-based multi-tenant stream processing. We explore Henge's approach to unify user requirements by translating different

types of performance requirement to resource need in multi-tenant environment. Henge provides SLO satisfaction for jobs with latency and/or throughput SLOs. Henge also proposes a new metric called juice, that helps systems to interpret throughput satisfaction independent of input rate and topology structure. Henge makes dynamic reconfiguration operations based on these proposed metrics and is able to provide 19.3× better satisfaction rate than Storm with less than 1% reconfiguration overhead.

CHAPTER 4: POPULAR IS CHEAPER: CURTAILING MEMORY COSTS IN INTERACTIVE ANALYTICS ENGINES.

4.1 INTRODUCTION

Real-time analytics is projected to grow annually at a rate of 31% [6]. Apart from stream processing engines, which have received much attention [7, 8, 84], real time analytics now also includes the burgeoning area of interactive data analytics engines such as Druid [11], Redshift [13], Mesa [14], Presto [15] and Pinot [12]. These systems have seen widespread adoption [18, 19] in companies which require applications to support sub-second query response time. Applications span usage analytics, revenue reporting, spam analytics, ad feedback, and others [31]. Typically large companies have their own on-premise deployments while smaller companies use a public cloud. The internal deployment of Druid at Yahoo! (now called Oath) has more than 2000 hosts, stores petabytes of data and serves millions of queries per day at sub-second latency scales [31].

Many typical interactive analytics engines today follow Lambda [85] or Kappa architecture [86], handle both real-time and interactive off-line queries, requires fast responses. These engines are expected to cache popular historical data and relies on in-memory processing to provide short query makespan. A typical interactive analytics engine like Druid [11] parallelize query processing over multiple compute nodes that contain required data (called data segments) to reduce query latency. On the other hand, serving multiple queries concurrently accessing popular data requires parallelization through data replication. While full replication for all data provides best performance, it is impossible due to the limited memory.

Interactive analytics engines employ two forms of parallelism. First, data is organized into data blocks, called *segments*–this is standard in all engines. For instance, in Druid, hourly data from a given source constitutes a segment. Second, a query that accesses multiple segments can be run in parallel on each of those segments, and then the results are collected and aggregated. Query parallelization helps achieve low latency. Because a query (or part thereof) running at a compute node needs to have its input segment(s) cached at that node's memory, *segment placement* is a problem that needs careful solutions. Full replication is impossible due to the limited memory.

Figure 4.1 shows the query latency for two cluster sizes (15, 30 compute nodes) and query rates (1500, 2500 qps). For each configuration (cluster size / query rate pair), as the replication factor (applied uniformly across segments) is increased, we observe the curve hits a "knee", beyond which further replication yields marginal latency improvements. The knee

for 15 / 2500 is 9 replicas, and for the other two is 6 replicas. Our goal is to achieve the knee of the curve for individual segments (which is a function of their respective query loads), in an adaptive way.

Getafix's philosophy is developed from the production pattern we have observed from Yahoo!'s Druid cluster. We discovered that at any given time, the most popular data (1%) is an order of magnitude more popular, in terms of number of accesses, than the least popular (40%). Getafix is built atop intuition arising from our optimal solution to the static version of the replication problem (which we call ModifiedBestFit). Our static solution is provably optimal in *both* makespan (runtime of the query set) as well as memory costs. In the dynamic scenario, Getafix makes replication decisions by continually measuring query injection rate, segment popularity, and current cluster state.

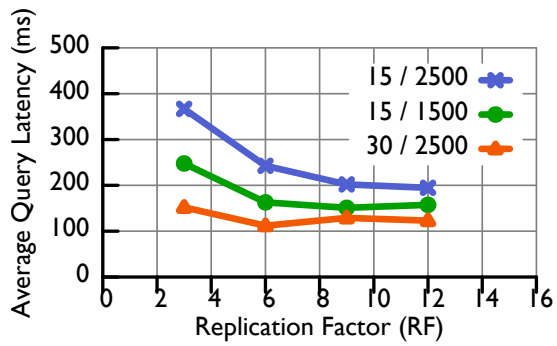


Figure 4.1: *Average Query Latency observed with varying replication factors for different (cluster size / query injection rate) combinations.*

4.2 THE REPLICATION STRATEGY

Problem Formulation: Given m segments, n historical nodes (HNs), and k queries that access a subset of these segments, our goal is to find a segment allocation (segment assignment to HNs) that both: 1) minimizes total runtime (makespan), and 2) minimizes the total number of segment replicas. For simplicity we assume: a) each query takes unit time to process each segment it accesses, b) initially HNs have no segments loaded, and c) HNs are homogeneous in computation power. Our implementation relaxes these assumptions.

Consider the query-segment pairs in the given static workload, i.e., all pairs (Q_i, S_j) where query Q_i needs to access segment S_j. Spreading these query-segment pairs uniformly across all HNs, in a load-balanced way, automatically gives a time-optimal schedule: no two HNs finish more than 1 time unit apart from each other. A load balanced assignment is desirable as it *always* achieves the minimum runtime (makespan) for the set of queries. However, arbitrarily (or randomly) assigning query-segment pairs to HNs may not minimize the total amount of replication across HNs.

Consider an example with 6 queries accessing 4 segments. The access characteristics C

for the 4 segments are: $\{S_1$:6, S_2:3, S_3:2, S_4:1$\}$. In other words, 6 queries access segment S_1, 3 access S_2 and so on. A possible time-optimal (balanced) assignment of the query-segment pair could be: bin $HN_1 = \{S_1$:3, S_2:1$\}$, $HN_2 = \{S_2$:2, S_3:1, S_4:1$\}$, $HN_3 = \{S_1$:3, S_3:1$\}$. However, this assignment is not optimal in replication factor (and thus storage). The total number of replicas stored in the HNs in this assignment is 7. The minimum number of replicas required for this example is 5. An allocation that achieves this minimum is: $HN_1 = \{S_1$:4$\}$, $HN_2 = \{S_2$:3, S_4:1$\}$, $HN_3 = \{S_1$:2, S_3:2$\}$ (Figure 4.2).

Formally, the input to our problem is: 1) segment access counts $C = \{c_1, \ldots c_m\}$ for k queries accessing m segments, and 2) n HNs each with capacity $\lceil \frac{\sum_i c_i}{n} \rceil$ (in our paper, "capacity" always means "compute capacity"). We wish to find: *Allocation* $X = \{x_{ij} = 1, \text{if segment } i \text{ is replicated at HN } j\}$, such that it minimizes $\sum_i \sum_j x_{ij}$.

Algorithm 4.1 Generalized Allocation Algorithm.

```
1: function ModifiedFit(C, nodelist)  ▷ C: Access counts for each segment  ▷ nodelist:
   List of HNs
2:     n ← Length(nodelist)
3:     capacity ← ⌈ Σ_{C_i ∈ C} |C_i| / n ⌉
4:     binCap ← InitArray(n, capacity)
5:     priorityQueue ← BuildMaxHeap(C)
6:     while !Empty(priorityQueue) do
7:         (segment, count) ← Extract(priorityQueue)
8:         (left, bin) ← ChooseHistoricalNode
9:             (count, binCap)
10:        LoadSegment(nodelist, bin, segment)
11:        if left > 0 then
12:            Insert(priorityQueue, (segment, left))
```

We solve this problem as a *colored* variant of the traditional bin packing problem [87]. A query-segment pair is treated as a *ball* and a HN represents a *bin*. Each segment is represented by a *color*, and there are as many balls of a color as there are queries accessing it. The number of distinct colors assigned to a bin (HN) is the number of segment replicas this HN needs to store. The problem is then to place the balls in the bins in a load-balanced way that minimizes the number of "splits" for all colors, i.e., the number of bins each color is present in, summed up across all colors. This number of splits is the same as the total number of segment replicas. Unlike traditional bin packing which is NP-hard, this version of the problem is solvable in polynomial time.

ModifiedBestFit: Algorithm 4.1 depicts our solution to the problem. The algorithm maintains a priority queue of segments, sorted in decreasing order of popularity (i.e., number of queries accessing the segment). The algorithm works iteratively: in each iteration

it extracts the next segment S_j from the head of the queue, and allocates the query-segment pairs corresponding to that segment to a HN, selected based on a heuristic called ChooseHistoricalNode. If the selected HN's current capacity is insufficient to accommodate all the pairs, then the remaining available compute capacity in that HN is filled with a subset of it. Subsequently, the segment's count is updated to reflect remaining unallocated query-segment pairs, and finally, the segment is re-inserted back into the priority queue at the appropriate position.

The total number of iterations in this algorithm equals the total number of replicas created across the cluster. The ChooseHistoricalNode problem bears similarities with segmentation in traditional operating systems [88]. We explored three strategies to solve ChooseHistoricalNode: First Fit, Largest Fit, and Best Fit. Of the three, we only describe Best Fit here as it gives an optimal allocation.

In each iteration, we choose the next HN that would have the least compute capacity (space, or number of slots for balls) remaining after accommodating all the queries for the picked segment (head of queue). Ties are broken by picking the lower HN id. If none of the nodes have sufficient capacity to fit all the queries for the picked segment, we default to Largest Fit for this iteration, i.e., we choose the HN with the largest available capacity (ties broken by lower HN id), fill it as much as possible, and re-insert unassigned queries for the segment back into the sorted queue. We call this algorithm ModifiedBestFit. Consider our running example (Figure 4.2) where C is $\{S_1{:}6,\ S_2{:}3,\ S_3{:}2,\ S_4{:}1\}$. The algorithm assigns S_1 to HN_1 and S_2 to HN_2. Next, it picks segment S_1 (again tie broken with S_3) and assigns it to HN_3 because it has sufficient space to fit all the balls. The final assignment produced is optimal in both makespan and replication factor.

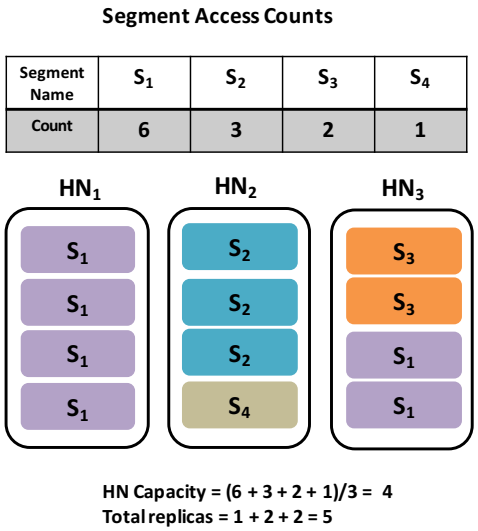


Figure 4.2: *Problem depicted with balls and bin. Query-segment pairs are balls and historical nodes represent bins. All balls of same color access the same segment. HN capacity refers to compute capacity. Optimal assignment shown.*

4.3 REDUCING NETWORK TRANSFER THROUGH MATCHING

Dynamically, Getafix performs ModifiedBestFit periodically by collecting popularity of each data segment. Consider the example shown in Figure 4.3. In the configuration at

time T_1 (top part of figure), HN_1 has segments S_2 and S_3, HN_2 has S_4 only, and HN_3 has segments S_1 and S_2. At time T_2, ModifiedBestFit expects the following configuration: $E_1 = \{S_1\}$, $E_2 = \{S_2, S_4\}$, $E_3 = \{S_1, S_3\}$. If each HN_i chooses to host the segments in E_i, then the algorithm needs to fetch 3 segments in total. However the minimum required is 2, given by the following assignment: E_1 to HN_3, E_2 to HN_2, E_3 to HN_1.

We model this problem as a bipartite graph shown in Figure 4.3 where vertices on the bottom represent expected configurations (E_j) and vertices on the top represent HNs (HN_i) with the current set of replicas. An $HN_i - E_j$ edge represents the network cost to transfer all of E_j's segments to HN_i (except those already at HN_i). Network transfer is minimized by finding the minimum cost matching in this bipartite graph. We use the classical Hungarian Algorithm [89] to find the minimum matching. It has a complexity of $O(n^3)$ where n is the number of HNs. This is acceptable because interactive analytics engine clusters only have a few hundred nodes. The coordinator uses the results to set up data transfers for the segments to appropriate HNs.

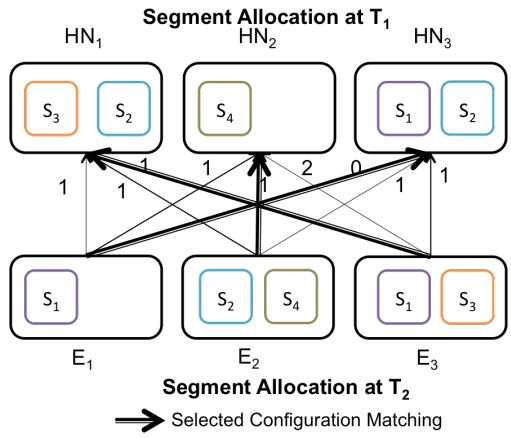
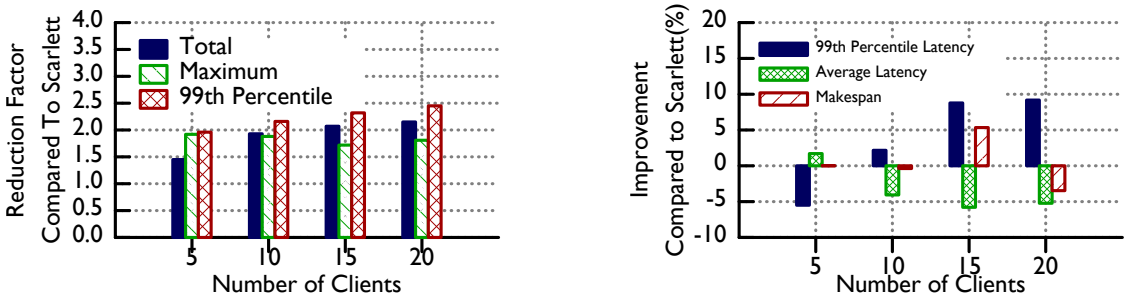


Figure 4.3: *Physical HN Mapping problem from Figure 4.2 represented as a bipartite graph.*

4.4 DYNAMIC REPLICATION COMPARING TO PRIOR APPROACH

We compare Getafix with Scarlett [90] where the replication factors is determined by the number of accesses and the placement is determined by machines load factor. We implement Scarlett replication and placement strategy on top of Druid and performs comparison of memory usage and query latency using number of HNs ranges from 5 to 20.

Comparison vs. Scarlett: We increase the query load (number of workload generator clients varied from 5 to 20) while keeping the compute capacity (HNs) fixed (20). Figure 4.4a plots the savings in Getafix's memory usage compared to Scarlett's. Getafix uses 1.45 - 2.15× less total memory (across HNs), and 1.72 - 1.92× less maximum memory in a single HN. Scarlett alleviates query hotspots by creating more replicas of popular segments, while Getafix carefully balances replicas of popular and unpopular segments to keep overall



(a) *Scarlett memory divided by Getafix total memory. Higher is better.*

(b) *Reduction in Makespan, Average and 99th Percentile Latency of Getafix compared to Scarlett. Higher is better.*

Figure 4.4: *AWS Experiments: Getafix vs. Scarlett with increasing load (number of client varying from 5 to 20).*

replication (and memory usage) low. Getafix's memory savings also increases as more clients are added.

Memory Dollar Cost Savings in Public Cloud: We perform a back of the envelope calculation, based on our experimental numbers. For the 20 HN + 20 client experiment, Getafix has an effective replication factor of 1.9 compared to Scarlett's 4.2. (Our trace study shows heavy-tailed nature of segment popularity [3]. This implies the very popular segments influence effective replication factor.) In a public cloud deployment, where popular data size is 100 TB [1], Getafix thus can reduce memory usage by approximately 230 TB (100 TB $\times$ (4.2 - 1.9)). This amounts to cost savings of 230 $\times$ 10^3 GB $\times$ \$0.005/GB/hour = \$1150 per hour. Annually, this would amount to \$10 million worth of savings.

To quantify the impact of this memory savings on performance, Figure 4.4b plots the reduction in makespan, average and 99th percentile latency for Getafix compared to Scarlett. Getafix completes all the queries within $\pm 5\%$ of Scarlett for all the experiments. Query latency is also comparable.

We conclude that compared to Scarlett, Getafix significantly reduces memory usage in a private cloud, dollar cost in a public cloud, with small impact on query performance.

[1]Most present day production clusters in Google, Yahoo handle petabytes of data [14] per day. Of this only a fraction of the data is most popular and hosted in memory. We conservatively estimated 100 TB as the ballpark of popular data size.

4.5 CONCLUSION

In this chapter, we proposed techniques for segment management in interactive data analytics systems. We use segment popularity information to make decisions on which segments to load and how many replicas to assign. Our ModifiedBestFit strategy is optimal in a static setting and we adapt to workload changes by reallocating segments with minimized bandwidth consumption. We show that Getafix uses memory more efficiently by using 1.45 - 2.15× less total memory (across historical nodes) and 1.72 - 1.82× (in a single historical node) less maximum memory comparing to existing data replication mechanism like Scarlett, without compromising query makespan.

CHAPTER 5: MOVE FAST AND MEET DEADLINES: FINE-GRAINED REAL-TIME STREAM PROCESSING WITH CAMEO.

5.1 INTRODUCTION

Stream processing applications in large companies handle tens of millions of events per second [16, 17, 91]. In an attempt to scale and keep total cost of ownership (TCO) low, today's systems: a) parallelize operators across machines, and b) use multi-tenancy, wherein operators are collocated on shared resources. Yet, resource provisioning in production environments remains challenging due to two major reasons:

(i) High workload variability. In a production cluster at a large online services company, we observed orders of magnitude variation in event ingestion and processing rates, *across time*, *across data sources*, *across operators*, and *across applications*. This indicates that resource allocation needs to be dynamically tailored towards each operator in each query, in a nimble and adept manner at run time.

(ii) Latency targets vary across applications. User expectations come in myriad shapes. Some applications require quick responses to events of interest, i.e., short end-to-end latency. Others wish to maximize throughput under limited resources, and yet others desire high resource utilization. Violating such user expectations is expensive, resulting in breaches of service-level agreements (SLAs), monetary losses, and customer dissatisfaction.

To address these challenges, we explore a new *fine-grained* philosophy for designing a multi-tenant stream processing system. Our key idea is to provision resources to each operator based solely on its *immediate* need. Concretely we focus on deadline-driven needs. Our fine-grained approach is inspired by the recent emergence of event-driven data processing architectures including actor frameworks like Orleans [92, 93] and Akka [94], and serverless cloud platforms [95, 96, 97, 98].

Our motivation for exploring a fine-grained approach is to enable resource sharing directly among operators. This is more efficient than the traditional *slot-based* approach, wherein operators are assigned dedicated resources. In the slot-based approach, operators are mapped onto processes or threads—examples include task slots in Flink [99], instances in Heron [8], and executors in Spark Streaming [10]. Developers then need to either assign applications to a dedicated subset of machines [100], or place execution slots in resource containers and acquire physical resources (CPUs and memory) through resource managers [32, 33, 34].

While slot-based systems provide isolation, they are hard to dynamically reconfigure in the face of workload variability. As a result it has become common for developers to "game" their resource requests, asking for over-provisioned resources, far above what the job needs [35].

Aggressive users starve other jobs which might need immediate resources, and the upshot is unfair allocations and low utilization.

At the same time, today's fine-grained scheduling systems like Orleans, as shown in Figure 5.1, cause high tail latencies. The figure also shows that a slot-based system (Flink on YARN), which maps each executor to a CPU, leads to low resource utilization. The plot shows that our approach, Cameo, can provide both high utilization and low tail latency.

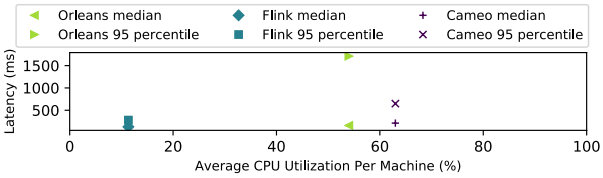
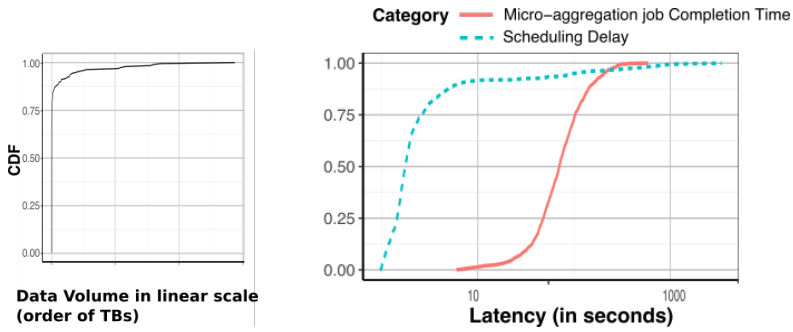


Figure 5.1: *Slot-based system (Flink), Simple Actor system (Orleans), and our framework Cameo.*

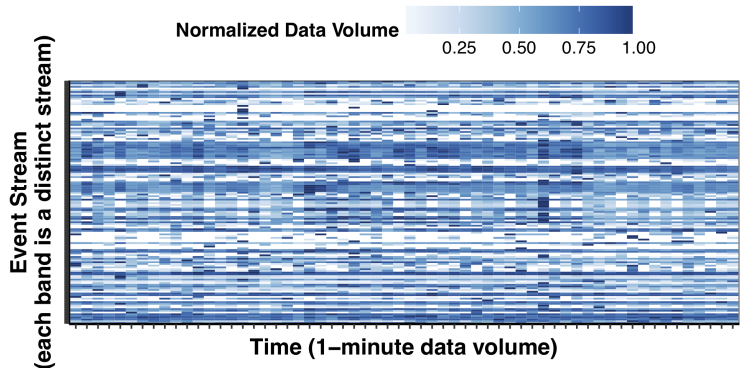
To realize our approach, we develop a new priority-based framework for fine-grained distributed stream processing. This requires us to tackle several *architectural* design challenges including: 1) translating a job's performance target (deadlines) to priorities of individual messages, 2) developing interfaces to use real-time scheduling policies such as earliest deadline first (EDF) [101], least laxity first (LLF) [102] etc., and 3) low-overhead scheduling of operators for prioritized messages. We present *Cameo*, a new scheduling framework designed for data streaming applications. Cameo:

- *Dynamically* derives priorities of operators, using both: a) *static input*, e.g., job deadline; and b) *dynamic stimulus*, e.g., tracking stream progress, profiled message execution times.
- Contributes new mechanisms: i) *scheduling contexts*, which propagate scheduling states along dataflow paths, ii) a *context handling* interface, which enables pluggable scheduling strategies (e.g., laxity, deadline, etc.), and iii) tackles required scheduling issues including per-event synchronization, and semantic-awareness to events.
- Provides low-overhead scheduling by: i) using a stateless scheduler, and ii) allowing scheduling operations to be driven purely by message arrivals and flow.

We build Cameo on Flare [17], which is a distributed data flow runtime built atop Orleans [92, 93]. Our experiments are run on Microsoft Azure, using production workloads. Cameo, using a laxity-based scheduler, reduces latency by up to 2.7× in single-query scenarios and up to 4.6× in multi-query scenarios. Cameo schedules are resilient to transient workload spikes and ingestion rate skews across sources. Cameo's scheduling decisions incur less than 6.4% overhead.



(a) *Data Volume Distribution* (b) *Job Scheduling & Completion Latencies*



(c) *Ingestion Heatmap*

Figure 5.2: *Workload characteristics collected from a production stream analytics system.*

5.2 BACKGROUND AND MOTIVATION

5.2.1 Workload Characteristics

We study a production cluster that ingests more than 10 PB per day over several 100K machines. The shared cluster has several internal teams running streaming applications which perform debugging, monitoring, impact analysis, etc. We first make key observations about this workload.

Long-tail streams drive resource over-provisioning. Each data stream is handled by a standing *streaming* query, deployed as a dataflow job. As shown in Figure 5.2a, we first observe that 10% of the streams process a majority of the data. Additionally, we observe that a long tail of streams, each processing small amount data, are responsible for over-provisioning—their users rarely have any means of accurately gauging how many nodes are required, and end up over-provisioning for their job.

Temporal variation makes resource prediction difficult. Figure 5.2c is a heat map showing incoming data volume for 20 different stream sources. The graph shows a high

degree of variability across both sources and time. A single stream can have spikes lasting one to a few seconds, as well as periods of idleness. Further, this pattern is continuously changing. This points to the need for an agile and fine-grained way to respond to temporal variations, as they are occurring.

Users already try to do fine-grained scheduling. We have observed that instead of continuously running streaming applications, our users prefer to provision a cluster using external resource managers (e.g., YARN [32], Mesos [33]), and then run periodic micro-batch jobs. Their implicit aim is to improve resource utilization and throughput (albeit with unpredictable latencies). However, Figure 5.2b shows that this ad-hoc approach causes overheads as high as 80%. This points to the need for a common way to allow all users to perform fine-grained scheduling, without a hit on performance.

Latency requirements vary across jobs. Finally, we also see a wide range of latency requirements across jobs. Figure 5.2b shows that the job completion time for the micro-aggregation jobs ranges from less than 10 seconds up to 1000 seconds. This suggests that the range of SLAs required by queries will vary across a wide range. This also presents an opportunity for priority-based scheduling: applications have longer latency constraints tend to have greater flexibility in terms of *when* its input can be processed (and vice versa).

5.2.2 Prior Approaches

Dynamic resource provisioning for stream processing. Dynamic resource provisioning for streaming data has been addressed primarily from the perspective of dataflow reconfiguration. These works fall into three categories as shown in Figure 5.3:

i) *Diagnosis And Policies*: Mechanisms for when and how resource re-allocation is performed;

ii) *Elasticity Mechanisms*: Mechanisms for efficient query reconfiguration; and

iii) *Resource Sharing*: Mechanisms for dynamic performance isolation among streaming queries.

These techniques make changes to the dataflows in reaction to a performance metric (e.g., latency) deteriorating.

