Linearize, Predict and Place: Minimizing the Makespan for Edge-based Stream Processing of Directed Acyclic Graphs

Shweta Khare
Vanderbilt University
Nashville, TN
shweta.p.khare@vanderbilt.edu

Hongyang Sun
Vanderbilt University
Nashville, TN
hongyang.sun@vanderbilt.edu

Julien Gascon-Samson
ETS Montreal
Montreal, Quebec
julien.gascon-samson@etsmtl.ca

Kaiwen Zhang
ETS Montreal
Montreal, Quebec
kaiwen.zhang@etsmtl.ca

Aniruddha Gokhale
Vanderbilt University
Nashville, TN
a.gokhale@vanderbilt.edu

Yogesh Barve
Vanderbilt University
Nashville, TN
yogesh.barve@vanderbilt.edu

Anirban Bhattacharjee
Vanderbilt University
Nashville, TN
anirban.bhattacharjee@vanderbilt.edu

Xenofon Koutsoukos
Vanderbilt University
Nashville, TN
xenofon.koutsoukos@vanderbilt.edu

ABSTRACT

Many IoT applications found in cyber-physical systems, such as smart grids, must take control actions in response to critical events, such as supply-demand mismatch, which requires low-latency processing of streaming data for rapid event detection and anomaly remediation. These streaming applications generally take the form of directed acyclic graphs (DAGs), where vertices represent operators and edges represent the flow of data between these operators. Edge computing has recently attracted significant attention as a means to readily meet the requirements of latency-critical IoT applications due to its ability to provide low-latency processing near the source of data. To accrue the benefits of edge computing, the constituent operators of these applications must be placed in a manner that intelligently trades-off inter-operator communication costs with the cost of interference incurred due to co-location of operators on the same resource-constrained edge devices. To address these challenges and to substantially simplify the placement problem for DAGs of arbitrary sizes and topologies, we present an algorithm that first transforms any arbitrary stream processing DAG into an approximate set of linear chains. Subsequently, a data-driven latency prediction model for co-located linear chains is used to inform the placement of operators such that the makespan, defined as the maximum latency of all paths in the DAG, is minimized. We empirically evaluate our algorithm using a variety of DAG placement scenarios on a Beagle Bone cluster, which is representative of an edge computing environment.

SEC '19, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363315

CCS CONCEPTS

• **Theory of computation → Streaming models**; • **Computer systems organization → Embedded and cyber-physical systems**; • **Computing methodologies → Distributed computing methodologies**;

KEYWORDS

Stream Processing, Edge Computing, Latency Minimization, Latency Prediction, Operator Placement

ACM Reference Format:
Shweta Khare, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Aniruddha Gokhale, Yogesh Barve, Anirban Bhattacharjee, and Xenofon Koutsoukos. 2019. Linearize, Predict and Place: Minimizing the Makespan for Edge-based Stream Processing of Directed Acyclic Graphs. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318216.3363315

1 INTRODUCTION

The Internet of Things (IoT) paradigm has enabled a large number of physical devices or "things" equipped with sensors and actuators to connect over the Internet to exchange information. IoT applications typically involve continuous processing of data streams produced by these devices for the control and actuation of intelligent systems. In most cases, such processing needs to happen in near real-time to gain insights and detect patterns of interest. For example, in smart grids, energy usage data from millions of smart meters is continuously assimilated to identify critical events, such as demand-supply mismatch, so that corrective action can be taken to maintain grid stability [50]. Similarly, in video surveillance systems, video streams are continuously analyzed to detect traffic violations, such as jay walking and collisions [30].

Distributed Stream Processing Systems (DSPS) are used for scalable and continuous processing of data streams, such as sensor data streams produced by IoT applications [49]. In DSPS, an application

is structured as a Directed Acyclic Graph (DAG), where vertices represent operators that process incoming data and directed edges represent the flow of data between operators. The operators perform user-defined computations on the incoming stream(s) of data. Storm [51], Spark [54], Flink [14], Millwheel [5], etc. are examples of widely used DSPSs. These systems have, however, been designed for resource-rich, cloud/cluster environments, where a master node distributes both data and computation over a cluster of worker nodes for large-scale processing (e.g., as in Storm). Using such cloud-centric solutions for IoT applications will require transferring huge amounts of sensor data produced by devices at the network edge to data-centers located at the core of the network [44]. However, moving data over a bandwidth-constrained backhaul network incurs high latency cost, which makes such cloud-centric solutions infeasible for latency-sensitive IoT applications.

