Ps-Stream: Proactive Straggler Mitigation Using Spatio-Temporal Features for Elastic Stream Computing Systems

Minghui Wu, Dawei Sun, Shang Gao, Member, IEEE,
Keqin Li, Fellow, IEEE and Rajkumar Buyya, Fellow, IEEE

***Abstract*—Recent stream computing systems have shifted from static, threshold-based straggler detection to adaptive, temporal-pattern-based prediction. Existing prediction approaches typically rely on autoregressive time-series analysis to model each node's historical resource usage within a cluster. However, they often overlook cross-node state correlations, resulting in isolated prediction models that fail to capture the dynamic spatio-temporal dependencies among nodes. To address these limitations, we propose Ps-Stream, a proactive framework for straggler prediction and mitigation. First, Ps-Stream constructs a unified spatio-temporal graph by modeling compute nodes as a spatial graph at each time step and linking each node across adjacent time steps. Second, it learns spatio-temporal representations for straggler prediction by combining Graph Convolutional Networks (GCNs) to capture spatial dependencies with attention mechanisms to model temporal dynamics. Third, it incorporates a fine-grained task migration strategy that proactively reallocates tasks from predicted stragglers to the least-loaded nodes while considering task dependencies and migration costs to minimize migration overhead. Experimental results show that Ps-Stream improves straggler prediction accuracy by up to 26%, reduces maximum system latency by 62%, more than doubles maximum throughput, and improves resource utilization by up to 34% compared with state-of-the-art baselines.**

***Index Terms*—Stream computing systems, straggler prediction, straggler mitigation, graph convolutional network, attention mechanism.**

I. Introduction

DISTRIBUTED stream computing systems serve as a core infrastructure for real-time big-data applications [1], distributing continuous, high-throughput data streams across clusters of compute nodes to meet the millisecond-level responsiveness required by services such as real-time fraud detection, traffic monitoring, and financial trading [2], [3]. However, skewed data distributions, commonly generated by spatio-temporally sensitive services such as hotspot aggregation in geographic sensor networks, can lead to the emergence of straggler nodes that lag behind their peers in clusters [3], [4]. Static schedulers, which cannot adapt to dynamic workload fluctuations, may inadvertently assign compute-intensive tasks to these already resource-constrained nodes, degrading overall system performance [5], [6]. To address this, dynamic online scheduling must be employed to redistribute tasks and mitigate stragglers, thereby maximizing resource utilization across the cluster [7].

Based on the timing of straggler mitigation, dynamic online scheduling approaches can be categorized as either reactive or proactive.

Reactive scheduling has been widely studied to handle stragglers at runtime [8], [9]. These approaches monitor node performance in real time and reconfigure workloads only when overload conditions are detected [10]. Techniques such as dynamic thresholding trigger task migration or resource scaling once a node's resource consumption exceeds a predefined limit [11]–[13]. However, these methods often lack the ability to predict future resource loads, especially when facing uncertainties such as sudden traffic spikes or drastic workload fluctuations [14]. As a result, systems may struggle to react in time, motivating the shift toward proactive scheduling.

Proactive scheduling utilizes historical data and predictive models to anticipate load changes [15], [16], adjusting task deployment before nodes reach critical thresholds [17]–[19]. While proactive strategies have brought notable performance gains in distributed stream processing systems [20], [21], they still face challenges, especially when clusters exhibit strong spatio-temporal interdependencies. In such systems, dozens or even hundreds of compute nodes are tightly coupled: Inter-node data transmissions and collaborative processing create a complex spatial graph [22]–[24]. Over time, this graph evolves, as each node's load at a given time step both influences and is influenced by the load of its neighbors at subsequent steps.

However, many existing studies [8], [11], [15], [16] fail to capture this complexity. They typically model each node's load as an independent time series, neglecting the correlations between nodes and across time. Consequently, their prediction models often struggle to accurately forecast the spatio-temporal evolution of resource loads. Furthermore, when straggler mitigation actions, such as task redistribution, alter the

This work is supported by National Natural Science Foundation of China under Grant No. 62372419; Fundamental Research Funds for the Central Universities under Grant No. 265QZ2021001; and MelbourneChindia Cloud Computing (MC3) Research Network. (Corresponding author: Dawei Sun.)

Minghui Wu and Dawei Sun are with the School of Artificial Intelligence, China University of Geosciences, Beijing, 100083, China (e-mail: wuminghui@email.cugb.edu.cn; sundaweicn@cugb.edu.cn)

Shang Gao is with the School of Information Technology, Deakin University, Waurn Ponds, Victoria, 3216, Australia (e-mail: shang.gao@deakin.edu.au)

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, New York, 12561, USA (e-mail: lik@newpaltz.edu)

Rajkumar Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, the University of Melbourne, Australia (e-mail: rbuyya@unimelb.edu.au)

system's load patterns, new data variations can emerge. These variations may deviate significantly from historical data patterns, thereby undermining the effectiveness of conventional prediction models.

These issues motivate us to develop **Ps-Stream**, a proactive framework for predicting and mitigating stragglers in distributed stream-computing systems. Ps-Stream begins by constructing a spatial graph with either a predefined or data-adaptive adjacency matrix to describe relationships among compute nodes. Graph neural networks (GCNs) [25] are applied to this spatial graph to model correlations between neighboring nodes at each time step, reflecting data flows and computational dependencies throughout the cluster.

To incorporate temporal dynamics, Ps-Stream links each compute node to its own representation across adjacent time steps. This allows the model to learn temporal dependencies, capturing trends and correlations in the evolution of resource loads. By modeling both space and time, Ps-Stream can then forecast which nodes are likely to become stragglers.

Building on these predictions, Ps-Stream constructs a fine-grained task-migration model. Upon anticipating a straggler, the system proactively triggers task scheduling to offload tasks to the least-loaded nodes before bottlenecks occur, balancing the system load in advance. In doing so, Ps-Stream also accounts for task dependencies and migration costs to minimize resource consumption and system overhead during migration.

Our main contributions are:

(1) **Spatio-Temporal Graph Construction**: We model compute nodes as a spatial graph at each time step and establish temporal connections across steps, creating a unified spatio-temporal graph structure.
(2) **Joint Feature Learning**: We use graph convolutional networks to model spatial dependencies and attention mechanisms to capture temporal dynamics, enabling the learning of robust spatio-temporal representations for straggler prediction.
(3) **Proactive, Cost-Aware Migration**: We develop a fine-grained task migration model that offloads tasks from predicted stragglers to underutilized nodes, while considering task dependencies and migration overhead.

The rest of this paper is organized as follows. Section II reviews related work. Section III introduces the system model, including the application model, communication load model, and resource model. Section IV formalizes the problems of resource prediction and load-balancing. Section V details the Ps-Stream methodology. Section VI evaluates the performance of Ps-Stream. Finally, Section VII concludes the paper and outlines future work.

II. Related work

In this section, we review recent works in two related areas: reactive scheduling and proactive scheduling.

A. Reactive scheduling

Elastic resource allocation strategies in clusters largely adopt a reactive approach, where resources are adjusted dynamically in response to real-time changes in the system state. These strategies focus on optimizing resource utilization while maintaining system QoS performance. Many approaches respond to workload fluctuations by reallocating resources to address sudden changes, essentially formulating resource reallocation as a combinatorial task-to-resource assignment problem [26], [27].

For example, Li et al. [8] proposed a cost-efficient load-balancing algorithm for distributed stream computing systems, which reduces job execution costs while optimizing load distribution. TOP-Storm [9] enhances resource utilization through a topology-aware scheduler that logically groups tasks to reduce inter-task communication, followed by resource-aware task allocation. An adaptive scheduling method [12] concurrently schedules multiple jobs using different policies based on data dependencies and dynamically adjusts resource allocation among jobs according to workload characteristics. Brown et al. [11] focused on scheduling and provisioning for dynamic workflows with unpredictable workloads, prioritizing critical tasks to ensure reliable and efficient stream processing.

Although these reactive mechanisms can adjust resource configurations quickly based on feedback from the current system state, they often incur latency due to their post hoc nature. This delay becomes critical when facing sudden load spikes, potentially leading to inadequate resource allocation, delayed mitigation of performance bottlenecks, and degraded system responsiveness.

B. Proactive scheduling

Prediction-driven scheduling aims to predict future workload demands or resource requirements by analyzing historical data. These predictions are then used to proactively allocate resources and mitigate performance degradation caused by workload fluctuations or resource shortages.

For example, Pec [16] addresses proactive elastic scheduling for computation- and communication-intensive applications, with a focus on workload prediction, stable resource pre-allocation, and communication-aware resource placement. Similarly, a predictive scheduling framework [17] incorporates topology-aware modeling to predict average tuple processing time and guide task assignment, based on runtime statistics and application graph structure.