Cameo's approach does not involve changes to the dataflow. It is based on the insight that the streaming engine can delay processing of those query operators which will not violate performance targets right away. This allows us to quickly prioritize and provision resources proactively for those other operators which could immediately need resources. At the same time, existing reactive techniques from Figure 5.3 are orthogonal to our approach and can be used alongside our proactive techniques.

The promise of event-driven systems. To achieve fine-grained scheduling, a promising

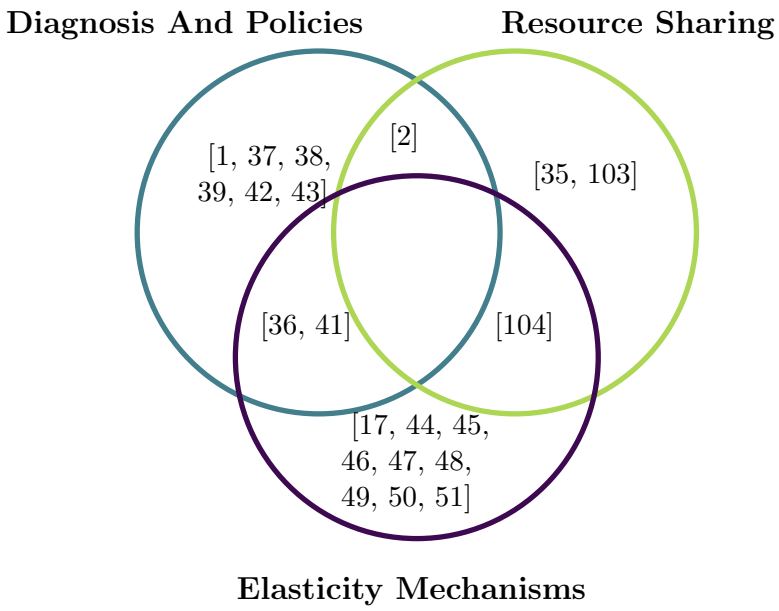


Figure 5.3: *Existing Dataflow Reconfiguration Solutions.*

direction is to leverage emerging event-driven systems such as actor frameworks [105, 106] and serverless platforms [107]. Unlike slot-based stream processing systems like Flink [9] and Storm [108], operators here are not mapped to specific CPUs. Instead event-driven systems maintain centralized queues to host incoming messages and dynamically dispatch messages to available CPUs. This provides an opportunity to develop systems that can manage a unified queue of messages across query boundaries, and combat the over-provisioning of slot-based approaches. Recent proposals for this execution model also include [96, 107, 109, 110].

Cameo builds on the rich legacy of work from two communities: classical real-time systems [111, 112] and first-generation stream management systems (DSMS) in the database community [59, 113, 114, 115]. The former category has produced rich scheduling algorithms, but unlike Cameo, none build a full working system that is flexible in policies, or support streaming operator semantics. In the latter category the closest to our work are event-driven approaches [59, 116, 117]. But these do not interpret stream progress to derive priorities or support trigger analysis for distributed, user-defined operators. Further, they adopt a centralized, stateful scheduler design, where the scheduler *always* maintains state for all queries, making them challenging to scale.

Achieving Cameo's goal of dynamic resource provisioning is challenging. Firstly, messages sent by user-defined operators are a black-box to event schedulers. Inferring their impact on query performance requires new techniques to analyze and re-prioritize said messages. Secondly, event-driven schedulers must scale with message volume and not bottleneck.

5.3 DESIGN OVERVIEW

Assumptions, System Model: We design Cameo to support streaming queries on clusters shared by cooperative users, e.g., within an organization. We also assume that the user specifies a latency target at query submission time, e.g., derived from product and service requirements.

The architecture of Cameo consists of two major components: (i) a scheduling strategy which determines message priority by interpreting the semantics of query and data streams given a latency target. (Section 5.4), and (ii) a scheduling framework that 1. enables message priority to be generated using a pluggable strategy, and 2. schedules operators dynamically based on their current pending messsages' priorities (Section 5.5).

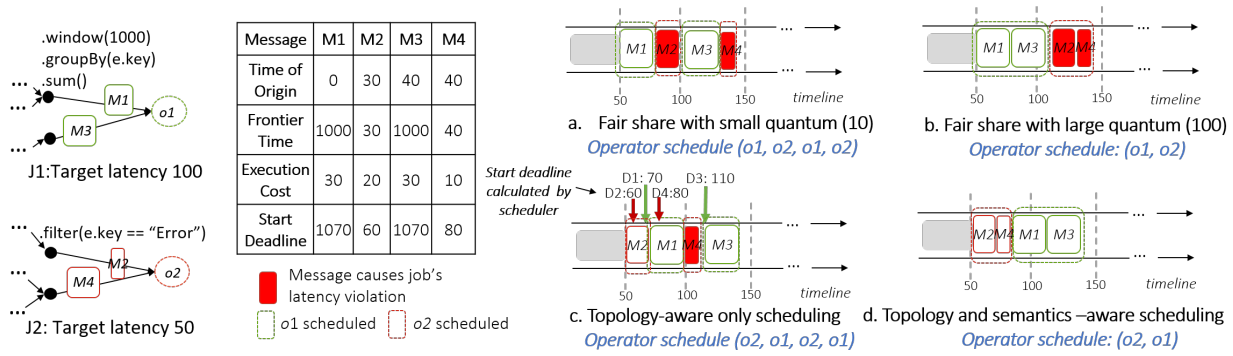
Cameo prioritizes operator processing by computing the *start deadlines* of arriving messages, i.e., latest time for a message to start execution at an operator without violating the downstream dataflow's latency target for that message. Cameo continuously reorders operator-message pairs to prioritize messages with earlier deadlines.

Calculating priorities requires the scheduler to continuously book-keep both: (i) per-job static information, e.g., latency constraint/requirement[1] and dataflow topology, and (ii) dynamic information such as the timestamps of tuples being processed (e.g., stream progress [118, 119]), and estimated execution cost per operator. To scale such a fine-grained scheduling approach to a large number of jobs, Cameo utilizes *scheduling contexts*— data structures attached to messages that capture and transport information required to generate priorities.

The scheduling framework of Cameo has two levels. The upper level consists of *context converters*, embedded into each operator. A context converter modifies and propagates scheduling contexts attached to a message. The lower level is a *stateless scheduler* that determines target operator's priority by interpreting scheduling context attached to the message. We also design a programmable API for a pluggable scheduling strategy that can be used to handle scheduling contexts. In summary, these design decisions make our scheduler scale to a large number of jobs with low overhead.

Example. We present an example highlighting our approach. Consider a workload, shown in Figure 5.4, consisting of two streaming dataflows $J1$ and $J2$ where $J1$ performs a batch analytics query and $J2$ performs a latency sensitive anomaly detection pipeline. Each has an output operator processing messages from upstream operators. The default approach used by actor systems like Orleans is to: i) order messages based on arrival, and ii) give each operator a fixed time duration (called "quantum") to process its messages. Using

[1] We use latency constraint and latency requirement interchangeably.



Message	M1	M2	M3	M4
Time of Origin	0	30	40	40
Frontier Time	1000	30	1000	40
Execution Cost	30	20	30	10
Start Deadline	1070	60	1070	80

Figure 5.4: *Scheduling Example: J1 is batch analytics, J2 is latency-sensitive. Fair-share scheduler creates schedules "a" and "b". Topology-aware scheduler reduces violations ("c"). Semantics-aware scheduler further reduces violations ("d"). We further explain these examples in Section 5.4.2*

this approach we derive the schedule "a" with a small quantum, and a schedule "b" with a large quantum — both result in two latency violations for $J2$. In comparison, Cameo discovers the opportunity to postpone less latency-sensitive messages (and thus their target operators). This helps $J2$ meet its deadline by leveraging topology and query semantics. This is depicted in schedules "c" and "d". This example shows that *when* and *how long* an operator is scheduled to run should be dynamically determined by the priority of the next pending message. We expand on these aspects in the forthcoming sections.

5.4 SCHEDULING POLICIES IN CAMEO

One of our primary goals in Cameo is to enable fine-grained scheduling policies for dataflows. These policies can prioritize messages based on information, like the deadline remaining or processing time for each message, etc. To enable such policies, we require techniques that can calculate the priority of a message for a given policy.

We model our setting as a non-preemptive, non-uniform task time, multi-processor, real-time scheduling problem. Such problems are known to be NP-Complete offline and cannot be solved optially online without complete knowledge of future tasks [120, 121]. Thus, we consider how a number of commonly used policies in this domain, including Least-Laxity-First (LLF) [102], Earliest-Deadline-First (EDF) [101] and Shortest-Job-First (SJF) [122], and describe how such policies can be used for event-driven stream processing.We use the LLF policy as the default policy in our description below.

The above policies try to prioritize messages to avoid violating latency constraints. Deriving the priority of a message requires analyzing the impact of each operator in the dataflow on query performance. We next discuss how being deadline-aware can help Cameo derive

appropriate priorities. We also discuss how being aware of query semantics can further improve prioritization.

Symbol	Definition
ID_M	ID of Message M.
ddl_M	Message start deadline.
o_M	target operator of M.
C_{o_M}	Estimated execution cost of M on its target operator.
t_M, and p_M	Physical (and logical) time associated with the last event required to produce M.
L	Dataflow latency constraint of the dataflow that M belongs to.
p_{M_F}, and t_{M_F}	Frontier progress, and frontier time.

Table 5.1: *Notations used in paper for message M.*

5.4.1 Definitions and Underpinnings

Event. Input data arrives as *events*, associated with a *logical time* [123] that indicates the *stream progress* of these events in the input stream.

Dataflow job and operators. A dataflow job consists of a DAG of *stages*. Each stage operates a user-defined function. A stage can be parallelized and executed by a set of dataflow *operators*.

We say an operator o_k is *invoked* when it processes its input message, and o_k is *triggered* when it is invoked and leads to an output message, which is either passed downstream to further operators or the final job output.

Cameo considers two types of operators: i) regular operators that are triggered immediately on invocation; and ii) windowed operators [118] that partitions data stream into sections by logical times and triggers only when all data from the section are observed.

Message timestamps. We denote a message M as a tuple $(o_M, (p_M, t_M))$, where: a) o_M is the operator executing the message; b) p_M and t_M record the *logical* and *physical* time of the input stream that is associated with M, respectively. Intuitively, M is influenced by input stream with logical time $\leq p_M$. Physical time t_M marks the system time when p_M is observed at a source operator.

We denote C_{o_M} as the estimated time to process message M on target operator O, and L as the latency constraint for the dataflow that M belongs to.

Latency. Consider a message M generated as the output of a dataflow (at its sink operator). Consider the set of all events E that influenced the generation of M. We define latency as the difference between the last arrival time of any event in E and the time when M is generated.

5.4.2 Calculating Message Deadline

We next consider the LLF scheduling policy where we wish to prioritize messages which have the least laxity (i.e., flexibility). Intuitively, this allows us to prioritize messages that are closer to violating their latency constraint. To do this, we discuss how to determine the *latest time* that a message M can start executing at operator O without violating the job's latency constraint. We call this as the *start deadline* or in short the *deadline* of the message M, denoted as ddl_M. For the LLF scheduler, ddl_M is the message priority (lower value implies higher priority).

We describe how to derive the priority (deadline) using topology-awareness and then query (semantic)-awareness.

- Topology Awareness

  Single-operator dataflow, Regular operator. Consider a dataflow with only one regular operator o_M. The latency constraint is L. If an event occurs at time t_M, then M should complete processing before $t_M + L$. The start deadline, given execution estimate C_{o_M}, is:

$$ddl_M = t_M + L - C_{o_M} \tag{5.1}$$

  Multiple-operator dataflow, Regular operator. For an operator o inside a dataflow DAG that is invoked by message M, the start deadline of M needs to account for execution time of downstream operators. We estimate the maximum of execution times of critical path [42] from o to any output operator as C_{path}. The start deadline of M is then:

$$ddl_M = t_M + L - C_{O_M} - C_{path} \tag{5.2}$$

  Schedule "c" of Figure 5.4 showed an example of topology-aware scheduling and how topology awareness helps reduce violations. For example, $ddl_{M2} = 30 + 50 - 20 = 60$ means that $M2$ is promoted due to its urgency. We later show that even when query semantics are not available (e.g., UDFs), Cameo improves scheduling with topology information alone. Note that upstream operators are not involved in this calculation. C_{O_M} and C_{path} can be calculated by profiling.

- Query Awareness

Cameo can also leverage dataflow semantics, i.e., knowledge of user-specified commands inside the operators. This enables the scheduler to identify messages which can tolerate further delay without violating latency constraints. This is common for windowed operations, e.g., a `WindowAggregation` operator can tolerate delayed execution if a message's logical time is at the start of the window as the operator will only produce output at the end of a window. Window operators are very common in our production use cases.

Multiple-operator dataflow, Windowed operator. Consider M that targets a windowed operator o_M, Cameo is able to determine (based on dataflow semantics) to what extent M can be delayed without affecting latency. This requires Cameo to identify the minimum logical time (p_{M_F}) required to trigger the target window operator. We call p_{M_F} *frontier progress*. Frontier progress denotes the stream progress that needs to be observed at the window operator before a window is complete. Thus a windowed operator will not produce output until frontier progresses are observed at all source operators. We record the system time when all frontier progresses become available at all sources as *frontier time*, denoted as t_{M_F}.

Processing of a message M can be safely delayed until all the messages that belong in the window have arrived. In other words when computing the start deadline of M, we can extend the deadline by $(t_{M_F} - t_M)$. We thus rewrite Equation 5.2 as:

$$ddl_M = \mathbf{t_{M_F}} + L - C_{O_M} - C_{path} \tag{5.3}$$

An example of this schedule was shown in schedule "d" of Figure 5.4. With query-awareness, scheduler derives t_{M_F} and postpones $M1$ and $M3$ in favor of $M2$ and $M4$. Therefore operator $o2$ is prioritized over $o1$ to process $M2$ then $M4$.

The above examples show the derivation of priority for a LLF scheduler. Cameo also supports scheduling policies including commonly used policies like EDF, SJF etc. In fact, the priority for EDF can be derived by a simple modification of the LLF equations. Our EDF policy considers the deadline of a message prior to an operator executing and thus we can compute priority for EDF by omitting C_{O_M} term in Equation 5.3. For SJF we can derive the priority by setting $ddl_M = C_{O_M}$—while SJF is not deadline-aware we compare its performance to other policies in our evaluation.

5.4.3 Mapping Stream Progress

For Equation 5.3 frontier time t_{M_F} may not be available until the target operator is triggered. However, for many fixed-sized window operations (e.g., `SlidingWindow`, `TumblingWindow`, etc.), we can *estimate* t_{M_F} based on the message's logical time p_M. Cameo performs two steps: first we apply a TRANSFORM function to calculate p_{M_F}, the logical time of the message that triggers o_M. Then, Cameo infers the frontier time t_{M_F} using a PROGRESSMAP function. Thus $t_{M_F} = \text{PROGRESSMAP}(\text{TRANSFORM}(p_M))$. We elaborate below.

Step 1 (Transform): For a windowed operator, the completion of a window at operator o triggers a message to be produced at this operator. Window completion is marked by the increment of window ID [118, 124], calculated using the stream's logical time. For message M that is sent from upstream operator o_u to downstream operator o_d, p_{M_F} can be derived using p_M using on a TRANSFORM function. With the definition provided by [124], Cameo defines TRANSFORM as:

$$p_{M_F} = \text{TRANSFORM}(p_M) = \begin{cases} (p_M/S_{o_d} + 1) \cdot S_{o_d} & S_{o_u} < S_{o_d} \\ p_M & \text{otherwise} \end{cases} \tag{5.4}$$

For a sliding window operator o_d, S_{o_d} refers to the *slide size*, i.e., value step (in terms of logical time) for each window completion to trigger target operator. For the tumbling window operation (i.e., windows cover consecutive, non-overlapping value step), S_{o_u} equals the window size. For a message sent by an operator o_u that has a shorter slide size than its targeting operator o_d, p_{M_F} will be increased to the logical time to trigger o_d, that is, $= (p_M/S_{o_d} + 1) \cdot S_{o_d}$.

For example if we have a tumbling window with window size 10 s, then the expected frontier progress, i.e., p_{M_F}, will occur every 10th second (1, 11, 21 ...). Once the window operator is triggered, the logical time of the resultant message is set to p_{M_F}, marking the latest time to influence a result.

Step 2 (ProgressMap): After deriving the frontier progress p_{M_F} that triggers the next dataflow output, Cameo then estimates the corresponding frontier time t_{M_F}. A temporal data stream typically has its logical time defined in one of three different time domains:

(1) *event time* [125, 126]: a totally-ordered value, typically a timestamp, associated with original data being processed;

(2) *processing time*: system time for processing each operator [119]; and

(3) *ingestion time*: the system time of the data first being observed at the entry point of the system [125, 126].

Cameo supports both event time and ingestion time. For processing time domain, M's

timestamp could be generated when M is observed by the system.

To generate t_{M_F} based on progress p_{M_F}, Cameo utilizes a ProgressMap function to map logical time p_{M_F} to physical time t_{M_F}. For a dataflow that defines its logical time by data's ingestion time, logical time of each event is defined by the time when it was observed. Therefore, for *all* messages that occur in the resultant dataflow, logical time is assigned by the system at the origin as $t_{M_F} = \text{ProgressMap}(p_{M_F}) = p_{M_F}$.

For a dataflow that defines its logical time by the data's event time, $t_{M_F} \neq p_{M_F}$. Our stream processing run-time provides channel-wise guarantee of in-order processing for all target operators. Thus Cameo uses linear regression to map p_{M_F} to t_{M_F}, as: $t_{M_F} = \text{ProgressMap}(p_{M_F}) = \alpha \cdot p_{M_F} + \gamma$, where α and γ are parameters derived via a linear fit with running window of historical p_{M_F}'s towards their respective t_{M_F}'s. E.g., For same tumbling window with window size 10s, if p_{M_F} occurs at times $(1, 11, 21 \ldots)$, with a 2s delay for the event to reach the operator, t_{M_F} will occur at times $(3, 13, 23 \ldots)$.

We use a linear model due to our production deployment characteristics: the data sources are largely real time streams, with data ingested soon after generation. Users typically expect events to affect results within a constant delay. Thus the logical time (event produced) and the physical time (event observed) are separated by only a small (known) time gap. When an event's physical arrival time cannot be inferred from stream progress, we treat windowed operators as regular operators. Yet, this conservative estimate of laxity does not hurt performance in practice.

5.5 SCHEDULING MECHANISMS IN CAMEO

We next present Cameo's architecture that addresses three main challenges:

1 How to make static and dynamic information from both upstream and downstream processing available during priority assignment?

2 How can we efficiently perform fine-grained priority assignment and scheduling that scales with message volume?

3 How can we admit pluggable scheduling policies without modifying the scheduler mechanism?

Our approach to address the above challenges is to separate out the priority assignment from scheduling, thus designing a two-level architecture.This allows priority assignment for user-defined operators to become programmable.To pass information between the two levels (and across different operators) we piggyback information atop messages passed between operators.

More specifically, Cameo addresses challenge 1 by propagating *scheduling contexts* with

messages. To meet challenge 2, Cameo uses a two-layer scheduler architecture. The top layer, called the *context converter*, is embedded into each operator and handles scheduling contexts whenever the operator sends or receives a message. The bottom layer, called the *Cameo scheduler*, interprets message priority based on the scheduling context embedded within a message and updates a priority-based data structure for both operators and operators' messages. Our design has advantages of: (i) avoiding the bottleneck of having a centralized scheduler thread calculate priority for each operator upon arrival of messages, and (ii) only limiting priority to be per-message. This allow the operators, dataflows, and the scheduler, to all remain stateless.

To address 3 Cameo allows the priority generation process to be implemented through the context handling API. A context converter invokes the API with each operator.

5.5.1 Scheduling Contexts

Scheduling contexts are data structures attached to messages, capturing message priority, and information required to perform priority-based scheduling. Scheduling contexts are *created*, *modified*, and *relayed* alongside their respective messages. Concretely, scheduling contexts allow capture of scheduling states of both upstream and downstream execution. A scheduling context can be seen and modified by both context converters and the Cameo scheduler. There are two kinds of contexts:

1. **Priority Context** (`PC`): `PC` is necessary for the scheduler to infer the priority of a message. In Cameo `PC`s are defined to include local and global priority as (ID, PRI_{local}, PRI_{global}, $Dataflow_DefinedField$). PRI_{local} and PRI_{global} are used for applications to enclose message priorities for scheduler to determine execution order, and $Dataflow_DefinedField$ includes upstream information required by the pluggable policy to generate message priority.

A `PC` is attached to a message before the message is sent. It is either created at a source operator upon receipt of an event, or inherited and modified from the upstream message that triggers the current operator. Therefore, a `PC` is seen and modified by all executions of upstream operators that lead to the current message. This enables `PC` to address challenge 1 by capturing information of dependant upstream execution (e.g., stream progress, latency target, etc.).

2. **Reply Context** (`RC`): `RC` meets challenge 1 by capturing periodic feedback from the downstream operators. `RC` is attached to an acknowledgement message [2], sent by the target operator to its upstream operator after a message is received. `RC`s provide processing

[2] A common approach used by many stream processing systems [7, 8, 9] to ensure processing correctness

feedback of the target operator and all its downstream operators. `RC`s can be aggregated and relayed recursively upstream through the dataflow.

Cameo provides a programmable API to implement these scheduling contexts and their corresponding policy handlers in context converters. API functions include:

1. **function** BuildCxtAtSource(Event e) that creates a `PC` upon receipt of an event e;

2. **function** BuildCxtAtOperator(Message M) that modifies and propagates a PC when an operator is invoked (by M) and ready to send a message downstream;

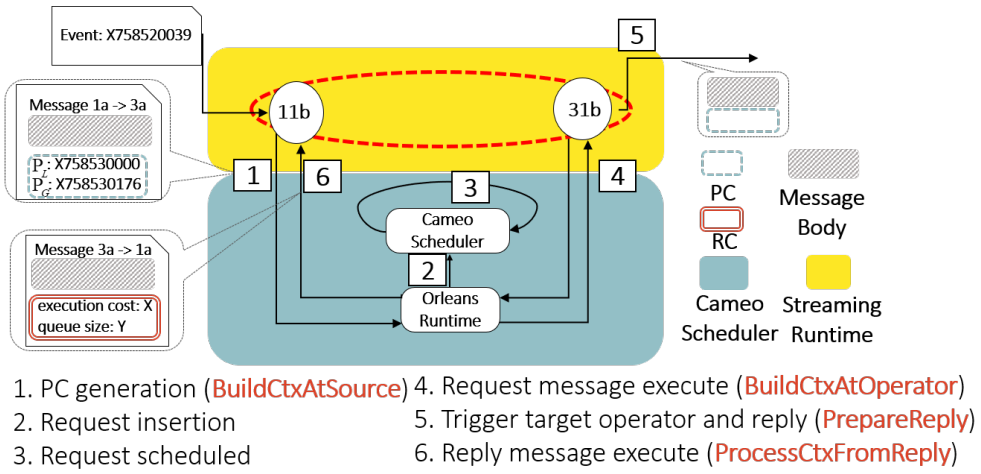
3. **function** ProcessCtxFromReply(Message r) that processes RC attached to an acknowledgement message r received at upstream operator; and

4. **function** PrepareReply(Message r) that generates RC containing user-defined feedbacks, attached to r sent by a downstream operator.

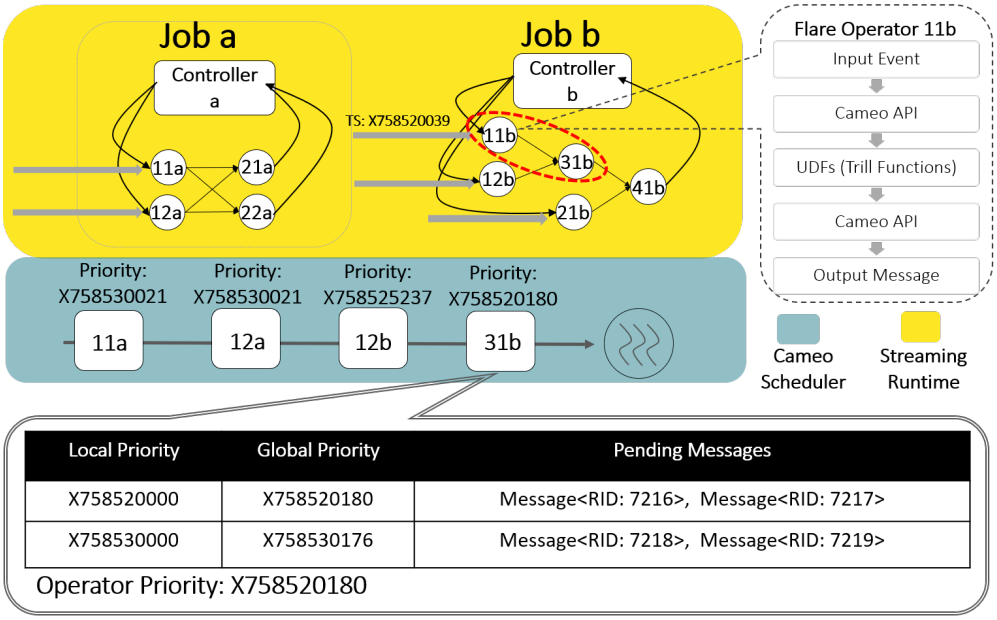
5.5.2 System Architecture

Figure 5.5a shows context converters at work. After an event is generated at a source operator $1a$ (step 1), the converter creates a `PC` through BuildCxtAtSource and sends the message to Cameo scheduler. The target operator is scheduled (step 2) with the priority extracted from the `PC`, before it is executed. Once the target operator $3a$ is triggered (step 4), it calls BuildCtxAtOperator, modifying and relaying PC with its message to downstream operators. After that $3a$ sends an acknowledgement message with an `RC` (through PrepareReply) back to $1a$ (step 5). `RC` is then populated by the scheduler with runtime statistics (e.g, CPU time, queuing delays, message queue sizes, network transfer time, etc.) before it is scheduled and delivered at the source operator (step 6).