To address this concern, the edge computing paradigm has been proposed [46] to enable computations to execute near the source of data on low-cost edge devices and small-scale data-centers called cloudlets [47]. Edge-based stream processing systems, such as Frontier [40], Amazon Greengrass [2], Microsoft Azure IoT [3] and Apache Edgent [1], support data stream processing on multiple edge devices thereby reducing the need for costly data transfers. However, to meet the low response time requirements of latency-sensitive applications, it is also important to distribute the constituent operators of the DSPS over resource-constrained edge devices intelligently. An optimal placement approach should minimize the end-to-end response time or *makespan* of a stream processing DAG while trading-off communication costs incurred due to distributed placement of operators across edge devices, and interference costs incurred due to co-location of operators on the same edge device [15].

The above-mentioned edge-based stream processing platforms, however, provide only the mechanisms for IoT stream processing but not the solution for optimal operator placement. As such, framework-specific solutions for operator placement [34, 35, 42] are not directly applicable for edge-based stream processing. Likewise, existing framework-agnostic solutions [13, 15, 27, 28, 52] for operator placement make simplifying assumptions about the interference costs of co-located operators. These solutions do not consider the impact of incoming data rates and DAG-based execution semantics on the response time of an application. Due to these simplifying assumptions, their estimation of response time for DAG execution is less accurate and the produced placement of operators on the basis of this response time estimation is less effective.

To address these limitations, in this paper, we present a data-driven latency prediction model which takes the impact of DAG-based execution semantics and data rate into consideration to predict the latency of all paths in a DAG. Subsequently, this latency prediction model is used by a greedy operator placement algorithm to inform the placement of operators such that the makespan of the DAG is minimized.

Learning a latency prediction model for arbitrary DAG structures, however, has significantly high overhead in terms of computational costs and model training time. Moreover, formulating the model training problem itself is very complex. Therefore, our solution transforms a DAG into an approximate set of linear chains, which makes learning a latency prediction model for co-located operators significantly less expensive and easier to construct than learning a model for arbitrary DAG structures. Accordingly, to estimate the latency of a path in a given DAG, we first linearize the DAG into multiple linear chains and then use the latency prediction model for co-located linear chains to approximate the response time of the path in the original DAG. It is important to note that the original DAG structure is what gets executed using our solution. DAG linearization is only used for approximating path latencies in the original DAG structure so as to guide the operator placement decisions of the greedy placement algorithm.

Our paper makes the following key contributions:

- **DAG Linearization:** We present an algorithm that transforms any given DAG into an approximate set of linear chains in order to approximate the latency of a path in the DAG. This set includes the target path of the DAG, whose latency we are interested in approximating, in addition to other mapped linear chains. Upon the execution of this set of linear chains, the latency of the path we are interested in is observed to be very close to the measured latency along that path when the original DAG structure is executed. The transformation algorithm considers both the split (or fork), and join (or merge) points in DAGs.
- **Latency Prediction Model:** We present a model for predicting the 90th percentile latency of a linear chain of operators on the basis of its length (i.e., number of operators in the linear chain), the incoming data rate, the sum of execution times of all operators in the linear chain and a characterization of background load imposed by other co-located linear chains. For higher accuracy, we learn a separate prediction model for different numbers of co-located chains present at an edge device. All the learned models presented in this paper have a prediction accuracy of at least 92%.
- **Greedy Placement Heuristic:** We present a greedy placement heuristic for makespan minimization, which leverages the DAG linearization algorithm and the latency prediction model to guide its placement decisions. Experimental results show that, compared with two baseline approaches, our placement heuristic significantly reduces the prediction error while achieving low makespan.