START [18] introduces a straggler prediction and mitigation based on a Long-Short Term Memory (LSTM) Encoder, which dynamically adapts task scheduling by analysing resource consumption patterns. Tang et al. [28] propose a two-dimensional parallel LSTM model to predict software faults in cloud virtual machines. They further introduce a low-complexity heuristic greedy algorithm to optimize service scheduling, aiming to minimize cost, rejection rate, and enhance execution reliability.

While these methods demonstrate notable progress, they often treat node loads as independent time series and overlook the complex spatial relationships among compute nodes in stream processing environments. Such spatial dependencies, which reflect data flow and task coordination across nodes, are critical for understanding system dynamics. Ignoring them can limit the learning capacity of the model.

Spatio-temporal deep learning techniques have also been explored in other prediction-oriented domains, such as urban

TABLE I
RELATED WORK COMPARISON.

Related work	Aspects				
	Scheduling type	Methodology	Straggler Prediction	Prediction Objective	Topology Awareness
Li et al. [8]	Reactive	Cost-efficient load balancing	✗	Nodes Busyness	✔
TOP-Storm [9]	Reactive	Resource-aware task assignments	✗	NULL	✔
Brown et al. [11]	Reactive	Resource scheduling and provisioning	✗	NULL	✔
Pec [16]	Proactive	Resource pre-allocation	✔	Operation workload	✔
START [18]	Proactive	Long-Short-Term-Memory network	✔	Number of straggler tasks	✗
Tang et al. [28]	Proactive	Long-Short-Term-Memory network	✔	Fault of nodes	✗
Our work	Proactive	Spatial-temporal convolution network	✔	Straggler nodes	✔

traffic flow forecasting [29]–[31]. However, these works focus on prediction accuracy within a fixed application context and do not address how prediction outcomes can be translated into proactive, cost-aware scheduling decisions.

In summary, existing approaches provide valuable insights into both reactive and proactive scheduling mechanisms. However, our proposed Ps-Stream framework distinguishes itself by incorporating spatio-temporal graph modeling through graph convolutional networks and attention mechanisms. This enables the learning of spatial and temporal features for accurate straggler prediction. In addition, Ps-Stream introduces a fine-grained, cost-aware task migration model that considers task dependencies and migration costs when offloading tasks from stragglers to underutilized nodes, thereby minimizing overhead. A comparison between our work and the relevant literature is summarized in Table I.

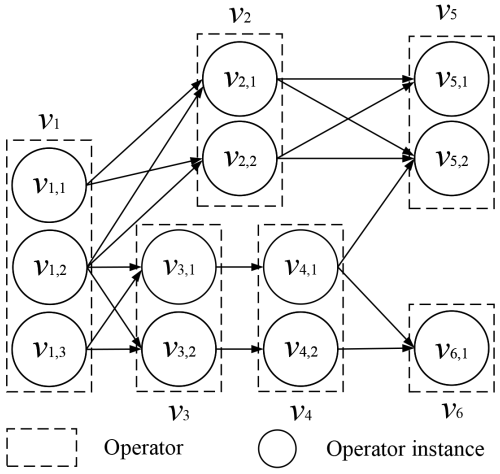

Fig. 1. Logical graph of operator instances.

III. SYSTEM MODEL

Before addressing the straggler problem and introducing our proposal, we first explain the streaming application model, communication load model, and the resource model in stream computing environments.

A. Streaming application model

In stream computing environments, a streaming application is conceptualized as a set of computational tasks designed to process continuously generated real-time data streams [32]. It describes the data processing logic through a directed acyclic graph (DAG) [6], denoted as $G_T = \{V(G_T), E(G_T)\}$. Here, $V(G_T) = \{v_1, v_2, ..., v_n\}$ represents a finite set of operators, each corresponding to a specific computation or transformation on the data stream. Each operator $v_i \in V(G_T)$ is associated with a unique function $f(v_i)$, ensuring that no two operators perform the same tasks, i.e., $\forall v_i, v_j \in V(G_T), i \neq j \implies f(v_i) \neq f(v_j)$. The edge set $E(G_T) = \{e_{i,j} | v_i, v_j \in V(G_T), i \neq j\}$ represents a finite set of directed edges between operators, where each edge $e_{i,j}$ indicates the flow of data from operator v_i to operator v_j, and can be used to represent their communication load.

When a streaming application is submitted to the data centre, it undergoes two key processes: initialization of operator instances and deployment of these instances.

(1) Initialization of operator instances. Multiple instances of operator v_i can be initialized based on the parallelism level specified by the user, where $\forall v_{i,k}, v_{i,m} \in v_i$, then $\exists f(v_{i,k}) = f(v_{i,m})$. As shown in Fig. 1, the initial operator topology consists of 6 operators, denoted as $\{v_i, v_2, ..., v_6\}$, each serving a different function. An instance topology is then constructed based on the configuration of operators: operator v_1 has 3 parallel instances, v_2, v_3, v_4 and v_5 each have 2, and v_6 has 1 instance. All instances under the same operator share the same function. For example, $f(v_{3,1}) = f(v_{3,2})$, indicating that instances $v_{3,1}$ and $v_{3,2}$ perform the same computation.

(2) Deployment of instances. The instance topology is deployed to workers on compute nodes by the scheduler. As shown in Fig. 2, the instance topology in Fig. 1 is evenly deployed across two compute nodes, with each node containing three workers. Communication between instances forms runtime dependencies between workers, thereby constructing a directed graph $G_W = \{V(G_W), E(G_W)\}$ with workers as vertices, where $E(G_W) \subseteq E(G_T)$. In this context, workers are the basic units for resource allocation and scheduling in the cluster.

B. Communication load model

In stream computing systems, the deployment of operator instances creates intricate topological relationships among workers. Quantifying communication load between workers is essential for understanding these topological dependencies and improving the accuracy of resource usage predictions.

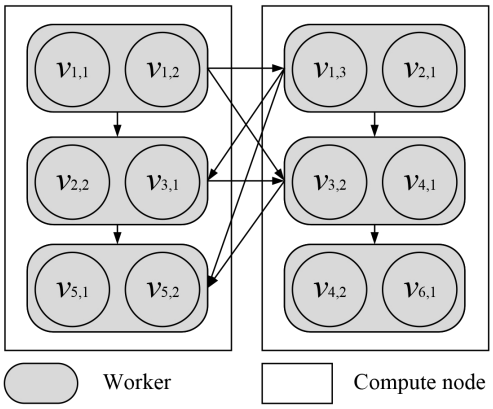

Fig. 2. Spatial topology of workers

We define a mapping function $map : V(G_T) \longrightarrow V(G_W)$, which assigns an operator instance $v_{i,k}$ to a worker w_p, where $V(G_W) = \{w_1, w_2, ..., w_M\}$ denotes the set of workers in the cluster, M denotes the number of user-defined workers. Each instance must be assigned to a worker, i.e. $w_p = map(v_{i,k})$, where $w_p \in V(G_W)$, $v_{i,k}$ is an instance of v_i, and $v_i \in V(G_T)$. Let $\rho(v_{i,k}, v_{j,m})$ represent the data transmission rate (in tuples per unit time) between two operator instances $v_{i,k}$ and $v_{j,m}$. This rate is defined by Eq. (1).

$$\rho(v_{i,k}, v_{j,m}) = \begin{cases} 0, & \text{if } map(v_{i,k}) = map(v_{j,m}) \\ E_\rho, & \text{otherwise} \end{cases}, \quad (1)$$

where E_ρ represents the average transmission rate over the sampling interval $[t_s, t_e]$. t_s and t_e represent the start and end times, respectively. Although short-term fluctuations in arrival rates may occur, E_ρ smooths out these variations by averaging them. E_ρ is calculated by Eq. (2).

$$E_\rho = \frac{\int_{t_s}^{t_e} E_{\rho,t} dt - \max(E_{\rho,t}) - \min(E_{\rho,t})}{t_e - t_s - \Delta t}, \quad (2)$$

where $E_{\rho,t}$ denotes the transmission rate at time t, and $t \in [t_s, t_e]$. Δt is the total duration corresponding to the maximum and minimum transmission rates.