Cameo enables scheduling states to be managed and transported alongside the data. This allows Cameo to meet challenge 2 by keeping the scheduler away from centralized state maintenance and priority generation. The Cameo scheduler manages a two level priority-based data structure, shown in Figure 5.5b. We use PRI_{local} to determine M's execution priority within its target operator, and PRI_{global} of the next message in an operator to order all operators that have pending messages. Cameo can schedule at either message granularity or a coarser time quanta. While processing a message, Cameo *peeks* at the priority of the next operator in the queue. If the next operator has higher priority, we swap with the current operator after a fixed time quantum (tunable).



(a) *Scheduling contexts circulating between two operators.*



(b) *Cameo Scheduler Architecture. Operators sorted by global priority. Messages at an operator sorted by local priority.*

Figure 5.5: *Cameo Mechanisms.*

5.5.3 Implementing the Cameo Policy

To implement the scheduling policy of Section 5.4, a `PC` is attached to message M (denoted as $PC(M)$) with fields as shown in Table 5.2.

Algorithm 5.1 demonstrates how Cameo performs context conversions. The core of Algorithm 5.1 is CxtConvert, which generates `PC` for downstream message M_d (denoted as $\texttt{PC}(M_d)$), triggered by $\texttt{PC}(M_u)$ from the upstream triggering message. To schedule a downstream message M_d triggered by M_u, Cameo first retrieves stream progress p_{M_u} contained in

ID	PRI_{local}	PRI_{global}	$Dataflow - DefinedField$
ID_M	p_{M_F}	ddl_{M_F}	(p_{M_F}, t_{M_F}, L)

Table 5.2: *Cameo Message Example.*

Algorithm 5.1 Priority Context Conversion

1: **function** BuildCxtAtSource(Event e) ▷ Generate PC for message M_e at source triggered by event e
2: $\texttt{PC}(M_e) \leftarrow$ InitializePriorityContext()
3: $\texttt{PC}(M_e).(PRI_{local}, PRI_{global}) \leftarrow (e.p_e, e.t_e)$
4: $\texttt{PC}(M_e) \leftarrow$ ContextConvert$(\texttt{PC}(M_e), \texttt{RC}_{\texttt{local}})$
5: **return** $\texttt{PC}(M_e)$

6: **function** BuildCxtAtOperator(Message M_n) ▷ Generate PC for message M_d at an intermediate operator triggered by upstream message M_u
7: $\texttt{PC}(M_d) \leftarrow \texttt{PC}(M_u)$
8: $\texttt{PC}(M_d).(PRI_{local}, PRI_{global}) \leftarrow \texttt{PC}(M_u).(p_{M_F}, t_{M_F})$
9: $\texttt{PC}(M_d) \leftarrow$ ContextConvert$(\texttt{PC}(M_d), \texttt{RC}_{\texttt{local}})$
10: **return** $\texttt{PC}(M_d)$

11: **function** CxtConvert$(\texttt{PC}(M), \texttt{RC})$ ▷ Calculating message priority based on $\texttt{PC}(M), \texttt{RC}$ provided
12: $p_{M_F} \leftarrow$ Transform$(\texttt{PC}(M).p_M)$
13: $t_{M_F} \leftarrow$ ProgressMap(p_{M_F}) ▷ As in Section 5.4.3
14: **if** t_{M_F} defined in stream event time **then**
15: ProgressMap.update$(\texttt{PC}.t_M, \texttt{PC}.p_M)$ ▷ Improving prediction model as in Section 5.5.3
16: $\texttt{PC}(M).p_M, \texttt{PC}(M).t_M \leftarrow p_{M_F}, t_{M_F}$
17: $ddl_M \leftarrow t_{M_F} + \texttt{PC}(M).L - \texttt{RC}.C_m - \texttt{RC}.C_{path}$
18: $\texttt{PC}(M).(PRI_{local}, PRI_{global}) \leftarrow (p_{M_F}, ddl_M)$

19: **function** ProcessCtxFromReply(Message r) ▷ Retrieve reply message's RC and store locally
20: $\texttt{RC}_{\texttt{local}}.update(r.\texttt{RC})$

21: **function** PrepareReply(Message r) ▷ Recursively update maximum critical path cost C_{path} before reply
22: **if** Sender(r) = Sink **then**
23: $r.\texttt{RC} \leftarrow$ InitializeReplyContext()
24: **else** $r.\texttt{RC}.C_{path} \leftarrow \texttt{RC}.C_m + \texttt{RC}.C_{path}$

PC(M_u). It then applies the two-step process (Section 5.4.3) to calculate frontier time t_{M_F} using p_{M_u}. This may extend a message's deadline if the operator is not expected to trigger immediately (e.g., windowed operator). We capture p_{M_F} and estimated t_{M_F} in PC as message priority and propagate this downstream. Meanwhile, p_{M_u} and t_{M_u} are fed into a linear model to improve future prediction towards t_{M_F}. Finally, the context converter computes message priority ddl_{M_u} using t_{M_F} as described in Section 5.4.

Cameo utilizes RC to track critical path execution cost C_{path} and execution cost C_{o_M}. RC contains the processing cost (e.g., CPU time) of the downstream critical path up to the current operator, obtained via profiling.

5.5.4 Customizing Cameo: Proportional Fair Scheduling

We next show how the pluggable scheduling policy in Cameo can be used to support other performance objectives, thus satisfying 3. For instance, we show how a token-based rate control mechanism works, where token rate equals desired output rate. In this setting, each application is granted tokens per unit of time, based on their target sending rate. If a source operator exceeds its target sending rate, the remaining messages (and all downstream traffic) are processed with operator priority reduced to minimum. When capacity is insufficient to meet the aggregate token rate, all dataflows are downgraded equally. Cameo spreads tokens proportionally across the next time interval (e.g., 1 sec) by tagging each token with the timestamp at each source operator. For token-ed messages, we use token tag PRI_{global}, and interval ID as PRI_{local}. Messages without tokens have PRI_{global} set to MIN_VALUE. Through PC propagation, all downstream messages are processed when no tokened traffic is present.

Figure 5.6 shows Cameo's token mechanism. Three dataflows start with 20% (12), 40% (24), and 40% (24) tokens as target ingestion rate per source respectively. Each ingests 2M events/s, starting 300 s apart, and lasting 1500 s. Dataflow 1 receives full capacity initially when there is no competition. The cluster is at capacity after Dataflow 3 arrives, but Cameo ensures token allocation translates into throughput shares.

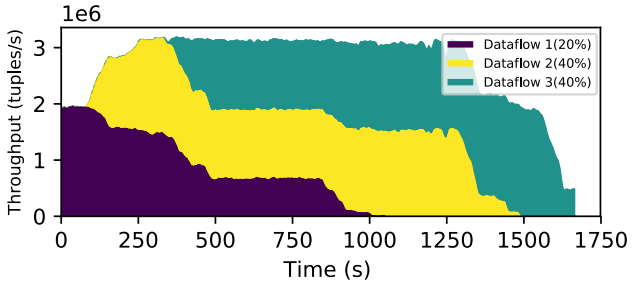


Figure 5.6: *Proportional fair sharing using Cameo.*

5.6 EXPERIMENTAL EVALUATION

We next present experimental evaluation of Cameo. We first study the effect of different queries on Cameo in a single-tenant setting. Then for multi-tenant settings, we study Cameo's effects when:

- **Varying environmental parameters** (Section 5.6.2): This includes: a) workload (tenant sizes and ingestion rate), and b) available resources, i.e., worker thread pool size, c) workload bursts.
- **Tuning internal parameters and optimization** (Section 5.6.3): We study: a) effect of scheduling granularity, b) frontier prediction for event time windows, and c) starvation prevention.

We implement streaming queries in Flare [17] (built atop Orleans [92, 93]) by using Trill [123] to run streaming operators.We compare Cameo vs. both i) default **Orleans** (version 1.5.2) scheduler, and ii) a custom-built **FIFO** scheduler. By default, we use the 1 ms minimum re-scheduling grain (Section 5.5.2). This grain is generally shorter than a message's execution time. Default Orleans implements a global run queue of messages using a ConcurrentBag [127] data structure. ConcurrentBag optimizes processing throughput by prioritizing processing thread-local tasks over the global ones. For the FIFO scheduler, we insert operators into the global run queue and extract them in FIFO order. In both approaches, an operator processes its messages in FIFO order.

Machine configuration. We use DS12-v2 Azure virtual machines (4 vCPUs/56GB memory/112G SSD) as server machines, and DS11-v2 Azure virtual machines (2 vCPUs/14GB memory/28G SSD) as client machines [128]. Single-tenant scenarios are evaluated on a single server machine. Unless otherwise specified, all multi-tenant experiments are evaluated using a 32-node Azure cluster with 16 client machines.

Evaluation workload. For the multi-job setting we study performance isolation under concurrent dataflow jobs. Concretely, our workload is divided into two control groups:

- **Latency Sensitive Jobs (Group 1):** This is representative of jobs connected to user dashboards, or associated with SLAs, ongoing advertisement campaigns, etc. Our workload jobs in Group 1 have sparse input volume across time (1 msg/s per source, with 1000 events/msg), and report periodic results with shorter aggregation windows (1 second). These have strict latency constraints.
- **Bulk Analytic Jobs (Group 2):** This is representative of social media streams being processed into longer-term analytics with longer aggregation windows (10 seconds). Our Group 2 jobs have input of both higher and variable volume and high speed, but with

lax latency constraints.

Our queries feature multiple stages of windowed aggregation parallelized into a group of operators. Each job has 64 client sources. All queries assume input streams associated with event time unless specified otherwise.

Latency constraints. In order to determine the latency constraint of one job, we run multiple instances of the job until the resource (CPU) usage reaches 50%. Then we set the latency constraint of the job to be twice the tail (95th percentile) latency. This emulates the scenario where users with experience in isolated environments deploy the same query in a shared environment by moderately relaxing the latency constraint. Unless otherwise specified, a latency target is marked with grey dotted line in the plots.

5.6.1 Single-tenant Scenario

In Figure 5.7 we evaluate a single-tenant setting with 4 queries: IPQ1 through IPQ4. IPQ1 and IPQ3 are periodic and they respectively calculate sum of revenue generated by real time ads, and the number of events generated by jobs groups by different criteria. IPQ2 performs similar aggregation operations as IPQ1 but on a sliding window (i.e., consecutive window contains overlapped input). IPQ4 summarizes errors from log events via running a windowed join of two event stream, followed by aggregation on a tumbling window (i.e., where consecutive windows contain non-overlapping ranges of data that are evenly spaced across time).

From Figure 5.7a we observe that Cameo improves median latency by up to 2.7×and tail latency by up to 3.2×.We also observe that default Orleans performs almost as well as Cameo for IPQ4. This is because IPQ4 has a higher execution time with heavy memory access, and performs well when pinned to a single thread with better access locality.

Effect on intra-query operator scheduling. The CDF in Figure 5.7b shows that Orleans' latency is about 3× higher than Cameo. While FIFO has a slightly lower median latency, its tail latency is as high as in Orleans. Cameo's prioritization is especially evident in Figure 5.7c, where dots are message starts, and red lines separate windows. We first observe that Cameo is faster, and it creates a clearer boundary between windows. Second, messages that contribute to the first result (colored dots) and messages that contribute to the second result (grey dots) do not overlap on the timeline. For the other two strategies, there is a *drift* between stream progress in early stages vs. later stages, producing a prolonged delay. In particular, in Orleans and FIFO, early-arriving messages from the next window are executed *before* messages from the first window, thus missing deadlines.

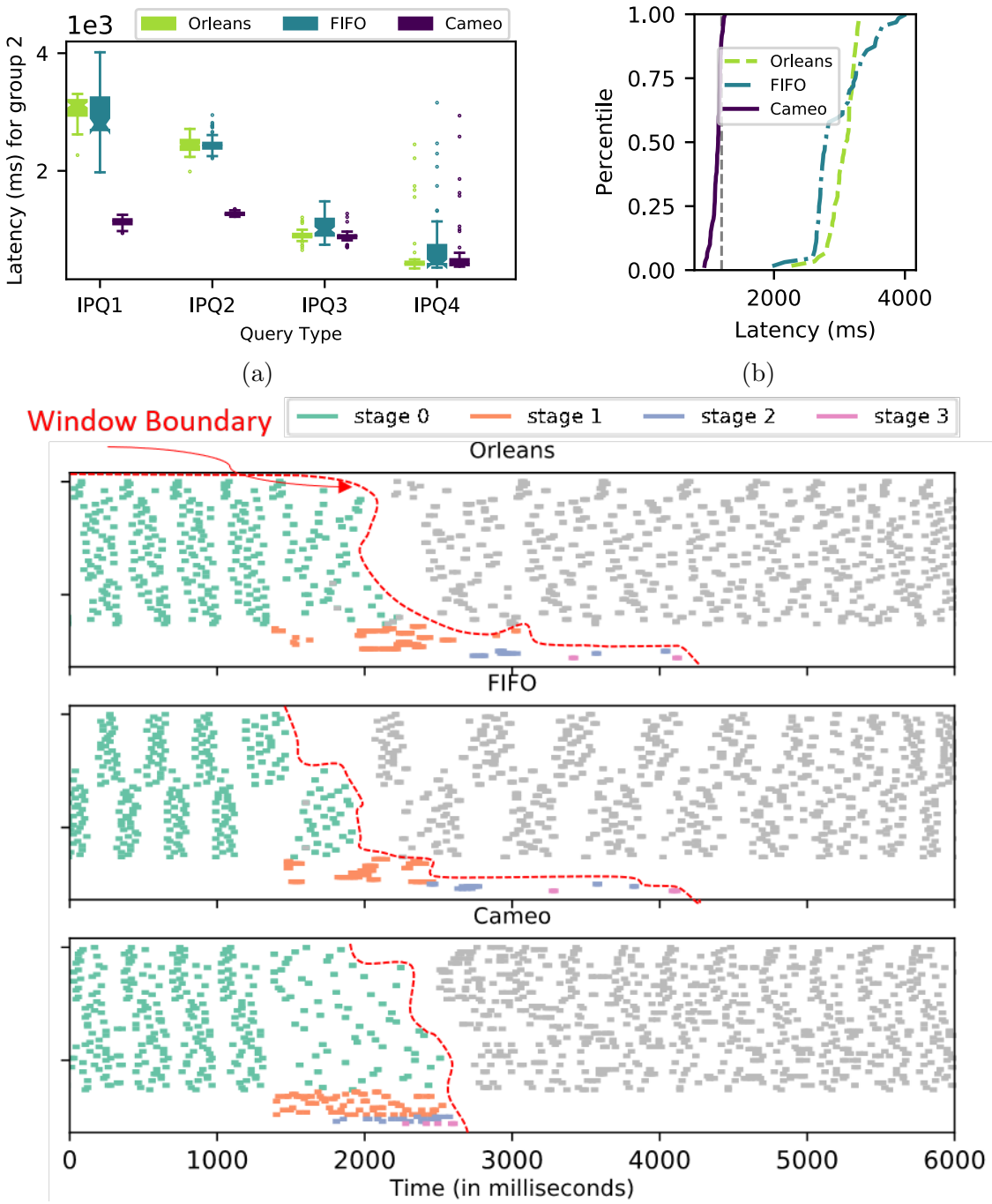
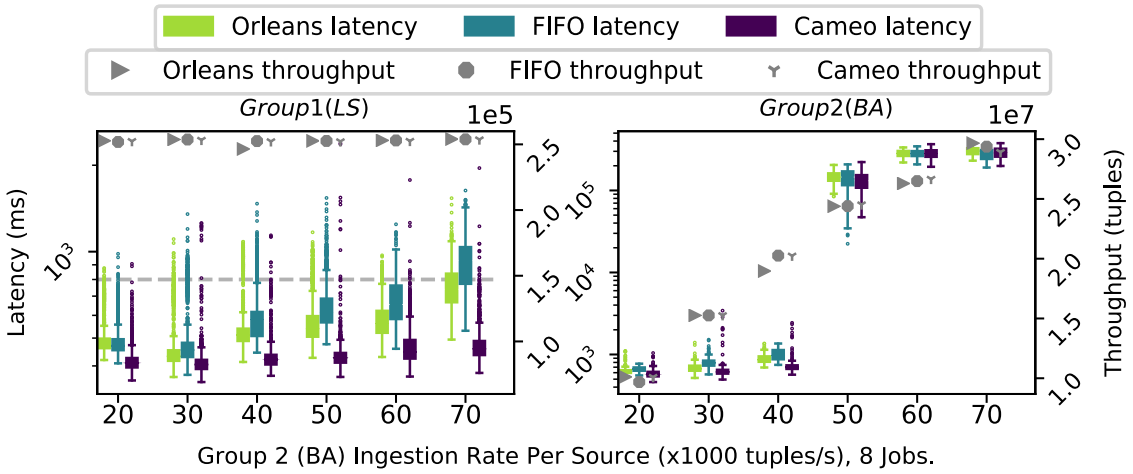
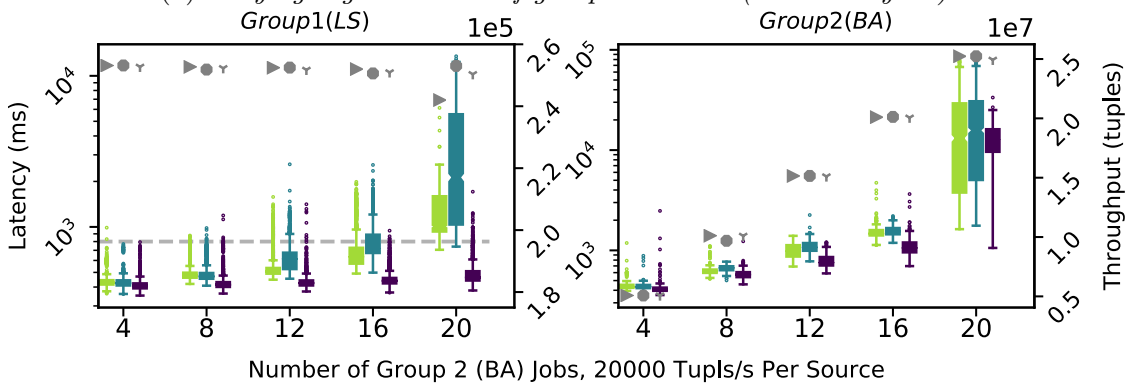


Figure 5.7: *Single-Tenant Experiments: (a) Query Latency. (b) Latency CDF. (c) Operator Schedule Timeline: X axis = time when operator was scheduled. Y axis = operator ID color coded by operator's stage. Operators are triggered at each stage in order (stage 0 to 3). Job latency is time from all events that belong to the previous window being received at stage 0, until last message is output at stage 3.*

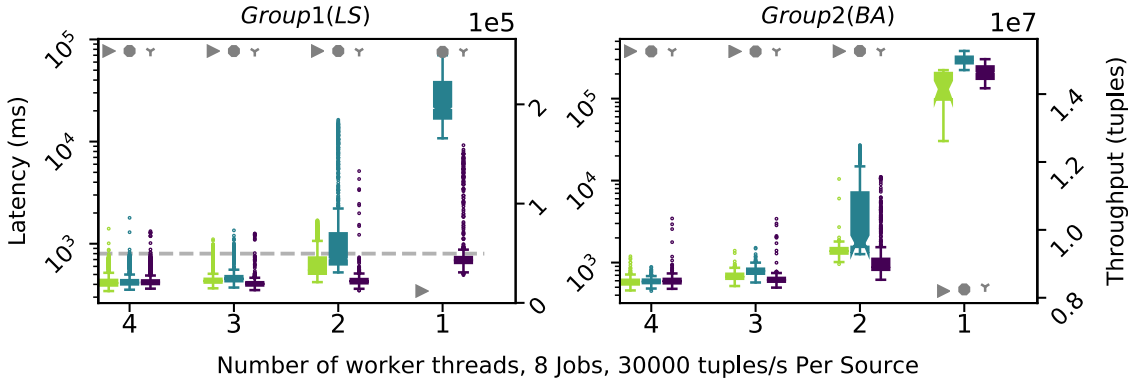
5.6.2 Multi-tenant Scenario



(a) *Varying ingestion rate of group 2 tenants (Bulk Analytics).*



(b) *Varying number of group 2 tenants (Bulk Analytics).*



(c) *Varying worker thread pool size.*

Figure 5.8: *Latency-sensitive jobs under competing workloads.*

Figure 5.8 studies a control group of latency-constrained dataflows (group 1 LS jobs) by fixing both job count and data ingestion rate. We vary data volume from competing workloads (group 2 BA jobs) and available resources. For LS jobs we impose a latency

target of 800 ms, while for BA jobs we use a 7200s latency constraint.

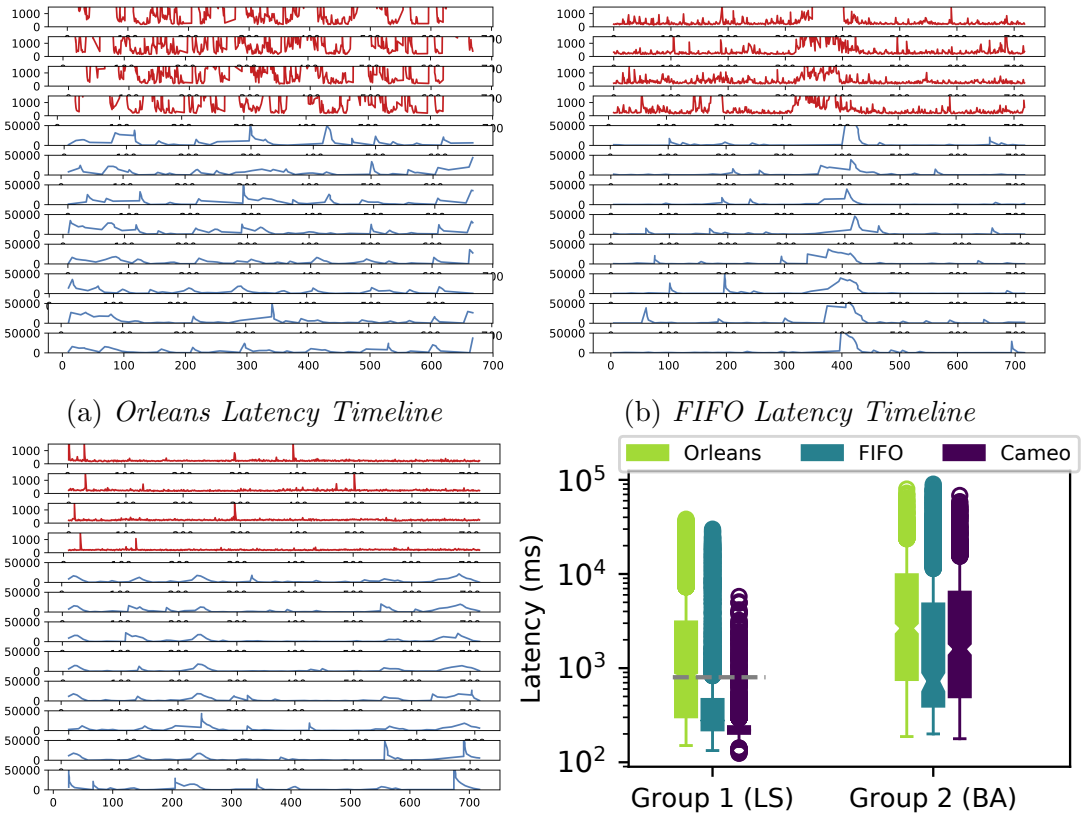
Cameo under increasing data volume. We run four group 1 jobs alongside group 2 jobs. We increase the competing group 2 jobs' traffic, by increasing the ingestion speed (Figure 5.8a) and number of tenants (Figure 5.8b). We observe that all three strategies (Cameo, Orleans, FIFO) are comparable up to per-source tuple rate of 30K/s in Figure 5.8a, and up to twelve group 2 jobs in Figure 5.8b. Beyond this, overloading causes massive latency degradation, for group 1 (LS) jobs at median and 99 percentile latency (respectively): i) Orleans is worse than Cameo by up to 1.6 and 1.5× in Figure 5.8a, up to 2.2 and 2.8× in Figure 5.8b, and ii) FIFO is worse than Cameo by up to 2 and 1.8× in Figure 5.8a, up to 4.6 and 13.6× in Figure 5.8b. Cameo stays stable. Cameo's degradation of group 2 jobs is small— with latency similar or lower than Orleans and FIFO, and Cameo's throughput only 2.5% lower.

Effect of limited resources. Orleans' [105] underlying SEDA architecture [129] resizes thread pools to achieve resource balance between execution steps, for dynamic reprovisioning. Figure 5.8c shows latency and throughput when we decrease the number of worker threads. Cameo maintains the performance of group 1 jobs except in the most restrictive 1 thread case (although it still meets 90% of deadlines). Cameo prefers messages with impending deadlines and this causes back-pressure for jobs with less-restrictive latency constraints, lowering throughput. Both Orleans and FIFO observe large performance penalties for group 1 and 2 jobs (higher in the former). Group 2 jobs with much higher ingestion rate will naturally receive more resources upon message arrivals, leading to back-pressure and lower throughput for group 1 jobs.

Effect of temporal variation of workload. We use a Pareto distribution for data volume in Figure 5.9, with four group 1 jobs and eight group 2 jobs. (This is based on Figures 5.2a, 5.2c, which showed a Power-Law-like distribution.) The cluster utilization is kept under 50%.

