High-throughput Real-time Edge Stream Processing with Topology-Aware Resource Matching

Peng Kang*, Samee U. Khan†, Xiaobo Zhou§ and Palden Lama*
*The University of Texas at San Antonio
†Mississippi State University
§The University of Colorado Colorado Springs

***Abstract*—With the proliferation of Internet of Things (IoT) devices, real-time stream processing at the edge of the network has gained significant attention. However, edge stream processing systems face substantial challenges due to the heterogeneity and constraints of computational and network resources and the intricacies of multi-tenant application hosting. An optimized placement strategy for edge application topology becomes crucial to leverage the advantages offered by Edge computing and enhance the throughput and end-to-end latency of data streams. This paper presents Beaver, a resource scheduling framework designed to deploy stream processing topologies across distributed edge nodes efficiently. Its core is a novel scheduler that employs a synergistic integration of graph partitioning within application topologies and a two-sided matching technique to optimize the strategic placement of stream operators. Beaver aims to achieve optimal performance by minimizing bottlenecks in the network, memory, and CPU resources at the edge. We implemented a prototype of Beaver using Apache Storm and Kubernetes orchestration engine and evaluated its performance using an open-source real-time IoT benchmark (RIoTBench). Compared to state-of-the-art techniques, experimental evaluations demonstrate at least $1.6\times$ improvement in the number of tuples processed within a one-second deadline under varying network delay and bandwidth scenarios.**

***Index Terms*—Edge Stream Processing, Topology Placement, Resource Scheduling, Multi-tenancy**

I. Introduction

There are growing interests in processing continuous and time-sensitive data streams generated by Internet of Things (IoT) devices at the edge of the network to meet low latency, privacy, and location-aware computing requirements of modern applications [1]–[3]. The traditional approach of *stream processing* in remote data centers suffers from wide-area network delays, heavy traffic that may overload the network, and jitters caused by untrusted and unpredictable network [4]. As a result, recent works [5], [6] have focused on enabling Edge stream processing by utilizing distributed edge computing nodes, such as IoT gateways, edge routers, regional micro-datacenters, etc. as shown in Figure 1.

A stream processing application is defined by a logical topology of operators connected into a Directed Acyclic Graph (DAG), which processes data streams as they flow from the source to the sink. Multiple such applications can share the edge computing infrastructure to process real-time data streams for quickly deriving insights and making timely decisions. However, delivering optimal performance for these applications requires careful placement of stream processing operators on distributed edge nodes. In contrast to traditional cloud computing scenario, operator placement in edge computing framework is more complex due to the presence of a hierarchy of compute layers, limited network bandwidth between the edge nodes, variable network delays, and heterogeneous hardware. Furthermore, recent studies [7]–[9] have shown that IoT workloads from different application domains can have diverse and highly dynamic characteristics. These factors pose significant challenges for resource scheduling and application deployment in the edge environment.

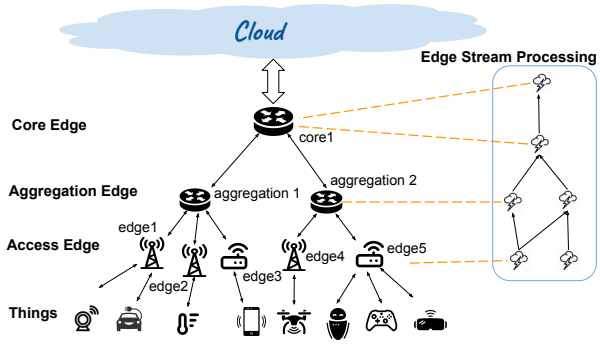


Fig. 1: IoT-Edge-Cloud architecture.

Existing works [10]–[12] that focus on reducing the end-to-end stream processing time employ operator placement strategies that either disregard scenarios involving multiple applications sharing the edge infrastructure or are inherently biased towards applications that are scheduled earlier. Importantly, they lack the system resilience to handle the uncertainties within edge networks effectively. Our case study in Section II using an open-source stream processing engine, Apache Storm and a real-time IoT benchmark (RIoTBench) [13] shows that existing resource scheduling techniques are severely impacted by an increase in network delay between the edge nodes, resulting in a significant drop in application performance. Designing a scheduler that can achieve robust performance for edge stream processing is a key challenge, as it requires minimizing the worst-case resource bottleneck caused by the complex data flow dynamics between stream operators, the variable network delay between edge nodes, and the large number of possible ways to split application topologies across heterogeneous nodes.

To address these challenges, we develop a resource schedul-

ing framework for strategic placement of stream operators on distributed edge nodes. Our key contributions are:

- We develop a resource scheduling framework, `Beaver`, that provides robust performance for stream processing applications on a shared multi-layered edge environment. It achieves consistent performance even in the face of increased network delay and reduced bandwidth. In contrast to prior work [12], which depends on application-level data rate control, our approach is non-intrusive and does not require any modification of user's code.
- We design a novel operator placement algorithm that integrates graph theory-based application topology partitioning and game theoretic two-sided matching techniques to iteratively identify an operator placement strategy that minimizes the bottleneck across CPU, memory, and network resources.
- We design and implement `Beaver` using a representative stream processing engine, Apache Storm, Docker for containerization of stream operators, and Kubernetes orchestration engine.
- We demonstrate `Beaver's` robust performance gains and resource utilization efficiency over Coda [11], Amnis [12], RStorm [10], and default Apache Storm under varying network delays and bandwidth between the edge nodes. Extensive evaluation using an open-source real-time IoT benchmark (RIoTBench) [13] show that `Beaver` outperforms state-of-the-art techniques with at least $1.6\times$ improvement in the number of processed tuples within a one-second deadline.

The rest of the paper is organized as follows: Section II covers the background and motivation, Section III elaborates on the design and implementation, Section IV details the testbed setup and experimental results, Related work is discussed in Section V, and conclusions are presented in Section VI.

II. Background & Motivation

A. Edge Stream Processing

Stream processing is a dominant distributed computing paradigm for processing and analysis of high-volume, heterogeneous, and continuous data streams to extract insights and actionable results in real time. Naturally, there is growing interest in deploying stream-processing engines (SPEs), such as Apache Storm, Apache Flink, etc., on IoT Gateways and edge routers which are close to the end users and IoT devices. These edge nodes have more computing resources than wireless sensor networks but are still limited compared to the cloud data centers. Data streams produced by IoT devices are processed on these edge nodes using a data flow programming model, where each application is packaged as a directed acyclic graph (DAG) data structure, called a topology. Individual data points (tuples) flow through a topology from sources to sinks. The vertices correspond to stream operators, while the edges denote the data flows between these operators. An application's *query latency* is determined by the end-to-end stream processing time. The SPE scheduler maps the operators from the DAG into a physical execution plan comprising multiple execution stages. Each stage can be further subdivided into numerous tasks that can run in parallel.