Then, the total transmission rate $c_{w_{p,q}}$ between worker w_p and w_q is defined as the sum of the communication rates between all operator instances assigned to w_p and w_q. $c_{w_{p,q}}$ can be calculated by Eq. (3).

$$c_{w_{p,q}} = \sum_{\substack{map(v_{i,k})=w_p \\ map(v_{j,m})=w_q}} \rho_{(v_{i,k}, v_{j,m})} \quad (3)$$

C. Resource model

In a cluster CN with K compute nodes, denoted as $CN = \{cn_1, cn_2, ..., cn_k, ..., cn_K\}$, the resource load of compute node cn_k can be measured in different dimensions, such as CPU, memory, and I/O [33]. We assume that each compute node in the cluster only serves streaming applications. There are M workers in the cluster, represented as $V(G_W) = \{w_1, w_2, ..., w_M\}$. Resource usage information for each worker is collected from the stream processing platform, e.g., by implementing Apache Storm's built-in `Metric` interface. At time t, the CPU utilization of worker w_p is denoted as $R^{cpu}_{w_p}(t)$, where $R^{cpu}_{w_p}(t) \in [0, 1]$. The memory usage of worker w_p at time t is $mem(t)$, and the maximum available memory for the worker (as configured by the user) is Mem_{max}. The memory utilization $R^{mem}_{w_p}(t)$ can be calculated by Eq. (4).

$$R^{mem}_{w_p}(t) = \frac{mem(t)}{Mem_{max}} \quad (4)$$

At time t, let $io^r_{w_p}(t)$ and $io^w_{w_p}(t)$ denote the data read and write rates of worker w_p, respectively. Given a total I/O bandwidth BW, the I/O utilization $R^{io}_{w_p}(t)$ of worker w_p can be calculated by Eq. (5).

$$R^{io}_{w_p}(t) = \frac{io^r_{w_p}(t) + io^w_{w_p}(t)}{BW} \quad (5)$$

Over a sampling interval $[t_s, t_e]$, the average CPU utilization $R^c_{w_p}$, memory utilization $R^m_{w_p}$, and I/O utilization $R^{io}_{w_p}$ of worker w_p can be calculated by Eq. (6).

$$\begin{cases} R^{cpu}_{w_p} = \dfrac{1}{t_e - t_s} \cdot \displaystyle\int_{t_s}^{t_e} R^{cpu}_{w_p}(t)\, dt \\ R^{mem}_{w_p} = \dfrac{1}{t_e - t_s} \cdot \displaystyle\int_{t_s}^{t_e} R^{mem}_{w_p}(t)\, dt \\ R^{io}_{w_p} = \dfrac{1}{t_e - t_s} \cdot \displaystyle\int_{t_s}^{t_e} R^{io}_{w_p}(t)\, dt \end{cases} \quad (6)$$

Each compute node hosts multiple workers that process data streams. Therefore, the average resource utilization of compute node cn_k, including CPU utilization $R^{cpu}_{cn_k}$, memory utilization $R^{mem}_{cn_k}$, and I/O utilization $R^{io}_{cn_k}$, can be calculated by aggregating the utilizations of all its assigned workers, along with the node's static system overhead with Eq. (7).

$$\begin{cases} R^{cpu}_{cn_k} = \displaystyle\sum_{w_p \in cn_k} R^{cpu}_{w_p} + S^{cpu}_{cn_k} \\ R^{mem}_{cn_k} = \displaystyle\sum_{w_p \in cn_k} R^{mem}_{w_p} + S^{mem}_{cn_k} \\ R^{io}_{cn_k} = \displaystyle\sum_{w_p \in cn_k} R^{io}_{w_p} + S^{io}_{cn_k} \end{cases}, \quad (7)$$

where $S^{cpu}_{cn_k}$, $S^{mem}_{cn_k}$ and $S^{io}_{cn_k}$ denote the static CPU, memory, and I/O utilization of compute node cn_k when no user workloads are running.

Finally, the overall resource consumption R_{cn_k} of compute node cn_k is defined as a weighted sum of its CPU, memory, and I/O utilization and calculated by Eq. (8).

$$R_{cn_k} = \gamma \cdot R^{cpu}_{cn_k} + \beta \cdot R^{mem}_{cn_k} + (1 - \gamma - \beta) \cdot R^{io}_{cn_k}, \quad (8)$$

where γ and β are the weighting factors for CPU and memory, respectively, satisfying $0 < (\gamma + \beta) < 1$.

IV. Problem Formalization

In this section, we formalize the proactive scheduling problem in stream computing systems, which mainly involves two aspects: resource prediction and load imbalance across compute nodes.

A. Resource prediction

A worker's resource usage can be influenced by the rate, volume, and distribution of data from its upstream workers. For example, a sudden increase in data volume or a shift in distribution (e.g., from uniform to skewed) can significantly increase the resource demand on downstream workers. This effect may propagate along the data flow in the directed graph, potentially overloading multiple workers.

To model these dependencies, we represent the workers as a directed graph $G_W = \{V(G_W), E(G_W)\}$, where $V(G_W)$ is the set of workers and $E(G_W)$ is the set of directed edges representing communication dependencies.

We define the adjacency matrix derived from the worker topology G_W as $\mathbf{A} \in \mathbf{R}^{M\times M}$, where M is the number of workers in the cluster. If $w_p, w_q \in V(G_W)$ and $(w_p, w_q) \in E(G_W)$, then $\mathbf{A}_{p,q}$ is "1" otherwise it is "0".

At each time step t, the worker topology G_W has a dynamic feature matrix $\mathbf{X}^{(t)} \in \mathbf{R}^{M\times(M+3)}$, and it can be represented by Eq. (9).

$$\mathbf{X}^{(t)} = \begin{bmatrix} c_{w_{1,1}} & \cdots & c_{w_{1,M}} & R_1^{cpu} & R_1^{mem} & R_1^{io} \\ c_{w_{2,1}} & \cdots & c_{w_{2,M}} & R_2^{cpu} & R_2^{mem} & R_2^{io} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ c_{w_{M,1}} & \cdots & c_{w_{M,M}} & R_M^{cpu} & R_M^{mem} & R_M^{io} \end{bmatrix}. \quad (9)$$

where $c_{w_{p,q}}$ denotes the transmission rate from worker w_p to worker w_q, which is computed using Eq. (3), $1 \leqslant p, q \leqslant M$.

Given the worker topology G_W and the historical resource and communication data $\mathbf{X}^{(t-S):t}$ over the past S time steps, the objective is to predict the future resource load $\mathbf{Y}^{(t+1):(t+T)}$ of all workers over the next T time steps. The target matrix $\mathbf{Y}^{(t)}$, corresponding to CPU, memory, and I/O utilization, consists of the last three columns of the feature matrix $\mathbf{X}^{(t)}$. It can be represented by Eq. (10).

$$\mathbf{Y}^{(t)} = \mathbf{X}^{(t)}[:, M+1 : M+3], \quad (10)$$

Then, our task is to learn a prediction function f that maps the historical data and graph structure to the future resource loads at time T. This task is represented by Eq. (11).

$$\left[\mathbf{X}^{(t-S):t}, G_W\right] \xrightarrow{f} \mathbf{Y}^{(t+1):(t+T)}, \quad (11)$$

where $\mathbf{X}^{(t-S):t} \in \mathbf{R}^{M\times(M+3)\times S}$ and $\mathbf{Y}^{(t+1):(t+T)} \in \mathbf{R}^{M\times 3\times T}$.

B. Load imbalance across compute nodes

Improper instance deployment can lead to excessive resource consumption on certain compute nodes, resulting in stragglers, which are nodes that significantly lag in performance. For example, when tasks with data dependencies are deployed on a small number of nodes, these nodes may experience network or CPU bottlenecks, while others remain underutilized. In addition, as data streams and task resource demands fluctuate, delayed or reactive scheduling strategies often fail to redistribute workloads in time, intensifying the load imbalance. This imbalance not only wastes resources and increases the risk of node overloading, but also degrades system response time, potentially leading to cascading failures or even system-wide collapse.

Let $CN = \{cn_1, cn_2, ..., cn_K\}$ be the set of compute nodes in the cluster, and let their resource consumption be $R_{cn} = \{R_{cn_1}, R_{cn_2}, ..., R_{cn_K}\}$. We define the load imbalance degree L_{imb} as a composite metric that captures both the maximum relative deviation and the coefficient of variation (CV) of resource usage across compute nodes. It can be calculated by Eq. (12).

$$L_{imb} = \max\left(\frac{\max(R_{cn})}{\bar{R}_{cn}} - 1, 1 - \frac{\min(R_{cn})}{\bar{R}_{cn}}\right) + CV, \quad (12)$$

where $\bar{R}_{cn}$ denotes the average resource consumption across compute nodes, and is calculated by Eq. (13).

$$\bar{R}_{cn} = \frac{1}{K} \cdot \sum_{i=1}^{K} R_{cn_i}, \quad (13)$$

and the coefficient of variation CV can be calculated by Eq. (14).

$$CV = \frac{1}{\bar{R}_{cn}} \cdot \sqrt{\frac{\sum_{i=1}^{K}\left(R_{cn_i} - \bar{R}_{cn}\right)^2}{K}} \quad (14)$$

To mitigate the impact of stragglers on system latency, it is important to keep L_{imb} within an acceptable range, typically below a given threshold α. Thus, the resource load across compute nodes in the cluster should satisfy the condition: $L_{imb} \leqslant \alpha$.

V. Ps-Stream: Architecture and Algorithm

In this section, we introduce Ps-Stream's architecture and describe the algorithm used for straggler prediction and resource migration.

A. System architecture

As shown in Fig. 3, the system architecture of Ps-Stream comprises five main modules: Feature Extractor, Straggler Prediction, Straggler Mitigation, Schedule Generator, and Custom Scheduler.