High ingestion rate can suddenly prolong queues at machines. Visualizing timelines in Figures 5.9a, 5.9b, and 5.9c shows that for latency-constrained jobs (group 1), Cameo's latency is more stable than Orleans' and FIFO's. Figure 5.9d shows that Cameo reduces (median, 99th percentile) latency by (3.9×, 29.7×) vs. Orleans, and (1.3×, 21.1×) vs. FIFO. Cameo's standard deviation is also lower by 23.2× and 12.7× compared to Orleans and FIFO respectively. For group 2, Cameo produces smaller average latency and is less affected by ingestion spikes. Transient workload bursts affect many jobs, e.g., all jobs around $t = 400$ with FIFO, as a spike at one operator affects all its collocated operators.

Ingestion pattern from production trace. Production workloads exhibit high degree of skew across data sources. In Figure 5.10 we show latency distribution of dataflows consuming



(a) *Orleans Latency Timeline*

(b) *FIFO Latency Timeline*

(c) *Cameo Latency Timeline*

(d) *Latency Distribution*

Figure 5.9: *Latency under Pareto event arrival.*

two workload distributions derived from Figure 5.2c: Type 1 and 2. Type 1 produces twice as many events as Type 2. However, Type 2 is heavily skewed and its ingestion rate varies by 200× across sources. This heavily impacts operators that are collocated. The success rate (i.e., the fraction of outputs that meet their deadline) is only 0.2% and 1.5% for Orleans and 7.9% and 9.5% for FIFO. Cameo prioritizes critical messages, maintaining success rates of 21.3% and 45.5% respectively.

5.6.3 Cameo: Internal Evaluation

We next evaluate Cameo's internal properties.

LLF vs. EDF vs. SJF. We implement three scheduling policies using the Cameo context API and evaluate using Section 5.6.1's workload. The three scheduling policies are: Least Laxity First (LLF, our default), Earliest Deadline First (EDF), and Shortest Job First (SJF). Figure 5.11 shows that SJF is consistently worse than LLF and EDF (with the exception of query IPQ4– due to the lack of queuing effect under lighter workload). Second, EDF and

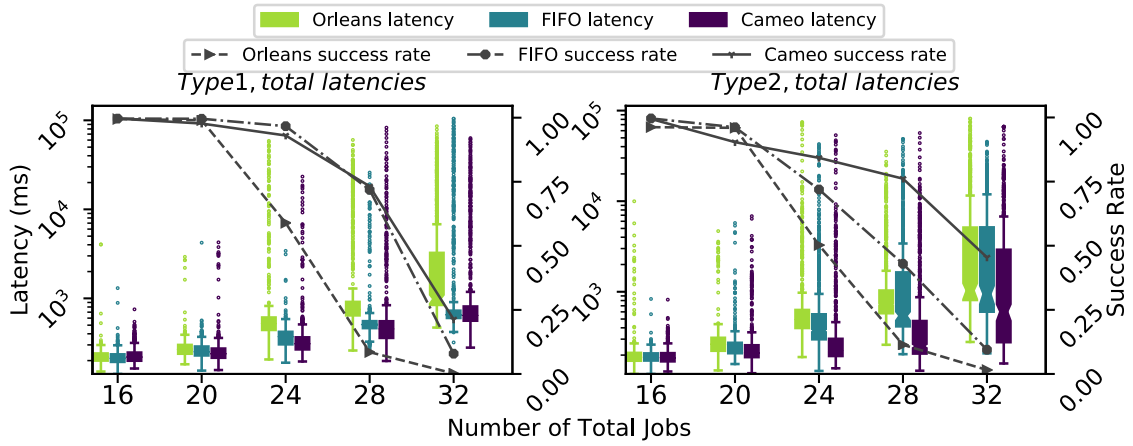


Figure 5.10: *Spatial Workload Variation.*

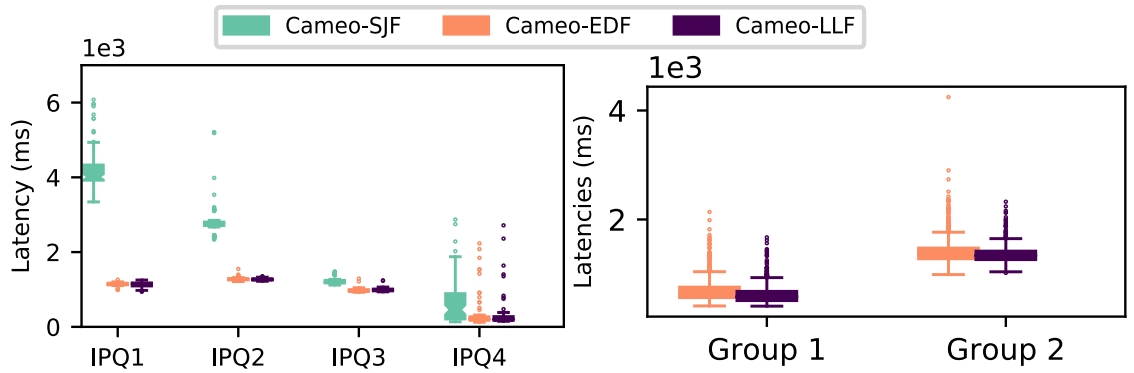


Figure 5.11: *Cameo Policies. Left: Single query latency distribution. Right: Multi-Query Latency Distribution.*

LLF perform comparably.

In fact we observed that EDF and LLF produced similar schedules for most of our queries. This is because: i) our operator execution time is consistent within a stage, and ii) operator execution time is $\ll$ window size. Thus, excluding operator cost (EDF) does not change schedule by much.

Scheduling Overhead. To evaluate Cameo with many small messages, we use one thread to run a no-op workload (300-350 tenants, 1 msg/s/tenant, same latency needs). Tenants are increased to saturate throughput.

Figure 5.12 (left) shows breakdown of execution time (inverse of throughput) for three scheduling schemes: FIFO, Cameo without priority generation (overhead only from priority scheduling), and Cameo with priority generation and the LLF policy from Section 5.4 (overhead from both priority scheduling and priority generation). Cameo's scheduling overhead is $< 15\%$ of processing time in the worst case, comprising of 4% overhead from priority-based scheduling and 11% from priority generation.

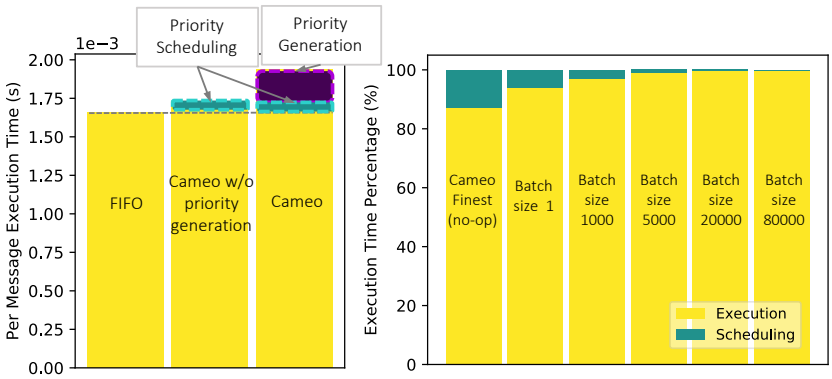


Figure 5.12: *Cameo Scheduling Overhead.*

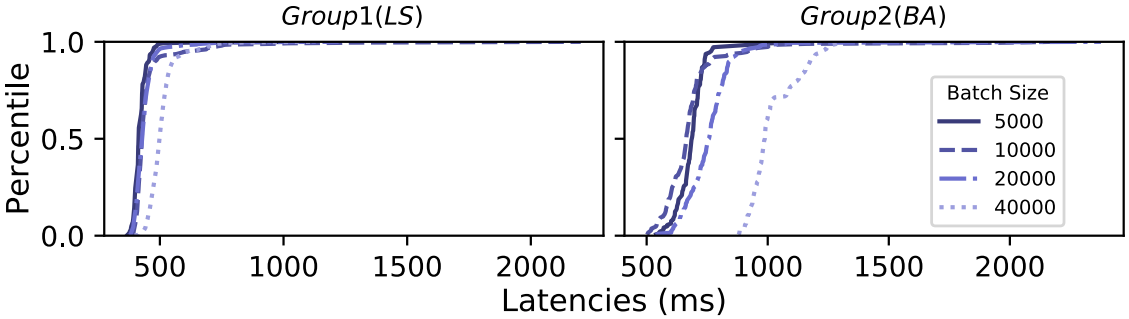


Figure 5.13: *Effect of Batch Size.*

In practice, Cameo encloses a columnar batch of data in each message like Trill [123]. Cameo's overhead is small compared to message execution costs. In Figure 5.12 (right), under Section 5.6's workload, scheduling overhead is only 6.4% of execution time for a local aggregation operator with batch size 1. Overhead falls with batch size. When Cameo is used as a generalized scheduler and message execution costs are small (e.g., with < 1 ms), we recommend tuning scheduling quantum and message size to reduce scheduling overhead.

In Figure 5.13, we batch more tuples into a message, while maintaining same overall tuple ingestion rate. In spite of decreased flexibility available to the scheduler, group 1 jobs' latency is unaffected up to 20K batch size. It degrades at higher batch size (40K), due to more lower priority tuples blocking higher priority tuples. Larger messages hide scheduling overhead, but could starve some high priority messages.

To evaluate the effect of increasing the scheduling quantum, we evaluate Cameo's performance under varying scheduling quantum (Section 5.5.2) using workload described in Figure 5.10. Figure 5.14 (Left) shows the latency distribution with *all* Type 1 and Type 2 jobs trigger dataflow output on the *same* stream progress (e.g., 10, 20, 30s ...), whereas Figure 5.14 (Right) shows the result with jobs's output triggered on *interleaved* stream progress (e.g., job 1 on 10, 20, 30s etc., job 2 on 12, 22, 32s, etc.). The left figure reveals the potential benefits of using coarser scheduling quantum, as resources are contended for

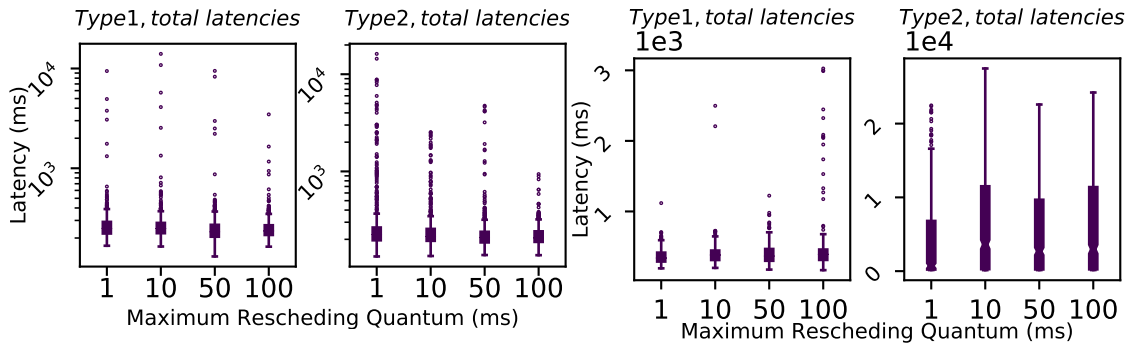


Figure 5.14: *Effect of Varying Scheduling Quantum. Left: Jobs Triggered By Clustered Stream Progress. Right: Jobs Triggered By Interleaved Stream Progress.*

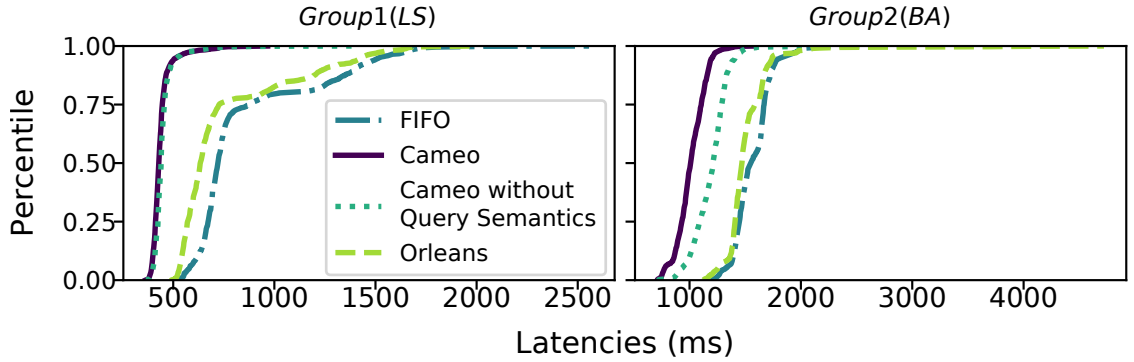


Figure 5.15: *Benefit of Query Semantics-awareness in Cameo.*

by many high priority messages, using finest scheduling granularity causes longer latency tail due to frequent context switches. However, a very large scheduling quantum (100ms) can hurt Cameo's performance by prohibiting the scheduler from preempting low-priority operators that arrive early, creating head-of-line blocking for high priority messages.

Varying Scope of Scheduler Knowledge. If Cameo is unaware of query semantics (but aware of DAG and latency constraints), Cameo conservatively estimates t_{M_F} without deadline extension for window operators, causing a tighter ddl_M. Figure 5.15 shows that Cameo performs slightly worse without query semantics (19% increase in group 2 median latency). Against baselines, Cameo still reduces group 1 and group 2's median latency by up to 38% and 22% respectively. Hence, even without query semantic knowledge, Cameo still outperforms Orleans and FIFO.

Effect of Measurement Inaccuracies. To evaluate how Cameo reacts to inaccurate monitoring profiles, we perturb measured profile costs (C_{O_M} from Equation 5.3) by a normal distribution (μ=0), varying standard deviation (σ) from 0 to 1 s. Figure 5.16 shows that when σ of perturbation is close to window size (1 s), latency is: i) stable at the median, and ii) modestly increases at tail, e.g., only by 55.5% at the 90th percentile. Overall, Cameo's performance is robust when standard deviation is $\leq$ 100ms, i.e., when measurement error is

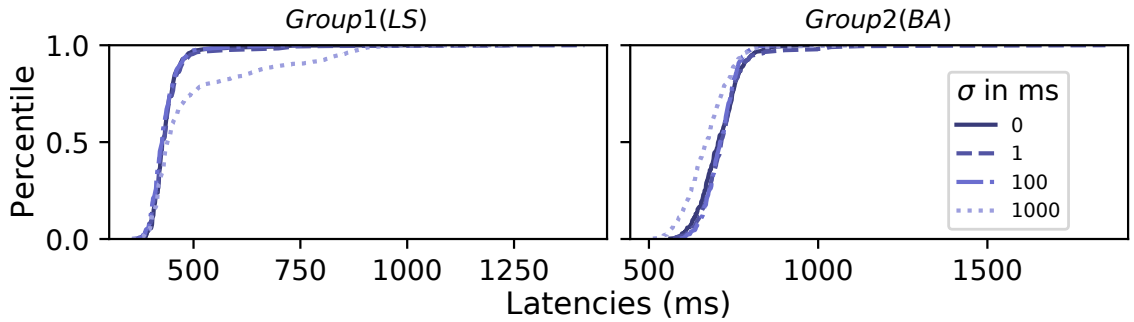


Figure 5.16: *Profiling Inaccuracy. Standard deviation in ms.*

reasonably smaller than output granularity.

5.7 RELATED WORK

Streaming system schedulers. The first generation of Data Stream Management Systems (DSMS) [113, 130], such as Aurora [116], Medusa [131] and Borealis [59], use QoS based control mechanisms with load shedding to improve query performance at run time. These are either centralized (single-threaded) [116], or distributed [59, 131] but do not handle timestamp-based priorities for partitioned operators. TelegraphCQ [114] orders input tuples before query processing [132, 133], while Cameo addresses operator scheduling within and across query boundaries. Stanford's STREAM [115] uses chain scheduling [117] to minimize memory footprints and optimize query queues, but assumes all queries and scheduler are execute in a single-thread. More recent works in streaming engines propose operator scheduling algorithms for query throughput [134] and latency [135, 136]. Reactive and operator-based policies include [134, 135], while [136] assumes arrivals are periodic or Poisson—however, these works do not build a framework (like Cameo), nor do they handle per-event semantic awareness for stream progress.

Modern stream processing engines such as Spark Streaming [10], Flink [99], Heron [8], MillWheel [137], Naiad [138], Muppet [139], Yahoo S4 [140]) do not include *native* support for multi-tenant SLA optimization. These systems also rely on coarse-grained resource sharing [100] or third-party resource management systems such as YARN [32] and Mesos [33].

Streaming query reconfiguration. Online reconfiguration has been studied extensively [141]. Apart from Figure 5.3, prior work addresses operator placement [103, 142], load balancing [143, 144], state management [44], policies for scale-in and scale-out [37, 45, 51, 145]. Among these are techniques to address latency requirements of dataflow jobs [45, 51], and ways to improve vertical and horizontal elasticity of dataflow jobs in containers [146]. The performance model in [38] focuses on dataflow jobs with latency constraints, while we fo-

cus on interactions among operators. Online elasticity was targeted by System S [36, 41], StreamCloud [49] and TimeStream [147]. Others include [2, 148]. Neptune [104] is a proactive scheduler to suspend low-priority batch tasks in the presence of streaming tasks. Yet, there is no operator prioritization *within* each application. Edgewise [149] is a queue-based scheduler based on operator load but not query semantics. All these works are orthogonal to, and can be treated as pluggables in, Cameo.

Event-driven architecture for real-time data processing. This area has been popularized by the resource efficiency of serverless architectures [95, 97, 150]. Yet, recent proposals [109, 110, 151, 152] for stream processing atop event-based frameworks do not support performance targets for streaming queries.

5.8 CONCLUSION

We proposed Cameo, a fine-grained scheduling framework for distributed stream processing. To realize flexible per-message scheduling, we implemented a stateless scheduler, contexts that carry important static and dynamic information, and mechanisms to derive laxity-based priority from contexts. Our experiments with real workloads, and on Microsoft Azure, showed that Cameo achieves $2.7 \times -4.6\times$ lower latency than competing strategies and incurs overhead less than 6.4%.

CHAPTER 6: SCHEDULING PERFORMANCE CRITICAL STATEFUL FUNCTIONS WITH DIRIGO

6.1 INTRODUCTION

Real-time data are known to bring unique challenges to dataflow engine design due to their unpredictable volume, velocity, and arrival patterns. Handling real-time data requires elastic resource provisioning to provide predictable service under the influence of many environmental variables. Most state-of-the-art real-time dataflow engines rely on *application-level reconfigurations* performed manually by end-users or resource managers, which re-compiles and re-generates dataflow plans when resource overloads (or underloads) are detected [36, 37]. Recently, we have observed a trend that real-time data processing engines are evolving to adopt *serverless architecture* [107, 153]. Contradict to most dataflow engines today, which statically associate dataflow operator(s) with isolated resources, this approach models dataflow operators as serverless functions and assigns resources dynamically for functions with pending work.

This approach provides the obvious benefit of reducing resource idleness: The event-driven architecture allows resources (e.g., CPUs) to be work-conserving, creating high resource utilization [129]. Through this deployment paradigm, service providers gain fine-grained access to pending work in the queue, allowing them to make better decisions in response to user-level performance requirements or resolving performance bottlenecks without requiring user-level intervention [4]. And therefore, users are not required to monitor and adapt their dataflow deployment constantly.

While this architecture can fundamentally change how to achieve system elasticity within real-time data processing engines, one of the significant issues with serverless platforms today is the lack of *native* support to *function states*. As many real-time dataflow applications contain operations requiring access to functions states, today's serverless platforms treat all functions as stateless. All function states used during an invocation are considered ephemeral.Users who aim to reduce remote storage accesses prefer a longer instance lifetime that allows function states to be reused across many requests [154]. For data-centric large-scale applications, users need to rely on the maximum time span enforced by the provider (900 seconds for AWS Lambda [155]) to maintain in-memory states [156]. As a result, users are again required to explicitly reason about the trade-off between frequent cold starts or resource idleness, making it challenging to deploy stateful dataflows that contain fine-grained, state-accessing tasks that are difficult to predict.

Dirigo builds a serverless function processing stack designed for real-time dataflow appli-

cations based on actor-based architecture. It proposes that the future serverless runtime systems should aim to perform elastic actions fully automatically without user involvement. This framework should satisfy the following criteria: i) providing native support for *addressable function state* ii) scheduling mechanism that enables *fine-grained*, *decentralized* function dispatch iii) supporting user-intent awareness. Dirigo allows functions to communicate with each other *directly* through *messages*, and proposes a scheduling framework to perform *decentralized* message scheduling. The design of Dirigo's scheduling policies is guided by the following problems:

1. How does user-specified performance intent translate to machine/worker choices for pending messages to be executed.

2. How do accesses to states affect scheduling choices and how to reason about the trade-off between load-balancing and state locality.

3. How should scheduler supports advanced stateful operations such as distributed aggregation [157].

We also study a set of scheduling policies and the best scheduling policy for different types of real-time workloads.

Because real-time systems are "fast-moving" with messages arriving continuously, the queue states of workers change very quickly. Therefore it is prohibitive to design "static" or "optimal" solutions to this scheduling problem, and our approach is to design fast and efficient mechanisms and policies with Dirigo.

6.2 MOTIVATION

Real-time data processing systems have long been one of the major processing frameworks adopted by many user scenarios. Traditionally designed for data streaming applications, these systems must provide timely services for long-running applications consuming both online and offline data streams. With low processing response time and close integration with network data storage, today these systems are used for applications far beyond the realm of stream processing, including event-driven applications, microservice pipelines, and ML training/inference applications [109], etc. This emerging trend brings new challenges in designing real-time data processing frameworks that serve as unified processing engines for emerging applications, including adaptation to various user requirements, exploiting the processing semantics of different applications, and adaptively managing application states. Recent efforts that bring us closer to this vision attempt to build real-time data processing

frameworks using serverless architecture. This new design decouples operator logic from operator deployment, leaving a significant challenge for service providers to optimize function scheduling guided by user requirements for application performance.

We discovered that scheduler design targeting today's serverless platforms could not be directly applied to our scenario. Dirigo extends an actor-like architecture that we studied in Chapter 5 and aims to exploit opportunities to parallelize operators without violating processing semantics. The benefit of using an actor-like underlying framework is that each function is associated with its addressable state(s), allowing future events that target the same function to access its previously accessed state(s). Therefore, the end-user could assume that states objects are present whenever the functions are invoked. However, the actor-based framework uses single-threaded, lock-free processing semantics. As a result, the users need to *constantly* reason about how to provision resources for each stage of their applications in real-time, preventing elastic actions from being performed efficiently. Aiming to minimize user effort, Dirigo performs dynamic scale-out and scale-in for operators if permitted by processing semantics.

However, this design poses a different challenge to scheduler design: functions in Dirigo are directly addressable, meaning that any function in the system can invoke another function by its globally identifiable address by inserting a message in its target function's queue. Whereas for most serverless frameworks today, functions do not directly communicate with each other, and all pending events are sent to a queue structure before being dispatched by an external, logically centralized entity (e.g., [150]).

Dirigo requires a *fully decentralized*, *per-worker* fine-grained function scheduler that can serve stateful dataflow applications with user intent. As Dirigo also manages function states within the runtime, it could exploit state locality between invocations that target the same function in its scheduling policy design[1]. In this chapter, we also attempt to determine whether the state-of-the-art techniques for fine-grained task scheduling can be adapted and improve policy design. To evaluate these design choices, we design a decentralized scheduler and investigate scheduling strategies that meet the following criteria in this work:

- **Ability to interpret user-intent and perform scheduler operation globally:** A scheduling policy should be able to translate user intent to scheduling decisions – in Chapter 3 and Chapter 5 we discuss how different user intent should be interpreted by scheduler. However, these solutions either rely on reactive application-level recofiguration (Chapter 3) or can only be applied locally at each machine in a centralized

[1][158] can also suppport preservation of intermediate data through allocating communicating functions to the same worker.

fashion (Chapter 5).

- **Awareness of stateful operations:** A scheduling policy should be designed to accommodate function statefulness by being able to exploit memory locality as well as in-order processing.

6.3 RELATED WORKS

In Dirigo we explore how to build a distributed, fine-grained scheduler framework that can accommodate scheduling policies that could dynamically schedule stateful function invocation by being aware of user intent. Scheduling dataflow applications as function DAGs requires scheduler to perform resource planning both within and across application boundaries. Past works on SLA-aware function scheduling performs scheduling operations from the application layer's function handler, with only limited information of system-wide information [4]. Similar to Dirigo, [158] discusses a fully decentralized scheduler framework that resides on each lambda worker. However, [158]'s can be seen as an application-layer scheduling technique as the scheduler performs scale-out(-in) decisions while relying on the underlying framework (AWS Lambda [150]) to map functions to workers. Dirigo, on the other hand, exploits recurring tasks with performance targets.

Most works focusing on resource provisioning for serverless architecture assume that function scheduling can be performed with (semi-) global centralized entities. The scheduling framework proposed by [159] supports latency SLA performs laxity-aware scheduling for serverless functions. However, its laxity-aware scheduling is performed *locally* at each work pool in a centralized manner. [159] also supports load-balancing performed *reactively* based on queuing metrics, while Dirigo focuses on proactively schedules function requests before processing. A platform such as [151] adopts a bottom-up approach by sending tasks to pre-designated workers and offload requests to the global scheduler when the worker is overloaded. This is similar to our optimistic offloading scheduling from the perspective of scheduling philosophy, but we explore a fully decentralized approach where no global view could be obtained. [160] discusses QoS aware scheduling for serverless functions. It develops a client-side scheduler that adaptively dispatches function requests to a black-boxed serverless framework. Whereas in Dirigo we discuss function scheduling performed within the serverless runtime. As a result, both user-specified intent and resource allocation within the cluster is visible to the scheduler.

Distributed fine-grained scheduling has been explored before serverless computing platforms have emerged. [161] explores a fully decentralized scheduler design where workers

obtain no global view of resource usage for the entire cluster. It uses reservation-based scheduling, derived from Power-of-Two [162] scheduling scheme, which places tasks reservations in multiple candidate worker queues and waits for the first available worker to fetch and execute the task. Similar scheduler design that is also inspired by Power-of-Two scheme includes [163]. In Dirigo we adopt similar philosophy in our dynamic-binding scheduling policies, while combining priority/laxity awareness into our approach. fine grained scheduling [164] (long short jobs, randomized stealing).