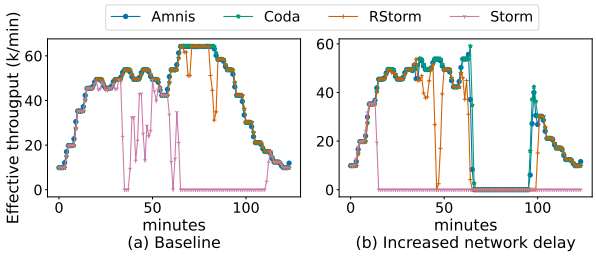


Fig. 2: Impact of increased network delay on the performance of ETL application. State-of-the-art resource scheduling techniques were evaluated under a dynamic workload, while concurrently executing three other stream processing applications.

B. Limitations of Existing Resource Schedulers

The resource schedulers in current SPEs, which are designed for resource-rich cloud data centers may not be well-suited for the resource-constrained edge environment. Even new methods proposed for the edge are insufficient to handle the challenges of geographically distributed edge nodes, such as variable network delays and heterogeneous computing capacity. To demonstrate the limitations of state-of-the-art techniques, we conducted a case study on a testbed of virtual machines (VMs) emulating the edge environment depicted in Fig. 1. The edge network has a tiered structure, encompassing nodes with diverse capacities across the access, aggregation, and core layers. We used Linux TC (traffic control) tool to control the network delay between the edge nodes. The testbed hosted four concurrently running applications from the RIoTBench [13], each facing a dynamic workload. We evaluated the resource scheduling techniques of Amnis [12], Coda [11], RStorm [10], and default Storm by measuring the effective throughput of an ETL (Extraction, Transform, and Load) application. We define effective throughput as the number of tuples processed within a one-second deadline.

Fig. 2 (a) shows the results of a baseline case where the network delay between the access and aggregation edge nodes was set at 15 ms. Similarly, the network delay between the aggregation and core edge nodes was fixed at 50 ms. We observed that Amnis, Coda and RStorm achieve better throughput than the default Storm in the face of dynamic workload. However, when we repeated the experiments while doubling the network delay between each tier of edge nodes, there was a significant drop in the effective throughput for all competing methods, as shown in Fig. 2 (b). Amnis and Coda failed to process the tuples within one second when the workload peaks around 65 minutes time, whereas RStorm and Storm failed even earlier.

C. Challenges of Stream Operator Placement

Designing a robust scheduler for edge stream processing is challenging as it requires minimizing the worst-case resource

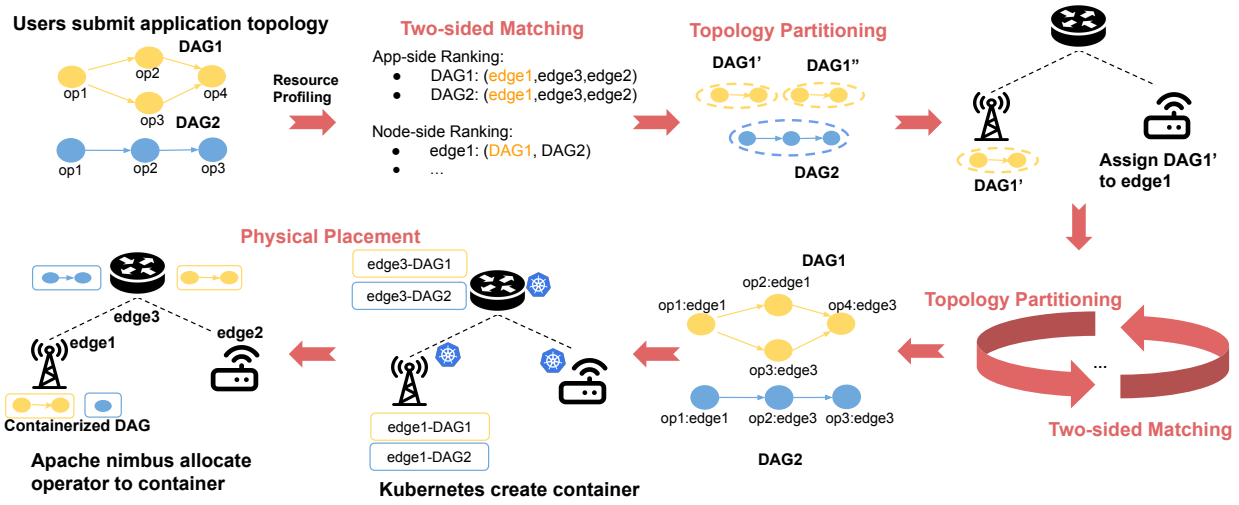


Fig. 3: Stream application execution pipeline in Beaver framework.

bottleneck in the face of complex and dynamic data flow, network delay, and heterogeneous node distribution. Stream operator placement, that minimizes the CPU, memory, and network bottleneck, can be roughly modeled as a minimax optimization problem in this context.

$$
\begin{aligned}
&\min_{\forall j \in \{1,..,m\}} \max\Big(\sum_{1\le k\le K} \frac{X_j^k \times CPU^k}{CPU_j}, \\
&\sum_{1\le k\le K} \frac{X_j^k \times MEM^k}{MEM_j}, \sum_{1\le k\le K} \frac{X_j^k \times DataRate^k}{BW_j} \Big) \\
&\text{s.t.} \sum_{1\le k\le K} \frac{X_j^k \times CPU^k}{CPU_j} \le 1, \forall j \in \{1,..,m\} \\
&\sum_{1\le k\le K} \frac{X_j^k \times MEM^k}{MEM_j} \le 1, \forall j \in \{1,..,m\} \\
&\sum_{1\le k\le K} \frac{X_j^k \times DataRate^k}{BW_j} \le 1, \forall j \in \{1,..,m\} \\
&\sum_{1\le j\le m} X_j^k \le 1, \quad \forall k \in \{1,...,K\}
\end{aligned}
\tag{1}
$$

Here, X_j^i is assigned the value of 1 if node j hosts operator k, and 0 otherwise. $k \in \{1, ..., K\}$ includes all application's operators. The CPU, memory, and network utilization of operator k are denoted as CPU^k, MEM^k, and $DataRate^k$, respectively. The total available CPU, memory, and network bandwidth resources available on node j are denoted as CPU_j, MEM_j, and BW_j respectively. The optimization problem gets further complicated when we consider minimizing network delay between stream operators placed on distributed edge nodes. Thus, a heuristic solution is needed to tackle the complexity of stream operator placement problem.

III. System Design

We present the design and implementation of `Beaver`.