The **Feature Extractor** module analyzes resource information collected by the Monitor component within each Worker. It extracts both resource dependencies and spatio-temporal features among workers. This module leverages Graph Convolutional Networks (GCNs) to learn spatial relationships and incorporates an attention mechanism to capture the temporal dynamics of the evolving worker topology.

The **Straggler Prediction** module consists of multiple feed-forward neural network layers. These layers iteratively learn the spatial structure of the worker topology to construct an efficient prediction function. This function forecasts each worker's future resource load, including CPU, memory, and I/O utilization, enabling early detection of potential stragglers.

The **Straggler Mitigation** module analyzes the predicted resource loads of workers and their distribution across compute nodes to identify which nodes are likely to become stragglers. Once identified, the module forwards this information to the Schedule Generator module for further optimization.

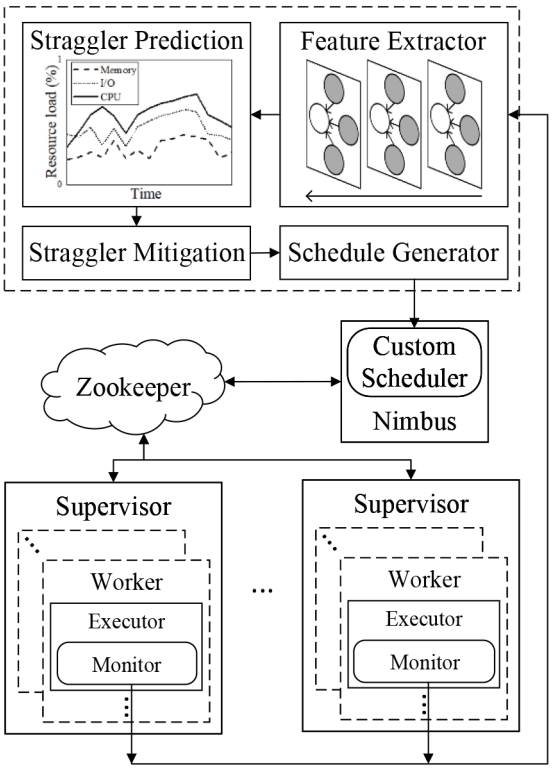

Fig. 3. Ps-Stream architecture

The **Schedule Generator** module evaluates the data dependencies among workers deployed on predicted stragglers and estimates the migration costs for these affected workers. Based on this analysis, it determines which workers need to be migrated at the minimal cost and generates a migration plan.

The **Custom Scheduler** module is implemented using interfaces provided by the underlying stream computing platform (e.g., Apache Storm's `IScheduler`) to support dynamic scheduling. During initial deployment, it identifies the communication paths of each data stream in the instance topology (e.g., Fig.1) and evenly distributes interdependent instances across workers to reduce communication overhead. Based on the migration plan generated by the Schedule Generator, the Custom Scheduler dynamically adjusts instance placements, reallocating workers to different compute nodes to mitigate potential stragglers.

Compared to traditional straggler mitigation techniques, such as node-level or single-node prediction models, Ps-Stream offers several advantages: (1) Fine-grained resource modeling: Traditional methods often predict resource consumption at the compute-node level, but these predictions can become inaccurate once task migration alters data distribution patterns. Ps-Stream addresses this by modeling resource usage at the worker level, maintaining consistent data patterns even after migration. (2) Spatio-temporal awareness: Traditional methods often neglect the spatio-temporal dependencies between compute nodes, limiting their ability to learn complex interactions. Ps-Stream overcomes this limitation by incorporating a spatio-temporal model that more effectively captures dynamic relationships and load propagation among compute nodes.

B. Spatial-temporal prediction

Ps-Stream incorporates a spatial-temporal prediction model (corresponding to the Feature Extractor and Straggler Prediction in Fig. 3) to capture dynamic spatial and temporal correlations within the worker topology. Regardless of how the cluster size changes, the logical worker topology remains unchanged, allowing the prediction model to generalize across different cluster sizes without retraining. As shown in Fig. 4, the model consists of two key modules: spatial feature extraction and temporal feature extraction.

We use historical time series data X as input. A two-layer GCN is used to capture the worker network's topological structure and extract spatial features. The spatially enriched time series are then passed to a Transformer model, where information flow across time steps captures temporal dependencies. The final output is the predicted resource load for each worker over the next T time steps.

1) Spatial feature extraction: The spatial feature extraction module uses the worker topology and worker features to learn worker embeddings that represent both individual worker characteristics and their structural relationships. GCNs [25] are used to iteratively aggregate feature information from each worker's neighbors. By stacking multiple GCN layers, the model expands its receptive field and captures higher-order topological dependencies.

Given the adjacency matrix G_W and feature matrix $\mathbf{X}$, the GCN constructs a filter in the Fourier domain. This spectral filter, when applied over the worker topology, captures spatial features by considering the first-order neighborhood between workers. This process can be described by Eq. (15).

$$H^{(l+1)} = \sigma\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{G}_W\widetilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right), \tag{15}$$

where $\widetilde{G}_W = G_W + I_W$ is the adjacency matrix with added self-connections. I_W is the identity matrix, and $\widetilde{D}$ is the degree matrix of $\widetilde{G}_W$, with $\widetilde{D}_{ii} = \sum_j \widetilde{G}_{W_{ij}}$. $H^{(l)}$ is the hidden feature representation at l-th layer, with $H^{(0)} = \mathbf{X}$. $W^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes the activation function.

We use a two-layer GCN model to extract spatial dependencies, which can be described by Eq. (16).

$$H^{(2)} = \sigma\left(\widetilde{A}\mathrm{ReLU}\left(\widetilde{A}\mathbf{X}W^{(0)}\right)W^{(1)}\right), \tag{16}$$

where $\widetilde{A} = \widetilde{D}^{-\frac{1}{2}}\widetilde{G}_W\widetilde{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix. $W^{(0)} \in \mathbf{R}^{(M+3)\times M}$ is the trainable weight matrix from the input $\mathbf{X}$ to the hidden layer, and $W^{(1)} \in \mathbf{R}^{M\times T}$ maps from the hidden layer to the output. $\mathrm{ReLU}(\cdot)$ is the activation function, $\mathrm{ReLU} = \max(0, \cdot)$. The resulting matrix $H^{(2)} \in \mathbf{R}^{M\times P}$ denotes the final spatial feature embedding of the worker topology, where P is the embedding dimension of a worker.

We convert the 2D feature embedding $H^{(2)}$ into a 1D vector H, where $H[k] = H^{(2)}_{i,j}$, $k = (i-1)\cdot P + j - 1, i \in [1, M], j \in [1, P]$. M is the number of workers used for streaming applications. Our model uses a sequence of S such embeddings to predict the future resource load of each worker

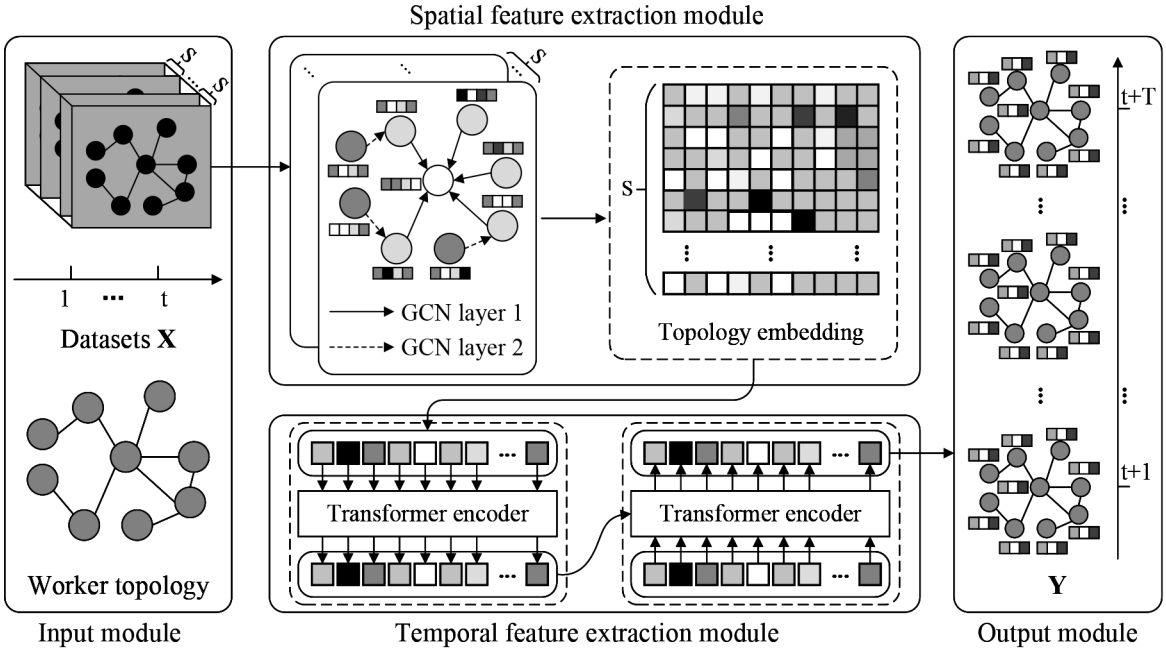

Fig. 4. Spatio-temporal prediction model

for T time steps. Therefore, a Topology Embedding matrix $B \in \mathbf{R}^{S\times(M\times P)}$ integrates the feature information from S worker topologies, where $B = [H_1, H_2, ..., H_S]^{\mathbf{T}}$.