6.4 TARGET WORKLOADS

Dataflow Applications: Dirigo framework supports real-time data processing applications composed of dataflows of user-implemented event-driven functions. This chapter explores how Dirigo works with real-time data processing applications that contain operators that utilize Dirigo's in-memory state storage. In particular, we investigate two scenarios: i) dataflow applications that leverage the in-memory state store as a data cache to speed up data access; and ii) dataflow applications with streaming aggregation operators that perform window aggregations on a single, in-ordered real-time data stream. In the former scenario, we investigate how different scheduling policies would influence applications that try to balance between achieving load-balancing and preserving data locality. In the latter scenario, we explore using different scheduling policies to perform automatic parallelization of single-threaded stateful operators.

Capturing message dependencies: The entire Dirigo system ensures the following dependencies are satisfied: i) when incoming messages have timestamps, Dirigo executes them in order of timestamps— thus any dependencies captured by timestamps are also satisfied by Dirigo; and ii) when upstream messages generate downstream messages, the former are executed completely before the latter start execution. In particular, i) is in fact satisfied by the underlying channel delivery framework in Dirigo.

6.5 SCHEDULER DESIGN

This chapter introduces the architecture of Dirigo's scheduler and the scheduling API that we use to implement scheduling policies.

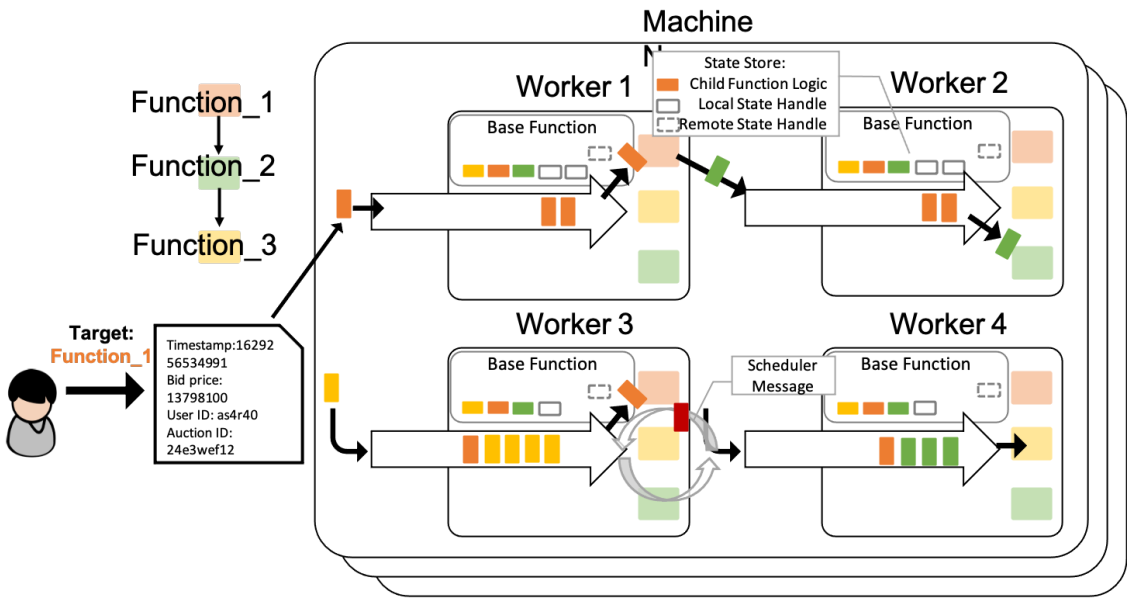


Figure 6.1: *Dirigo architecture.*

6.5.1 Scheduler Architecture

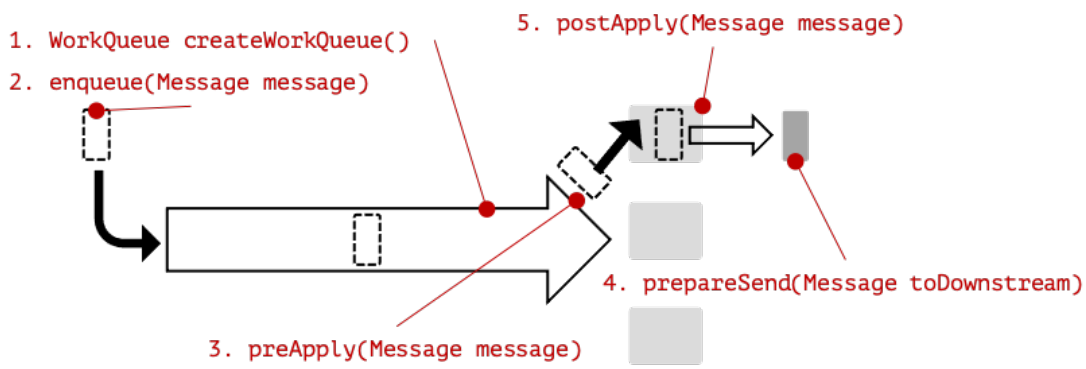
All dataflow applications in Dirigo are composed of a DAG of *functions* containing user-specified message handling logic. When a message is received, the scheduler finds the target function specified by the message target *function address* by loading the corresponding function with all its dependencies by creating a *function activation*. Once the execution finishes, the activation is recycled and used for future messages. At any given time, the number of function activations is less or equal to the number of workers within each machine.

Once the function DAG is submitted, each function within the DAG is registered to *every* worker in the system before execution starts. Therefore in Dirigo, a target function can be triggered at any worker where the function has registered. Dirigo achieves message scheduling by creating bindings between user-specified virtual function addresses with function workers. Figure 6.1 illustrates Dirigo's architecture. Beyond data messages exchanged for function processing, Dirigo uses *scheduler messages* to implement communication protocols to carry out scheduling operations.

Dirigo does not use a standalone scheduler thread to manage message execution, and all scheduling operations are triggered by messages. All messages received by a worker pass through a series of scheduling API functions that potentially trigger scheduling operations. For scheduling operations that need to be carried out by scheduling policy, the scheduling API can also create scheduler messages that specify actions to be performed and define how receiving workers should handle these messages.

Each worker manages both function class loaders that have been registered locally, and data object handles created during execution. Dirigo supports both heap memory backend and Redis [165] remote backend. For remote storage backend, users can use a global identifiable state name to access the same object despite the physical location where the function is invoked. For heap storage backend, the state object is non-fault-tolerant and is accessible within the scope of each worker, meaning that only the functions invoked on the same worker can share a state handle that points to the same state object.

6.5.2 Scheduling API



Message Type	Purpose
REGISTRATION	Initialization message with function logics.
INGRESS	Messages received from external client.
ERESS	Messages sent to external client.
REQUEST	Dataflow messages transfer between functions.
SCHEDULE_REQUEST	Scheduler message initialized by scheduling policy.
SCHEDULE_REPLY	Response message created upon receiving SCHEDULE_REQUEST.
FORWARDED	Dataflow messages

Figure 6.2: *Scheduling API and types of messages defined in Dirigo.*

We depict the scheduling API in Figure 6.2. The scheduling policy should first specify the work queue to hold messages through `createWorkQueue()`. It then defines the actions before the message is inserted into the work queue (through `enqueue(Message message)`), before (through `preApply(Message message)` and after execution (through

`postApply(Message message)` of a message, and before a downstream message is sent (through `prepareSend(Message message)`).

The scheduling policy can also specify whether to create a *scheduler message* and the handling mechanism within each scheduling API function. A scheduler message can target any function that resides on any workers in the system or targets any workers without specifying the function address. Figure 6.2 also specifies the types of messages that we use to implement scheduling policies. Within these message types, `SCHEDULE_REQUEST` and `SCHEDULE_REPLY` are the main types of scheduler messages we use to propagate scheduling information and scheduling decisions. The data types transported by these messages are customizable.

6.5.3 Customizing Priority-based Message Queue

Dirigo allows a scheduling policy to specify the message queue used by the policy. For a latency-sensitive dataflow application, Dirigo associates each message with a deadline that ensures the end-to-end job latency does not exceed expectations. Unless otherwise specified, we use the priority assignment strategy in [4] to assign priority to messages for a policy that targets latency-constraint applications, and we order all messages using Earliest-Deadline-First (EDF) [101] policy within each worker queue. Checking whether a worker could insert a message without introducing violations for the message itself or existing messages in the queue requires traversing the entire queue. To speed up this process, we use *minimum laxity priority queue* to track the minimum laxity of any messages that rank lower than a given message. Inserting a message M only influences messages in the queue with lower priorities than M. Therefore, to check whether inserting a message M would cause potential violations, we search whether there is a lower-prioritized message with a laxity value that is shorter or equal to M's execution cost. Minimum laxity priority queue achieves this goal by mapping each message m to the minimum-laxity of any message whose priority is lower than m. This process takes $\mathcal{O}(\log n)$ due to binary search. Maintaining a laxity-based queue involves updating the minimum laxity of every index upon queue insertion and deletion, which could cost $\mathcal{O}(n)$ in the worst case. To speed up this process, we only track the position in the priority queue where the minimum laxity changes, meaning that the minimum laxity queue is monotonically increasing. We list the algorithms of these operations in Algorithm 6.1.

Algorithm 6.1 Minimum-Laxity Priority Queue

Require: $workQueue$, $laxityMap$

Algorithm 6.1 Minimum-Laxity Priority Queue (cont.)

▷ *workQueue* maps each message to its current laxity, which is updated when messages are added or removed from the queue. *laxityMap* maintains the mapping between messages and minimum laxity of all lower-prioritized messages. *laxityMap* only keeps one entry (i.e., the lowest prioritized message) for each minimum laxity value.

```
1: function LaxityCheck(Message M)
2:     if workQueue.isEmpty() then
3:         return M.Priority − M.getCost() ≥ currentTime
4:     floorEntry ← workQueue.floorEntry(M.priority)        ▷ Binary search for the
       message laxity pair that has greater or equal to M.priority
5:     ecTotal ← floorEntry.priority − floorEntry.getValue()▷ Get total execution cost
       of all messages that have higher priorities.
6:     if ecTotal ≤ M.getLaxity() then
7:         ceilingEntry ← laxityMap.ceiling(M.priority)      ▷ Binary search for the first
       entry that have less priority than M.priority
8:         if ceilingEntry is null OR ceilingEntry.getLaxity() > M.getCost() then
9:             return true
10:    return false

11: function Add(Message M)
12:    if workQueue.isEmpty() then
13:        workQueue.add(M, M.getLaxity())
14:        workQueue.add(M, M.getLaxity())
15:    else
16:        minLaxitySoFar ← MAX_LONG
17:        minLaxityHead ← MAX_LONG
18:        laxity ← M.getLaxity()
19:        insert ← false
20:        for M′ in workQueue.descending() do
21:            if M′.getPriority() > M.getPriority() then
22:                updatedLaxty ← workQueue.get(M′) − M.getCost()
23:                workQueue.get(M′) ← updatedLaxity        ▷ Deduct time budget for all
       messages that have lower priority
24:                if minLaxitySoFar > updatedLaxity then
25:                    minLaxitySoFar ← updatedLaxity
```

Algorithm 6.1 Minimum-Laxity Priority Queue (cont.)

```
                    laxityMap.put(M', updatedLaxity)
                if updatedLaxity < minLaxityHead then
                    minLaxityHead ← updatedLaxity
            else
                laxity ← laxity − M'.getCost()    ▷ Deduct time budget from message to be added if the message in queue has higher priority
        if minLaxityHead > laxity then
            insert ← true
        headLaxityMap ← laxityMap.headMap(M)
        for MarkerM in laxityMap.headMap(M) do
            if laxityMap.get(MarkerM) >= laxity then
                laxityMap.remove(MarkerM) ▷ Remove entry with higher priority with higher laxity
        if insert is true then
            laxityMap.put(M, laxity)
        workQueue.add(M, laxity)
        return insert

function Poll(Message M)
    poll = null
    if workQueue.size is 0  then
        return poll
    poll = workQueue.poll()
    if laxityMap.has(poll) then
        laxityMap.remove(poll)
    ec = poll.getCost()
    for queuedMsg in workQueue do
        updatedLaxity = workQueue.get(queuedMsg) + ec
        workQueue.set(updatedlaxity)
        if laxityMap.has(poll) then
            laxityMap.update(poll, updatedLaxity)
    return poll
```

6.6 SCHEDULING POLICIES

In this work, we investigate how to design scheduling policies that help runtime achieve user-specified performance targets and evaluate how different design decisions influence performance. Specifically, we conduct an initial investigation on how different scheduling policies influence performance of stateful dataflow applications with user-specified latency targets.

6.6.1 Serving Latency-sensitive Stateful Functions

Dynamically scheduling stateful functions with latency target requires the scheduler to obtain information of all available resources and be able to estimate whether a request can satisfy its performance target once a scheduling decision is made. Dirigo performs scheduling by creating *bindings* between user-specified target function addresses and function workers. Specifically, we split all scheduling policies into three categories: dynamic-binding scheduling, optimistic offloading scheduling, and static-binding scheduling:

- **Static-binding Scheduling (SBS):** A static-binding scheduling (SBS) policy creates a fixed binding between the message's target function address and a physical worker in the cluster. It is the default scheduling strategy used by frameworks such as [166]. Static-binding scheduling allows all messages targeting a single function address to be processed in a single-threaded fashion. For stateful functions, function states objects can be stored locally without being explicitly transferred. The downside of this approach is that messages targeting the same function address cannot be parallelized, leading to load imbalance among workers.

- **Dynamic-binding Scheduling (DBS):** A dynamic-binding scheduling (DBS) policy does not create a static association between a message's target function address and the physical worker that executes this message. The targeted worker is determined once the message is created before the message is dispatched. The state-of-the-art strategy is the Power-of-Two choices technique [162], implemented as a part of a fine-grained, distributed scheduler in [161, 163]. It is a well-known technique that tackles workload skew among workers and provides good load-balancing within a cluster. However, this category of policies does not exploit state locality and may require an explicit transfer of states when the unction is triggered.

- **Optimistic Offloading Scheduling (OOS):** An optimistic offloading scheduling (OOS) assumes a static binding between the message's target function address and the physical worker that executes the message. The upstream message sender (client

or upstream function) optimistically assumes a request message will be completed by meeting its latency target. The target function keeps gauging its pending queue and offloading messages that cannot be executed before its deadline by seeking a new physical worker to bind. For stateful function, these policies find a middle ground between DBS and SBS policies by favoring physical worker that potentially holds function states used processing a message, while only offloading messages while necessary. However, the flexibility to balance locality awareness and load balancing does come with a cost: OOS could lead to scheduling overhead for fine-grained, heavily skewed workload that targets a single function address and the worker's scheduler is required to perform all scheduling operations.

We summarize the types of scheduling polices in Table 6.1.

Categories of Scheduling Policies			
Scheduling Types	Locality Awareness	Load Balancing	Scheduler Overhead
Dynamic-binding scheduling [162]	Low	High	Low
Static-binding scheduling [4]	High	Low	Low
Optimistic offloading scheduling (this work)	Medium	Medium	High

Table 6.1: *Types of scheduling policies.*

6.6.2 Scheduling Policies

In this chapter we explain scheduling policies we explore in detail. Figure 6.3 summarizes a hierarchy of available policies we describe.

Static-binding Scheduling (SBS): For SBS policies, we utilize the default binding technique [166] where all messages are sent to a worker identified by target address ID hash. Through SBS, the user could assume that the message will be invoked on the same physical worker. In Dirigo, the worker maintains state objects accessed by previously invoked messages. When function object cache is enabled (assuming no eviction), the object will present in the cache once first accessed. Then all future state access (with the same state ID) will be resolved locally. All messages received on the same worker are ordered by its message priority, similar to the approach described in [4]. We denote this policy as `PriorityBased`.

Dynamic-binding Scheduling (DBS): For DBS policies, we explore two scheduling policies: `QueueBasedDirect` and `LaxityBasedDirect`. `QueueBasedDirect` simulates a default power-of-two-choices scheduling policy by first selecting two random candidate workers, then

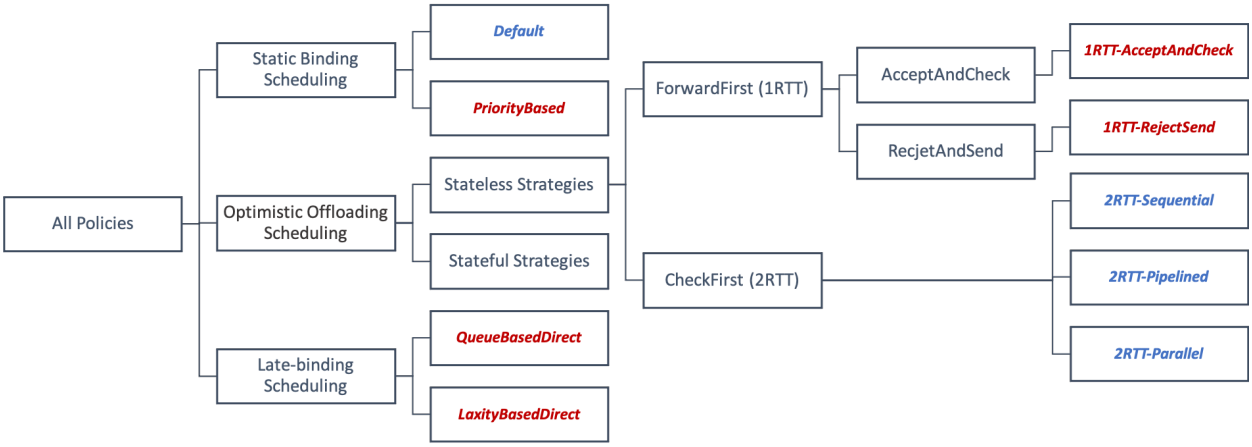


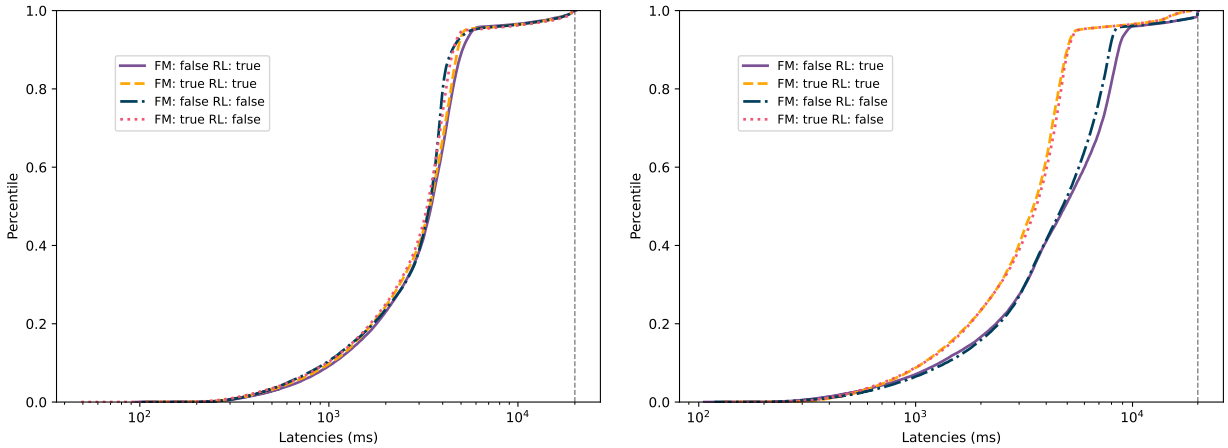
Figure 6.3: *List of Scheduling Policies. We focus on scheduling policies marked in red in this chapter.*

sending `SCHEDULE_REQUEST` to both workers, and finally waiting for `SCHEDULE_REPLY` that contains current queue sizes. `QueueBasedDirect` selects candidate that has a shorter queue length. `LaxityBasedDirect`, on the other hand, is a variation to `QueueBasedDirect` where `SCHEDULE_REQUEST` contains the estimated priority and laxity information for a message. The candidate worker performs a laxity check based on the priority/laxity pair received, providing an estimation on whether the latency target can be satisfied (using the priority queue structure described in Chapter 6.5.3). `SCHEDULE_REPLY` contains a boolean value on whether the message can be accommodated without leading to performance target violation.The sender chooses a candidate based on this response — if both candidates reply with the same responses, the sender makes its decision based on the candidates' queue length. The pending message is buffered until both `SCHEUDLE_REPLY`s are received so that no further processing is blocked.

Optimistic Offloading Scheduling (OOS) For OOS, all messages assume a static binding between function address and target worker, which we call the *lessor* of the function. If all messages in the queue are expected to meet their deadlines, all messages are directed to the function lessor and executed as-is. When the lessor detects a potential violation, the scheduler takes action to offload messages to other workers, which we call *lessees*.

We divide the OOS policies into two categories, based on the types of leasing requests sent by a function when a potential deadline miss is detected.

- **Forward-first** strategies perform scheduling action greedily (i.e., by forwarding messages to potential candidate lessee directly) and wait for the lessee's response determining whether a message is scheduled successfully.



(a) *1RTT-RejectSend* (b) *1RTT-AcceptAndCheck*

Figure 6.4: *Forward-first methods: Comparing effect of four parameters tuning RANDOM_LESSEE (RL) and FORCE_MIGRATE (FM). The policy selects potential lessees randomly when RL is set to True, and selects lessees with shortest queue when RL is set to false.*

- **Check-first** strategies require a lessor function sending a request to a potential lessee and checks reply before the scheduler makes a scheduling decision and dispatches the message.

Forward-first strategies require only one round-trip time (RTT) while Check-first strategies require at least 2 RTTs to complete each scheduling action. Under the Forward-first strategy group, we explore two scheduling policies: `1RTT-AcceptAndCheck` and `1RTT-RejectSend`. `1RTT-AcceptAndCheck` adopts an *accept-and-check* approach, where messages are inserted into the work queue. The policy later determines whether to take scheduling action by examining the work queue. On the other hand, `1RTT-RejectSend` adopts a *reject-and-reallocate* approach, where the scheduler takes action before the message is inserted into the work queue, i.e., all successfully inserted messages will be processed without being reallocated.

A forward-first strategy uses the following preset parameters:

- `FORCE_MIGRATE` specifies a forwarded message is required to be executed on lessee activation (when set to true), or lessee activation is required to respond whether such message can be executed locally.

- `RANDOM_LESSEE` specifies whether the lessee activation is selected at random. When specified as false, the scheduler chooses the candidate lessee that has the shortest queue size.

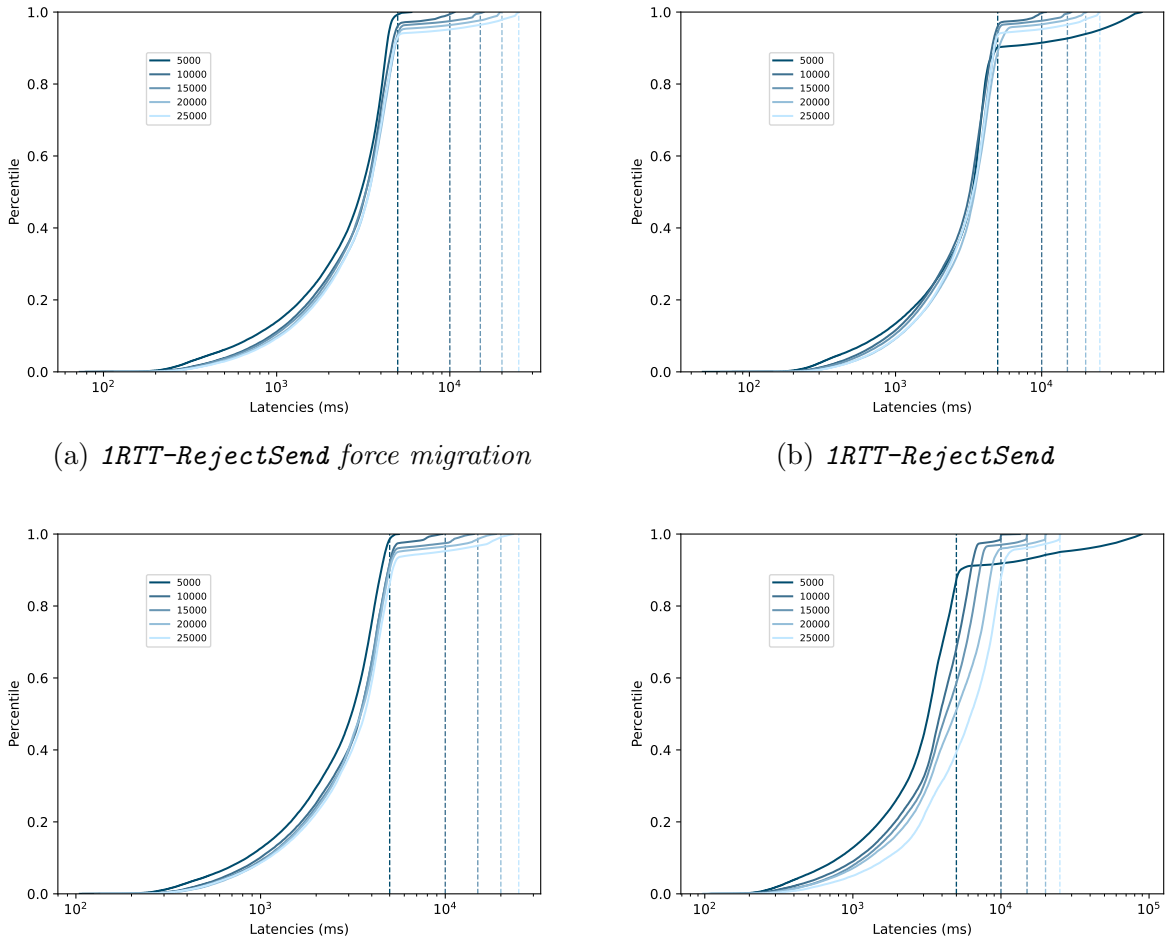
Figure 6.4 shows the comparison of variations for two policies: `1RTT-RejectSend` and

1RTT-AcceptAndCheck. We apply 128 parallel data sources. Each source sends messages targeting a single function address, and we observe how the scheduler offloads messages from the designated lessor. For each policy, we compare four different settings. Both Figure 6.4a and Figure 6.4b show that disabling RANDOM_LESSEE (and selecting lessee with a shorter queue) does not provide performance benefit over random selection in this scenario. Our queue-based lessee selection mechanism tracks lessee queue size lazily, and the workers could make scheduling decisions on the stale information when the task granularity is small. As we only have one worker in the cluster that serves as the lessor to all messages, queue-based lessee selection does not provide additional performance benefits. On the other hand, Figure 6.4a shows that FORCE_MIGRATE has little effect on performance as both policy variances use the same priority queue structures and the lax latency constraints set by the application. Whereas for 1RTT-AcceptAndCheck (Figure 6.4b), FORCE_MIGRATE has a more substantial effect on dataflow latency (increases by 40%). This effect is caused by significantly fewer queue operations (e.g., laxity checks) during requesting processes.[2] Both policies provide similar levels of success rates achieving latency constraints.