A. Overview

The `Beaver` framework comprises three key components: Resource Profiler, Logical Mapping Generator, and Physical Placement Module. As shown in Fig. 3, the Resource Profiler obtains the baseline resource usage profile of user-provided application topologies (DAGs) on a dedicated edge node. Next, the Logical Mapping Generator employs a novel scheduling algorithm (Algorithm 1) to generate a list of matching pairs that associate each operator group with a corresponding edge node, where each group contains one or more DAG operators. This is achieved by synergistically integrating two-sided matching and application topology partitioning to iteratively identify the optimal operator placement strategy. Finally, the Physical Placement Module deploys each operator group as Docker container on the designated edge node using the Kubernetes Pod scheduler. This module implements a pluggable scheduler for Apache Storm and automatically configures Kubernetes Pod scheduler to realize the physical placement of stream operators.

B. Resource Profiler

The Resource Profiler runs each application on a dedicated edge node and measures the average utilization of CPU, memory, and network for each stream operator using Prometheus [1]. We profile an application using the maximum input data rate based on the user's specification. In this paper, we use VMs with various vCPU and memory configurations to emulate heterogeneous edge nodes. In the future, we will conduct experiments with heterogeneous hardware including Raspberry Pis with different CPU frequencies. Resource profiling can be adapted to handle such scenarios by measuring normalized resource units such as Google Compute Units (GCUs) [14]. An allocation of 1 GCU should provide about the same amount of computational power on any machine.

[1] https://prometheus.io/

C. Logical Mapping Generator

The Logical Mapping Generator aims to provide a stream operator placement strategy that minimizes the worst-case resource bottleneck. It incorporates the game theoretic two-sided matching technique, where the application side attempts to find edge nodes with minimum resource bottleneck to place its stream operators, and the edge infrastructure side attempts to find stream operators with minimum resource demands. However, classical two-sided matching treats each entity independently and ignores the relationship between them [15]. This approach does not ensure optimal performance due to data dependency between stream operators. To address this issue, we introduce a topology-aware two-sided matching algorithm that integrates iterative topology partitioning to optimize operator placement.

*1) **Two-sided matching**:* We model stream operator placement as a many-to-one matching problem using two distinct sets of entities: a) $G = \{g_1, ..., g_i, ...g_n\}$ where g_i is the DAG of application i, which can be further split into groups of stream operators; b) $R = \{r_1, \ldots, r_j, \ldots, r_m\}$ where r_j is a vector of available CPU, memory and network bandwidth resources available on edge node j. The matching theory aims to find the best matches between the two sets of entities based on their mutual preferences [16]. The main idea is to find a stable matching so that no pair of entities would rather be matched with each other than with their assigned matches. The preferences of these entities are determined by the following ranking methods.

Application-side ranking involves evaluating each edge node for DAG g_i using Equation 2 and ranking them based on the metric $l(g_i, r_j)$, which represents the dominant factors affecting the query latency. Lower metric values are preferred.

$$l(g_i, r_j) = \max(\frac{CPU^{g_i}}{CPU_j}, \frac{MEM^{g_i}}{MEM_j}, \sum_{u \in U} \sum_{z \in Z} \frac{DataRate^u}{BW_z}) + \frac{\delta_{qj}}{\delta_{qs}} \quad (2)$$

where CPU^{g_i} and MEM^{g_i} are the CPU and memory utilization of DAG g_i. CPU_j and MEM_j are CPU and memory available on node j. The network utilization in terms of input data rate from an upstream operator is denoted as $DataRate^u$, where the set U represents the upstream operators for g_i. BW_z denotes the network bandwidth available on node z, where the set Z includes all nodes in the path between an upstream operator u and the current DAG g_i. δ_{qj} is the network delay between node j and the farthest node q where an upstream operator is assigned. δ_{qs} is delay between the sink node s and the farthest node q where an upstream operator is assigned.

Node-side ranking involves evaluating each DAG for edge node j using Equation 3 and ranking them based on the minimum residual resource ratio $t(g_i, r_j)$, which represents the fraction of node resource r_j that will be left if DAG g_i is assigned to node j. Higher metric values are preferred.

$$t(g_i, r_j) = \min(\frac{CPU_j - CPU^{g_i}}{CPU_j}, \frac{MEM_j - MEM^{g_i}}{MEM_j}, \frac{BW_j - \sum_{u \in U} DataRate^u}{BW_j}) \quad (3)$$

TABLE I: NOTATIONS USED

Notation	Description
$DAGs$	a list of directed acyclic graphs
g_i	DAG of application i
T	a list of DAGs waiting to enter matching phase
G	a list of DAGs in the matching phase
m	the number of edge nodes
r_j	resource vector of node j
$Assign[j]$	a DAG matched to node j
$Alloc(g_i)$	resource allocation needed by DAG g_i
$t(g_i, r_j)$	$t(g_i, r_j)$ is calculated by Eq. 3
$PrefNodes[g_i]$	a ranked list of nodes based on Eq. 2
$Final[j]$	final list of DAGs matched with node j

*2) **Topology Partitioning**:* We apply topology partitioning when a DAG's resource requirement exceeds the available resource on an assigned edge node, or the node finds a DAG that is ranked higher than the one currently assigned. Let (V, E) be a weighted graph representing an application topology, where V is the set of vertices and E is the set of edges. Let $W = (w_{uv})$ be the weight matrix associated with the graph, where the weight w_{uv} between vertices u and v represents the data flow rate between the corresponding stream operators. We aim to partition the weighted graph into two subsets, V_1 and V_2 such that the cut, $C(V_1, V_2)$ defined as the sum of the weights of the edges between V_1 and V_2, is minimized. By minimizing the cut, we can effectively reduce network traffic and improve overall performance. For this purpose, we employ the spectral partitioning method, which is known to provide high-quality solution by leveraging the properties of the entire graph [17]. Spectral partitioning utilizes the spectral properties of the graph Laplacian matrix to achieve effective partitioning. The Laplacian matrix L is computed as $L = D - A$, where D is the degree matrix and A is the weighted adjacency matrix of the graph. By calculating the eigenvalues and eigenvectors of L, we identify the Fiedler vector, which is the eigenvector associated with the second smallest eigenvalue. This Fiedler vector provides insights into the graph's connectivity facilitating optimal partitioning. Vertices with positive values in the Fiedler vector are assigned to V_1, while those with negative values are assigned to V_2.