2) Temporal feature extraction: In the temporal dimension, there are correlations between worker topologies at different time steps, and these correlations vary under different load conditions. After the graph convolution operations capture neighboring information for each worker in the spatial dimension, a standard convolution layer in the temporal dimension is subsequently applied to update sequential dependencies across time slices. This process enables the prediction model to account for both spatial and temporal dependencies.

To model longer-term dependencies, the topology embedding B is input into a Transformer [34], where B encodes the temporal evolution of worker relationships and load variations over time. The Transformer relies on attention mechanisms to model the temporal correlations between time slices of worker topologies. This enables refined predictions of future resource consumption for each worker.

At each time slice t, the signal $H_t \in B$ for topology embedding can be updated by aggregating information from its neighboring workers in the previous time slices. We transform the signal H_t into a query vector Q_t, a key vector K_t , and a value vector U_t through three learnable weight matrices $W_Q \in \mathbf{R}^{(M\times P)\times(M\times P)}$, $W_K \in \mathbf{R}^{(M\times P)\times(M\times P)}$ and $W_U \in \mathbf{R}^{(M\times P)\times(M\times P)}$, and Q_t can be described by Eq. (17).

$$Q_t = H_t W_Q, K_t = H_t W_K, U_t = H_t W_U \tag{17}$$

The attention embedding AE between different time slices can be calculated by Eq. (18).

$$AE = Attention\,(Q_t, K_t, U_t) = \sum_{H_j \in B} \alpha_{tj} U_t, \tag{18}$$

where α_{tj} is the attention weight that reflects the strength of the temporal relationships between time steps t^{th} and j^{th} topology embedding. α_{tj} can be calculated by Eq. (19).

$$\alpha_{tj} = \frac{\exp\left(\cos\left(Q_t, K_j\right)\right)}{\sum_{H_i \in B} \exp\left(\cos\left(Q_t, K_i\right)\right)}, \tag{19}$$

where $\cos\left(Q_t, K_i\right)$ denotes the cosine value between vector Q_t and vector K_i.

Finally, the attention embedding AE is passed through a feed-forward network to obtain the prediction matrix $\mathbf{Y}$, representing the resource load for each worker over the next T time steps.

C. Resource migration

Our spatio-temporal model (i.e., Feature Extractor and Straggler Prediction in Fig. 3) predicts the resource load of each worker over the next T time steps. Using Eq. (8), the resource load of compute nodes can be aggregated over this interval, providing data support for proactive scheduling.

If the predicted resource loads consistently satisfy the condition $L_{imb} \geqslant \alpha$ over a certain period, we consider the node with the highest resource load a straggler (a performance bottleneck). To mitigate its impact, Ps-Stream proactively migrates some instances from the straggler node to the node with the lowest resource load. The goal is to balance the cluster's load and optimize overall resource allocation. However, it is challenging to select the optimal subset of instances to migrate while minimizing migration costs.

Let $SN = \{w_1, w_2, ..., w_i\}$ be the set of workers running on the straggler node, and let $LN = \{w_{i+1}, w_{i+2}, ..., w_{i+j}\}$ be the set of workers running on the node with the lowest resource load. Here, $SN \subset V(G_W)$ and $LN \subset V(G_W)$. These workers in both SN and LN form a new topology $G_N =$

$\{V(G_N), E(G_N)\}$, where $V(G_N) \subset V(G_W), E(G_N) \subset E(G_W)$, and SN and LN are two subgraphs of G_N. We define a vector $\Phi = [\Phi_1, \Phi_2, ..., \Phi_i, ..., \Phi_j]^{\mathbf{T}}$ to describe worker affiliation with LN, where the size j of Φ is equal to the number of workers in topology G_N. If worker $w_i \in LN$, then $\Phi_i = 1$; otherwise, $\Phi_i = 0$.

Assume that some workers in SN are selected for migration to LN. We define a binary vector $\lambda = [\lambda_1, \lambda_2, ..., \lambda_i, ..., \lambda_j]^{\mathbf{T}}$ to describe which workers in G_N are selected for migration, where the size j of λ equals the number of workers in topology G_N. If worker w_i is migrated, then $\lambda_i = 1$; otherwise, $\lambda_i = 0$.

Let $C \in \mathbf{R}^{j \times j}$ represent the communication rate matrix among workers in topology G_N, where each element $c_{p,q} \in C$ represents the transmission rate from worker w_p to w_q, computed using Eq. (3).

The communication cost MC for the migrated workers from SN to LN can be calculated using Eq. (20).

$$MC = \frac{\lambda^{\mathbf{T}} C\,(\mathbf{1} - \lambda) - \Phi^{\mathbf{T}} C \lambda}{\Phi^{\mathbf{T}} C \lambda}, \tag{20}$$

where the denominator represents the communication between the migrated workers and the destination node (i.e., LN), and the numerator represents the communication between the migrated workers and the remaining workers in the source node SN.

To evaluate the overall migration benefit, we define the migration factor η, given by Eq. (21).

$$\eta = MC + \frac{\sum \lambda}{card\,(SN)} + \left| \frac{L'_{imb}}{\alpha} \right|, \tag{21}$$

where $card\,(SN)$ denotes the cardinality (size) of the set SN. $\sum \lambda$ denotes the number of migrated workers. L'_{imb} denotes the load imbalance degree in the cluster after migration.

To minimize η, Ps-Stream applies a simulated annealing algorithm, shown in Algorithm 1. The algorithm iteratively explores possible migration plans, accepting better ones and occasionally worse ones (with a probability governed by a cooling schedule) to escape local minima.

The input of Algorithm 1 includes the worker set SN in the straggler node, the worker set LN in the least-loaded node, the predicted future resource load matrix Y of workers used by streaming applications, the communication matrix C, and the reconstructed topology G_N. The output is the set RM of workers to be migrated from the straggler to the least-loaded node. Steps 1 to 5 initialize the data required. Steps 6 to 13 randomly select a worker to migrate. If it is in SN, move it to LN; otherwise, move it back. Steps 14 to 15 evaluate the new solution's cost using the migration factor. Steps 16 to 19 accept the new solution if it improves η, or accept worse solutions with a certain probability. Step 20 reduces the temperature that determines whether to accept worse solutions. Step 22 outputs the final set of workers to migrate.

Algorithm 1 is executed iteratively until the cluster's load imbalance degree L_{imb} falls below the threshold α. The final result is stored in a Map structure (with target node IDs as keys and sets of workers to migrate as values), which is passed to the scheduler for worker redeployment.

Algorithm 1: Minimize migration costs via simulated annealing.

Input: Worker sets SN, LN, prediction matrix $\mathbf{Y}$, communication matrix C, topology G_N;
Output: Set of migrated workers RM;

1 Initialize temperature T and the drop rate φ;
2 Set initial migration factor $\eta \leftarrow +\infty$;
3 Initialize best worker sets: $BSN \leftarrow SN, BLN \leftarrow LN$
4 **for** $i = 0$ *to* $MaxIteration$ **do**
5 Initialize current best sets: $NewSN \leftarrow BSN, NewLN \leftarrow BLN$;
6 $rw \leftarrow$ Randomly select a worker in SN;
7 **if** $rw \in NewSN$ **then**
8 Remove rw from $NewSN$;
9 Add rw to $NewLN$;
10 **else**
11 Remove rw from $NewLN$;
12 Add rw to $NewSN$;
13 **end**
14 Calculate communication cost MC between $NewLN$ and $NewSN$ using Eq. (20) ;
15 Calculate migration factor η' using Eq. (21) ;
16 **if** $\eta' - \eta < 0$ || *Math.random()* $< \mathrm{e}^{\frac{(\eta - \eta')}{\mathrm{T}}}$ **then**
17 $BSN \leftarrow NewSN, BLN \leftarrow NewLN$;
18 $\eta \leftarrow \eta'$;
19 **end**
20 $Updatetemperature : \mathrm{T} = \mathrm{T} \times \varphi$;
21 **end**
/* RM represents the set of workers to migrate: those in SN but not in BSN. */
22 $RM \leftarrow SN \setminus BSN$;
23 **return** RM

VI. Performance evaluation

In this section, we evaluate the performance of the Ps-Stream system. We first describe the experimental setup, followed by a detailed analysis of system performance, resource utilization, and the effectiveness of the spatio-temporal prediction model.

