Online Resource Optimization for Elastic Stream Processing with Regret Guarantee

Yang Liu
Shanghai University
China

Huanle Xu
University of Macau
Macau

Wing Cheong Lau
The Chinese University of Hong Kong
Hong Kong

ABSTRACT

Recognizing the explosion of large-scale real-time analytics needs, a plethora of stream processing systems, such as Apache Storm and Flink, have been developed to support such applications. Under these systems, a stream processing application is realized as a directed acyclic graph (DAG) of operators, where the resource configuration of each operator has a significant impact on its overall throughput and latency performance. However, there is a lack of dynamic resource allocation schemes, which are theoretically sound and practically implementable, especially under the drastically changing offered load. To address this challenge, we present *Dragster*[1], an online-optimization-based dynamic resource allocation scheme for elastic stream processing. By combining the online optimization framework with upper confidence bound (UCB) techniques, *Dragster* can guarantee, in expectation, a sub-linear increase in the throughput regret w.r.t. time. To demonstrate the efficacy, we implement *Dragster* to improve the throughput of Flink applications over Kubernetes. Compared to the state-of-the-art algorithm *Dhalion*, *Dragster* can achieve a 1.8X-2.2X speed-up in converging to the optimal configuration. It can contribute to 20.0%-25.8% gain in tuple-processing goodput and 14.6%-15.6% cost-savings.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**.

KEYWORDS

elastic stream processing, cloud computing, resource allocation, online optimization, gaussian-process UCB, kubernetes

ACM Reference Format:
Yang Liu, Huanle Xu, and Wing Cheong Lau. 2022. Online Resource Optimization for Elastic Stream Processing with Regret Guarantee. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545063

[1] *Dragster* is a Dynamic Resource Allocation scheme which provides a Guarantee on Streaming Throughput by bounding its Expected Regret compared to the offline optimal resource allocation.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545063

1 INTRODUCTION

As the concept of *Cloud Native* becomes more and more popular, a plethora of distributed stream processing systems, such as Spark and Flink, have been migrated to modern clouds to process large-scale dynamic streams with low latency and high throughput on the fly. However, such naive migration does not take full advantage of the cloud computing model. Distributed stream processing applications ignore the elasticity in cloud computing and still suffer from low resource utilization. In this sense, a crucial challenge drawn the attention of researchers and developers is the complexity of dynamically adjusting the configuration of such applications to fully utilize the computing resource and guarantee QoS requirements, especially under drastically changing workloads.

With the above observations in mind, in this paper, we aim to design an online-optimization-based dynamic resource allocation scheme that can track the optimal resource allocation to maximize the application throughput with high resource utilization for elastic stream processing. However, it is challenging to achieve this goal without careful design. First, operators in the stream processing application compose a directed acyclic graph (DAG), where the performance of each operator heavily depends on each other. In this sense, dynamically adjusting the resource allocation of all operators to improve the throughput or latency is not a good choice without understanding the topology of a DAG. Second, the performance of an operator is affected by the amount of configured resources in an unpredictable way. Increasing resources by adding an extra executor may only result in a marginal performance gain. The relationship between the resource configuration and operator performance is non-trivial (e.g., non-linear and multi-modal) and difficult to characterize using a simple well-formed function. Meanwhile, multiple users/ jobs share computing resources in public clouds with non-perfect separation/ isolation leading to dynamic cloud noises. For example, Google Cloud always overcommits [6] computing resources to guarantee the CPU utilization above 90% in each server. As such, the running-time performance of the same application under the same configuration may still suffer from a large variance, which makes performance prediction much more complex. Third, long-running stream processing applications inevitably face gradual drifts/ unexpected changes of offered loads that can threaten their stability. The stream processing system must react to external shocks by appropriately scaling up or scaling down the resource allocation to guarantee stability at all times.

In this paper, we present *Dragster*, an online-optimization-based dynamic resource allocation strategy for elastic stream processing. *Dragster* models a stream processing application as a DAG and takes a two-level online optimization framework to maximize the throughput. First, *Dragster* adopts online optimization algorithms to identify the bottleneck operator according to its service capacity.

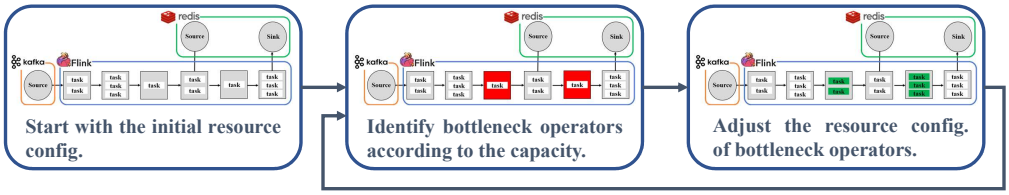


Figure 1: Two-level optimization framework of *Dragster*.

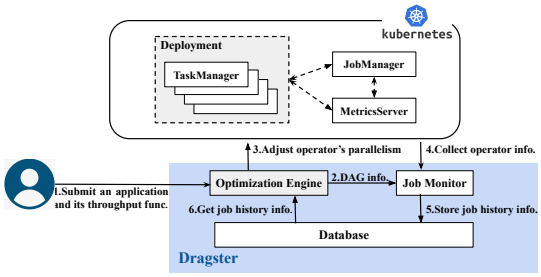


Figure 2: System Architecture of *Dragster* over Kubernetes.

Second, *Dragster* relies on Bayesian optimization techniques to adjust the resource configuration of the bottleneck operator. In addition, we consider the case where each operator contains a buffer and formulate a long-term constraint [23] to control the buffer size. More importantly, *Dragster* can provide a performance guarantee of sub-linearly increasing regret [8] of the application throughput and upper bounded buffer size, compared to the optimal solution. The regret guarantees the high throughput of *Dragster*, while the upper-bounded buffer size results in the low latency.

To evaluate the performance of *Dragster*, we have conducted extensive experiments on our Kubernetes-based [20] implementations aiming at maximizing the throughput of Flink applications under the Nexmark benchmark and Yahoo streaming benchmarks [30][25]. Experimental results show that *Dragster* achieves a 1.8X-2.2X speedup in convergence time to reach a near-optimal configuration (i.e., within 10% of the optimal throughput) and processes 20.0%-25.8% more tuples, compared to the state-of-art algorithm *Dhalion*. Meanwhile, *Dragster* can fully utilize computing resources which contributes to 14.6%-15.6% cost-savings for processing the same number of tuples as *Dhalion*.

The rest of the paper is organized as follows. After reviewing related works in Section 2, Section 3 presents the system design of *Dragster*. Section 4 and Section 5 present the optimization engine and its algorithmic approach. We evaluate *Dragster* via running real experiments in Section 6 and conclude this paper in Section 7.

2 RELATED WORK

In recent years, heuristic rule-based algorithms, such as *Dhalion* [16] in Twitter Heron and *dynamic allocation* [3] in Spark, have been implemented in main-stream systems. *Dhalion* [16] is the most powerful rule-based algorithm running on Twitter Heron. Users can define rules comprising a set of symptoms, diagnosis, and resolution actions. Symptoms provide information about observed anomalies and lead to the generation of a single diagnosis. However, rule-based algorithms can not learn from history and suffer from the same adjusting process facing recurrent workload changes.

Bayesian optimization is a popular online framework to handle black-box optimization problems [12], which has been widely applied for selecting the optimal resource configuration in batch processing systems [9][22]. *BO4CO*[17] directly adopts it in the stream processing case. Many other black-box optimization strategies, such as Latin hypercube sampling and hill-climbing algorithm, have also been studied in [11, 21, 32]. These kinds of works ignore the DAG information of the streaming application and easily lead to heavy overheads, i.e., hundreds or thousands of evaluations before obtaining the optimal configuration.

Recently, a body of works start to consider the DAG information of the stream processing application. *FIRM*[24] and *Re-storm*[28] define the path with the longest latency as a critical path and iteratively adjust the configuration among the critical path to optimize the latency. *DS2*[18] is a dynamic scaling controller which linearly increases/ decreases the number of executors in each operator based on the processing rate of upstreams. However, these works usually assume the computation performance, e.g., latency or throughput, for candidate configurations known beforehand or following a simple, fixed mapping. In the real world, the relationship between resource configuration and performance is non-trivial (e.g., non-linear and multi-modal) and difficult to characterize using a simple well-formed function.

3 SYSTEM OVERVIEW OF DRAGSTER

In this section, we provide an overview of *Dragster*. Figure 1 presents our two-level optimization framework. For a stream processing application, *Dragster* starts with an initial configuration, runs it, and collects the performance metrics (e.g., throughput, latency, and CPU utilization) to evaluate the current configuration. Then *Dragster* sequentially identifies the bottleneck operator and adjusts its configuration to improve the throughput. We implement *Dragster* as a plugin module in clouds. Figure 2 shows its system architecture.