*3) **Beaver Scheduling Algorithm**:* Algorithm 1 presents the pseudo-code for our topology-aware two-sided matching algorithm utilized by the Logical Mapping Generator. The notations used in the algorithm are explained in Table I. The algorithm takes a list of application topologies (DAGs) as input and generates a mapping between subsets of the DAGs and edge nodes. To initiate the scheduling process, we employ the `NodeRanking` procedure to rank all the edge nodes for each DAG g_i in the matching list G, and store the ranked list in $PrefNodes[g_i]$ (lines 2-3). Then, the algorithm enters a loop that continues until it finds an appropriate

Algorithm 1: `Beaver` Scheduling Algorithm

Input : $G \leftarrow DAGs$
Output: $\{Final[j] \mid j \in (1, m)\}$

1 $T \leftarrow \varnothing$;$Final \leftarrow \varnothing$;$PrefNodes \leftarrow \varnothing$; $Assign \leftarrow \varnothing$
2 **foreach** $g_i \in G$ **do**
3 $PrefNodes[g_i] \leftarrow$ `NodeRanking`(g_i)
4 **while** $G \neq \varnothing$ **do**
5 $g_i \leftarrow G.pop()$
6 $j \leftarrow PrefNodes[g_i].pop()$
7 **if** $Assign[j] = \varnothing$ **and** $r_j >$ `Alloc`(g_i) **then**
8 $Assign[j] \leftarrow g_i$
9 **Continue**
10 **else if** $r_j >$ `Alloc`(g_i) **and** $t(Assign[j], r_j) < t(g_i, r_j)$ **then**
11 $Assign[j] \leftarrow g_i$
12 $g', g'' \leftarrow$ `AppPartition`$(Assign[j])$
13 **else**
14 $g', g'' \leftarrow$ `AppPartition`(g_i)
15 **if** $t(g', r_j) < t(Assign[j], r_j)$ **then**
16 $g \leftarrow g' \cup g''$
17 $G.append(g)$
18 **Continue**
19 $G.append(g')$
20 $T.append(g'')$
21 $PrefNodes[g'] \leftarrow$ `NodeRanking`(g')
22 **if** $G = \varnothing$ **and** $T \neq \varnothing$ **then**
23 **for** $j = 1$ **to** m **do**
24 $Final[j] \leftarrow Final[j] \cup \{Assign[j]\}$
25 $r_j \leftarrow r_j -$ `Alloc`$(Assign[j])$
26 $Assign[j] \leftarrow \varnothing$
27 $G \leftarrow T$
28 $T \leftarrow \varnothing$
29 **foreach** $g_i \in G$ **do**
30 $PrefNodes[g_i] \leftarrow$ `NodeRanking`(g_i)
31 **return** $\{Final[j] \mid j \in (1, m)\}$

operator placement strategy that satisfies the preferences of all applications (see lines 4-30). On each iteration, a DAG g_i is taken from the list G, and its highest ranking node j is popped from the list $PrefNodes[g_i]$, to establish a possible match. If no other DAG has been assigned to node j and the available resource r_j is sufficient to meet the resource demand $Alloc(g_i)$, then g_i is temporarily assigned to the node (see lines 7-9). If another DAG $Assign[j]$ is already assigned to node j, the algorithm replaces it with g_i if its residual resource ratio $t(Assign[j], r_j)$ is lower than that of g_i, and the node has sufficient resource available. When the DAG fails to be assigned to the node, the `AppPartition` function divides it into two partitions, denoted as g' and g'' (see lines 10-12). If the residual resource ratio of the new DAG g' is lower than that of the assigned DAG $Assign[j]$, the algorithm combines g' and g'' and appends to the matching list G (see lines 15-18). Otherwise, g' and g'' are appended to the matching list G and the waiting list T respectively. In addition, it ranks all edge nodes for the new DAG g'. If the matching list G becomes empty while there are still DAGs in the waiting list T, the DAG assigned to node j is recorded in the final matched list $Final[j]$, and the resource available in the node is updated based on the resource demand $Alloc(Assign[j])$ (see lines 23-25). Next, the DAGs in the waiting list T are moved to the matching list G and the edge nodes are ranked for all DAGs in the list (see lines 29-30). **`NodeRanking` function** ranks each edge node for a given DAG g_i based on the metric $l(g_i, r_j)$ given by Eq. 2. Nodes with lower metric values are ranked higher. **`AppPartition` function** applies the Spectral Partitioning technique described in Section III-C2 to split a given DAG g_i into two partitions, g' and g''.

Complexity Analysis: The worst-case time complexity of the algorithm is $O(m \times n \times k^3)$. Here, m represents the number of nodes within the edge cluster, n signifies the number of stream processing applications, and k indicates the total number of operators in the largest DAG. k^3 is the time complexity of the spectral partitioning technique. Note that this worst-case estimation provides an upper bound; in practice, the algorithm's performance is reasonable.

D. Physical Placement Module

This module is responsible for executing the operator placement strategy determined by Algorithm 1. We assume that the edge nodes are set up as a Kubernetes cluster. Our implementation includes a pluggable scheduler [2] for Apache Storm and automatic configuration of Kubernetes Pod scheduler [3] to enable precise assignment of grouped operators to specific containers, and scheduling of the containers on their designated edge nodes. The placement module configures Kubernetes pod specification for each group of stream operators constituting a DAG partition. The specification includes a container image of Apache Storm and a *nodeName* field that is mapped to the name of the designated edge node. The Kubernetes pod scheduler places the pod on the designated edge node based on this specification. Each pod runs a daemon called the Supervisor, which listens for work assigned to it and starts and stops worker processes as needed. Our placement module configures the Supervisor with a tag labeled as NodeName-DAGName, indicating the mapping of the designated edge node with the DAG partition. The pluggable scheduler assigns stream operators across the pods based on the tags associated with the Supervisor process running on them. We use the terms 'pod' and 'container' interchangeably since we run a single container in a pod.

IV. Evaluation

A. Experimental TestBed

We setup a prototype testbed running the KVM hypervisor to host eight Ubuntu (v16.04) Virtual Machines (VMs) em-

[2]https://storm.apache.org/releases/1.2.4/Storm-Scheduler.html
[3]https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

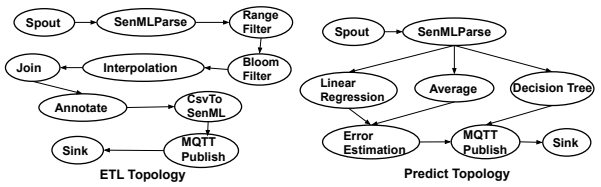


Fig. 4: DAG adapted from RIOTBench.

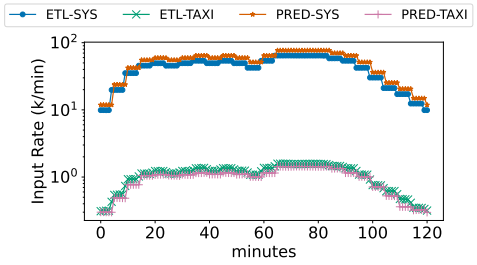
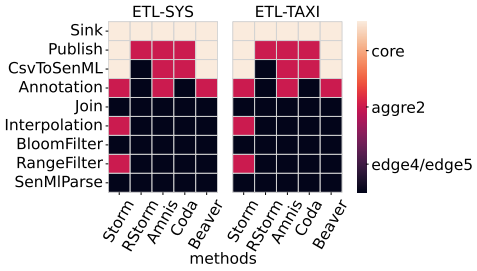
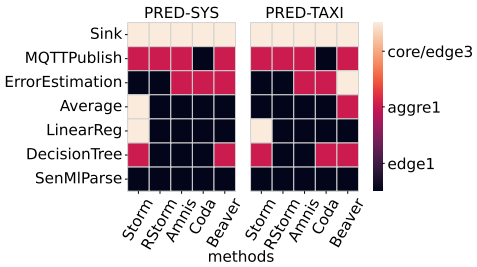


Fig. 5: Input data rate for RIOTBench applications.