A. Experimental setup

The Ps-Stream system is developed on top of Apache Storm 2.4.0 and deployed on Ubuntu 20.04. The cluster comprises 11 machines, each equipped with an Intel(R) Xeon(R) X5650 CPU (quad-core, 2.4 GHz), 4 GB of RAM, and a 100 Mbps Ethernet interface. Among these, one machine serves as the master node, running ZooKeeper and Nimbus, while the remaining 10 machines serve as supervisor nodes, each configured to run 4 workers with a JVM heap size of 256 MB per worker.

1) Implementation: Ps-Stream is implemented as two decoupled components. The scheduling and task migration logic is implemented in Java as a custom `IScheduler` plugin

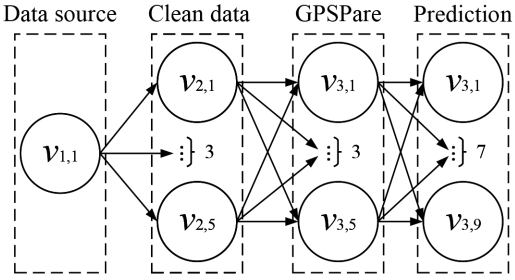

Fig. 5. Logical graph of DTPrediction.

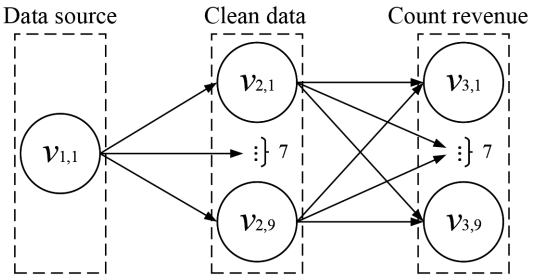

Fig. 6. Logical graph of TRevenue.

that overrides Storm's default scheduling behavior. The spatio-temporal prediction model is implemented in Python using PyTorch, with the GCN layers built on `torch_geometric`. The prediction service runs on a dedicated server equipped with an Intel Core i5-9300H CPU and performs inference on the CPU. The two components exchange resource-usage features and prediction results through a Redis instance deployed on the master node.

2) Dataset: We use the New York City Taxi Trip Dataset [35] to evaluate system performance. Each record in the dataset represents a completed taxi trip and includes details such as the taxi's license ID, pickup and drop-off locations and timestamps, travel distance, and total fare (including taxes and tolls). The data naturally exhibits a bimodal event rate distribution, reflecting morning and evening commute peaks, with rates reaching up to 4,500 events per second. To emulate real-time stream behavior, we apply the temporal scaling method from RIoTBench [36], which compresses message intervals to accelerate the stream rate. The temporal scaling factor is set to 450, enabling us to simulate peak stream conditions.

3) Streaming applications: We evaluate Ps-Stream with two streaming applications: DTPrediction and TRevenue. DTPrediction predicts the estimated time of arrival (ETA) for a taxi to reach the passenger's destination. TRevenue computes and reports each driver's cumulative daily revenue in real time. The logic graphs of these topologies are shown in Fig. 5 and Fig. 6, respectively.

4) Baseline schemes and evaluation metrics: We compare Ps-Stream against three state-of-the-art scheduling baselines: EvenScheduler (the default scheduler in Apache Storm [37]), R-Storm (a resource-aware scheduling framework [38]), and SP-ant (a scheduling algorithm based on ant colony optimization [10]). The evaluation focuses on the following metrics: system latency (the average end-to-end processing time per tuple, from ingestion to completion), maximum throughput (the highest number of tuples processed per second before task failure occurs), resource utilization (the average CPU, memory, and I/O utilization across all supervisor nodes), and prediction accuracy (the ability of the Ps-Stream model to forecast future resource loads).

B. System latency

In our experiments, the observed load imbalance degree L_{imb} fluctuates between 1.0 and 1.5. To avoid frequent worker migration operations, we set the threshold α for triggering migrations to 1.3. In addition, we employ a time scaling factor (default value 1.0) to control the intensity of the input data stream, computed as $Timestamp = Timestamp \times streamRate$, where $streamRate$ denotes the scaling factor. A smaller scaling factor compresses messages generated over a longer original interval into a shorter emission period in the preprocessed dataset (e.g., a factor of 0.2 compresses 5 seconds of data into 1 second), resulting in a higher-intensity, more bursty stream.

Under different streaming application scenarios, Ps-Stream consistently exhibits lower system latency than EvenScheduler, R-Storm, and SP-ant. As shown in Figs. 7 and 8, system latency fluctuates over time. Ps-Stream achieves average latencies of 30.7 ms and 36.5 ms under DTPrediction and TRevenue topologies, respectively, with relatively small fluctuation. In contrast, **EvenScheduler** shows significantly higher latency, with averages of 74.1 ms and 80.8 ms, and notable fluctuations due to its round-robin scheduling strategy. **R-Storm** performs better by allocating resources based on application requirements, resulting in lower average latencies of 59.6 ms and 63.2 ms. **SP-ant**, which uses ant colony optimization for dynamic scheduling, further reduces latency to 47.2 ms and 51.1 ms, but still performs worse than Ps-Stream. Although SP-ant dynamically allocates resources at runtime, it tends to deploy tasks in clusters on a few nodes. This clustering behavior may lead to localized overloads and increased latency. In addition, Ps-Stream also achieves the lowest latency variance among all methods, with standard deviations of 5.3 ms and 6.2 ms under DTPrediction and TRevenue, respectively, compared to 6.1-10.9 ms and 9.1-11.2 ms for the baselines. These results demonstrate that Ps-Stream has both lower latency and greater stability even under fluctuating data stream rates.

Across different temporal scaling factors (i.e., different input stream rates), Ps-Stream consistently maintains lower average latency than the baseline schedulers. As shown in Figs. 9 and 10, latency generally decreases as the temporal scaling factor increases. Key observations include: **EvenScheduler** has the highest latency due to its inability to adapt to load variation. **R-Storm** performs better due to its load-aware strategy but suffers under skewed data (e.g., at a scaling factor of 0.8), showing significant latency increases. **SP-ant** performs better than EvenScheduler and R-Storm but cannot fully overcome the effects of extreme skew in some cases. Ps-Stream, however, maintains low latency across all scaling factors. Notably,

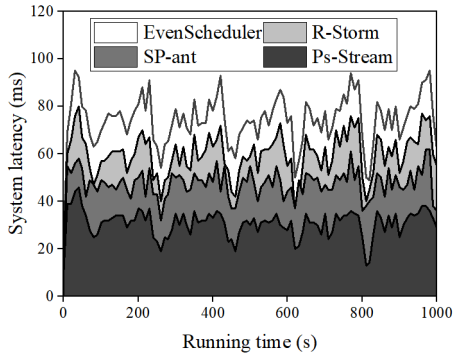

Fig. 7. System latency of DTPrediction.

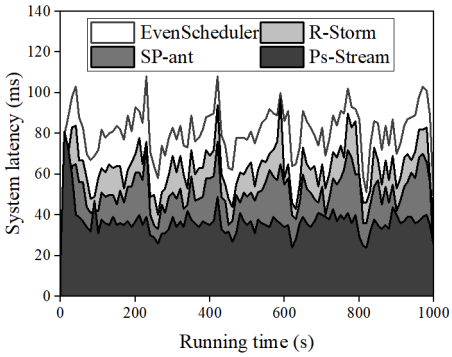

Fig. 8. System latency of TRevenue.

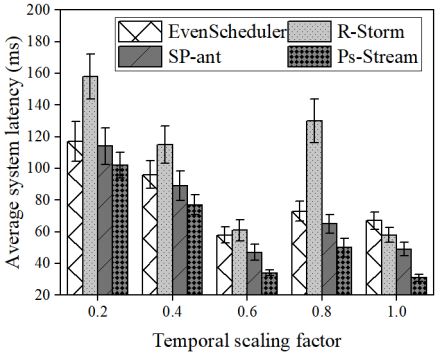

Fig. 9. Average system latency of DTPrediction under different temporal scaling factors.

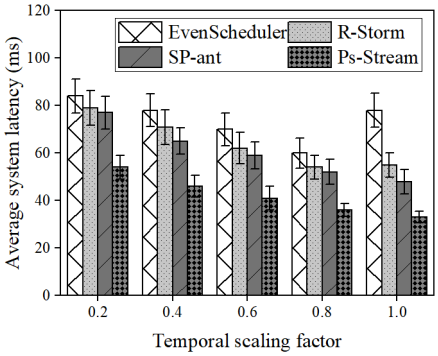

Fig. 10. Average system latency of TRevenue under different temporal scaling factors.