3.1 Dragster on Kubernetes

Flink 1.10 can run on the Kubernetes platform as a Flink session cluster containing three basic components: i) a Flink JobManager pod; ii) a deployment of Flink TaskManager pods and iii) a service exposing the JobManager's REST and UI ports. Each Flink TaskManager is packaged as a Kubernetes pod that can provide one slot to execute one Flink task. A Flink application is realized as a DAG of operators and each operator can be parallelized as multiple concurrent tasks across different TaskManager pods.

We use Kubernetes Horizontal Pod Autoscaler (HPA)[5] and Vertical Pod Autoscaler (VPA)[7] to dynamically adjust the resource configuration of TaskManager pods. Then re-allocate those pods to each operator via Flink checkpoint [13]. Because Flink checkpoint takes around 30s to stop-and-resume, fork, or update an application, *Dragster* adjusts the configuration every 10 mins in the current implementation. The experiments in Section 6 presents, even through such checkpoint mechanism may sacrifice 5% processing time, it can achieve 5X-6X improvement in application throughput. *Dragster* can also take advantage of a faster, more dynamic reconfiguration mechanism, such as *Cameo*[31], to perform at shorter time intervals.

3.2 System Architecture

In Figure 2, we illustrate the system implementation of *Dragster* which has three major components as below:

Database: Database contains the list of candidate configurations and stores the history information, including the resource configuration, throughput, capacity, and resource utilization of each operator with timestamps.

Optimization Engine: Optimization Engine is an implementation of *Dragster*. Based on the job history information, it can identify the bottleneck operator and adjust its configuration to improve the application throughput. The detailed design will state in Section 4 and Section 5.

- Optimization Engine uses PyTorch autograd[4] to do automatic differentiation to identify bottleneck operators.
- Optimization Engine implements the extended GP-UCB algorithm via the Python sklearn package to search the optimal configuration for bottleneck operators.

Job Monitor: When a Flink application is running on Kubernetes, Job Monitor communicates with Flink JobManager and Kubernetes Metrics Server to monitor the application:

- Flink JobManager provides the monitoring REST API [1]. Job Monitor can send an HTTP request to get the operator's status, locality, input throughput, and output throughput.
- Kubernetes Metrics Server [2] collects resource metrics from Kubelet and exposes them in the Kubernetes API server.

We can also apply *Dragster* in Storm and Heron to adjust the number of executors for each Bolt via *rebalancing*[25]. However, *Dragster* is not suitable for Spark Streaming. Spark Streaming processes streams as a series of small-batch tasks. *ExecutorAllocationManager*[26] schedules these tasks to all the executors. Executors do not have a fixed processing logic and can not fit into our model.

4 OPTIMIZATION FRAMEWORK UNDER DRAGSTER

In this section, we first present the basic paradigm of stream processing systems. Then, address the dynamic resource allocation problem in a two-level online optimization framework.

4.1 Modeling a Stream Processing Application with a DAG

Stream processing applications can process incoming data in real-time via a data stream graph that provides a logical view of the data transformation. In Figure 3, we take the data stream graph of the Yahoo streaming benchmark as an example. The processing node is called component and stream processing systems contain three basic components: source, operator, and sink. A source reads data from external message queues and emits an unbounded number of tuples further downstream. An operator can consume tuples from other components, do some processes, and emit new streams for further processing. A sink receives tuples and writes the results into the HDFS or Redis. Edges indicate how tuples are passed around. Each component can be parallelized with multiple tasks containing the same processing logic but running at different physical machines. We use throughput as the evaluation metric, i.e., the number of tuples emitted over a period of time. Without loss of generality, we assume there exists only one sink. And the throughput of the sink is the application throughput. If there are multiple sinks in the application, we can add a virtual sink to fix the situation.

Formally, we consider an application containing N sources indexed from 1 to N, and M operators indexed from $(N+1)$ to $(N+M)$. Each operator can have multiple predecessors and multiple successors. The predecessors of operator i form a predecessor set, i.e., $P_i = \{p_i^j | \ p_i^j \text{ is a predecessor of operator } i\}$ and the successors form a successor set, i.e., $S_i = \{s_j^i | \ s_j^i \text{ is a successor of operator } i\}$. We use e_j^i to measure the throughput emitted from operator i to operator j. Furthermore, let $\overrightarrow{\boldsymbol{e}_j} = \{e_j^i\}_i$ denote the throughput vector received by operator j. If operator i has unlimited service capacity, i.e., can process all the received tuples, the throughput function $h_{i,j} | \ \overrightarrow{\boldsymbol{e}_i} \rightarrow e_j^i$ captures the input to output throughput mapping. It can be expressed by:

$$e_j^i = h_{i,j}(\overrightarrow{\boldsymbol{e}_i}). \tag{1}$$

In this work, we assume the throughput function $h_{i,j}$ is an increasing and concave function, which can be either known beforehand or learned along with running the application. In particular, when developing an application, the developer could understand the logic of an operator and exactly provide its throughput function $h_{i,j}$. Meanwhile, for a user who does not know the logic of an operator, he can also provide an arbitrary concave function, e.g., hyperbolic tangent function, as an initial starting point and learn its parameters via regression in an online manner. Below are some possible forms of the throughput function:

$$h_{i,j}(\overrightarrow{\boldsymbol{e}_i}) = \overrightarrow{\boldsymbol{k}_{i,j}} \cdot \overrightarrow{\boldsymbol{e}_i}, \tag{2a}$$

$$h_{i,j}(\overrightarrow{\boldsymbol{e}_i}) = \min(\overrightarrow{\boldsymbol{k}_{i,j}} \circ \overrightarrow{\boldsymbol{e}_i}), \tag{2b}$$

$$h_{i,j}(\overrightarrow{\boldsymbol{e}_i}) = k_{i,j}^1 \cdot \tanh(\overrightarrow{\boldsymbol{k}_{i,j}^2} \cdot \overrightarrow{\boldsymbol{e}_i}), \tag{2c}$$

where $\cdot$ represents inner product (i.e., $(a_1, b_1) \cdot (a_2, b_2) = a_1a_2 + b_1b_2$), while $\circ$ represents dot product (i.e., $(a_1, b_1) \circ (a_2, b_2) = (a_1a_2, b_1b_2)$). (2a) states the output throughput is linearly increasing with a constant rate vector $\overrightarrow{\boldsymbol{k}_{i,j}}$, while (2b) states the output throughput depends on the bottleneck predecessor with a weight of $\overrightarrow{\boldsymbol{k}_{i,j}}$. In addition, the throughput function can be any concave and increasing function, such as the hyperbolic tangent function in (2c). The throughput function can be generalized as follows:

$$e_j^i = h_{i,j}(\overrightarrow{\boldsymbol{e}_i}, \ \overrightarrow{\boldsymbol{k}_{i,j}}), \tag{3}$$

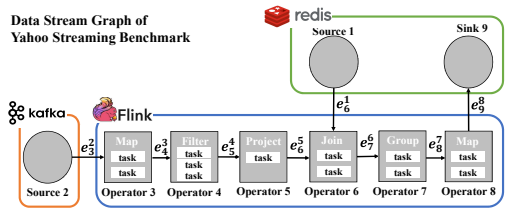


Figure 3: Data stream graph of Yahoo streaming benchmark.

where $\overrightarrow{k_{i,j}}$ is the parameter of the throughput function.

In reality, the computing resource limits the service capacity (i.e., maximal processing rate). As such, the output throughput of operator i is truncated by its service capacity y_i as follows:

$$e_j^i = \min\left(\alpha_{i,j} y_i,\ h_{i,j}(\overrightarrow{e_i}, \overrightarrow{k_{i,j}})\right), \tag{4}$$

where $\{\alpha_{i,j} | \sum_{j\in S_i} \alpha_{i,j} = 1\}$ is the weight for splitting capacity among successors. In the meanwhile, we define the service capacity vector $\boldsymbol{y}_t = \{y_i(t)\}_i$.

4.2 Dynamic Resource Allocation Problem with Buffers

In this part, we address the dynamic resource allocation problem in a two-level online optimization problem. First, how to identify the bottleneck operator. Second, how to adjust the resource configuration of the bottleneck operator.

4.2.1 Identify the Bottleneck Operator. Since the throughput of incoming source rate is time-varying, we formulate the application throughput function as a time-varying function $f_t(\boldsymbol{y}_t)$. It can be obtained by applying (4) multiple times. As the composition of a group of monotonic increasing, concave functions is still a concave function, $f_t(\boldsymbol{y}_t)$ is concave. The objective of our system model is to maximize the accumulated application throughput over time T:

$$\max_{\boldsymbol{y}_t} \sum_{t=1}^{T} f_t(\boldsymbol{y}_t). \tag{5}$$

In stream processing systems, unprocessed tuples are stored in the buffer which may lead to latency and data loss. Therefore, we add a long-term constraint (6) to ensure that each operator has enough service capacity to process the accumulated received tuples.

$$\sum_{t=1}^{T} \left(\sum_{j\in S_i} h_{i,j}(\overrightarrow{e_i}) - y_i(t) \right) \le 0,\ \forall i. \tag{6}$$

Since we only know the objective function $f_t(\boldsymbol{y}_t)$ one-slot later, i.e., the current source rate is unknown, no online algorithms can obtain the optimal service capacity $\boldsymbol{y}_t^*$. At each time slot, we want to adjust the service capacity $\boldsymbol{y}_t$ to track the optimum $\boldsymbol{y}_t^*$ closely, where we define the operator required to adjust its service capacity as a bottleneck operator.