(a) ETL-SYS use edge4 node. ETL-TAXI use edge5 node.



(b) PRED-SYS use edge3 node. PRED-TAXI use core node.

Fig. 6: Operator placement at different edge nodes.

ulating five access edge nodes, two aggregation edge nodes, and one core edge node as shown in Fig. 1. Each access edge node was equipped with 2 vCPUs and 4 GB RAM. The aggregation edge node had 4 vCPUs and 8 GB RAM. The core edge node had 8 vCPUs and 16 GB RAM. The network delay and bandwidth between the edge nodes were emulated using Linux's TC tool [4]. The network parameters were chosen based on the previous works [12], [18], [19]. A Kubernetes (Version 1.23.0) cluster using these eight VMs was built for container orchestration and management. Docker (Version 20.10.7) was used as the container runtime engine. Apache Storm (Version 2.1.0) served as the edge stream processing engine. Each Storm supervisor encapsulated by a container had 2 worker slots. We used Prometheus to collect CPU, memory, and network usage data.

1) Datasets: We used two IoT datasets, including Sense your City[5] and NY city taxi trips[6]. The **Sense your City (SYS)** is a real-world Smart Cities data stream collected from sensors deployed across different cities on three continents. Each city has multiple sensors, providing six types of observations every minute: temperature, humidity, ambient light, sound, dust, and air quality. The **New York City taxi trips (TAXI)** data includes smart transportation messages derived from 2 million trips taken in 2013. Each trip record includes pickup and drop-off dates, taxi and license details, start and end coordinates with timestamps, the distance measured by the taximeter, and taxes and tolls paid. For our benchmark runs, we utilized aggregated data from January 2013 [8], [20].

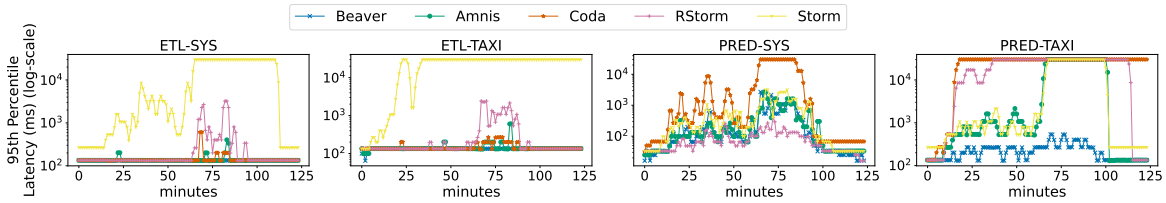
2) Benchmarks: We used the RIoTBench benchmark suite [13], which includes four IoT applications based on common IoT patterns for data pre-processing, statistical summarization, and predictive analytics. For our analysis, we selected two applications: ETL (Extraction, Transform, and Load) and PRED (Predictive Analytics), as illustrated in Fig. 4. The remaining two applications (STATS and TRAIN) were excluded in our study since they are integrated with public cloud services and are unsuitable for low-latency Edge stream processing. Throughout our experiments, we concurrently ran four applications: **PRED-SYS**, **PRED-TAXI**, **ETL-SYS**, and **ETL-TAXI**. To produce a dynamic workload, we modified RIoTBench's input generator. At one-minute intervals, this generator supplied Storm's source task (Spout) with multiple fixed-sized batches (10 tuples) of data. We adjusted the total number of data batches at each interval to generate dynamic input data rates as shown in Figure 5.

B. Evaluation Methodology

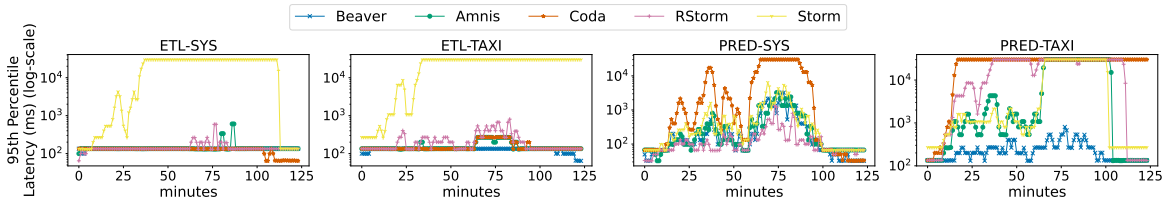
We conducted a comparative evaluation of `Beaver` against four different methods: (i) **Storm**, the default scheduler in Apache Storm that follows a round-robin algorithm, placing operators based on alphabetical sequence onto each worker node. The number of nodes needs to be specified by users. (ii) **RStorm** [10], a greedy method implemented in Apache Storm, designed to maximize CPU utilization and minimize network latency for increased overall throughput. (iii) **Amnis** [12], a state-of-the-art approach that considers data locality and resource constraints during physical plan generation and operator placement for stream queries. (iv) **Coda** [11], an approach that utilizes a stable many-to-one matching algorithm to place the microservice-based topology in a heterogeneous computing environment.

1) Topology Placement: Fig. 6 depicts the operator placement for four applications utilizing the five strategies mentioned above. The operator names can be cross-referenced with Fig. 4. Note that, except for Storm' default scheduler,

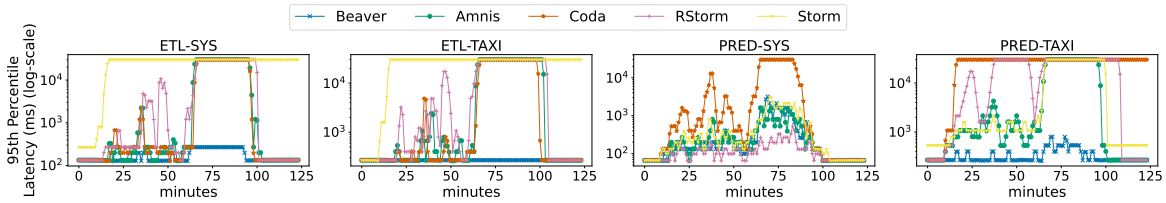
[4]https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html
[5]http://datacanvas.org/sense-your-city/
[6]https://databank.illinois.edu/datasets/IDB-9610843



(a) The base scenario.



(b) The network bandwidth decreased from 100 Mb to 50Mb.