To further demonstrate the effect of forced migration, we apply an increasingly strict latency target and observe changes in latency distribution, as shown in Figure 6.5. When FORCE_MIGRATE is enabled, 1RTT-AcceptAndCheck and 1RTT-RejectSend behave similarly (Figure 6.5a and Figure 6.5c), and the job latency decreases as we tighten the latency constraints. When FORCE_MIGRATE is disabled, performance is only influenced significantly (with median latency increasing by 10% and tail latency increasing by 8×) when the performance target is strict (i.e., 5000 ms), as most forwarded messages are rejected and executed locally for 1RTT-RejectSend (Figure 6.5b). For the rest of the settings, FORCE_MIGRATE has little influence on performance as few messages are rejected by lessor workers when performance constraint is lax. For 1RTT-AcceptAndCheck (Figure 6.5b), disabling FORCE_MIGRATE causes overall latency to increase due to more expensive queue operations as described above. Similar to 1RTT-RejectSend, we also observe a substantial latency increase (with median latency increasing by 4% and tail latency increasing by 15×) when the latency constraint is strict (5000 ms).

Under the Check-first category, we explore one base scheduling policy: 2RTT-Sequential and its two variations: 2RTT-Pipelined and 2RTT-Parallel. These policies perform a violation check after message execution. If a scheduling action is required, the scheduler sends out SEARCH_RANGE number of scheduling requests to different candidate lessees. It then determines whether all violations should be relocated to the lessor worker(s) once a response

[2] 1RTT-AcceptAndCheck with force execution uses a default priority queue that contains no laxity information.



(a) *1RTT-RejectSend force migration*

(b) *1RTT-RejectSend*

(c) *1RTT-AcceptAndCheck force migration*

(d) *1RTT-AcceptAndCheck*

Figure 6.5: *Forward-first methods: Scheduling behaviors under increasingly strict performance target.*

is received. Unlike Forward-first policies, Check-first policies avoid blindly reallocating messages by first performing a laxity check with the priority/laxity pair. `2RTT-Sequential` dispatches all violation messages to one lessor worker and waits for `REPLY_REQUIRED` amount of replies before deciding whether to relocate messages. `2RTT-Pipelined` is a variation policy of `2RTT-Sequential`. This policy waits for `SEARCH_RANGE` responses to complete a scheduling action while also starting a new violation search once `REPLY_REQUIRED` replies from the previous round are received. `2RTT-Parallel` dispatches parallel requests when violations are determined. Violations are partitioned to `SEARCH_RANGE` different chunks, and each chunk is dispatched to one lessee worker that responds positively to the request. The scheduler keeps track of all pending messages and processes these messages locally if a negative response(s) is received. Figure 6.6 applies `2RTT-Parallel` with increasing `SEARCH_RANGE`. We observe that

request latency decreases as we expand SEARCH_RANGE due to the fewer requests we dispatch to a single lessee, potentially creating a workload hot spot. This behavior is especially true when we increase SEARCH_RANGE from 1 to 2, with 16× improvement on median latency and success rate increases by 0.09. As we keep increasing SEARCH_RANGE (beyond 1 workers), the benefit of exploring more candidates starts decreasing as the marginal benefit created by additional candidate lessee starts to diminish.

Our micro-benchmark finds that Forward-first (1RTT) approaches perform generally better than Check-first (2RTT) approaches. For this work, we choose 1RTT approaches as our default approaches for OOS policies, and we plan to explore Check-First approaches in our future work.

Figure 6.7 illustrates the protocols of communication among workers, initiated by the lessor worker, using timelines while applying the following four scheduling policies respectively: LaxityBasedDirect (Figure 6.7a), 1RTT-AcceptAndCheck (Figure 6.7c), 1RTT-RejectSend (Figure 6.7b), and 2RTT-Parallel (Figure 6.7d).

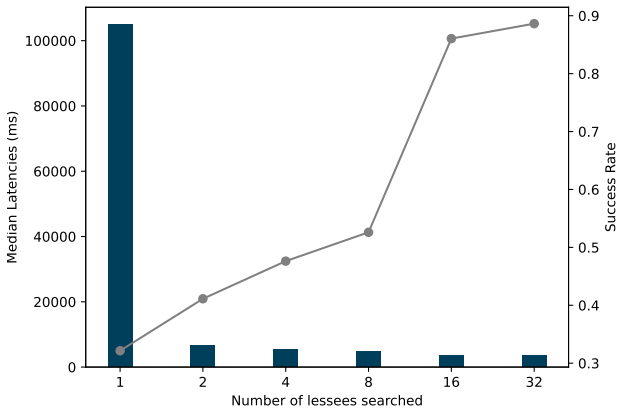


Figure 6.6: *Check-first methods: median latency distribution and success rates change while using* ***2RTT-Parallel*** *with increasing number of candidates explored.*

6.7 EVALUATIONS

6.7.1 Cluster Setup

We build Dirigo within Flink Statefun[166] framework. We deploy all experiments through Emulab cluster. Each cluster contains 32 D430 nodes (16 2.4Hz cores, 64GB memory, 10GB NICs). All dataflow applications have a default parallelism of 128, and our runtime is set up with 128 workers in total. We use 16 D430 nodes for Redis instances.

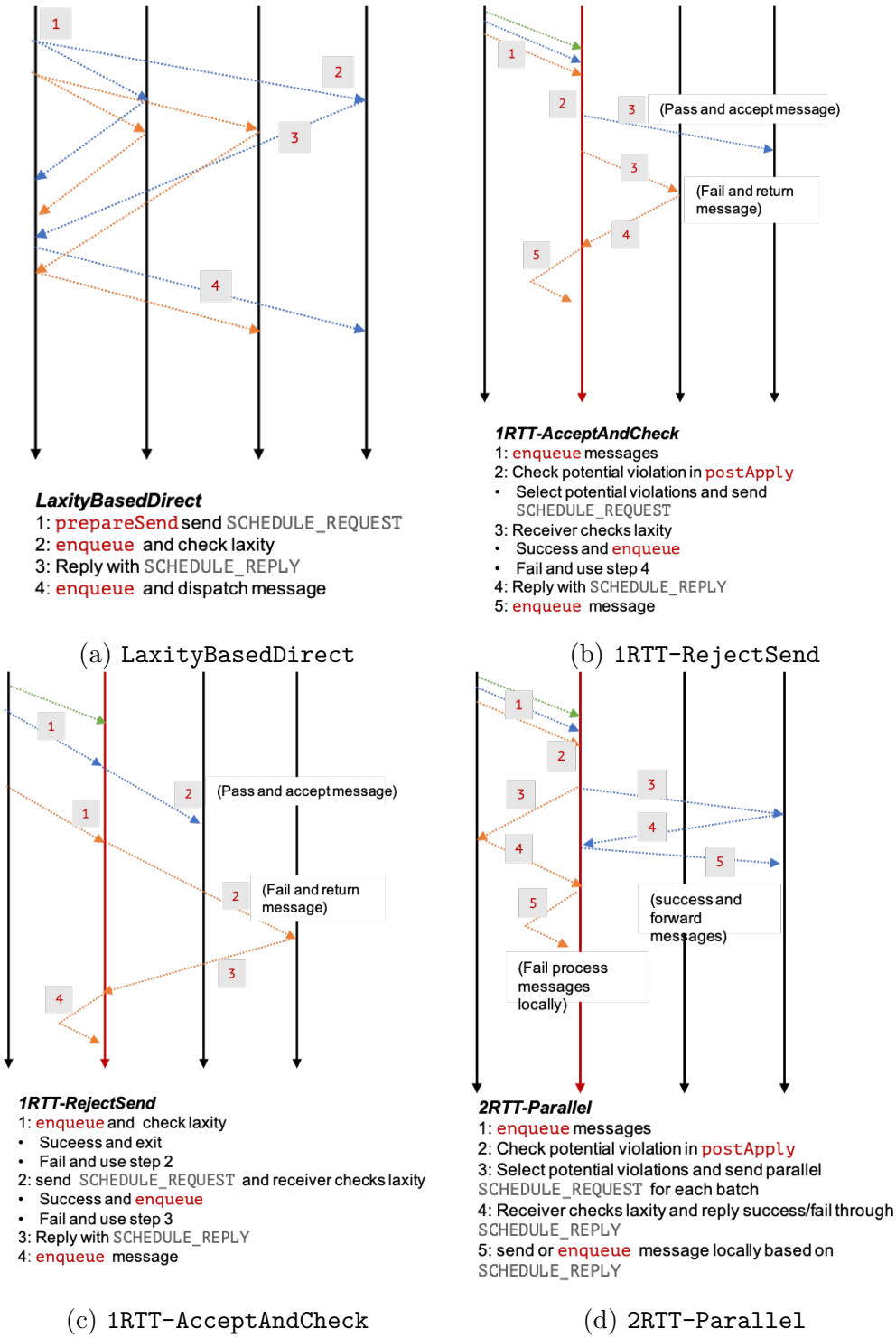


Figure 6.7: *Scheduler communication in timelines by applying different scheduling policies in Dirigo.*

6.7.2 Controlled Dataflow

For the controlled dataflow experiment, we apply a set of concurrent running two-staged dataflows where the stage 1 function sends messages to a designated stage 2 function (as shown in Figure 6.8). To create an accurate estimation of execution cost, we control the execution time of each function invocation. We also control request skewness by generating stage 2 requests that access state objects forming a Zipfian distribution. By default, each application can generate requests for $N_{objects}$ objects, where $N_{objects}$ equals the number of workers in total. Each object is associated with one worker, and the target worker is chosen based on the requested object. For a dataflow application that uses local state storage as cache space, we set the number of states stored at each worker to equal the total number of dataflows. As a result, each dataflow application can store one object on average on each worker. Here we attempt to create a read-only workload where a state fetch is forced when a function is assigned to a worker(s) that have not previously served a request targeting the same address. The default state object size is 512 bytes. For comparison, we perform the same experiment with an identical setting without state object access. This setting simulates a scenario where no storage constraint is enforced, and all objects can be accessed locally, where state accesses do not influence request latency.

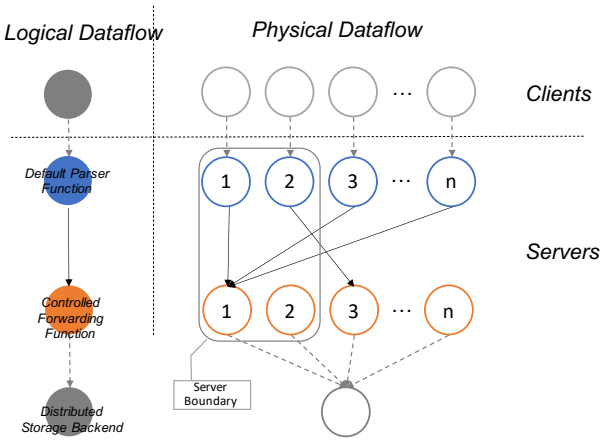
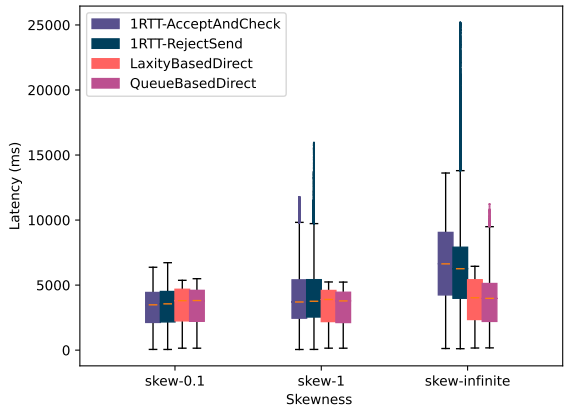


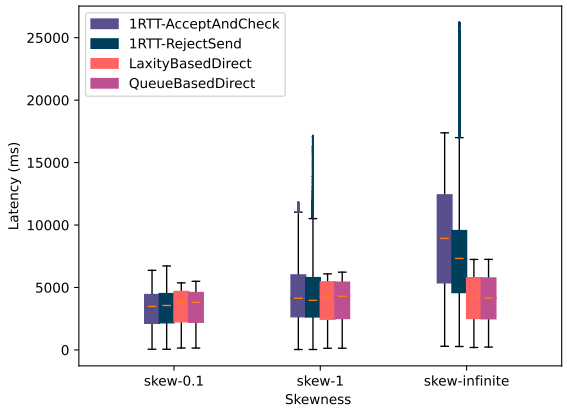
Figure 6.8: *Default Workload.*

DBS is preferable for large, stateless tasks, while OOS performs better for small, stateful tasks: We apply 10ms task execution time to all functions invocations. Figure 6.9 shows end-to-end latency using four scheduling policies in Dirigo: two OOS policies: `1RTT-AcceptAndCheck` and `1RTT-RejectSend`, as well as two DBS policies: `QueueBasedDirect` and `LaxityBasedDirect`. We vary the skewness of requests by varying the skew factor from 0.1, 1, to infinite[3]. Figure 6.9a and Figure 6.9b show latency distribution of all requests without and with state object accesses, respectively. Despite the type of scheduling policy being applied, we observe that latency (both median and tail) increases due to the increasingly uneven distribution of workload. Despite triggering more remote state accesses, the DBS

[3]When skew factor is infinite, all requests are directed to the same worker.



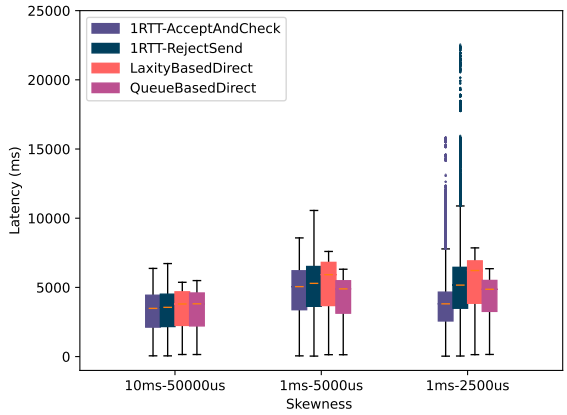
(a) *No state object accesses.*



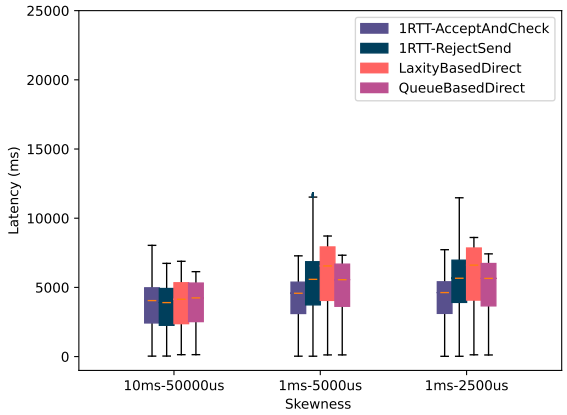
(b) *With state object accesses.*

Figure 6.9: OOS *vs* DBS *policies applied to controlled dataflows with increasing degree of skewness toward target functions.*

policies are more robust against workload skew as the state operation overhead is relatively small compared to function execution time. To show how the comparison between state operation and scheduling granularity[4] can influence the requests' latency distribution, we perform the experiments with fixed skewness (0.1) with decreasing scheduling granularity in Figure 6.10.



(a) *No state object accesses.*



(b) *With state object accesses.*

Figure 6.10: OOS *vs* DBS *with decreasing scheduling granularity. Each category is denoted by the function invocation cost and interval between requests.*

Figure 6.10 shows how varying scheduling granularity influences default workload running

[4]It can be determined by request rate and function execution time.

without (in Figure 6.10a) and with (in Figure 6.10b) state object accesses. Compared to the longer function invocation shown in Figure 6.9, Figure 6.10 demonstrates workloads where state operations become a more dominant factor towards latency distribution, as we observe latency increases slightly for all workloads after state accesses are applied. Meanwhile, we observe that the OOS policies perform better when task granularity is small – this is because the OOS policies do not require communication between local schedulers when latency targets are determined to be met. For fine-grained (e.g., 1 ms) function invocations, OOS policies trigger scheduling operations lazily, and the scheduler can accumulate more messages (comparing to longer tasks) given a fixed latency target. Meanwhile, both DBS policies force 2 RTTs to dispatch all messages. For fine-grained tasks, this communication becomes more expensive than overhead created by queue maintenance and potential load imbalances using OOS approaches.

6.7.3 Social Network Micro Service Simulation

We simulate Social Network Microservice from DeathStarBench [167]. This workload captures three types of user operations performed by social network microservices, as shown in Figure 6.11. We build a dataflow DAG that contains a parser function triggered by every event batch sent by each client. The parser functions handle event batch by sending each request to one of the downstream operator functions: i) `ComposePost` handles a write request by writing a post content to remote storage as well as post owners' and their followers' timeline; ii) `ReadUserTimeline` reads the latest posts from all followers; iii) `ReadHomeTimeline` reads the latest timeline posted by a specific user. The data objects that have been written/read during invoking functions would be temporarily stored in the local storage, and the local storage evicts old objects based on Least-Recently-Used (LRU) policy.

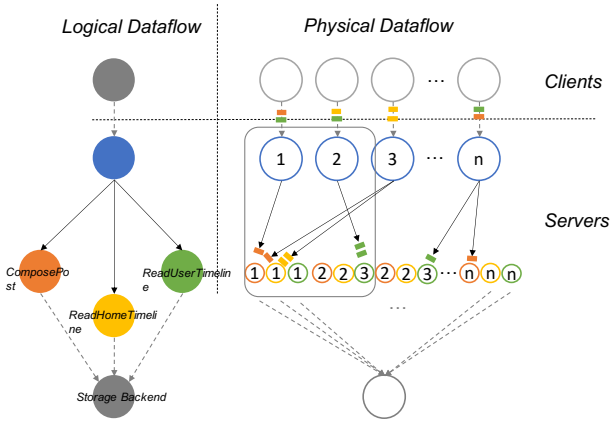
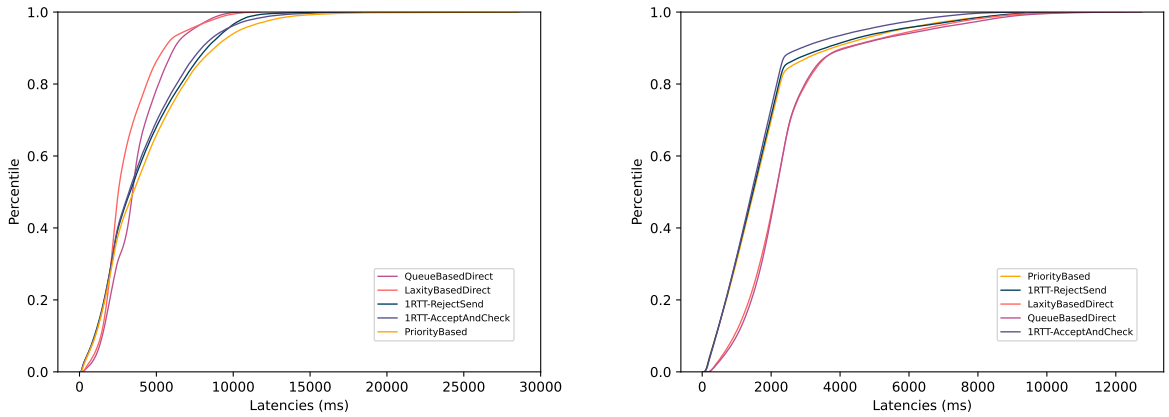


Figure 6.11: *Social Network Microservice Workload.*

For non-skewed web-access workload, `LaxityBasedDirect` is preferable when no cache is present, while `1RTT-AcceptAndCheck` is preferable with caching enabled. We compare two OOS policies and two DBS policies with default `PriorityBased` scheduling



(a) *Redis-only state storage.* (b) *Local object caches with Redis state backup.*

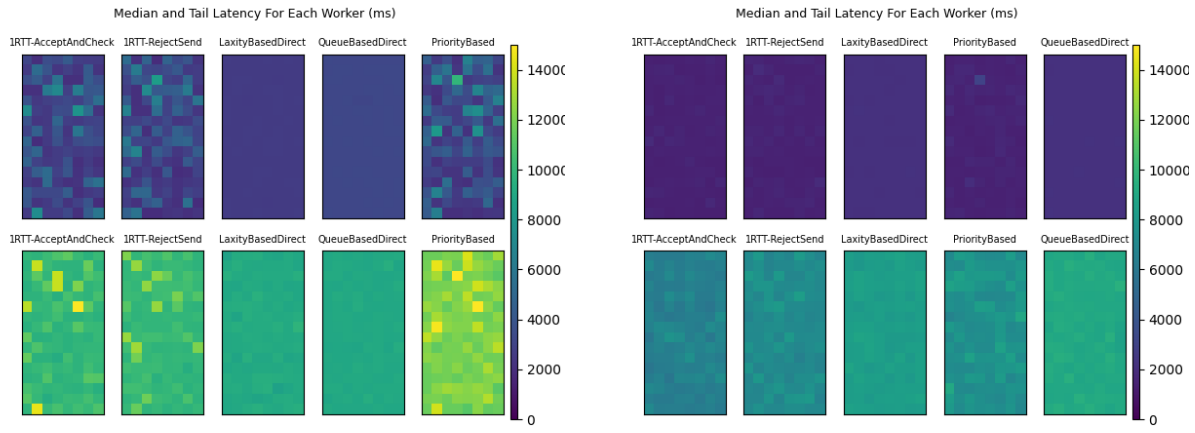
Figure 6.12: SBS, DBS *and* OOS *policies using social network graph: end-to-end latency.*

.

policy as an SBS policy. Figure 6.12a shows applying these scheduling policies to workloads that use Redis-only as a storage backup. As all functions are essentially treated as stateless by function runtime, the locality-agnostic late binding policies outperform all other scheduling policies: `QueueBasedDirect` provides up to 26% shorter tail (99 percentile) latency than OOS policies and 36% shorter tail latency than `PriorityBased`. `LaxityBasedDirect` provides up to 23% shorter median and 26% shorter tail latency than OOS policies, as well as 28% shorter median latency and 33% shorter tail latency than `PriorityBased`. Both OOS and DBS policies are able to improve SBS scheduling (`PriorityBased`) on both median and tail latency due to their ability to modify function bindings on the fly. Being laxity-aware helps `LaxityBasedDirect` perform better than `QueueBasedDirect` by shortening median latency by 27%. All adaptive reallocation policies provide shorter tails than static priority-based strategies, providing that reallocation operations are needed to reduce straggler effects.

Figure 6.12b shows applying the same set of policies to the workload with local object cache enabled. Unlike Figure 6.12a, all policies depicted in this figure have shorter latencies due to reduced remote state accesses. Both OOS policies and SBS policy achieve shorter latencies by being locality aware, with `1RTT-AcceptAndCheck` providing the shortest median and tail latency: `1RTT-AcceptAndCheck` provides up to 33% shorter median and 21% shorter tail latency than DBS policies, as well as 17% shorter tail latency than SBS. `1RTT-RejectSend` provides up to 31% shorter median and 8% shorter tail latency than DBS policies and similar performance to SBS.

In Figure 6.13a and Figure 6.13b we show the median and tail latency of each individual

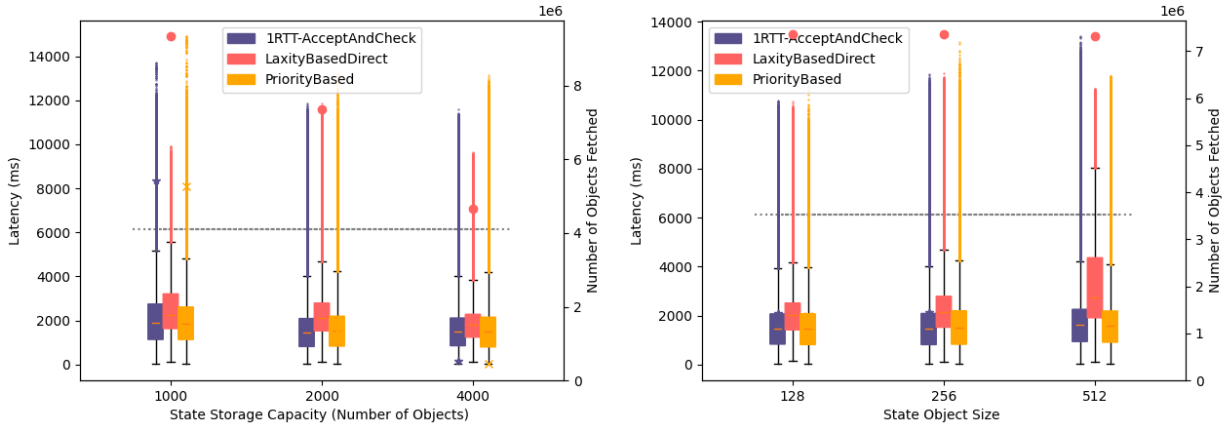


(a) *Redis-only state storage.* (b) *Local object caches with Redis state backup.*

Figure 6.13: SBS, DBS *and* OOS *policies using social network graph: per worker median (top row) and tail latency (bottom row). Darker color indicates shorter latency.*

worker, corresponding to scenarios described in Figure 6.12a and Figure 6.12b, respectively. Comparing these two scenarios, we observe that DBS provide near-perfect load balancing among workers compared to all other approaches with higher latency (both median and tail). This is because using these policies results in much more object fetches by proactively scheduling functions to workers that are less likely to contain objects locally. In contrast, the OOS approaches are able to favor workers that are more likely to cache state objects by only redirecting messages to lessee workers when violations are determined to occur. Meanwhile, SBS produces uneven latency distribution due to skewed user popularity, which directly translates to skewed workload distribution due to static bindings. We observe that the uneven workload distribution causes higher latency for the same set of workers using both SBS and OOS policies (Figure 6.13a). This pattern cannot be observed from Figure 6.13b as the number of state operations reduces significantly and becomes a less significant contributing factor toward request latencies.