4.2.2 Adjust the Configuration of the Bottleneck Operator. The service capacity of an operator can not be set directly since it depends on its resource configuration vector $\boldsymbol{x}_i$ (e.g., the number of executors, CPU cores, and memory size) in an unknown manner. Therefore, we adopt a Bayesian optimization framework that yields a small number of runs to search the optimal configuration for a bottleneck operator to achieve the target capacity. In our model, each operator follows an independent Bayesian optimization process. For operator i, we model its service capacity y_i under the configuration $\boldsymbol{x}_i$ sampled from a Gaussian Process as (7). We sequentially select configurations to track the target capacity $y_i(t)$.

Table 1: List of Notation.

Symbol	Definition
Decision Variable:	
$\boldsymbol{x}_i(t)$	Resource config. of operator i at time-slot t.
$\boldsymbol{x}_t$	Resource config. vector, i.e., $\boldsymbol{x}_t = \{\boldsymbol{x}_i(t)\}_i$.
Parameter:	
e_j^i	Throughput from operator i to operator j.
$\overrightarrow{e_i}$	Throughput vector received by operator i.
$\overrightarrow{k_{i,j}}$	Parameter of throughput function $h_{i,j}$.
$\alpha_{i,j}$	Capacity splitting weight from operator i to j.
$\boldsymbol{B}$	The overall computing resource budget.
Gaussian Process Model:	
$y_i \sim GP(\mu_i(\boldsymbol{x}_i), k_i(\boldsymbol{x}, \boldsymbol{x}_i))$.	
Dependent Variable and Function:	
$y_i(\boldsymbol{x}_i(t))$	Capacity of operator i running config. $\boldsymbol{x}_i(t)$.
$y_i(t)$	$y_i(t)$ is an abbreviation of $y_i(\boldsymbol{x}_i(t))$.
$\boldsymbol{y}_t$	Capacity vector, i.e., $\boldsymbol{y}_t = \{y_i(t)\}_i$.
$cpu_i(\boldsymbol{x}_i(t))$	CPU utilization of operator i on config. $\boldsymbol{x}_i(t)$.
$c_i(t)$	Observed noise sample of operator i's capacity.
$h_{i,j}(\overrightarrow{e_i})$	Input to output throughput mapping.
$f_t(\boldsymbol{y}_t)$	Application throughput under capacity $\boldsymbol{y}_t$.
$l_i(t)$	Soft-constraint of operator i for buffer size.

$$y_i \sim GP(\mu_i(\boldsymbol{x}_i), k_i(\boldsymbol{x}, \boldsymbol{x}_i)),\ \ \forall\, i. \tag{7}$$

Meanwhile, the service capacity can neither be observed directly, even after the configuration $\boldsymbol{x}_i(t)$ has already launched. It is reported that the throughput of an operator linearly increases with its CPU utilization [16][10] and upper bounded by its service capacity under a given configuration. In this sense, we use (8) to estimate the service capacity $y_i(t)$.

$$c_i(t) = \sum_{j\in S_i} e_j^i / cpu_i\big(\boldsymbol{x}_i(t)\big),\ \ \forall\, i. \tag{8}$$

where $cpu_i\big(\boldsymbol{x}_i(t)\big)$ is the CPU utilization. And the observed capacity $c_i(t)$ is a noise sample of $y_i(t)$ perturbed by a Gaussian noise, i.e., $c_i(t) = y_i(t) + \epsilon_i(t)$, where $\epsilon_i(t) \sim N(0, \sigma^2)$. Based on the historical observation, we can derive the posterior distribution of the Gaussian Process to estimate the mean and variance of the service capacity for not-yet observed configurations. After this, we select the current time-slot configuration to track the target capacity $y_i(t)$ and deploy it in the stream processing system.

4.2.3 Summarize in a Two-level Online Optimization Framework. In summary, we use an online optimization framework to identify the bottleneck operator based on the DAG information. Then, use a Bayesian optimization framework to search the optimal configuration for a bottleneck operator. It results in the following two-level

online optimization problem OPT1.

$$\max_{\{\boldsymbol{x}_i(t)\}} \quad \sum_{t=1}^{T} f_t(\boldsymbol{y}_t(\boldsymbol{x}_t)), \tag{9a}$$

$$s.t. \quad \sum_{t=1}^{T}\Big(\sum_{j\in S_i} h_{i,j}(\overrightarrow{\boldsymbol{e}_i}) - y_i(t)\Big) \le 0,\ \forall i, \tag{9b}$$

$$y_i \sim GP(\mu_i(\boldsymbol{x}_i), k_i(\boldsymbol{x}, \boldsymbol{x}_i)),\ \forall i, \tag{9c}$$

$$\sum_{i>N} \boldsymbol{x}_i(t) \le \boldsymbol{B},\ \forall t. \tag{9d}$$

where $\boldsymbol{x}_i(t)$ is the decision variable to maximize the application throughput $f_t(\boldsymbol{y}_t(\boldsymbol{x}_t))$. Constraint (9b) states that operator i can process all the received tuples. Constraint (9c) models the service capacity y_i sampled from a Gaussian process. Constraint (9d) states that overall allocated resources cannot exceed a budget $\boldsymbol{B}$. Algorithm 1 exhibits its pseudo-code.

Algorithm 1 Two-level Online Optimization Framework

Input: $h_{i,j}(\overrightarrow{\boldsymbol{e}_i}, \overrightarrow{\boldsymbol{k}_{i,j}})$: the throughput function;
$\alpha_{i,j}$: initial capacity splitting weight;
$\boldsymbol{x}_i(1)$: initial resource configuration;
Output: resource configuration $\boldsymbol{x}_i(t)$ for each time-slot;

1: Use the configuration $\boldsymbol{x}_i(1)$ to launch the application;
2: **repeat** for time-slot $t = 2, 3, \cdots$
3: Observe the application throughput $f_{t-1}(\boldsymbol{y}_{t-1})$, operator's throughput e^i_j, and operator's capacity c_i via (8);
4: Identify the bottleneck operator and compute its target service capacity $y_i(t)$.
5: Select and deploy the current time-slot configuration $\boldsymbol{x}_i(t)$ to track the target $y_i(t)$ for bottleneck operators;
6: **until** stop the application;

4.2.4 Performance evaluation. We adopt dynamic regret to evaluate the performance. Dynamic regret [8] is a commonly-used metric in the online optimization and multi-armed bandit literature. It captures the accumulated difference between the throughput achieved under the online solution $\boldsymbol{y}_t(\boldsymbol{x}_t)$ and that under the optimum $\boldsymbol{y}^*_t$. The existence of regret is due to the lack of knowledge about $f_t(\boldsymbol{y}_t)$ and $y_i(\boldsymbol{x}_i)$ beforehand. It is defined as follows:

$$\mathrm{Reg}_T = \sum_{t=1}^{T} f_t(\boldsymbol{y}^*_t) - \sum_{t=1}^{T} f_t(\boldsymbol{y}_t(\boldsymbol{x}_t)). \tag{10}$$

The long-term constraint (9b) states that operator i can process all the received tuples over time T. The long-term constraint is intractable in the online optimization literature, we need to separate it into each time slot and treat them as soft-constraints as follows:

$$l_i\big(y_i(t)\big) = \sum_{j\in S_i} h_{i,j}(\overrightarrow{\boldsymbol{e}_i}) - y_i(\boldsymbol{x}_i(t)) \le 0,\ \forall i, t. \tag{11}$$

Here, Constraint (11) is a soft-constraint that can be violated temporarily during some time-slots, as long as the accumulated constraint violations are under control. In other words, an operator does not need to process all the received tuples strictly at each time slot, as long as it can handle the accumulated received tuples. Following [23], we use the notion of dynamic fit to measure the accumulated amount of soft-constraint violations:

$$\mathrm{Fit}_T = \sum_{t=1}^{T}\sum_{i>N} l_i\big(y_i(\boldsymbol{x}_i(t))\big). \tag{12}$$

In this model, Fit_T gives an upper bound for the number of unprocessed tuples. We proceed to design a dynamic resource allocation scheme that can achieve both sub-linearly increasing dynamic regret and sub-linearly increasing dynamic fit w.r.t time. The dynamic regret guarantees the high throughput of *Dragster*. And the dynamic fit implies the upper-bounded buffer size and the low latency in stream processing systems.