(c) The network delay transitions from (15 ms, 50 ms) to (30 ms, 100 ms).

Fig. 7: The 95th percentile latency.

all other techniques assume that the resource requirements of the operators can be specified by the user or obtained through profiling. For instance, RStorm requires users to provide details about the resource usage of each operator as input parameters. Similarly, Amnis, Beaver, and CODA rely on network topology information within the edge environment.

2) Performance metrics: The assessed metrics include: (i) the **Effective throughput**: the number of tuples processed within a real-time processing deadline (one second in this paper) to generate outputs; (ii) the **95th percentile end-to-end latency**. Here, end-to-end latency is calculated as the difference between the output timestamp and the timestamp of the last input tuple that contributes to the output.

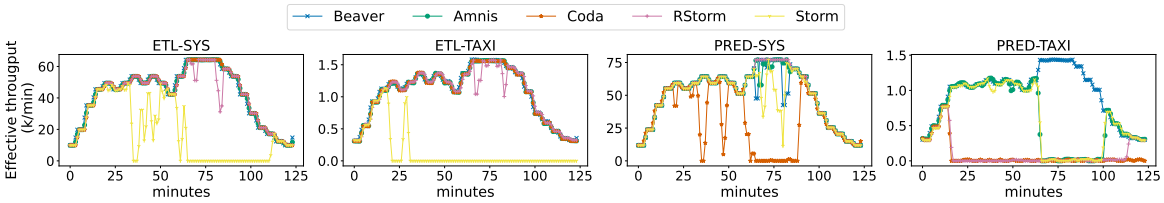
C. Performance Analysis

We performed a series of experiments encompassing diverse network delay and bandwidth scenarios to assess the robustness of `Beaver` in comparison to competing techniques.

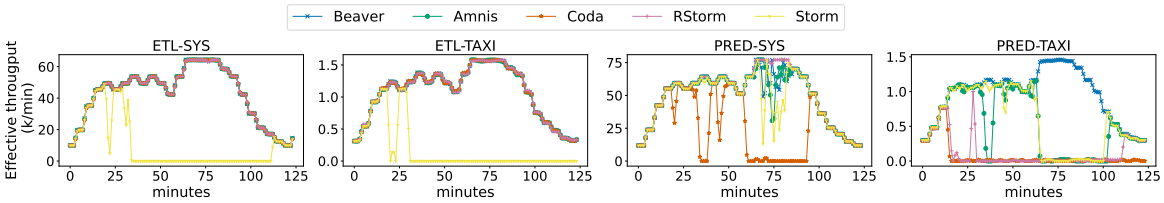
1) Baseline scenario: As a baseline, the network delay between the access and aggregation edge nodes was set at 15 ms, with a bandwidth of 100 Mbps. Similarly, the network delay between the aggregation and core edge nodes was fixed at 50 ms, with a bandwidth of 100 Mbps. Fig. 7a and Fig. 8a show the 95th percentile latency and the effective throughput of the four concurrent applications each facing a dynamic workload. In the case of default Storm, network bandwidth limitation at the aggregation2 node leads to poor tail latencies for ETL-SYS and ETL-TAXI. While Coda faces a CPU bottleneck at Edge1 between 60 to 100 minutes, impacting the latency of PRED-SYS. Furthermore, Amnis, Coda, Storm, and RStorm all experience a CPU bottleneck at Edge2, resulting in subpar latency and low effective throughput for PRED-TAXI. On the other hand, Beaver delivers exceptional performance for all applications.

2) Reduced network bandwidth scenario: We repeated the experiments with reduced network bandwidth (from 100 Mbps to 50 Mbps) between each tier of edge nodes. As depicted in Fig. 7b and Fig. 8b, it becomes evident that ETL-SYS encounters network bandwidth limitation leading to performance degradation earlier than in the baseline case when using the default Storm. Notably, the performance of other methods exhibits consistent behavior, with Beaver surpassing all competing techniques. These findings underscore the pivotal role of more efficient network utilization in optimizing the overall performance of diverse applications.

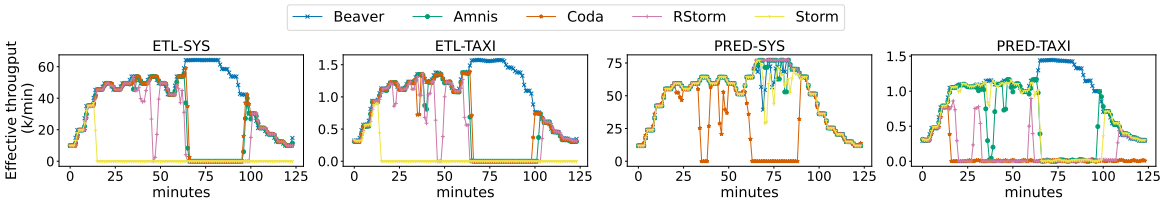
3) Increased network delay scenario: The spatial distribution of edge nodes can introduce significant variations in network delays between any pair of nodes. We repeated the experiments with increased network delay (from 15 ms to 30 ms) between the access and aggregation edge nodes. Similarly, the network delay between the aggregation and core edge nodes was increased from 50 ms to 100 ms. In the cases of Amnis, Coda, and RStorm, ETL-SYS and ETL-TAXI experience extremely poor performance between 60 to 100 minutes as shown in Fig. 7c and Fig. 8c. This is due to the accumulation of data tuples in the outbound message queue of stream operators, which is exacerbated by increased network delay in conjunction with high data arrival rates. In



(a) The base scenario.



(b) The network bandwidth decreased from 100 Mb to 50Mb.



(c) The network delay transitions from (15 ms, 50 ms) to (30 ms, 100 ms).

Fig. 8: Throughput of tuples processed with one second deadline.

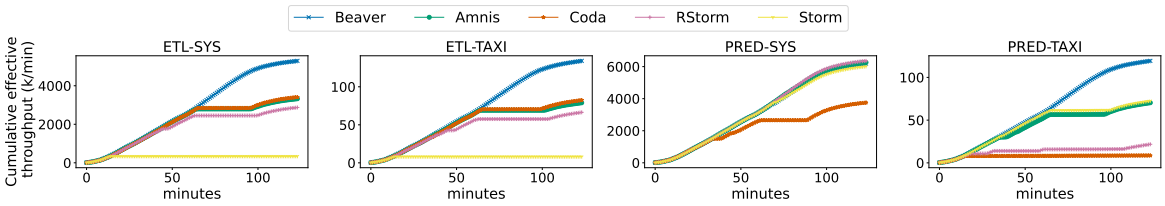


Fig. 9: Cumulative throughput of tuples processed with one second deadline for Fig. 8c.

contrast, PRED-CITY and PRED-TAXI were less affected by these challenges, primarily due to their lower input data rates. Fig. 9 compares the cumulative effective throughput of various methods. Beaver exhibits at least $1.6X$ performance improvement over the state-of-the-art techniques.