With the non-skewed workload, SBS is preferable for settings using large object sizes. At the same time, SBS and OOS work well for small cache capacity. We further extend the previous scenario by comparing SBS policy with one OOS policy (`1RTT-AcceptAndCheck`) and one DBS policy (`LaxityBasedDirect`). Here we explore how both cache capacity (in terms of the number of memory objects) and object sizes influence performance under different scheduling policies. Figure 6.14a shows how requests' latency progresses as we increase the maximum number of objects stored in the local object cache. We also show the number of objects fetched from remote storage applying each policy. From



(a) *Varying cache capacity (number of objects), object size 256.*

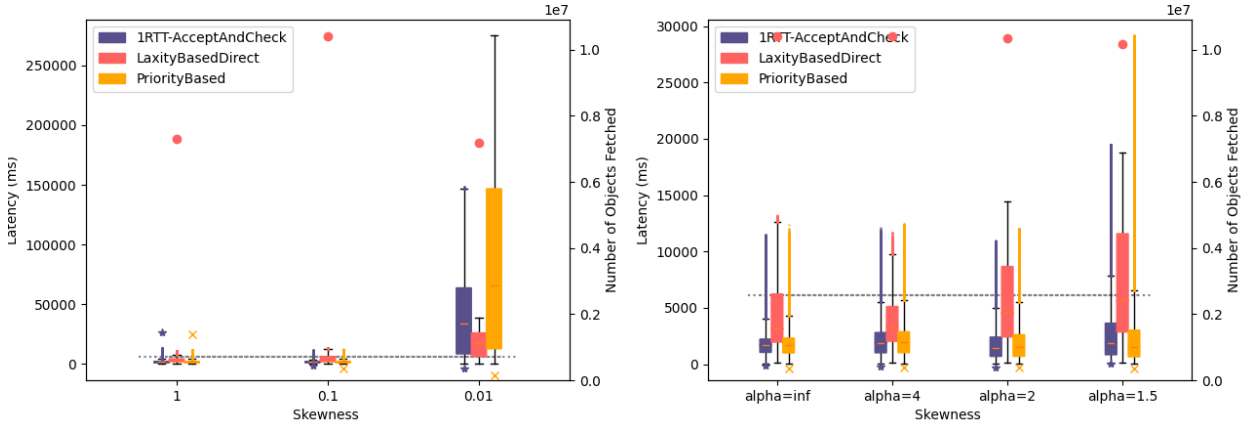
(b) *Varying state object sizes, cache capacity 2000.*

Figure 6.14: *Varying cache capacity and state object size: latency distribution and number of objects fetched.*

Figure 6.14a, we observe that increasing cache capacity helps overall requests latency to decrease. For locality-aware policy like `PriorityBased` and `1RTT-AcceptAndCheck`, relaxing the capacity constraint helps reduce request latencies at first (1000 to 2000) (the number of state operations is reduced by 3.7× for `1RTT-AcceptAndCheck` and 3.8× for `PriorityBased`). Further relaxing this constraint has little effect on latency, as the number of object fetches only reduces slightly and most of the objects accessed tend to be cached already. For DBS policy like `LaxityBasedDirect`, increasing storage capacity reduces the number of fetches, causing the majority of the request latencies to drop. We observe that `LaxityBasedDirect` provides the best tail latency (up to 17% compared to `1RTT-AcceptAndCheck` and 16% compared to `PriorityBased`) while its median latency is higher (up to 31% comparing to `1RTT-AcceptAndCheck`).

Varying the state object sizes has little effect on the number of objects fetched for each policy, as shown in Figure 6.14b. We observe that increased object size moderately increases the tail latency while using `1RTT-AcceptAndCheck` due to potential overhead caused by state accesses of reallocated functions. At the same time, the majority of the requests are not influenced by changing object sizes. Due to the higher number of object fetches (5× compared to locality-aware approaches), larger state object accesses impact performance significantly for `LaxityBasedDirect` (with 21% increase in tail latency and 26% increase in median latency). As we apply a relaxed latency target on the dataflow applications, `1RTT-AcceptAndCheck` triggers reallocation operations lazily, resulting in similar latency distribution compared to the SBS approach like `PriorityBased`. `PriorityBased` generates

less traffic to the Redis cluster by issuing only 4-5% more requests than the static-binding `PriorityBased` approach.



(a) *Handling constant popularity skewness across users.*

(b) *Handling dynamic popularity skewness across users (Pareto distribution).*

Figure 6.15: *Handling user popular skewness.*

`LaxityBasedDirect` is preferred for the workload that has a *static* set of popular accesses for *small* objects. `1RTT-AcceptAndCheck` is preferred for a workload that has data access with *changing popularity* over time. In Figure 6.15a, we explore latency distribution while the system receives requests that are skewed towards popular users, as described in [167]. The skewness indicates the proportion of users that are responsible for 90% of all requests. When the skewness is low, the latency distribution of different policies is similar, with `LaxityBasedDirect` with marginally higher latency due to the number of fetches performed. When we ingest requests that skew heavily (1% of users ingest 90% of requests), `LaxityBasedDirect` performs significantly better than the other approaches (with up to 10% improvement on success rate, 3.5× shorter tail latency than `1RTT-AcceptAndCheck` and 7.2× shorter tail latency than `PriorityBased`) as many heavily accessed data objects can be fetched once and reused many times (even on the lessee worker) due to small amount of popular accesses. This access pattern can be observed from the decreasing number of state operations as we increase skewness from 0.1 to 0.01, which leads to a drop of the number of remote fetches for `LaxityBasedDirect` (by 45%). The difference in performance in this scenario indicates that `LaxityBasedDirect` would be the preferred policy for skewed, eventual consistent requests with frequent accesses to a small amount of small, popular objects.

Figure 6.15b shows an experiment where we inject transient workload spikes for each

worker. The average workload rate we generate is equivalent to the workload we generate in Figure 6.15a. The rate of ingestion varies across time, forming a Pareto distribution with an increasing skewness factor. Figure 6.15b shows that `1RTT-AcceptAndCheck` is able to find a good balance between the SBS approach and DBS approach: It reduces the tail latency of SBS (by 60%) due to its ability to offload messages from workers that receive transient load spikes. Meanwhile, it can also provide an excellent overall latency compared to `LaxityBasedDirect` as it produces much fewer state accesses (19×) and therefore reduces median latency by 2×.

6.7.4 Nexmark Data Analytical Queries

The social network workload performs data read and writes in a highly concurrent fashion without strict requirements on data freshness. However, many dataflow applications require functions to be invoked in a certain order to provide a complete view of all past updates on function states. One crucial application scenario for real-time data processing frameworks is stream analytical applications. We implement two Nexmark benchmark queries based on [168, 169]. For a microservice application that we examined previously, the latency requirement set for the application is largely measured per request, meaning that the latency of one request may not be influenced by how later requests are handled. In contrast, many data streaming applications perform partial updates to results when updates are received by performing aggregation operations (14 out of 23 queries [168] perform aggregation operations). We implement two tumbling window aggregation queries (Highest Bid and Bid Quantization queries) from the Nexmark benchmark. Figure 6.16 shows the function DAG of Nexmark queries: we implement a 2-staged dataflow: The first stage receives streams of bid and auction information from clients. The second stage performs stateful aggregation operations on input batches. The highest bid query computes the highest bid price that has been received in the last tumbling window. The bid quantization query computes bid price distribution by building a full profile of all bid prices received in the last tumbling window. All window has a default duration of 10 seconds. To estimate

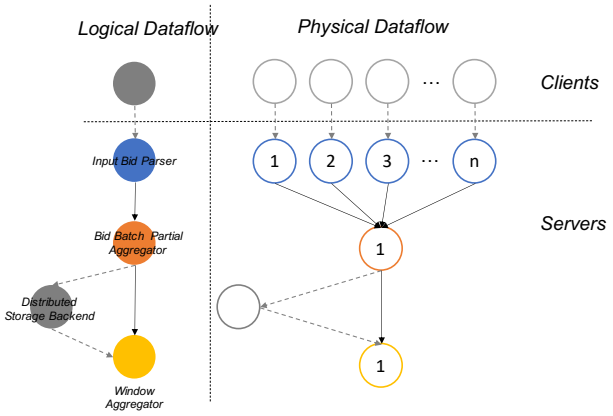
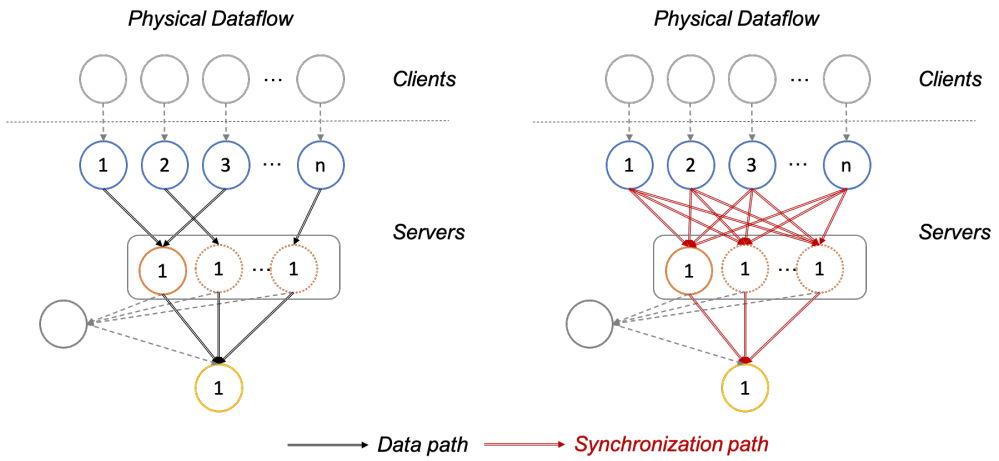
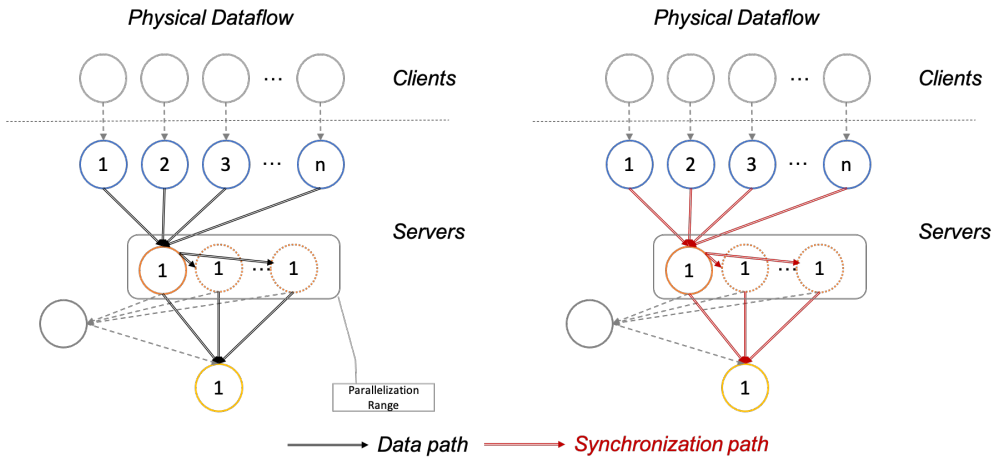


Figure 6.16: *Nexmark Aggregation Workload.*



(a) `LaxityBasedDirect` *synchronization.*



(b) `1RTT-RejectSend` *synchronization.*

Figure 6.17: *Nexmark queries: window aggregation latency.*

the execution cost of the downstream path, we use the sum of the execution cost of all operators along the execution path while calculating message priorities.

Directly schedule a message targeting aggregation function using one of the scheduling policies we described in Section 6.6 could lead to problems, as aggregation operation requires all past updates to be seen before new updates are applied. By default, scheduling policies could alter the order of messages executed between given pair and communicating functions. This could lead to missing updates when messages are processed after their window(s) close [119]. Most aggregations can be performed distributedly and incrementally [170], and messages targeting aggregation functions can be transparently offloaded when needed before the window closes. As a result, updates can be performed locally and then synchronized to global storage before final results are computed. Performing partial

aggregation allows messages to be reordered and executed on multiple workers without enforcing execution ordering globally, creating rooms for performance optimization. However, special messages (e.g., messages that trigger window boundary) still need to be executed after previous messages that contribute to the windowed result are seen and processed.

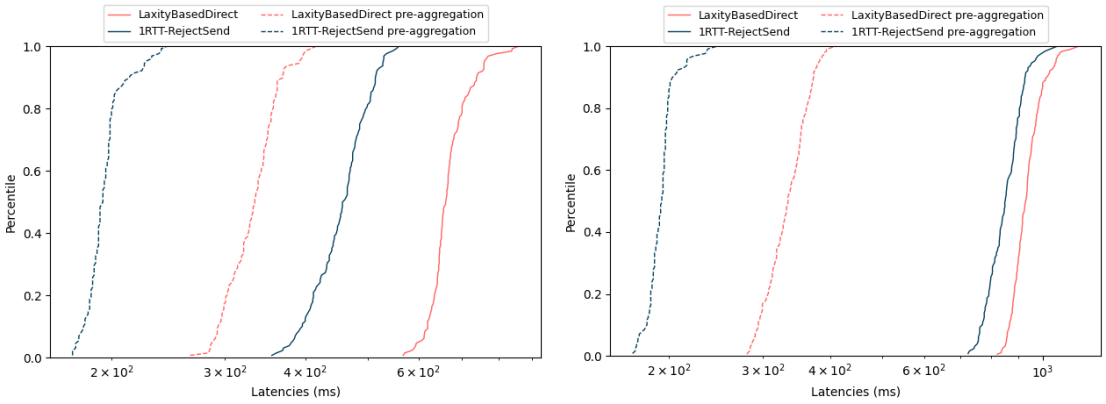
Here we show how scheduling policy can be designed to coordinate with this synchronization process. We create a special type of marker messages that originate from each data source when the window closes and propagate these messages downstream to ensure that messages that previously transferred between the same pair of operators are processed successfully.[5]These markers reach stage 2 function by reaching every worker that may have invoked this function. Once the marker is received, these functions flush partial states stored locally to remote storage and forward marker message to the final window aggregation function, where final results are retrieved and computed. We specify `PARALLELISM_RANGE`, so the scheduler only selects the neighboring range of `PARALLELISM_RANGE` from the designated lessor ID. Widening `PARALLELISM_RANGE` increases communication during synchronization, and narrowing this parameter reduces the number of choices for the scheduler to choose candidate workers.

Figure 6.17 depicts how function requests are transferred between communicating functions and how marker messages propagate through function DAG while applying two policies: `LaxityBasedDirect` (Figure 6.17a) and `1RTT-RejectSend` (Figure 6.17b). `LaxityBasedDirect` selects the target operator *before* a message is dispatched (Section 6.6). Therefore, marker messages need to be sent to *all possible downstream workers* to ensure all remaining messages between two workers that influence the next query result are delivered. We modify `LaxityBasedDirect` so users could specify messages (e.g., marker messages) that would be buffered until all previously pending messages on workers are delivered. On the other hand, `1RTT-RejectSend` dispatch all messages to the lessor worker before messages are either i) inserted into the lessor worker's queue or ii) forwarded to the selected lessee[6]. There exists no direct communication between stage 1 parser functions and any stage 2 functions except the lessor worker of the message. As a result, this synchronization can be achieved by two steps: i) each stage 1 function dispatch a marker message to the lessor worker (ensuring all messages sent from stage 1 functions are delivered); and ii) the lessor collects all markers and forwards a marker to each possible lessor within the `PARALLELISM_RANGE` neighboring range (ensuring all messages forwarded by lessors are delivered). For M stage 1 workers and `PARALLELISM_RANGE` of N, this process takes $M \times N$ messages for `LaxityBasedDirect` with

[5]We assume source operator and channels between each pair of workers deliver messages in order.

[6]We enable `FORCE_MIGRATION` in this example and we assume all channels between workers guarantee in-order message delivery.

1 Hop from stage 1 function to stage 2, and $(M + N)$ messages for `1RTT-RejectSend` with two hops from stage 1 function to stage 2 lessor and the lessor to all potential lessees.

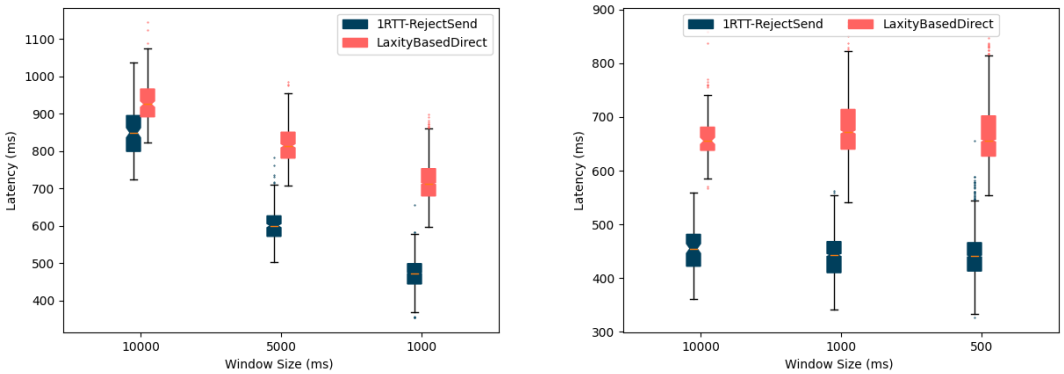


(a) *HighestBid Query Update Latency.* (b) *BidQuantization Query Update Latency.*

Figure 6.18: *Nexmark queries: latency distribution varied by policies. We show both final aggregation latencies and pre-aggregation latencies.*

OOS produces lighter synchronization overhead, making it a preferred policy to distribute frequent, in-order operations. In order to show how both DBS and OOS can work with dataflow applications to support distributed aggregation in Dirigo, in Figure 6.18, we plot both windowed query latency and pre-aggregation latency for highest bid and bid quantization queries. Here pre-aggregation latency indicates the latency between the generation time of the last contributing event and the time when the last state update is applied to any stage 2 function. For the highest bid query, latencies collected using `1RTT-RejectSend` are lower than `LaxityBasedDirect` by up to 30% (median latency) and 34% (tail latency). For the bid quantization query, latencies collected using `1RTT-RejectSend` are lower than `LaxityBasedDirect` by up to 9% for both median and tail latencies. We also observe that the aggregation stage is not the primary source of performance overhead. Instead, `LaxityBasedDirect`'s pre-aggregation latency is about 130ms higher than `1RTT-RejectSend` (measured at median latency) as synchronization messages from stage 1 function accumulate at stage 2 work queue, delaying updates sent from another source that targets the same window. We do not observe the same behavior for the bid quantization query. It collects large computational states, and synchronization of state objects generates significantly longer overhead than the highest bid query (Figure 6.18a).

Figure 6.19 shows how latency distribution progresses as we apply both policies with different `PARALLELISM_RANGE`. Figure 6.19a shows that reducing window size reduces overall latency for both `1RTT-RejectSend` and `LaxityBasedDirect` policies. We observe the gap



(a) *BidQuantization Query.* (b) *HighestBid Query.*

Figure 6.19: *Nexmark queries: latency distribution varying window sizes.*

between median latency `1RTT-RejectSend` and `LaxityBasedDirect` widens as we decrease window sizes (with median latency improvement of 9%, 35%, 51%). This increase shows that `1RTT-AcceptAndCheck` is preferable when the function states are small, and the synchronization process is the major contributor to the query latency. `LaxityBasedDirect`, on the other hand, is less sensitive to increasing function states as it distributes all partial aggregation operations evenly among operators. This effect is prominent for aggregation operations that cannot combine function states (e.g., quantization, topK, etc.). For `1RTT-AcceptAndCheck`, which prefers allocating messages on lessor workers, messages with lax deadline constraints are placed on the lessor worker's queue when the window size is large. This could lead to uneven distribution of state objects and influence state synchronization at the aggregation stage. We aim to improve Dirigo with dynamic state-operation awareness in our future work.

For an aggregation operator that can combine function states (e.g., average, sum, max, etc.) as in the highest bid query (Figure 6.19b), the size of function state is independent from the number of requests handled by each worker. Therefore, `1RTT-AcceptAndCheck`'s benefit is consistent as we change the window sizes (44%, 52%, 49%), showing that `1RTT-RejectSend` is the preferable policy using different window sizes.

We further show the benefit of operator parallelization through running overloading an aggregation operator using the bid quantization query and varying `PARALLELISM_RANGE` in Figure 6.20. The dashed line shows the aggregation operator's latencies being executed on a fixed worker in a single-threaded, in-order fashion (using `PriorityBased` scheduling policy). From the figure, we observe that when the `PARALLELISM_RANGE` is small, the system should opt to use the default `PriorityBased` policy as the synchronization process becomes the bottleneck of the processing pipeline. Further increasing `PARALLELISM_RANGE` reveals

the benefit of auto-parallelization provided by the system. Due to the lighter synchronization overhead, `1RTT-RejectSend` achieves a stable state (`PARALLELISM_RANGE` = 4) faster than `LaxityBasedDirect` (`PARALLELISM_RANGE` = 12) and provides a better median latency with fewer resources (`PARALLELISM_RANGE` = 4, compared to `PARALLELISM_RANGE` = 8 using `LaxityBasedDirect`).

6.8 DISCUSSION

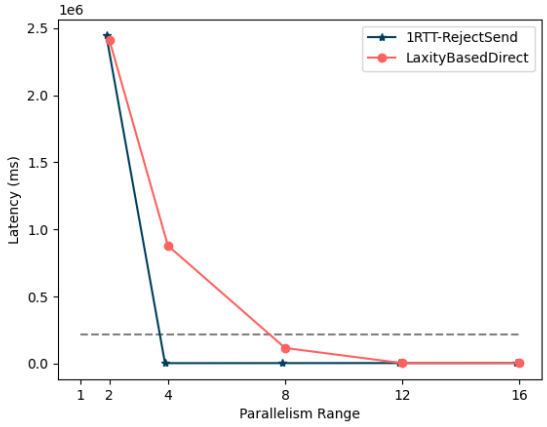


Figure 6.20: *Nexmark queries: latency distribution varying* ***PARALLELISM_RANGE****.*

Decentralized vs. centralized design: Generally speaking, nearly all Dirigo policies can be implemented in a decentralized and centralized way. We next compare the choice of decentralized vs. centralized mechanisms, albeit in a limited way, to justify our choice of decentralized mechanisms for Dirigo's policies.

A centralized message scheduler may lead to performance bottlenecks for millisecond granularity tasks performing computation for real-time data streams. In contrast, Dirigo's scheduling actions are performed either at the source or target workers on a per-message basis. This design prevents messages from being dispatched to a centralized entity. Here we show a comparison of performing centralized and decentralized scheduling using Dirigo using controlled dataflow with 1 ms granularity tasks.

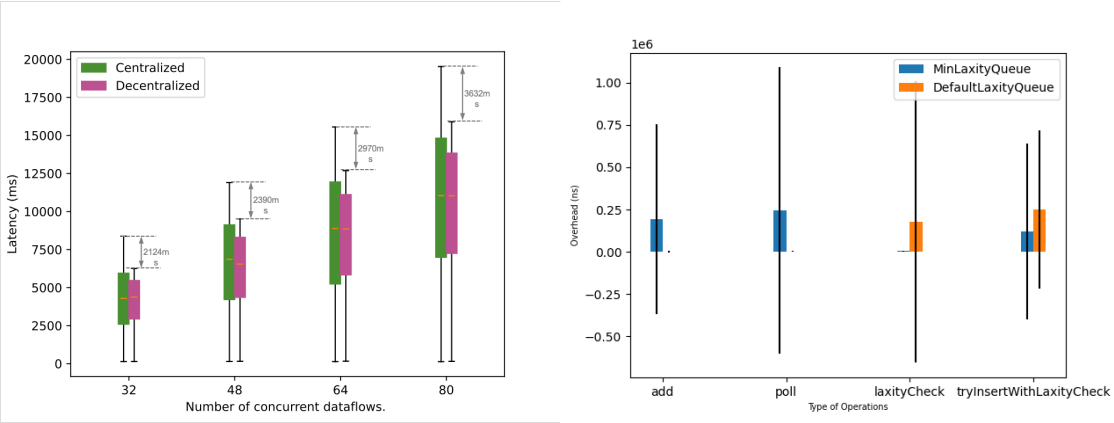
To perform centralized scheduling in Dirigo, we use one worker to serve as a scheduler worker and perform scheduling operations, and the rest of all workers perform message processing. All messages (from clients and function workers) are sent to the scheduler worker, and the scheduler dispatch messages to a random worker. For decentralized scheduling, we use Dirigo to run the `QueueBasedDirect` policy. As the goals of both scheduling policies are similar (load balancing all messages across all workers), we show that using centralized scheduling can result in higher request latency, as depicted in Figure 6.21a. Note that even though `QueueBasedDirect` performs more sophisticated scheduling actions (i.e., probing neighboring workers for queue size responses), it shows better scalability as we increase the number of concurrent dataflows running concurrently. As we increase the number of

dataflows from 32 to 80, the maximum latency gap between the two approaches expands from 2124ms to 3632ms (up by 71%).