5 OPTIMIZATION ALGORITHM DESIGN UNDER DRAGSTER

In this section, we present design details of *Dragster*. At a high level, it mainly includes two parts, i) applying the online saddle point/ gradient descent algorithm to identify the bottleneck operator and ii) adopting the extended Gaussian-Process UCB algorithm to adjust the configuration for a bottleneck operator.

5.1 Algorithm Design

First, *Dragster* handles the online optimization problem to determine the current time-slot target capacity $\boldsymbol{y}_t$ via the online saddle point algorithm. Let $\lambda_i(t)$ be the multiplier associated with operator i and $\boldsymbol{\lambda}_t = \{\lambda_i(t)\}_i$. The per-slot Lagrangian function for OPT1 can be expressed as:

$$L_t(\boldsymbol{y}_t, \boldsymbol{\lambda}_t) = f_t(\boldsymbol{y}_t) - \sum_i \lambda_i(t)\cdot l_i\big(y_i(t)\big). \tag{13}$$

The online saddle point algorithm works as follows: Given the dual variable $\boldsymbol{\lambda}_{t-1}$, the current time-slot capacity vector $\boldsymbol{y}_t$ is the optimum of the last time-slot Lagrangian function:

$$\boldsymbol{y}_t = \arg\max_{\boldsymbol{y}} L_{t-1}(\boldsymbol{y}, \boldsymbol{\lambda}_{t-1}). \tag{14}$$

The dual variable updates in the form of (15), where γ is the step-size to be designed in the sequel.

$$\lambda_i(t) = \max\big\{0, \lambda_i(t-1) + \gamma\cdot l_i\big(y_i(t)\big)\big\}. \tag{15}$$

Besides the online saddle point algorithm, we can also use other online optimization algorithms, such as the online gradient descent algorithm to adjust the service capacity:

$$y_i(t) = y_i(t-1) + \eta\cdot dL_{t-1}(\boldsymbol{y}_{t-1}, \boldsymbol{\lambda}_{t-1})/dy_i. \tag{16}$$

Second, *Dragster* handles the Bayesian optimization problem to adjust the configuration $\boldsymbol{x}_i(t)$ of the bottleneck operator via the extended Gaussian-Process UCB algorithm to achieve the target service capacity $y_i(t)$. Here, we model the service capacity y_i sampled from a Gaussian process for each operator. A $GP(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'}))$ is specified by its mean function $\mu(\boldsymbol{x}) = \mathbb{E}[y(\boldsymbol{x})]$ and covariance (or kernel) function $k(\boldsymbol{x}, \boldsymbol{x'}) = \mathbb{E}[(y(\boldsymbol{x}) - \mu(\boldsymbol{x}))(y(\boldsymbol{x'}) - \mu(\boldsymbol{x'}))]$. A major advantage of *GP* is the existence of an analytic formula for mean and covariance of the posterior distribution. For noisy samples $\boldsymbol{y}_t = \{y_1, y_2, \cdots, y_t\}$ at points $\boldsymbol{A}_t = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_t\}$, the posterior over y is still a GP distribution, with mean $u_t(\boldsymbol{x})$ and variance $\sigma^2_t(\boldsymbol{x})$ given by:

$$\begin{aligned} \mu_t(\boldsymbol{x}) &= \boldsymbol{k}_t(\boldsymbol{x})^T(\boldsymbol{K}_t + \sigma^2 I)^{-1}\boldsymbol{y}_t, \\ k_t(\boldsymbol{x}, \boldsymbol{x'}) &= k(\boldsymbol{x}, \boldsymbol{x'}) - \boldsymbol{k}_t(\boldsymbol{x})^T(\boldsymbol{K}_t + \sigma^2 I)^{-1}\boldsymbol{k}_t(\boldsymbol{x'}), \\ \sigma^2_t(\boldsymbol{x}) &= k_t(\boldsymbol{x}, \boldsymbol{x}). \end{aligned} \tag{17}$$

where $\boldsymbol{k}_t(\boldsymbol{x}) = [k(\boldsymbol{x}_1, \boldsymbol{x}), k(\boldsymbol{x}_2, \boldsymbol{x}), \cdots, k(\boldsymbol{x}_t, \boldsymbol{x})]^T$ and $\boldsymbol{K}_t = [k(\boldsymbol{x}, \boldsymbol{x'})]_{\boldsymbol{x}, \boldsymbol{x'}\in \boldsymbol{A}_t}$. In *Dragster*, we adopt the squared exponential

kernel [15]. Define $\boldsymbol{x} = \{\boldsymbol{x}_i\}_i$, $\boldsymbol{\mu}_t(\boldsymbol{x}) = \{\mu_t^i(\boldsymbol{x}_i)\}_i$ and $\boldsymbol{\sigma}_t(\boldsymbol{x}) = \{\sigma_t^i(\boldsymbol{x}_i)\}_i$. We adopt the extended Gaussian-Process UCB algorithm to adjust the current time-slot configuration to be the optimum of the following optimization problem:

$$\boldsymbol{x}_t = \Pi_{\mathcal{X}}[\arg\max_{\boldsymbol{x}} \ -|\boldsymbol{\mu}_{t-1}(\boldsymbol{x}) - \boldsymbol{y}_t| + \beta_{t-1}\boldsymbol{\sigma}_{t-1}^2(\boldsymbol{x})]. \tag{18}$$

where $\Pi_{\mathcal{X}}[\boldsymbol{x}]$ is the projection of $\boldsymbol{x}$ onto the set $\{\boldsymbol{x}| \ \sum_i \boldsymbol{x}_i \le \boldsymbol{B}\}$ and $\beta_t = 2\log(|\mathcal{X}|t^2\pi^2\delta/6)$ is the UCB weight with $\delta \in (1, \infty)$. The UCB weight β_t provides a trade-off between exploitation (i.e., $\mu_t(\boldsymbol{x})$) and exploration (i.e., $\sigma_t(\boldsymbol{x})$). The basic idea behind the UCB algorithm is to explore the configurations more often if they are promising or not well-explored. Algorithm 2 is the pseudo-code.

Algorithm 2 Dragster with Online Saddle Point Algorithm

Input: $h_{i,j}(\overrightarrow{\boldsymbol{e_i}}, \overrightarrow{\boldsymbol{k}_{i,j}})$: the throughput function;
Output: resource configuration $\boldsymbol{x}_i(t)$ for each time-slot;
1: Use the configuration $\boldsymbol{x}_i(1)$ to launch the application;
2: **repeat** for time-slot $t = 2, 3, \cdots$
3: Observe and store application throughput $f_{t-1}(\boldsymbol{y}_{t-1})$, operator's throughput $e_{i,j}$ and capacity c_i via (8);
4: Use (15) to update $\boldsymbol{\lambda}_t$ and use the online saddle point algorithm (14) to compute the target capacity $y_i(t)$;
5: Use (17) to update the posterior distribution of the $\boldsymbol{y}_t$ to get $\boldsymbol{\mu}_{t-1}(\boldsymbol{x}_t)$ and $\boldsymbol{\sigma}_{t-1}(\boldsymbol{x}_t)$;
6: Select and deploy configuration $\boldsymbol{x}_i(t)$ as (18).
7: **until** stop the application;

Remark 1. *The conventional UCB algorithm [27] uses* $\boldsymbol{x} = \arg\max_{\boldsymbol{x}\in\mathcal{X}} \ \mu_{t-1}(\boldsymbol{x}) + \beta_{t-1}\sigma_{t-1}^2(\boldsymbol{x})$ *to search the maximum. However, in our setting, we extend the UCB algorithm to find the configuration which can track the capacity closest to* y_t*. i.e., just have enough capacity to handle the incoming tuples. In this sense, the objective function is changed to* $\boldsymbol{x} = \arg\max_{\boldsymbol{x}\in\mathcal{X}} \ -|\mu_{t-1}(\boldsymbol{x}) - y_t| + \beta_{t-1}\sigma_{t-1}^2(\boldsymbol{x})$.

5.2 Performance Analysis

In this part, we proceed to show that under *Dragster*, the dynamic regret defined in (10) and the dynamic fit defined in (12) are both sub-linearly increasing. Before formally presenting the result, we assume that the following conditions are satisfied.

Assumption 1. *(Slater's Condition) There exists a constant* $\varepsilon > 0$*, and an interior point* $\tilde{y}_i(t)$ *such that* $l_i(\tilde{y}_i) = \sum_{j\in S_i} h_{i,j}(\overrightarrow{\boldsymbol{e_i}}) - y_i(\boldsymbol{x}_i(t)) \le -\varepsilon, \forall i > N$.