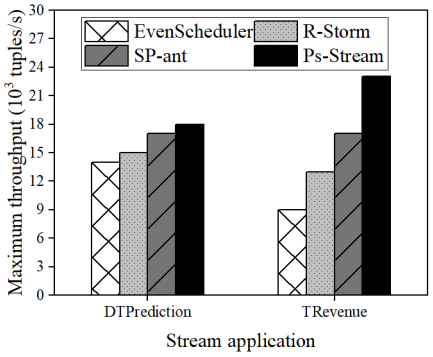

Fig. 11. Maximum throughput of the two streaming applications.

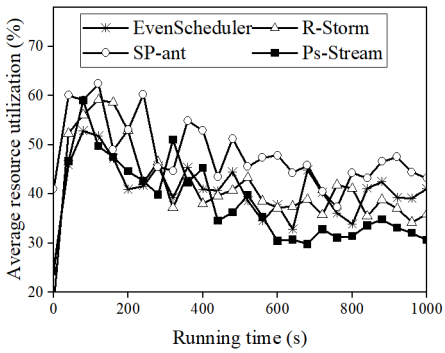

Fig. 12. Average resource load of DTPrediction.

under the DTPrediction topology and a scaling factor of 0.8, Ps-Stream avoids the dramatic latency spikes observed in other methods. This demonstrates its stronger load balancing capability and adaptability to data stream skewness.

For the TRevenue topology (Fig. 10), the standard-deviation intervals of Ps-Stream and the closest baseline (SP-ant) do not overlap at any scaling factor. For the DTPrediction topology (Fig. 9), overlap occurs at the lower scaling factors (0.2, 0.4), where the stream is more heavily compressed and thus higher in intensity, resulting in larger latency and variance across all methods. As the scaling factor increases toward the default (1.0), uncompressed rate, intensity decreases and Ps-Stream's advantage becomes clearly separable from all baselines.

In summary, the experiments confirm that Ps-Stream consistently outperforms the baselines in terms of average latency and resilience to stream rate variations. This is attributed to its proactive scheduling mechanism that perceives resource imbalances in advance and triggers timely task rescheduling.

C. System throughput

By measuring this metric, we can determine the system's performance ceiling under sufficient resource conditions, providing a benchmark for capacity planning. In our tests, we gradually increase the data input rate until task failure (i.e., system downtime) occurs.

With a data rate increment of 1,000 tuples/s, Ps-Stream exhibits significantly higher throughput than EvenScheduler, R-Storm, and SP-ant across both streaming applications. As shown in Fig. 11, for the DTPrediction topology, the system throughputs are 14,000, 15,000, 17,000, and 18,000 tuples/s for EvenScheduler, R-Storm, SP-ant, and Ps-Stream, respectively. Ps-Stream improves throughput by 5.8% over SP-ant. For the TRevenue topology, the system throughputs are 9,000 tuples/s for EvenScheduler, 13,000 tuples/s for R-Storm, 17,000 tuples/s for SP-ant, and 23,000 tuples/s for Ps-Stream. Ps-Stream outperforms SP-ant by 35.3%. These results demonstrate that Ps-Stream handles high input rates more effectively, with performance gains becoming more pronounced under heavier workloads. This suggests stronger scalability compared to other frameworks.

In summary, Ps-Stream consistently delivers higher system throughput due to its proactive ability to redistribute workers from overloaded nodes to underutilized ones, thus improving cluster-wide resource utilization.

D. Resource utilization

Average resource utilization reflects the efficiency of cluster resources used by streaming applications. While lower utilization may indicate underuse, it also provides an elastic buffer for burst workloads, preventing nodes from becoming overloaded during peak conditions.

Across different streaming applications, Ps-Stream exhibits lower average resource utilization than EvenScheduler, R-Storm and SP-ant. As shown in Figs. 12 and 13, for the DTPrediction topology, SP-ant exhibits relatively high resource

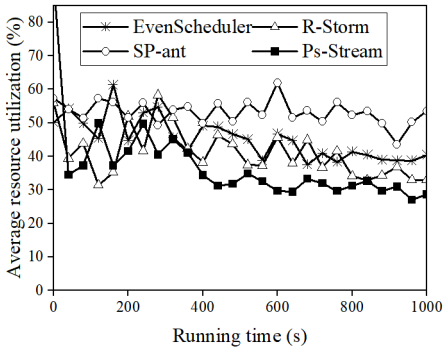

Fig. 13. Average resource load of TRevenue.

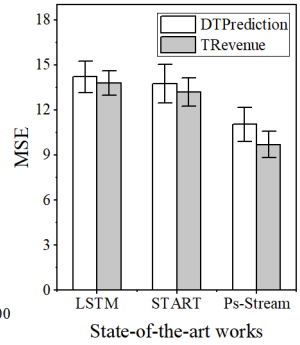

Fig. 14. MSE values of two topologies under LSTM, START and Ps-Stream.

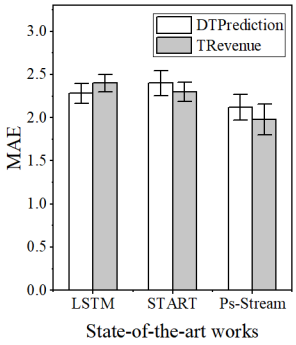

Fig. 15. MAE values of the two topologies under LSTM, START and Ps-Stream.

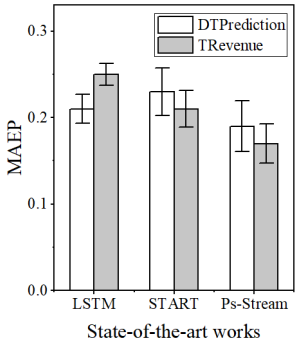

Fig. 16. MAPE values of the two topologies under LSTM, START and Ps-Stream.

usage (40–60%), with initial-stage fluctuations. EvenScheduler and R-Storm show slightly lower utilization (35–45%), while Ps-Stream maintains the lowest usage, stabilizing between 30–40%. For the TRevenue topology, fluctuations in resource utilization across the methods decrease. EvenScheduler and R-Storm stabilize around 40%, SP-ant sustains a higher utilization level (near 52%), while Ps-Stream stabilizes at approximately 33% after initial fluctuations.

In summary, Ps-Stream achieves lower and more balanced resource utilization. Although fluctuations may occur during the initial scheduling phase, its dynamic task placement strategy improves resource efficiency across the cluster over time.

E. Prediction Accuracy Comparison

In this experiment, we compare the prediction accuracy of Ps-Stream against two baselines: the START model [18] and the Long Short-Term Memory model (LSTM). LSTM is the general Long Short-Term Memory network used as a standard time-series prediction baseline, while START extends LSTM with an encoder network. However, both model each compute node's time series independently and do not capture dependencies between DAG operators. The prediction task is to forecast each worker's resource load, including CPU, memory, and I/O utilization, over the next 3 time steps. The ground truth is the corresponding resource utilization observed by the monitoring module at those future time steps.

We collect a dataset of 1,000 per-worker resource-usage samples and split it chronologically into 80% for training and 20% for test. All models are trained exclusively on the training split, and prediction accuracy is evaluated on the held-out 20% test split. Under identical experimental conditions, we use Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE) [39]. Each experiment is repeated 3 times, and the average result and standard deviation are reported.

As shown in Figs. 14 - 16, Ps-Stream exhibits consistently better accuracy across all metrics and applications: For MSE (Fig. 14), Ps-Stream improves prediction accuracy by 26% over LSTM and 23% over START, and the standard-deviation intervals do not overlap with either baseline. For MAE (Fig. 15), Ps-Stream achieves gains of 12% over LSTM and 14% over START. For MAPE (Fig. 16), Ps-Stream outperforms LSTM and START by 21% and 18%, respectively. For MAE and MAPE, the standard-deviation intervals show a marginal overlap with the closest baseline in a few configurations. Ps-Stream achieves the lowest mean error in all 3 runs across all six metric–topology combinations.

In summary, Ps-Stream provides significantly better prediction accuracy than both START and LSTM. Its advantage is attributed to Ps-Stream's ability to integrate the spatio-temporal structure of worker topologies to improve the prediction of future resource loads on compute nodes.

VII. Conclusion and Future Work

In distributed stream computing environments, the primary objectives of system implementation are to balance resource loads across the cluster, minimize system latency, and maintain overall system stability. Achieving these objectives requires a system that can sense both the communication dynamics between tasks in streaming applications and the resource consumption of workers running on compute nodes. Moreover, it should be capable of accurately predicting the resource load of each worker and proactively adjusting their deployment across nodes to prevent bottlenecks.

To address these challenges, we propose Ps-Stream, a proactive framework for straggler prediction and mitigation. Ps-Stream forecasts which compute nodes might become stragglers and dynamically adapts task scheduling to reduce system latency and improve resource utilization. The framework consists of two core components: a spatio-temporal prediction model that identifies stragglers by integrating GCNs to model spatial dependencies and attention mechanisms to capture temporal dynamics under fluctuating workloads; and a cost-sensitive scheduler that proactively migrates tasks from predicted stragglers to optimal target nodes, minimizing migration overhead and promoting balanced resource distribution across the cluster.