Therefore we conclude that the decentralized mechanism is generally preferable to the centralized mechanism for implementing Dirigo-like policies.

Operation Overhead: Next we measure the overhead of operations in Dirigo. Specifically, we measure—are the new record queue operations worth the overhead they entail? Figure 6.21b shows the average comparison of queue operations performing Dirigo scheduling policies. Specifically, we run our experiments to record queue operations overhead when the message queue size is between 1000-2000. We record four types of queue operations, including the three operations we describe in Algorithm 6.1 and a `tryInserWithLaxityCheck` operation, where the scheduling policy first performs an acceptance check, and then inserts the message directly if the acceptance check is successful. We show that using minimum laxity queue improves `laxityCheck` and `CheckAndInsert` by 48× and 2× but increases the overhead of `add` and `poll` by 91× and 109× on average respectively. The impact of the queue implementation depends on how scheduling policies use these operations.

We conclude that minimum laxity queue is beneficial for scheduling policies that use laxity check operations often (e.g., performing many acceptance checks on candidate workers). This is generally true when the load across workers is imbalanced and high, and each message may entail checks multiple times (in the system) before it is added.



(a) *Centralized vs. decentralized scheduling as we increase the number of concurrent running dataflows.*

(b) *Queue operation overhead.*

Figure 6.21: *Dirigo implementation.*

6.9 CONCLUSION

In this chapter, we present Dirigo, a serverless function runtime that supports proactive function scheduling. Dirigo natively supports in-memory function states. We build a set of scheduling policies that aims to find a middle ground between balancing workload skewness and preserving memory state locality. We test three categories of policies on state-of-art benchmarks to explore the state spaces of application scenarios where each policy category can provide the most benefit.

CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1 SUMMARY OF CONTRIBUTIONS

System elasticity has long been the focus of research for real-time data processing systems. With new types of data-centric applications that seek low-latency, near-real-time services counting to emerge, this problem will continue to be important in the future. This thesis proposed several techniques to help real-time data processing systems to *provide tunable performance under the influence of many environmental variables without compromising resource efficiencies.*

In Chapter 2, we proposed Stela [1], an stream processing system that performs on-demand scheduling that 1) optimizes the post-scaling throughput; and 2) minimizes the interruption to the ongoing computation. For scale-out, Stela selects which operators (inside the application) are given more resources, and does so with minimal intrusion. Similarly, for scale-in, Stela selects which machine(s) to remove in a way that minimizes the overall detriment to the application's performance.

We believe that deployers of the jobs should be able to specify performance targets as intents of these jobs. The intent of these jobs should be i) user-facing (i.e., not involving the system's internal metrics); and ii) interpretable across jobs with different types of performance targets (latency and throughput). In Chapter 3 described how we solve these challenges: In Henge [2], we proposed using a utility function to define job SLO, and Henge uses utility function in its adaptation state machine to enforce its reconfiguration policy. We also proposed Juice for Henge — a metric that captures the percentage of input data that is effectively processed per unit of time. Juice reflects the processing efficiency of a job with throughput-centric SLO.

Interactive data analytics engines typically handle real-time ingested data and analytics queries that involve both newly ingested and historical data. In Chapter 4, we discussed Getafix [3], a segment management solution that explores the query access pattern across data segments observed from production traces. We observed that the popular (in terms of the number of accesses per unit of time) shifts over time, and Getafix leverages an adaptive replication strategy (ModifiedBestFit) to i) determine the optimal plan that includes the number of replications and the placement of the replicas that minimizes query span and memory consumption (in the static scenario), and ii) minimize network transfer between replication placement plan periodically. Getafix is load-balance-aware, and the ModifiedBestFit policy can be adapted to replication in heterogeneous settings.

In Chapter 5, we built Cameo [4] to explore a new *fine-grained* philosophy for designing a multi-tenant stream processing system. Our key idea is to provide resources to each *operator* based solely on its *immediate* need that stems from the priority of the data just received at the operator. Our design handles temporal and spatial workload variation by i) *Proactively* deriving priorities for messages (and their target operators) to match performance goals in the workload specification; ii) Using a *stateless* scheduler that scales to many dataflows. Instead of tracking each dataflow application, we assign priorities using a scheduling context that is passed along with individual messages flowing between operators; iii) Improving this prioritization both *statically* (e.g., accounting for query windowing semantics) and *dynamically* (by profiling query execution at runtime).

In Chapter 6, we described Dirigo, a distributed scheduling framework for real-time stateful dataflow applications with user intent. We envision that future real-time dataflow processing should be adopting the serverless architecture with underlying runtime providing native support for *application-level* function states. We built a serverless stack and a scheduling framework prototype that can accommodate three categories of intent-aware scheduling policies. We further tested these policies against two stateful dataflow applications and studied the best scheduling policies under different application scenarios.

7.2 FUTURE WORK

Cloud computing has gone through many waves of innovations during the last decade. The most recent one was led by revolutions in the *cloud deployment paradigm* as well as the *emerging new cloud infrastructures*. This thesis lays the groundwork for many future directions:

Self-managing elastic real-time dataflows as functions: Despite being the most elastic, cost-efficient deployment option for cloud users, serverless platforms have not yet become the most optimal target framework for massive scale, stateful cloud applications. This is because i) runtime scheduler does not support automated, state-aware scheduling; ii) many stateful applications maintain large function state that cannot be migrated efficiently; iii) instant scale-out/in leads to many state updates that cannot be merged/aggregated efficiently; and iv) the state functions and their accesses are largely transparent to underlying runtime and therefore cannot be optimized towards in-network devices and state storage. Despite many applications showing significant resource efficiency improvement while deployed in a serverless fashion [171], they are mostly stateless or close to stateless, and supporting stateful applications with massive scale (e.g., data analytics, ML training) remains an open issue. To fully achieve this vision, it is crucial to explore the following directions:

- **Automated Stateful Function Scheduling:** In Chapter 6 we explored scenarios where different scheduling policies should be applied to stateful dataflow applications based on various environmental factors (e.g., processing semantics, user intent, data skewness, state access pattern, etc.). To best serve stateful functions in a multi-tenant environment, schedulers such as Dirigo should be extended to 1. support automated policy selection in order to adapt to workload needs on the fly and 2. support per-dataflow user-intent through user-facing resource provisioning API.

- **Scheduler/State Cache Co-design:** Dirigo fetches state objects synchronously, meaning that scheduling policy could be extended to predict whether a state object should be fetched to achieve a better estimation for resource planning. In order to achieve an accurate prediction, schedulers like Dirigo (or other runtime components) should be extended to track previously routed requests or previously accessed state object that remains in the object cache.

- **In-cache State Management:** Another potential optimization to build on the serverless runtime is that the state management (e.g., state read, write, aggregation, etc.) could be performed out of the critical path. This optimization requires function runtime to create a *data path* that asynchronously performs state operation and choose what (and how) to migrate (or offload) state objects intelligently. Once a scheduling decision is made, the runtime can coordinate with the scheduler to pre-load data dependencies in order to hide data fetching overhead.

Supporting hardware-agnostic function that can adapt to the heterogeneous cloud: Utilizing a hybrid cloud has become a prominent issue recently due to the recent progress in custom-designed chips and programmable hardware. The future cloud architecture creates new opportunities for us to rethink how to efficiently deploy real-time data processing applications and build middleware to better accommodate the scenarios we have discussed. Recent works [172, 173] propose solutions for cloud applications to better adapt to the next-generation hybrid cloud using traditional (i.e., monolithic) hardware architecture. Compared to existing architecture, disaggregated architecture provides two unique opportunities to achieve high resource utilization that today's cloud frameworks have yet to embrace: i)As data generally have weaker locality and computational resources can stay mostly stateless, functions can be dynamically re-allocated/parallelized to various types of hardware resources if desired; and ii) For applications that require different types of target devices, data does not need to be explicitly transferred between device memories but instead could be accessed from memory nodes and/or storage devices.

Cross-layer System Elasticity: Works discussed in this thesis all focus on improving system elasticity on an individual layer of the cloud stack. However, how these solutions can be best incorporated in a non-interfering fashion remains an open question. Past research in other areas of networking [174] has shown that supporting user-level requirements within the underlying framework introduces performance benefits but needs to be carefully thought out. Existing works such as [175, 176, 177] had to carefully design cross-layer solutions for systems in the areas including wireless networks, video streaming systems, and sensor networks. A potential direction of future research is to investigate how these elasticity solutions work concurrently in cloud applications.

REFERENCES

[1] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 22–31.

[2] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven Multi-Tenant Stream Processing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3267809.3267832 pp. 249–262.

[3] M. Ghosh, A. Raina, L. Xu, X. Qian, I. Gupta, and H. Gupta, "Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 40.

[4] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 389–405.

[5] Markets and Markets, "Streaming Analytics Market by Component, Application (Predictive Asset Management, Risk Management, Location Intelligence, Sales and Marketing, Supply Chain Management), Industry Vertical, Deployment Model, and Region - Global Forecast to 2024," Report. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html

[6] Research and Markets, "Streaming Analytics Market by Verticals - Worldwide Market Forecast & Analysis (2015 - 2020)," Report, June 2015. [Online]. Available: https://www.researchandmarkets.com/research/mpltnp/streaming

[7] The Apache Software Foundation, "Storm," 2015. [Online]. Available: https://storm.apache.org

[8] 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 2015 ACM SIGMOD international conference on Management of data*, 2015.

[9] The Apache Software Foundation, "Flink," 2014. [Online]. Available: https://flink.apache.org

[10] 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 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[11] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A Real-time Analytical Data Store," in *Proceedings of the 2014 ACM International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595631 pp. 157–168.

[12] LinkedIn, "Pinot," 2015. [Online]. Available: https://github.com/linkedin/pinot/wiki

[13] Amazon, "Redshift," 2012. [Online]. Available: https://aws.amazon.com/redshift/

[14] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal, "Mesa: A Geo-replicated Online Data Warehouse for Google's Advertising System," *Communications of the ACM*, vol. 59, no. 7, pp. 117–125, June 2016. [Online]. Available: http://doi.acm.org/10.1145/2936722

[15] Facebook, "PrestoDB," 2013. [Online]. Available: https://prestodb.io/

[16] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian et al., "Scuba: Diving into Data at Facebook," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1057–1067, 2013.

[17] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa et al., "Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.

[18] Amazon Redshift, "Customer Success," 2018. [Online]. Available: https://aws.amazon.com/redshift/customer-success/

[19] Metamarkets, "Powered by Druid," 2018. [Online]. Available: http://druid.io/druid-powered.html

[20] D. Laney, "3D Data Management: Controlling Data Volume, Velocity and Variety," *META group research note*, vol. 6, no. 70, p. 1, 2001.

[21] 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*. ACM, 2014, pp. 238–245.

[22] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin et al., "Aurora: A Data Stream Management System," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2003, pp. 666–666.

[23] A. Maskey, E. Ryvkina, M. Cherniack, Y. Ahmad, B. Berg, U. Cetintemel, H. Jeong-Hyon, O. Papaemmanouil, A. Rasin, N. Tatbul, Y. Xing, S. Zdonik, D. Abadi, M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Borealis, Distributed Stream Processing Engine," 2017. [Online]. Available: http://cs.brown.edu/research/borealis/public/

[24] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 23–27.

[25] 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.

[26] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic Scaling of Data Parallel Operators in Stream Processing," in *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009.* IEEE, 2009, pp. 1–12.

[27] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A Distributed, Scalable Platform for Data mining," in *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms.* ACM, 2006, pp. 27–37.

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

[29] K.-L. Wu, K. W. Hildrum, W. Fan, P. S. Yu, C. C. Aggarwal, D. A. George, B. Gedik, E. Bouillet, X. Gu, G. Luo et al., "Challenges and Experience in Prototyping a Multimodal Stream Analytic and Monitoring Application on System S," in *Proceedings of the 33rd International Conference on Very Large Databases.* VLDB Endowment, 2007, pp. 1185–1196.

[30] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," in *Proceedings of the International Conference on Management of Data (SIGMOD).* ACM, 2008, pp. 1123–1134.

[31] H. Gupta, "Beyond Hadoop at Yahoo!: Interactive analytics with Druid," Talk, September 2016. [Online]. Available: https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51640

[32] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth et al., "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing.* ACM, 2013, p. 5.

[33] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 11, 2011, pp. 22–22.

[34] The Linux Foundation, "Kubernetes: Production-Grade Container Orchestration." [Online]. Available: https://kubernetes.io/

[35] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-Regulating Stream Processing in Heron," *Proceedings of the VLDB Endowment*, August 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dhalion-self-regulating-stream-processing-heron/

[36] 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.

[37] 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 6th ACM Symposium on Cloud Computing*. ACM, 2015, pp. 276–287.

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

[39] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions For Distributed Streaming Dataflows," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 783–798.

[40] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven multi-tenant stream processing," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2018, pp. 249–262.

[41] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic Scaling of Data Parallel Operators in Stream Processing," in *Proceedings of International Parallel and Distributed Processing Symposium*. IEEE, 2009, pp. 1–12.

[42] M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. Dimitrova, S. Wicki, Z. Chothia, and T. Roscoe, "Snailtrail: Generalizing Critical Paths For Online Analysis of Distributed Dataflows," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 95–110.

[43] M. Hoffmann, A. Lattuada, and F. McSherry, "Megaphone: Latency-conscious State Migration For Distributed Streaming Dataflows," *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 1002–1015, 2019.

[44] 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 2013 ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 725–736.

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

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

[47] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch, "Overload Management in Data Stream Processing Systems with Latency Guarantees," in *Proceedings of the 7th IEEE International Workshop on Feedback Computing (Feedback Computing)*, 2012.

[48] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "Themis: Fairness in Federated Stream Processing under Overload," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 541–553.

[49] 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.

[50] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and Adaptable Stream Processing at Scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 374–389.

[51] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-Aware Elastic Scaling for Distributed Data Stream Processing Systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[52] M. Wall, "Big Data: Are you ready for blast-off?" 2016, ONLINE. [Online]. Available: http://www.bbc.com/news/business-26383058

[53] The Apache Software Foundation, "Apache Hadoop," last Visited: December 1, 2021. [Online]. Available: http://hadoop.apache.org/

[54] The Apache Software Foundation, "Apache Hive," 2016, ONLINE. [Online]. Available: https://hive.apache.org/

[55] The Apache Software Foundation, "Apache Pig," 2016, ONLINE. [Online]. Available: http://pig.apache.org/

[56] The Apache Software Foundation, "Apache Spark," last Visited: December 1, 2021. [Online]. Available: https://spark.apache.org/

[57] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[58] The Apache Software Foundation, "Apache Storm," http://storm.apache.org/, last Visited: December 1, 2021.

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

[60] 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 EDBT/ICDT Workshops*. ACM, 2012, pp. 55–60.

[61] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica et al., "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

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

[63] The Apache Software Foundation, "Apache Zookeeper," last Visited: December 1, 2021. [Online]. Available: http://zookeeper.apache.org/

[64] "Emulab," last Visited: December 1, 2021. [Online]. Available: http://emulab.net/

[65] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin et al., "Aurora: A Data Stream Management System," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 2003, pp. 666–666.

[66] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, "Load Management and High Availability in the Borealis Distributed Stream Processing Engine," in *GeoSensor Networks*. Springer, 2008, pp. 66–85.

[67] 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 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[68] B. Lohrmann, P. Janacik, and O. Kao, "Elastic Stream Processing with Latency Guarantees," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, June 2015, pp. 399–410.

[69] C. Jones, J. Wilkes, N. Murphy, and C. Smith, "Service Level Objectives," last Visited: December 1, 2021. [Online]. Available: https://landing.google.com/sre/book/chapters/service-level-objectives.html

[70] 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.

[71] "Uber Releases Hourly Ride Numbers In New York City To Fight De Blasio," last Visited: December 1, 2021. [Online]. Available: https://techcrunch.com/2015/07/22/uber-releases-hourly-ride-numbers-in-new-york-city-to-fight-de-blasio/

[72] "How to collect and analyze data from 100,000 weather stations," last Visited: December 1, 2021. [Online]. Available: https://www.cio.com/article/2936592/big-data/how-to-collect-and-analyze-data-from-100000-weather-stations.html

[73] "Storm Applications," last Visited: December 1, 2021. [Online]. Available: http://storm.apache.org/Powered-By.html

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

[75] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-Based Scheduling: If You're Late Don't Blame Us!" in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[76] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards Automated SLOs for Enterprise Clusters," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, p. 117.

[77] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[78] Wikipedia, "Pareto Efficiency — Wikipedia, The Free Encyclopedia," 2016, last Visited December 1, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pareto_efficiency&oldid=741104719

[79] "SLOs," last Visited: December 1, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Service_level_objective

[80] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[81] M. Naaman, A. X. Zhang, S. Brody, and G. Lotan, "On the Study of Diurnal Urban Routines on Twitter." in *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media*, 2012.

[82] "SDSC-HTTP Trace," last Visited: December 1, 2021. [Online]. Available: http://ita.ee.lbl.gov/html/contrib/SDSC-HTTP.html

[83] "EPA-HTTP Trace," last Visited: December 1, 2021. [Online]. Available: http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html

[84] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik, "Distributed Operation in the Borealis Stream Processing Engine," in *Proceedings of the 2005 ACM International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066274 pp. 882–884.

[85] "Lambda Architecture," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Lambda_architecture

[86] "Kappa Architecture," 2019. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[87] Wikipedia, "Bin Packing Problem," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem

[88] W. Stallings, *Operating Systems: Internals and Design Principles Edition: 5*. Pearson, 2005.

[89] Wikipedia, "Hungarian algorithm," 2018. [Online]. Available: http://en.wikipedia.org/wiki/Hungarian_algorithm

[90] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966472 pp. 287–300.

[91] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, "DShark: A General, Easy to Program and Scalable Framework for Analyzing in-Network Packet Traces," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19, USA, 2019, p. 207–220.

[92] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud Computing for Everyone," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2011, p. 16.

[93] "Orleans." [Online]. Available: https://dotnet.github.io/orleans/

[94] "Akka." [Online]. Available: https://akka.io/

[95] "Azure Functions." [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[96] "Serverless Streaming Architectures and Best Practices, amazon web services." [Online]. Available: https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf

[97] "Google Cloud Functions." [Online]. Available: https://cloud.google.com/functions

[98] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*, 2019.

[99] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[100] "Storm Multitenant Scheduler." [Online]. Available: https://storm.apache.org/releases/current/javadocs/org/apache/storm/scheduler/multitenant/package-summary.html

[101] C. L. Liu and J. W. Layland, "Scheduling Algorithms For Multiprogramming in a Hard-real-time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[102] A. K.-L. Mok, "Fundamental Design Problems of Distributed Systems For the Hard-real-time Environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

[103] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters." in *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018, pp. 4–1.

[104] P. Garefalakis, K. Karanasos, and P. Pietzuch, "Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019, pp. 233–245.

[105] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein, "Optimizing Distributed Actor Systems For Dynamic Interactive Services," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 38.

[106] P. Haller and M. Odersky, "Scala Actors: Unifying Thread-based and Event-based Programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[107] P. A. Bernstein, T. Porter, R. Potharaju, A. Z. Tomsic, S. Venkataraman, and W. Wu, "Serverless Event-Stream Processing over Virtual Actors," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2019.

[108] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., "Storm @ Twitter," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 147–156.

[109] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond Analytics: The Evolution of Stream Processing Systems," in *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, 2020, pp. 2651–2658.

[110] A. Kumar, Z. Wang, S. Ni, and C. Li, "Amber: A Debuggable Dataflow System Based on the Actor Model," *Proceedings of the VLDB Endowment*, vol. 13, no. 5, pp. 740–753, 2020.

[111] N. Li and Q. Guan, "Deadline-Aware Event Scheduling for Complex Event Processing Systems," in *Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2013, pp. 101–109.

[112] Z. Ou, G. Yu, Y. Yu, S. Wu, X. Yang, and Q. Deng, "Tick Scheduling: A Deadline Based Optimal Task Scheduling Approach For Real-time Data Stream Systems," in *Proceedings of the International Conference on Web-Age Information Management*. Springer, 2005, pp. 725–730.

[113] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A New Model and Architecture For Data Stream Management," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 120–139, 2003.

[114] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 668–668.

[115] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[116] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator Scheduling in a Data Stream Manager," in *Proceedings 2003 VLDB Conference*. Elsevier, 2003, pp. 838–849.

[117] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator Scheduling For Memory Minimization in Data Stream Systems," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 253–264.

[118] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and Evaluation Techniques For Window Aggregates in Data Streams," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 311–322.

[119] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, pp. 1792–1803, 2015.

[120] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, "An Application of Bin-packing to Multiprocessor Scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[121] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo, "Implications of Classical Scheduling Results for Real-time Systems," *Computer*, vol. 28, no. 6, pp. 16–25, 1995.

[122] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[123] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A High-performance Incremental Query Processor for Diverse Analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, 2014.

[124] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order Processing: A New Architecture for High-performance Stream Systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[125] "Flink time attribute." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/event_time.html

[126] "Apache Kafka Core Concepts." [Online]. Available: https://kafka.apache.org/11/documentation/streams/core-concepts

[127] ".NET ConcurrentBag." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1?view=netframework-4.8

[128] Azure, "Sizes for windows virtual machines in azure." [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes

[129] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proceedings of the 18th ACM Symposium on Operating System Principles SOSP*. ACM, 2001, pp. 230–243.

[130] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing." in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, vol. 3, 2003, pp. 257–268.

[131] M. Balazinska, H. Balakrishnan, and M. Stonebraker, "Load Management and High Availability in the Medusa Distributed Stream Processing System," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 929–930.

[132] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 261–272.

[133] V. Raman, B. Raman, and J. M. Hellerstein, "Online Dynamic Reordering For Interactive Data Processing," in *VLDB*, vol. 99, 1999, pp. 709–720.

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

[135] X. Li, Z. Jia, L. Ma, R. Zhang, and H. Wang, "Earliest Deadline Scheduling for Continuous Queries Over Data streams," in *Proceedings of the IEEE International Conference on Embedded Software and Systems*. IEEE, 2009, pp. 57–64.

[136] Y. Gu, G. Yu, and C. Li, "Deadline-aware Complex Event Processing Models Over Distributed Monitoring Streams," *Mathematical and Computer Modelling*, vol. 55, no. 3-4, pp. 901–917, 2012.

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

[138] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[139] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style Processing of Fast Data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, 2012.

[140] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of the IEEE Data Mining Workshops (ICDMW)*. IEEE, 2010, pp. 170–177.

[141] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A Catalog of Stream Processing Optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[142] 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, (ICDE)*. IEEE, 2006, pp. 49–49.

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

[144] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An Elastic and Highly Available Streaming Service in the Cloud," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*. ACM, 2012, pp. 55–60.

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

[146] 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 (ICDE)*. IEEE, 2015, pp. 723–734.

[147] 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, 2013, pp. 1–14.

[148] 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.

[149] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A Better Stream Processing Engine For the Edge," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019, pp. 929–946.

[150] "AWS Lambda." [Online]. Available: https://aws.amazon.com/lambda/

[151] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan et al., "Ray: A Distributed Framework for Emerging AI Applications," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[152] A. Akhter, M. Fragkoulis, and A. Katsifodimos, "Stateful Functions as a Service in Action," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1890–1893, 2019.

[153] "Using AWS Lambda with Amazon Kinesis." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html

[154] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[155] "Serverless Architectures with AWS Lambda." [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/serverless-architectures-lambda/timeout.html

[156] "Serverless Needs a Bolder, Stateful Vision." [Online]. Available: https://thenewstack.io/serverless-needs-a-bolder-stateful-vision/

[157] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[158] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A Scalable and Locality-enhanced Framework for Serverless Parallel Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[159] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A Scalable Low-latency Serverless Platform," *arXiv preprint arXiv:1911.09849*, 2019.

[160] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 311–327.

[161] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[162] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[163] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[164] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid Datacenter Scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 499–510.

[165] "Redis." [Online]. Available: https://redis.io

[166] "Stateful Functions: A Platform-Independent Stateful Serverless Stack." [Online]. Available: https://ci.apache.org/projects/flink/flink-statefun-docs-master/

[167] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson et al., "An Open-source Benchmark Suite for Microservices and Their Hardware-software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[168] "Nexmark Benchmark Suite." [Online]. Available: https://github.com/nexmark/nexmark/

[169] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, "NEXMark—A Benchmark for Queries over Data Streams DRAFT," Technical report, OGI School of Science & Engineering at OHSU, Septembers, Tech. Rep., 2008.

[170] S. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting Aggregate Queries Over Ad-hoc Wireless Sensor Networks," in *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 2002, pp. 49–58.

[171] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski et al., "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[172] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "AlloX: Compute Allocation in Hybrid Clusters," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[173] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.

[174] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguments in System Design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[175] S. Shakkottai, T. S. Rappaport, and P. C. Karlsson, "Cross-layer Design For Wireless Networks," *IEEE Communications magazine*, vol. 41, no. 10, pp. 74–80, 2003.

[176] E. Setton, T. Yoo, X. Zhu, A. Goldsmith, and B. Girod, "Cross-layer Design of Ad Hoc Networks for Real-time Video Streaming," *IEEE Wireless Communications*, vol. 12, no. 4, pp. 59–65, 2005.

[177] A. G. Ruzzelli, G. M. O'Hare, and R. Jurdak, "MERLIN: Cross-layer Integration of MAC and Routing For Low Duty-cycle Sensor Networks," *Ad Hoc Networks*, vol. 6, no. 8, pp. 1238–1257, 2008.