Assumption 2. *The accumulated optimum variance of the objective function* $f_t(\boldsymbol{y})$*(i.e.,* $V(\boldsymbol{y_t^*}) = \sum_{t=1}^{T} ||\boldsymbol{y}_{t+1}^* - \boldsymbol{y}_t^*||$*) is upper bounded.*

Assumption 1 is the Slater's condition in online optimization literature, which guarantees the existence of a bounded dual variable $\boldsymbol{\lambda}_t$. In *Dragster*, Assumption 1 states that there exists a configuration that can strictly process all the arrived tuples at each time slot. Given enough computing resources, Assumption 1 can always be satisfied. To achieve a sub-linearly regret, one has to impose regularity constraints on the objective function [8][29]. Assumption 2 is a typical constraint to guarantee the dynamic environment changes slow enough to contain a sub-linearly dynamic regret.

Under Assumptions 1-2, we present Theorem 1 to capture the theoretical performance of *Dragster* with the operator throughput function $h_{i,j}(\overrightarrow{\boldsymbol{e_i}}, \overrightarrow{\boldsymbol{k}_{i,j}})$ known beforehand.

Theorem 1. *Let* $\delta \in (1, \infty), \gamma = 1/\sqrt{t}$ *and* $\beta_t = 2\log(|\mathcal{X}|t^2\pi^2\delta/6)$. *Dragster with online saddle point algorithm can obtain a sub-linear dynamic regret and dynamic fit with probability* $(1 - 1/\delta)$*. Precisely,*

$$\Pr\Big\{Fit_T \le M^{2/3}H(1 + H/2\varepsilon) + H\sqrt{T}/\varepsilon + M\sqrt{8T\beta_T\Gamma_T/\log(1+\sigma^{-2})}\Big\} \ge 1 - \frac{1}{\delta}. \tag{19}$$

$$\Pr\Big\{Reg_T \le \sqrt{T}(G^2/2 + V(\boldsymbol{y_t^*})) + H(M + (2 + MH)/2\varepsilon)\cdot Fit_T + GM\sqrt{8T\beta_T\Gamma_T/\log(1+\sigma^{-2})}\Big\} \ge 1 - \frac{1}{\delta}. \tag{20}$$

where M *is the number of operators,* d *is the dimension of the configuration* $\boldsymbol{x}_i$*,* H *is the upper bound of the throughput function (i.e.,* $h_{i,j}(\overrightarrow{\boldsymbol{e_i}}, \overrightarrow{\boldsymbol{k}_{i,j}}) \le H$*),* G *is the upper bound of the objective function's gradient (i.e.,*$|df_t(\boldsymbol{y})/dy_i| \le G$*), and* $\Gamma_T = O((\log T)^{d+1})$ *is the maximum information gain of the squared exponential kernel.*

Proof. In summary, the existence of the regret is due to the lack of knowledge of the capacity function $y_i(\boldsymbol{x}_i)$ and throughput function $f_t(\boldsymbol{y})$ beforehand. For the capacity function $y_i(\boldsymbol{x}_i)$, we use the extended Gaussian Process UCB algorithm in (18) to analyze the regret leading to the term $O(\sqrt{T\beta_T\Gamma_T})$ in (19) and (20). For the throughput function $f_t(\boldsymbol{y})$, we use the online saddle point algorithm to track the optimum leading to the term $O(\sqrt{T})$ in (19) and (20).

First of all, we want to show $\{|y_i(\boldsymbol{x}_i) - \mu_{t-1}^i(\boldsymbol{x}_i)| \le \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i), \ \forall \boldsymbol{x}_i, \ \forall t > 1\}$ satisfied with $\ge (1 - 1/\delta)$ probability. If $r \sim N(0,1)$ is drawn from a Gaussian distribution, then $Pr\{r \ge c\} \le (1/2)e^{-c^2/2}$. Because $y_i(\boldsymbol{x}_i) \sim N(\mu_{t-1}^i(\boldsymbol{x}_i), (\sigma_{t-1}^i(\boldsymbol{x}_i))^2)$ is also drawn from a Gaussian distribution, then:

$$Pr\{|y_i(\boldsymbol{x}_i) - \mu_{t-1}^i(\boldsymbol{x}_i)| \ge \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i)\} \le exp(-\beta_t/2).$$

With $\beta_t = 2\log(|\mathcal{X}|t^2\pi^2\delta/6)$ and $\sum(1/t^2) = 6/\pi^2$, we can get $\{|y_i(\boldsymbol{x}_i) - \mu_{t-1}^i(\boldsymbol{x}_i)| \le \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i), \forall \boldsymbol{x}_i, \forall t > 1\}$ holds with $\ge (1 - \frac{1}{\delta})$ probability.

Secondly, we want to show that the difference between the target service capacity $\boldsymbol{y}_t$ solved by (14) and $\boldsymbol{y}(\boldsymbol{x}(t))$ achieved by *Dragster* has an upper bound. Because $\boldsymbol{x}_i(t)$ is the optimum of (18) and $\boldsymbol{x}_i^*$ is a feasible solution, then:

$$\begin{aligned} &-|\mu_{t-1}^i(\boldsymbol{x}_i(t)) - y_i(t)| + \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i(t)) \\ \ge &-|\mu_{t-1}^i(\boldsymbol{x}_i^*) - y_i(t)| + \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i^*). \end{aligned} \tag{21}$$

$\boldsymbol{x}_i^*$ is the optimum of OPT1 which can achieve the optimal capacity $y_i(t)$, i.e., $\{y_i(t) = \mu_{t-1}^i(\boldsymbol{x}_i^*)\}$. (21) can be transformed as follows:

$$|\mu_{t-1}^i(\boldsymbol{x}_i(t)) - y_i(t)| \le \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i(t)). \tag{22}$$

As operator capacity is positive, then $\{y_i(\boldsymbol{x}_i(t)), \ y_i(t) \ge 0\}$. Because $\sigma_{t-1}(\boldsymbol{x}_i^*(t))$ takes squared exponential kernel, then $\{\beta_t^{1/2}\sigma_{t-1}(\boldsymbol{x}_i^*(t)) \ge 0\}$. If $\{|y_i(\boldsymbol{x}_i) - \mu_{t-1}^i(\boldsymbol{x}_i)| \le \beta_t^{1/2}\sigma_{t-1}^i(\boldsymbol{x}_i)\}$

holds, (22) can be transformed as follows:

$$|\mu^i_{t-1}(\boldsymbol{x}_i(t)) + \beta_t^{1/2}\sigma^i_{t-1}(\boldsymbol{x}_i(t)) - y_i(t)| \le 2\beta_t^{1/2}\sigma^i_{t-1}(\boldsymbol{x}_i(t)).$$
$$|y_i(\boldsymbol{x}_i(t)) - y_i(t)| \le 2\beta_t^{1/2}\sigma^i_{t-1}(\boldsymbol{x}_i(t)).$$

By *Cauchy-Schwarz* inequality, the difference between the $y_i(\boldsymbol{x}_i(t))$ and $y_i(t)$ is upper bounded by:

$$|y_i(\boldsymbol{x}_i(t)) - y_i(t)|^2 \le 4\beta_t\left(\sigma^i_{t-1}(\boldsymbol{x}_i(t))\right)^2. \tag{23}$$

Thirdly, we want to prove $\sum_{t=1}^{T}\left(\sigma^i_{t-1}(\boldsymbol{x}_i(t))\right)^2$ in (23) is bounded by $2\Gamma_T/\log(1+\sigma^{-2})$ via analyzing the *maximum information gain* [14]. *Maximum information gain* comes from Bayesian Experimental Design, where the informativeness of a set of sampling points $A \subset \mathcal{X}$ for y is measured by the information gain defined as the mutual information between y and observations $\boldsymbol{c}_A = \boldsymbol{y}_A + \boldsymbol{\epsilon}_A$:

$$I(\boldsymbol{c}_A; y) = H(\boldsymbol{c}_A) - H(\boldsymbol{c}_A|y).$$

It measures the reduction in uncertainty about y from observing $\boldsymbol{c}_A$. The problem of finding the maximum information gain $\max_{|A|\le T} I(\boldsymbol{c}_A; y)$ is NP-hard. If the covariance function $k(\boldsymbol{x}, \boldsymbol{x'})$ follows the squared exponential kernel, it can be bounded by $\Gamma_T = O((\log T)^{d+1})$ [15], where d is the dimension of the configuration $\boldsymbol{x}_i$. In our setting, we observe the capacity function $y_i(\boldsymbol{x}_i)$ at points $\{\boldsymbol{c}_i(t)\}_t$ after T rounds. Following [19], we can prove that:

$$\sum_{t=1}^{T}\left(\sigma^i_{t-1}(\boldsymbol{x}_i(t))\right)^2 \le 2\Gamma_T/\log(1+\sigma^{-2}). \tag{24}$$

Fourthly, we want to show the dual variable $||\boldsymbol{\lambda}_t||$ is upper bounded by $(\gamma MH + (2H + \gamma MH^2)/2\varepsilon)$. Following (15):

$$||\boldsymbol{\lambda}_{t+1}||^2 = \sum\nolimits_{i>N}\left(\max\{0, \lambda_i(t) + \gamma \cdot l_i\big(y_i(t+1)\big)\}\right)^2.$$