D. System Overhead Analysis

We analyzed the resource utilization on each edge node to gain deeper insights into Beaver's overall efficiency in the increased network delay scenario, as depicted in Fig. 10, Fig. 11, and Fig. 12. In the case of Storm, when CPU becomes a bottleneck, there is a rapid increase in memory usage due to the accumulation of intermediate data in memory. Notably, this pattern was observed at Edge2, Edge4, Edge5, and Aggregation2. Simultaneously, Storm exhibits higher network bandwidth consumption at these edge nodes since it tries to distribute operators evenly across the worker nodes. Conversely, for Amnis and RStorm, the bottlenecks were observed at Edge2, with increased CPU and memory utilization leading to significantly higher tail latencies, particularly evident in PRED-TAXI. Coda, on the other hand, encountered resource bottlenecks at Edge1 and Edge2, resulting in poor tail latency for PRED-SYS and PRED-TAXI. Due to high network delay between aggregation2 and core1 (2X higher than baseline case), heightened outbound data queuing at Aggregation2 causes a surge in memory usage while diminishing CPU utilization. This effect cascades to the preceding nodes, Edge4 and Edge5, prompting an increase in their memory usage and a decrease in CPU utilization. This chain reaction ultimately results in significantly elevated latency. Notably, Beaver remains unaffected by the increased network delay due to its minimal data transfer between Aggregation2 and core1 nodes. As a result, Beaver outperformed other methods, significantly mitigating potential resource bottlenecks and contributing to a more effective system overall.

V. Related Work

Edge and fog computing are rapidly evolving fields with various approaches to address the challenges and opportunities of distributed computing. Many efforts focus on

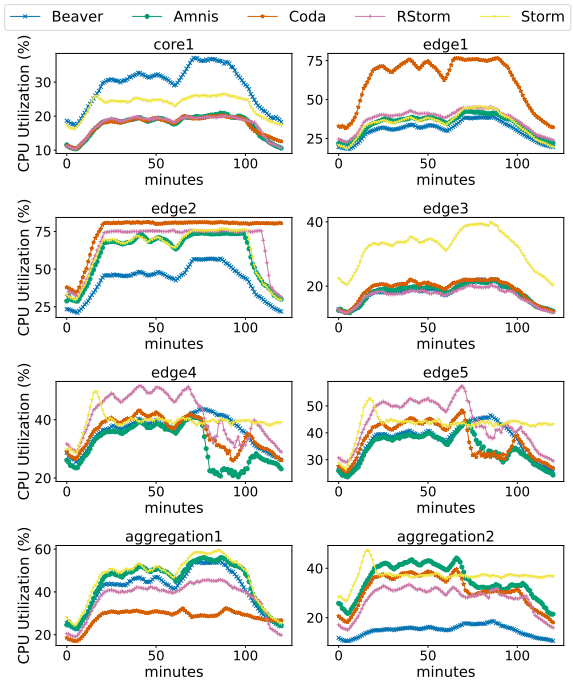


Fig. 10: CPU utilization for each edge node.

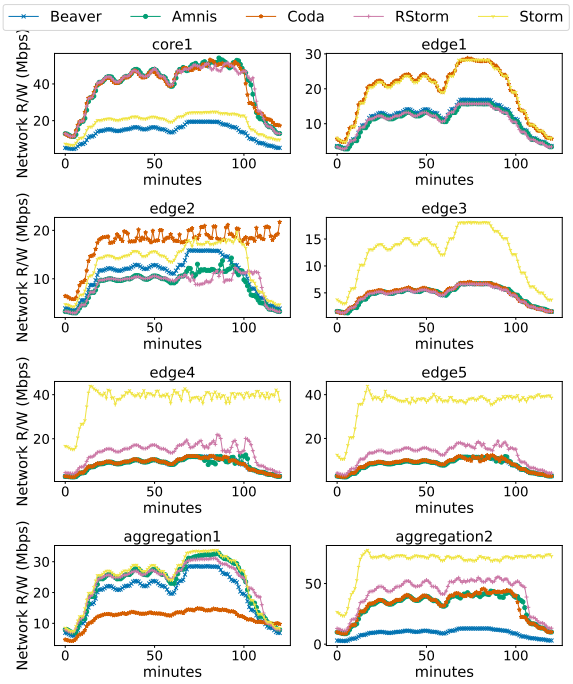


Fig. 12: Network Read/Write usage for each edge node.

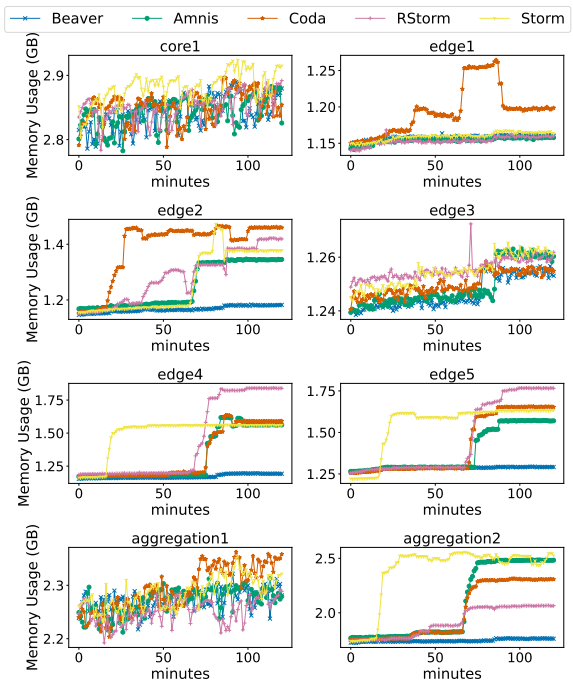


Fig. 11: Memory utilization for each edge node.

reducing access latency and bandwidth consumption while overcoming limitations like limited bandwidth and computing power in edge cloud environments, e.g., Cloudlets [21]. These limitations significantly impact the performance of edge and fog computing systems, requiring researchers and developers to consider them during design and implementation. Previous projects [6], [22] proposed programming abstractions to optimize available resources, but they often overlooked multi-tenant situations and complex real-world applications, leading to limitations in throughput and latency.

Existing industrial stream processing systems (e.g., Flink, Storm, Spark), are designed for low-latency intra-datacenter settings with powerful resources and high network connectivity. They are unsuitable for edge stream processing. Stream processing systems such as AWS IoT Greengrass and Azure IoT Edge aim to improve data processing at the edge without relying on cloud backend connectivity. These systems require a hub device with limited computational capabilities and lack support for distributed data-parallel processing, leading to limited throughput and a single point of failure [23].