In future work, we plan to incorporate incremental learning with larger training datasets to improve prediction accuracy. We will also optimize the spatio-temporal prediction model for better precision with fewer samples. Additionally, we aim to enhance the scheduler to further reduce communication overhead during task redeployment in streaming applications.

REFERENCES

[1] H. Jin, F. Chen, S. Wu, Y. Yao, Z. Liu, L. Gu, and Y. Zhou, "Towards low-latency batched stream processing by pre-scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 710–722, 2019.

[2] F. Farhat, D. Zad Tootaghaj, Y. He, A. Sivasubramaniam, M. Kandemir, and C. R. Das, "Stochastic modeling and optimization of stragglers," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 1164–1177, 2018.

[3] W. Li, D. Liu, K. Chen, K. Li, and H. Qi, "Hone: Mitigating stragglers in distributed stream processing with tuple scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2021–2034, 2021.

[4] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024.

[5] J. Tan, Z. Tang, W. Cai, W. J. Tan, X. Xiao, J. Zhang, Y. Gao, and K. Li, "A cost-aware operator migration approach for distributed stream processing system," *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 441–454, 2025.

[6] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo, "Runtime adaptation of data stream processing systems: The state of the art," *ACM Computing Surveys*, vol. 54, 2022.

[7] M. Barika, S. Garg, A. Y. Zomaya, and R. Ranjan, "Online scheduling technique to handle data velocity changes in stream workflows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 2115–2130, 2021.

[8] H. Li, J. Xia, W. Luo, and H. Fang, "Cost-efficient scheduling of streaming applications in apache flink on cloud," *IEEE Transactions on Big Data*, vol. 9, no. 4, pp. 1086–1101, 2023.

[9] A. Muhammad, M. Aleem, and M. A. Islam, "Top-storm: A topology-based resource-aware scheduler for stream processing engine," *Cluster Comput*, vol. 24, pp. 417–431, 2021.

[10] M. Farrokh, H. Hadian, M. Sharifi, and A. Jafari, "Sp-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters," *Expert Systems with Applications*, vol. 191, pp. 1–11, 2022.

[11] A. Brown, S. Garg, J. Montgomery, and U. KC, "Resource scheduling and provisioning for processing of dynamic stream workflows under latency constraints," *Future Generation Computer Systems*, vol. 131, pp. 166–182, 2022.

[12] D. Cheng, X. Zhou, Y. Wang, and C. Jiang, "Adaptive scheduling parallel jobs with dynamic batching in spark streaming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2672–2685, 2018.

[13] G. Siachamis, G. Christodoulou, K. Psarakis, M. Fragkoulis, A. van Deursen, and A. Katsifodimos, "Evaluating stream processing autoscalers," in *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*. Association for Computing Machinery, 2024, p. 110–122.

[14] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions," *ACM Computing Surveys*, vol. 53, no. 3, May 2020.

[15] J. Cai, W. Liu, Z. Huang, and F. R. Yu, "Task decomposition and hierarchical scheduling for collaborative cloud-edge-end computing," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 4368–4382, 2024.

[16] X. Wei, L. Li, X. Li, X. Wang, S. Gao, and H. Li, "Pec: Proactive elastic collaborative resource scheduling in data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1628–1642, 2019.

[17] T. Li, J. Tang, and J. Xu, "Performance modeling and predictive scheduling for distributed stream data processing," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 353–364, 2016.

[18] S. Tuli, S. S. Gill, P. Garraghan, R. Buyya, G. Casale, and N. R. Jennings, "Start: Straggler prediction and mitigation for cloud computing environments using encoder lstm networks," *IEEE Transactions on Services Computing*, vol. 16, no. 1, pp. 615–627, 2023.

[19] R. Bitar, M. Wootters, and S. El Rouayheb, "Stochastic gradient coding for straggler mitigation in distributed learning," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 277–291, 2020.

[20] S. Tuli, N. Basumatary, S. S. Gill, M. Kahani, R. C. Arya, G. S. Wander, and R. Buyya, "Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated iot and fog computing environments," *Future Generation Computer Systems*, vol. 104, pp. 187–200, 2020.

[21] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 101–116, 2022.

[22] G. Rosinosky, D. Schmitz, and E. Rivière, "Streambed: Capacity planning for stream processing," in *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*. Association for Computing Machinery, 2024, p. 90–102.

[23] H. Xu, P. Liu, S. T. Ahmed, D. Da Silva, and L. Hu, "Adaptive fragment-based parallel state recovery for stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2464–2478, 2023.

[24] L. Zhang, W. Zheng, K. Zheng, H. Zhu, C. Li, and M. Guo, "Bayesian-driven automated scaling in stream computing with multiple qos targets," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 7, pp. 1251–1267, 2024.

[25] J. Li, H. Peng, Y. Cao, Y. Dou, H. Zhang, P. S. Yu, and L. He, "Higher-order attribute-enhancing heterogeneous graph neural networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 560–574, 2023.

[26] X. Bai, Y. Ye, B. Zhang, and S. S. Ge, "Efficient package delivery task assignment for truck and high capacity drone," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 11, pp. 13 422–13 435, 2023.

[27] X. Bai, W. Yan, and M. Cao, "Clustering-based algorithms for multivehicle task assignment in a time-invariant drift field," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2166–2173, 2017.

[28] X. Tang, Y. Liu, Z. Zeng, and B. Veeravalli, "Service cost effective and reliability aware job scheduling algorithm on cloud computing systems," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1461–1473, 2023.

[29] A. Ali, A. Sharafian, H. Yasir Naeem, M. Zakarya, Z. Wu, and X. Bai, "Advanced computational models for urban traffic flow prediction: A comprehensive review and future directions," *Computer Science Review*, vol. 60, p. 100886, 2026.

[30] R. Ali, A. Ali, H. M. Y. Naeem, M. Asad, T. Alsarhan, and B. B. Heyat, "A comprehensive survey of deep learning-based traffic flow prediction models for intelligent transportation systems," *ICCK Transactions on Advanced Computing and Systems*, vol. 1, no. 3, pp. 117–137, 2024.

[31] A. Ali, I. Ullah, S. Ahmad, Z. Wu, J. Li, and X. Bai, "An attention-driven spatio-temporal deep hybrid neural networks for traffic flow prediction in transportation systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 26, no. 9, pp. 14 154–14 168, 2025.

[32] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Computing Surveys*, vol. 52, no. 2, 2019.

[33] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, vol. 21, no. 3, pp. 940–954, 2022.

[34] G. Brauwers and F. Frasincar, "A general survey on attention mechanisms in deep learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 3279–3298, 2023.

[35] B. Donovan and D. Work, "New york city taxi trip data (2010-2013)," 2016. [Online]. Available: https://doi.org/10.13012/J8PN93H8

[36] 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, pp. 1–22, 2017.

[37] "Apache storm," https://storm.apache.org/releases/2.4.0/Storm-Scheduler.html, 2022.

[38] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, 2015, p. 149–161.

[39] C. Zheng, X. Fan, S. Pan, H. Jin, Z. Peng, Z. Wu, C. Wang, and P. S. Yu, "Spatio-temporal joint graph convolutional networks for traffic forecasting," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 1, pp. 372–385, 2024.

Minghui Wu is a PhD student at the School of Information Engineering, China University of Geosciences, Beijing, China. He received his Bachelor's Degree in Network Engineering from Zhengzhou University of Aeronautics, Zhengzhou, China in 2020. His research interests include big data stream computing, distributed systems, and blockchain.

Dawei Sun is a Professor in the School of Information Engineering, China University of Geosciences, Beijing, P.R. China. He received his Ph.D. degree in computer science from Northeastern University, China in 2012, and conducted the Postdoctoral research in the department of computer science and technology at Tsinghua University, China in 2015. His current research interests include big data computing, cloud computing and distributed systems. In these areas, he has authored over 100 journal and conference papers.

Shang Gao received her Ph.D. degree in computer science from Northeastern University, China in 2000. She is currently a Senior Lecturer in the School of Information Technology, Deakin University, Geelong, Australia. Her current research interests include distributed system, cloud computing and cyber security.

Keqin Li is a SUNY Distinguished Professor at the State University of New York and a National Distinguished Professor at Hunan University (China). He has authored or co-authored more than 1290 journal articles, book chapters, and refereed conference papers. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked #2) and career-long impact (ranked #4) based on a composite indicator of the Scopus citation database. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an IEEE Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a Member of the European Academy of Sciences and Arts. He is a Member of Academia Europaea (Academician of the Academy of Europe).

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He has authored over 750 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 172 with 158,900+ citations). He is among the world' s top 2 most influential scientists in distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He served as the founding Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and now serving as EiC of Journal of Software: Practice and Experience.