Rearranging terms, it follows that:

$$\begin{aligned} &||\boldsymbol{\lambda}_{t+1}||^2 - ||\boldsymbol{\lambda}_t||^2 \\ &\le 2\gamma\sum\nolimits_{i>N}\lambda_i(t)\cdot l_i\big(y_i(t+1)\big) + \gamma^2\sum\nolimits_{i>N} l_i^2\big(y_i(t+1)\big). \end{aligned} \tag{25}$$

We assume the Slater's condition $l_i\big(\tilde{y}_i(t+1)\big) \le -\varepsilon$ in Assumption 1, where $\tilde{\boldsymbol{y}}_{t+1}$ is a feasible solution and $\boldsymbol{y}_{t+1}$ is the optimum of (14). As such $L_t(\tilde{\boldsymbol{y}}_{t+1}, \boldsymbol{\lambda}_t) \le L_t(\boldsymbol{y}_{t+1}, \boldsymbol{\lambda}_t)$. It follows that:

$$\sum\nolimits_i \lambda_i(t)\cdot l_i\big(y_i(t+1)\big) \le H - \varepsilon||\boldsymbol{\lambda}_t||. \tag{26}$$

We proceed to bound $\sum_{i>N} l_i^2\big(y_i(t+1)\big)$ as follows:

$$\sum\nolimits_{i>N} l_i^2\big(y_i(t+1)\big) <= MH^2. \tag{27}$$

Combining (25), (26)and (27) yields:

$$||\boldsymbol{\lambda}_{t+1}||^2 - ||\boldsymbol{\lambda}_t||^2 \le \gamma(2H + \gamma MH^2 - 2\varepsilon||\boldsymbol{\lambda}_t||). \tag{28}$$

Following (28), we claim that:

$$||\boldsymbol{\lambda}_t|| \le \gamma MH + (2H + \gamma MH^2)/2\varepsilon, \tag{29}$$

as otherwise, if (t_1+1) is the first time that $\boldsymbol{\lambda}_{t_1+1}$ violates (29). Due to triangle inequality and (15), it follows that:

$$||\boldsymbol{\lambda}_{t_1}|| \ge ||\boldsymbol{\lambda}_{t_1+1}|| - ||\boldsymbol{\lambda}_{t_1+1} - \boldsymbol{\lambda}_{t_1}|| \ge (2H + \gamma MH^2)/2\varepsilon \tag{30}$$

(28) and (30) together imply that $||\boldsymbol{\lambda}_{t_1+1}|| \le ||\boldsymbol{\lambda}_{t_1}||$, which leads to a contradiction. As such, (29) holds.

In the sequel, we shall show Fit_T defined in (12) is upper bounded. Since the update of dual variable $\lambda_i(t)$ is given by:

$$\lambda_i(t) = \max\Big\{0, \lambda_i(t-1) + \gamma\cdot l_i\big(y_i(t)\big)\Big\}.$$

it follows that:

$$\begin{aligned} \mathrm{Fit}_T &= \sum\nolimits_t\sum\nolimits_{i>N} l_i\big(y_i(\boldsymbol{x}_i(t))\big) \\ &\le M^{3/2}H(1+H/2\varepsilon) + H/\gamma\varepsilon + M\sqrt{8T\beta_T\Gamma_T/\log(1+\sigma^{-2})}. \end{aligned}$$

Choosing $\gamma = O(1/\sqrt{T})$, Fit_T is bounded by $O(\sqrt{T\beta_T\Gamma_T})$.

Finally, we present Reg_T defined in (10) is also upper bound. Because $\boldsymbol{y}_{t+1}$ is the optimal solution of $L_t(\boldsymbol{y}, \boldsymbol{\lambda}_t)$, while $\boldsymbol{y}_t^*$ is feasible. Because $L_t(\boldsymbol{y}_t^*, \boldsymbol{\lambda}_t) \le L_t(\boldsymbol{y}_{t+1}, \boldsymbol{\lambda}_t)$, it follows that $f_t(\boldsymbol{y}_t^*) - \sum_i \lambda_i(t) l_i(\boldsymbol{y}_t^*) \le f_t(\boldsymbol{y}_{t+1}) - \sum_i \lambda_i(t) l_i(\boldsymbol{y}_{t+1})$. It follows:

$$\begin{aligned} f_t(\boldsymbol{y}_t^*) - f_t(\boldsymbol{y}_t) &\le f_t(\boldsymbol{y}_{t+1}) - f_t(\boldsymbol{y}_t) + \sum\nolimits_i \lambda_i(t)\left(l_i(\boldsymbol{y}_t^*) - l_i(\boldsymbol{y}_{t+1})\right) \\ &\le G^2/2\eta + \eta||\boldsymbol{y}_{t+1} - \boldsymbol{y}_t||^2/2 + \sum\nolimits_i \lambda_i(t) l_i(\boldsymbol{y}_t^*), \end{aligned}$$

where η is an arbitrary positive constant. We want to present the accumulated $||\boldsymbol{y}_{t+1} - \boldsymbol{y}_t||^2$ is also upper bounded. By interpolating intermediate term $||\boldsymbol{y}_t - \boldsymbol{y}_{t-1}^*||^2$ in $||\boldsymbol{y}_t^* - \boldsymbol{y}_t||^2 - ||\boldsymbol{y}_t^* - \boldsymbol{y}_{t+1}||^2$:

$$\begin{aligned} &||\boldsymbol{y}_t^* - \boldsymbol{y}_t||^2 - ||\boldsymbol{y}_t^* - \boldsymbol{y}_{t+1}||^2 \\ &\le 2|\mathcal{Y}|\cdot||\boldsymbol{y}_t^* - \boldsymbol{y}_{t-1}^*|| + ||\boldsymbol{y}_t - \boldsymbol{y}_{t-1}^*||^2 - ||\boldsymbol{y}_t^* - \boldsymbol{y}_{t+1}||^2. \end{aligned}$$

In this sense, $f_t(\boldsymbol{y}_t^*) - f_t(\boldsymbol{y}_t)$ can be bounded as:

$$\begin{aligned} &f_t(\boldsymbol{y}_t^*) - f_t(\boldsymbol{y}_t) \\ \le& G^2/2\eta + \eta|\mathcal{Y}|\cdot||\boldsymbol{y}_t^* - \boldsymbol{y}_{t-1}^*|| + \sum\nolimits_i \lambda_i(t) l_i(\boldsymbol{y}_{t+1}) \\ &+ \eta(||\boldsymbol{y}_t - \boldsymbol{y}_{t-1}^*||^2 - ||\boldsymbol{y}_t^* - \boldsymbol{y}_{t+1}||^2)/2. \end{aligned}$$

The dynamic regret Reg_T defined in (10) is upper bound by:

$$\begin{aligned} \mathrm{Reg}_T &= \sum\nolimits_t f_t(\boldsymbol{y}_t^*) - \sum\nolimits_t f_t(\boldsymbol{y}_t(\boldsymbol{x}_t)) \\ &\le \frac{TG^2}{2\eta} + \eta|\mathcal{Y}|V(\boldsymbol{y}_t^*) + ||\boldsymbol{\lambda}_t||\cdot\mathrm{Fit}_T + GM\sqrt{\frac{8T\beta_T\Gamma_T}{\log(1+\sigma^{-2})}}. \end{aligned}$$

If we choose $\eta = O(\sqrt{T})$, then the dynamic regret is with the same increasing rate of dynamic fit, i.e.,$\mathrm{Reg}_T \le O(\sqrt{T(\log T)^{d+2}})$. This completes the proof of Theorem 1. □

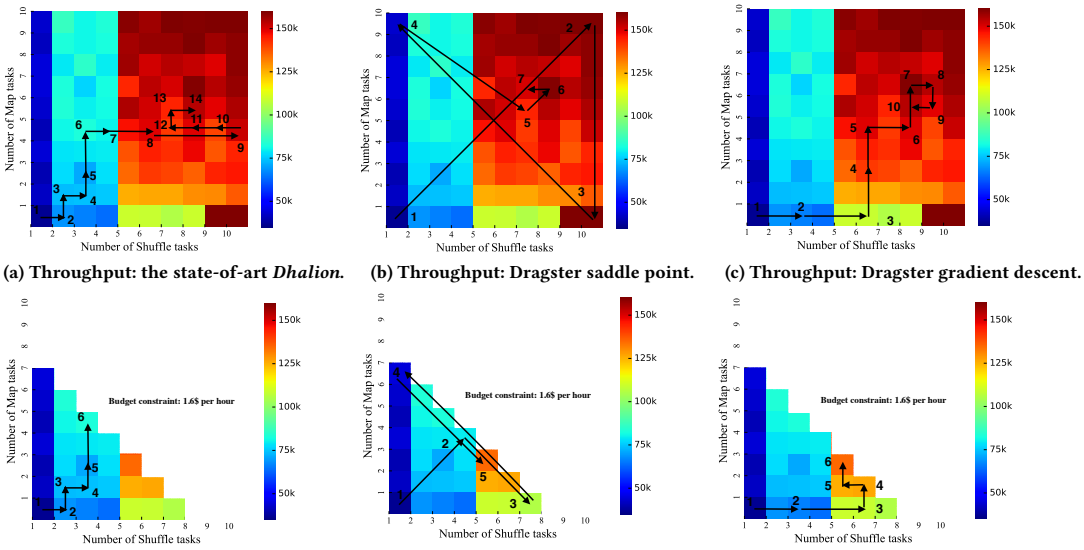
Paralleling Theorem 1, Theorem 2 shows, if the operator throughput function $h_{i,j}(\overrightarrow{\boldsymbol{e}_i}, \overrightarrow{\boldsymbol{k}_{i,j}})$ is unknown, but its prediction is accurate, *Dragster* can still achieve the same order dynamic regret.