Several academic projects have focused on edge stream processing systems. EdgeNet [24] is designed for individual IoT device data processing rather than distributed stream processing. EdgeWise [5] reduces backpressure but lacks scalability due to a centralized bottleneck. Frontier [25] focuses on fault-tolerance but overlooks edge dynamics and heterogeneity. DART [23] aims for scalability and adaptability but neglects resource bottleneck, leading to network congestion. Amnis [12] and Coda [11] encountered challenges in achieving a globally optimized placement and alleviating high queue delays, especially as network delays increased. Amnis also relies on intrusive mechanisms such as application-level data rate control. Another Kubernetes-based method solely considers the network score and a single application scenario to determine the priority of pod placement decisions [26]. However, these works do not handle multi-tenant use cases where multiple applications share the same hardware.

Several techniques have been explored to optimize resource

allocation in the edge-cloud continuum. Wang et al. [27] managed application quality of service through client workload reduction and allocation of cloudlet resources. Araldo et al. [28] implemented a polynomial-time resource allocation algorithm to maximize utility for edge network operators. Angelelli et al. [29] proposed multi-objective optimization-based scheduling of serverless functions on heterogeneous edge-cloud platforms. MicroSplit [30] utilized the Louvain method to partition microservices into the edge and the cloud. However, these works do not address the unique challenges of real-time stream processing in multi-tiered edge infrastructure, which is the focus of this paper.

VI. Conclusion

We designed a resource scheduling framework that incorporates a synergistic integration of application topology partitioning and a two-sided matching technique for optimal placement of stream operators across distributed edge nodes. Our prototype, `Beaver`, was implemented using Apache Storm and orchestrated through Kubernetes. We assessed its performance using RIoTBench, an open-source real-time IoT benchmark. In comparison to state-of-the-art techniques, `Beaver` showcases a minimum of a $1.6\times$ improvement in effective throughput, even in the presence of varying bandwidths and network delay scenarios. These results underscore the efficacy of our approach in mitigating resource bottlenecks across CPU, memory, and network at the edge, ensuring the timely processing of IoT data streams. The substantial performance enhancements achieved by `Beaver` underscore its potential to elevate the efficiency and reliability of real-time IoT applications.

Acknowledgments

This research is supported by National Science Foundation grants CNS 1911012, CNS 2124908, and CCF 2135439. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of National Science Foundation.

References

[1] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, pp. 64–71, 01 2017.

[2] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[3] C. Qin, H. Eichelberger, and K. Schmid, "Enactment of adaptation in data stream processing with latency implications-a systematic literature review," *Information and Software Technology*, 2019.

[4] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, 2009.

[5] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *Proc. of the 2019 Usenix Annual Technical Conference.*, p. 929–945, 2019.

[6] X. Wang, Z. Zhou, P. Han, T. Meng, G. Sun, and J. Zhai, "Edgestream: A stream processing approach for distributed applications on a hierarchical edge-computing system," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 14–27, IEEE, 2020.

[7] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of cloud versus grid workloads," in *2012 IEEE Intl. Conf. on Cluster Computing*, pp. 230–238, 2012.

[8] U. Tadakamalla and D. A. Menascé, "Characterization of iot workloads," in *Edge Computing*, pp. 1–15, Springer Intl. Publishing, 2019.

[9] F. Metzger, T. Hoßfeld, A. Bauer, S. Kounev, and P. E. Heegaard, "Modeling of aggregated iot traffic and its application to an iot cloud," *Proc. of the IEEE*, vol. 107, no. 4, pp. 679–694, 2019.

[10] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proc. of the 16th Annual Middleware Conf.*, 2015.

[11] N. Mehran, D. Kimovski, and R. Prodan, "A two-sided matching model for data stream processing in the cloud–fog continuum," in *2021 IEEE/ACM 21st Intl. Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 514–524, IEEE, 2021.

[12] J. Xu, B. Palanisamy, Q. Wang, H. Ludwig, and S. Gopisetty, "Amnis: Optimized stream processing for edge computing," *Journal of Parallel and Distributed Computing*, vol. 160, pp. 49–64, 2022.

[13] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[14] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *EuroSys'20*, (Heraklion, Crete), 2020.

[15] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, 1962.

[16] S. Bayat, Y. Li, L. Song, and Z. Han, "Matching theory: Applications in wireless communications," *IEEE Signal Processing Magazine*, 2016.

[17] F. McSherry, "Spectral partitioning of random graphs," in *Proc. 42nd IEEE Symposium on Foundations of Computer Science*, 2001.

[18] B. Varghese, E. de Lara, A. Y. Ding, C.-H. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis, "Revisiting the arguments for edge computing research," *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[19] T. Liu, S. He, S. Huang, D. Tsang, L. Tang, J. Mars, and W. Wang, "A benchmarking framework for interactive 3d applications in the cloud," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[20] S. Khare, H. Sun, K. Zhang, J. Gascon-Samson, A. Gokhale, and X. Koutsoukos, "Ensuring low-latency and scalable data dissemination for smart-city applications," in *2018 IEEE/ACM Third Intl. Conf. on Internet-of-Things Design and Implementation (IoTDI)*, 2018.

[21] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the leading edge of mobile-cloud convergence," in *6th Intl. Conf. on Mobile Computing, Applications and Services*, 2014.

[22] L. Huang, Z. Liang, N. Sreekumar, S. Kaushik, A. Chandra, and J. Weissman, "Towards elasticity in heterogeneous edge-dense environments," in *2022 IEEE 42nd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pp. 403–413, 2022.

[23] P. Liu, D. Da Silva, and L. Hu, "{DART}: A scalable and adaptive edge stream processing engine," in *2021 USENIX Annual Technical Conf. (USENIX ATC 21)*, pp. 239–252, 2021.

[24] M. Simioni, P. Gladyshev, B. Habibnia, and P. R. Nunes de Souza, "Monitoring an anonymity network: Toward the deanonymization of hidden services," *Forensic Science International: Digital Investigation*, vol. 38, p. 301135, 2021.

[25] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *Proc. of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[26] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *2022 22nd IEEE Intl. Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[27] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, "Towards scalable edge-native applications," in *Proc. of the 4th ACM/IEEE Symposium on Edge Computing*, p. 152–165, 2019.

[28] A. Araldo, A. D. Stefano, and A. D. Stefano, "Resource allocation for edge computing with multiple tenant configurations," in *Proc. of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[29] L. Angelelli, A. A. da Silva, Y. Georgiou, M. Mercier, G. Mounié, and D. Trystram, "Towards a multi-objective scheduling policy for serverless-based edge-cloud continuum," in *2023 IEEE/ACM 23rd Intl. Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023.

[30] A. Rahmanian, A. Ali-Eldin, B. Skubic, and E. Elmroth, "Microsplit: Efficient splitting of microservices on edge clouds," in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, pp. 252–264, 2022.