Theorem 2. *Under the same setting, running Dragster with the predicted throughput function $h_{i,j}(\overrightarrow{\boldsymbol{e}_i}, \overrightarrow{\boldsymbol{k}_{i,j}})$, when:*

$$\left|h_{i,j}(\overrightarrow{\boldsymbol{e}_i}, \overrightarrow{\boldsymbol{k}_{i,j}}) - h^*_{i,j}(\overrightarrow{\boldsymbol{e}_i})\right| = o(1/\sqrt{T}),\ \forall\ \overrightarrow{\boldsymbol{e}_i}, i, j. \tag{31}$$

where $h^*_{i,j}(\overrightarrow{\boldsymbol{e}_i})$ is the ground-truth, we can obtain the same order of dynamic regret and dynamic fit as Theorem 1.

In summary, the sub-linearly increasing dynamic regret guarantees the high throughput of *Dragster*. And the dynamic fit implies the upper-bounded buffer size and the low latency in stream processing systems.



(a) Throughput: the state-of-art *Dhalion*. (b) Throughput: Dragster saddle point. (c) Throughput: Dragster gradient descent.

(d) Throughput: the state-of-art *Dhalion*. (e) Throughput: Dragster saddle point. (f) Throughput: Dragster gradient descent.

Figure 4: (a)(b)(c) throughput without a budget constraint; (d)(e)(f) throughput under a tight constraint, i.e., 1.6 dollar per hour.

6 EXPERIMENTAL EVALUATION OF DRAGSTER IMPLEMENTATION

In this section, we present experimental results of *Dragster* on our Kubernetes-based implementation. We use *Dragster* to improve the throughput of 11 Apache Flink workloads chosen from the Nexmark benchmark[30] and Yahoo streaming benchmark[25].

6.1 Experimental Setup

Configurations: We use Apache Flink version 1.10 running on Kubernetes version 1.16. Each TaskManger pod can provide one slot with 1 CPU core and 2 GB Memory. Every 10 minutes, Dragster adjusts the number of tasks (i.e., from 1 to 10) for each operator.

Applications: There are 11 applications chosen from the Nexmark and Yahoo streaming benchmark under two source incoming rates. We provide the exact throughput function and capacity splitting weight.

- **Nexmark benchmark**: A diverse set of workloads is chosen from the Nexmark benchmark, including AsynclIO, Join, Window, Group, and WordCount.
- **Yahoo Streaming Benchmark**: It is an advertisement application, identifying relevant events from a number of advertising campaigns and advertisements.

Baseline Algorithms: In our experiments, we compare the following three schemes to show the efficacy of *Dragster*. *Dhalion* is the state-of-art algorithm, which is the most powerful rule-based scheme built-in main-stream stream processing systems and can handle the changing offered loads.

- ***Dhalion*** [16]: *Dhalion* linearly increases the number of tasks for an operator suffering from the backpressure and removes the idle one if its CPU utilization is lower than a threshold.
- ***Dragster* with the online saddle point:** *Dragster* uses the online saddle point algorithm in (14) to directly set the current service capacity as the last time-slot optimum.
- ***Dragster* with the online gradient descent:** *Dragster* adopts the online gradient descent algorithm in (16) to smoothly adjust the capacity for the bottleneck operator.

6.2 How *Dragster* Arrives in the Optimal Configuration

We present the behavior of *Dragster*, how it converges in the optimal configuration via the WordCount example. Figure 4(a,b,c) presents the sequence of chosen resource configurations running WordCount without a budget constraint. We project the search space into two dimensions. The x-axis is the number of Shuffle tasks, while the y-axis is the number of Map tasks. We collect the streaming throughput under 10 x 10 candidate configurations where the configuration with brighter color achieves higher throughput.

In Figure 4(a), the state-of-art algorithm *Dhalion* linearly increases the number of tasks for an operator suffering from backpressure, which leads to backward searching at the 9th time slot and convergence at the 14th time slot. *Dragster* with the online saddle algorithm directly sets the current time-slot capacity as the last time-slot optimum in Figure 4(b). After dramatically jumping among the search space to estimate the shape of the capacity model in the first 4 time slots, it quickly converges to the near-optimal (i.e., within 10% of the optimal throughput) configuration within 7 time slots. In Figure 4(c), *Dragster* with online gradient descent algorithm smoothly adjusts the resource configuration for the bottleneck operator. Thanks to the Gaussian-Process model capturing the relationship between the capacity and configuration, it can avoid the backward appearing in *Dhalion*.

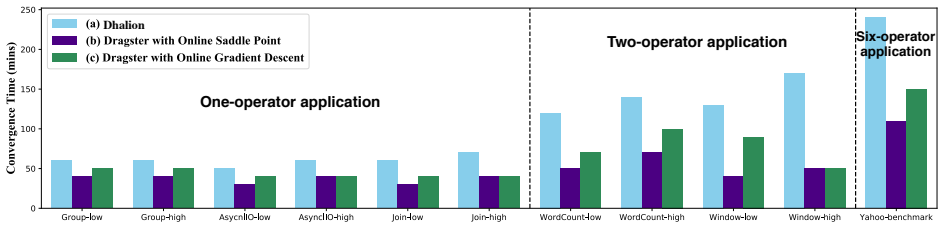


Figure 5: Convergence time under a diverse set of workloads chosen from Nexmark and Yahoo Streaming Benchmarks.

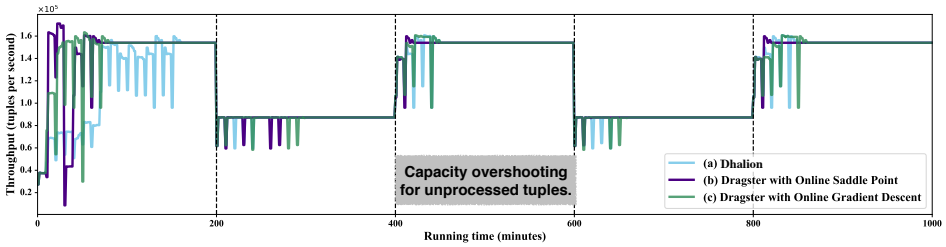


Figure 6: Streaming throughput under workload changes every 200 minutes running WordCount.

When users submit an application to a cloud, they inevitably need to consider the budget. In Figure 4(d,e,f), we present the behavior of *Dragster*, how it improves the throughput of a WordCound application under a tight budget constraint, i.e., 1.6 dollar per hour. In Figure 4(d), at each time slot, *Dhalion* selects one operator to adjust its configuration to reduce the backpressure. Without considering the DAG information, *Dhalion* converges to a non-optimal configuration at the 6th time slot. To be specific, the Map operator receives enough computing resources to process the offered workload, while the Shuffle operator does not receive enough resources and still suffers from heavy backpressure. *Dragster* takes the DAG information to balance the capacity among Map and Shuffle. In this case, two *Dragster* algorithms can converge in the optimal configuration leading to 64.7% higher throughput compared to *Dhalion*.

6.3 Performance Evaluation under a Diverse Set of Workloads

We illustrate the convergence time achieved by three schemes under a diverse set of workloads in Figure 5. Among these workloads, Group, AsynclIO, and Join have one operator, while Window and WordCount have two. Yahoo streaming benchmark has six operators and has more detailed discussions in Section 6.5.

In Figure 5, we sort the order of workloads based on the number of operators. With the increasing number of operators, i.e., a more complex application, the convergence time of *Dhalion* increases much faster than two *Dragster* algorithms. On average, *Dragster* with the online saddle point algorithm can arrive in the optimum for one-operator applications within 35 minutes, which can translate to 1.64X-speedup compared to *Dhalion*. Meanwhile, it can also converge within 52.5 minutes and achieve 2.67X-speedup in the two-operator case. *Dragster* with the online gradient descent algorithm can obtain 1.38X-speedup in the one-operator case and 1.81X-speedup in the two-operator case. For Yahoo streaming benchmark, two *Dragster* algorithms can converge in the optimal configuration within 110 and 140 minutes running time, which is 2.2X and 1.6X speedup compared to *Dhalion*.

6.4 Tracking Workload Changes

Long-running stream processing applications inevitably face gradual/ unexpected workload changes that threaten their stability. The dynamic allocation strategy must react to external shocks by appropriately scaling up or scaling down its resource allocation, especially some workload changes appearing on a regular basis. To highlight *Dragster's* capability to handle workload changes, we submit WordCount and scale up/ down its incoming rate without notifying systems every 200 minutes. The streaming throughput achieved by three schemes is shown in Figure 6. The y-axis is the throughput and the x-axis is the running time. Every 10 minutes, throughput curves in Figure 6 temporarily decrease a lot, since Flink stops processing tuples while adjusting the configuration. After adjusting is completed, the throughput comes back to the normal level. In these experiments, even though such stop-adjust-resume mechanism sacrifices 5% running time, it can achieve 5X-6X throughput improvement.Table 2 presents the convergence time, the number of processed tuples, and cost per billion tuples.

Decreasing the workload at 200 and 600 minutes running time, *Dhalion* and *Dragster* start to scale down the resource allocation. *Dhalion*, the blue curve, removes the idle task for an operator and stops this process as long as the CPU utilization constraint is satisfied. Meanwhile, *Dragster* wants to adjust the capacity to meet the input rate. In this sense, *Dragster* can further decrease the resource allocation and converge in a more economical resource configuration. Table 2 states *Dragster* can achieve 14.6%-15.6% cost-savings to process the same number of tuples compared to *Dhalion*. Increasing the workload at 0, 400, and 800 minutes running time, *Dhalion* and *Dragster* start to scale up the resource allocation. *Dragster* is an online learning-based algorithm that can learn from history to

Table 2: Convergence time, no. of processed tuples, and cost per billion tuples under workload changes running WordCount.

Running time (min)	0-200	200-400	400-600	600-800	800-1k
Offered workload	high	low	high	low	high
Convergence time: Dhalion (min)	140	**40**	40	40	40
Convergence time: Dragster saddle point (min)	**70**	70	**10**	**10**	**10**
Convergence time: Dragster online gradient (min)	80	90	40	40	40
# of processed tuples: Dhalion (10^9)	1.40	1.03	1.81	1.03	1.81
# of processed tuples: Dragster saddle point (10^9)	1.68	1.03	**1.83**	**1.04**	**1.83**
# of processed tuples: Dragster online gradient (10^9)	**1.70**	1.03	**1.83**	1.03	**1.83**
Cost per billion tuples: Dhalion ($)	**50.5**	78.3	51.2	78.3	51.2
Cost per billion tuples: Dragster saddle point ($)	53.2	**66.8**	**50.3**	**65.4**	**50.3**
Cost per billion tuples: Dragster online gradient ($)	52.4	71.8	51.0	67.6	**50.3**

capture the relationship between the capacity and the configuration. In this sense, it can quickly converge with workload changes, since the fluctuation happens many times before. However, *Dhalion* is rule-based, which may suffer from the same search process. In this case, when the workload increases again, *Dragster* with the online saddle point algorithm can only take 10 minutes to achieve the optimal configuration (i.e., directly converge to the optimum), while *Dhalion* always takes 40 minutes to do so.

6.5 Yahoo Streaming Benchmark

We illustrate that *Dragster* can handle a more complex streaming application, Yahoo streaming benchmark. There are six operators in the application, comprising one million candidate configurations. In this sense, exhaustively searching the optimum is impractical.

Figure 7 shows the streaming throughput achieved by three schemes running the Yahoo streaming benchmark. We scale up the input rate at 300 minutes running time without notifying the system. The y-axis is the throughput, while the x-axis is the running time. The violet curve shows that *Dragster* with the online saddle point algorithm can converge within 110 minutes, while *Dhalion* needs around 240 minutes as the blue curve. In this sense, *Dragster* performs even better in a more complex streaming application, a 2.2X-speedup (i.e., more than the 2X-speedup in WordCount application) to converge in the optimal configuration compared to *Dhalion*. When increasing the workload at 300 minutes running time, *Dragster* with online saddle point algorithm takes only 30 minutes to find out the optimal configuration, while *Dhalion* takes 90 minutes to do so. In addition, *Dragster* can process 11.2%-14.9% more tuples running the Yahoo streaming benchmark within 300 minutes shown in Table 3. *Dragster* can also converge in a more economical configuration and achieve 4.2% cost-savings to process the same number of tuples as *Dhalion*.

Table 3: Convergence time, tuple processing rate, and cost per billion tuples running Yahoo benchmark in 300 minutes.

	Dhalion	Dragster saddle pt.	Dragster online gd.
Convergence time (min)	240	**110**	150
Proc. rate b4 conv. (10^5/s)	1.93	2.15	**2.22**
Cost per billion tuples($)	120.4	**115.8**	**115.8**

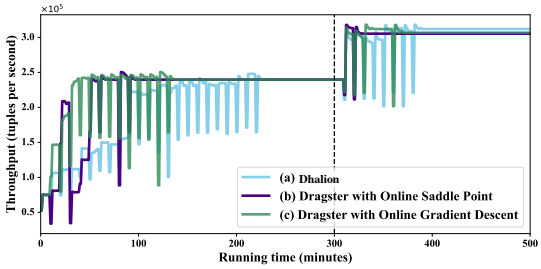


Figure 7: Streaming throughput running Yahoo benchmark.

7 CONCLUSION

In this paper, we design, analyze, and implement *Dragster*, an online-optimization-based dynamic resource allocation scheme for elastic stream processing. It is based on the solid mathematical foundation, i.e., sub-linearly increasing dynamic regret of throughput and upper-bounded buffer size. The sub-linearly increasing regret guarantees the high throughput of *Dragster*, while the upper-bounded buffer size results in the low latency in stream processing systems. More importantly, *Dragster* is implemented on the Kubernetes platform to improve the throughput of Apache Flink applications. Via running the Nexmark benchmark and Yahoo streaming benchmark, *Dragster* can achieve 1.8X-2.2X speedup to converge to a near-optimal configuration compared to the state-of-the-art algorithm, *Dhalion*. It can contribute to 20.0%-25.8% gain in tuple-processing goodput and 14.6%-15.6% cost-savings.

REFERENCES

[1] 2011. Flink REST API. https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/rest_api.html.
[2] 2014. Kubernetes Metrics Server. https://github.com/kubernetes-sigs/metrics-server.
[3] 2015. Spark Dynamic Allocation. https://jaceklaskowski.gitbooks.io/master-ing-apache-spark/spark-dynamic-allocation.html.
[4] 2016. PyTorch autograd: automatic differentiation. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.
[5] 2019. Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[6] 2019. Overcommiting CPUs one sole-tenant VMs. https://cloud.google.com/compute/docs/nodes/overcommitting-cpus-sole-tenant-vms.
[7] 2019. Vertical Pod Autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler.
[8] Mokhtari A, Shahrampour S, Jadbabaie A, and et al. 2016. Online optimization in dynamic environments: Improved regret rates for strongly convex problems[C].

In *Decision and Control (CDC)*. 7195–7201.

[9] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In *NSDI*, Vol. 2. 4–2.

[10] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*. 207–218.

[11] Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*. 189–200.

[12] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[13] Checkpoint. 2011. Flink Savepoints. https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html.

[14] TM Cover and JA Thomas. 1991. Elements of Information Theory,(pp 33-36) John Wiley and Sons. *Inc, NY* (1991).

[15] Varsha Dani, Thomas P Hayes, and Sham M Kakade. 2008. Stochastic linear optimization under bandit feedback. (2008).

[16] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[17] Pooyan Jamshidi and Giuliano Casale. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 39–48.

[18] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*. 783–798.

[19] Andreas Krause and Cheng S Ong. 2011. Contextual gaussian process bandit optimization. In *NIPS*. 2447–2455.

[20] Kuberntes. 2014. Kubernetes. https://kubernetes.io.

[21] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 165–176.

[22] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2020. Accordia: Adaptive Cloud Configuration Optimization for Recurring Data-Intensive Applications. In *ICDCS 2020*.

[23] Mahdavi M, Jin R, and Yang T. 2012. Trading regret for efficiency: online convex optimization with long term constraints[J]. In *Journal of Machine Learning Research*. 2503–2538.

[24] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 805–825.

[25] rebalance. 2014. Yahoo Streaming benchmark. https://storm.apache.org/releases/current/Understanding-the-parallelism-of-a-Storm-topology.html.

[26] Spark. 2011. Spark ExecutorAllocationManager. https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/ExecutorAllocationManager.scala.

[27] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *ICML*.

[28] Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee U Khan, and Keqin Li. 2015. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences* 319 (2015), 92–112.

[29] Yang T, Zhang L, Jin R, and et al. 2016. Tracking slowly moving clairvoyant: Optimal dynamic regret of online learning with true and noisy gradient[C]. In *ICML*. 449–457.

[30] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[31] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2020. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. *arXiv preprint arXiv:2010.03035* (2020).

[32] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 196–205.