The rest of this paper is organized as follows: Section 2 motivates our data-driven approach for estimating the impact of operator co-location on path latencies. Section 3 gives a formal statement of the problem we are studying and provides a greedy heuristic to solve it. Section 4 presents our approach for DAG linearization and the latency prediction model to estimate the 90th percentile latency of co-located linear chains. These predictions are needed for our greedy heuristic. Section 5 presents experimental results to validate our solution. Section 6 presents related work and compares our operator placement solution to existing solutions for makespan minimization. Finally, Section 7 offers concluding remarks, lessons learned and outlines future work.

2 IMPACT OF OPERATOR CO-LOCATION

Existing solutions make simplifying assumptions to estimate the cost of interference due to operator co-location. Some solutions [27, 28, 52] assume that the execution time of each operator becomes the

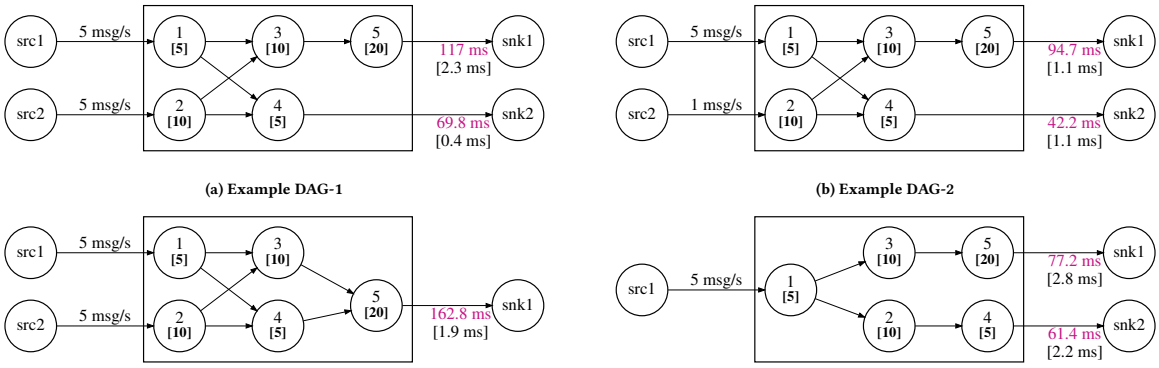


Figure 1: Impact of publication rate and DAG structure on latency

sum of execution times of all co-located operators when the underlying physical node is a single core and uses round robin scheduling. Other solutions [13, 15] ignore the impact of operator co-location and assume constant execution time. These solutions do not consider the impact of incoming data rate and DAG structure imposed execution semantics, both of which have a significant impact on the observed latency. We illustrate this using some representative examples in Figure 1.

We used a single core Beagle Bone Black (BBB) board [4] to run the DAGs depicted in Figure 1. All intermediate vertices which process incoming data, namely `vertex-1` to `vertex-5`, were hosted on the same BBB board while the source and sink vertices were hosted on a separate 2.83 GHz Intel Q9550 quad-core server. Source vertices send 64 Byte, time-stamped messages periodically at a configurable rate shown on their outgoing edges. Intermediate vertices perform a configurable amount of processing on each incoming message. The isolated execution time of an intermediate vertex, measured in milliseconds, is depicted within brackets below the `vertex-ID`. For intermediate vertices with multiple incoming edges, we assume interleaving semantics [28] wherein the vertex performs processing whenever it receives a message on any of its incoming edges. Sink vertices log the time-stamp of reception of messages after being processed by the intermediate vertices.

Each DAG was executed for two minutes in an experimental run. Average 90th percentile latency and standard deviation (shown in brackets) recorded by each sink vertex across 5 runs are shown along the incoming edge of the sink vertex. This implies that 90 percent of all messages received along all the paths which end at a sink vertex were observed to have an end-to-end path latency below the 90th percentile value. For example, in DAG-1 (Figure 1a), 90 percent of all messages received along paths $\langle src1, 1, 3, 5, snk1\rangle$ and $\langle src2, 2, 3, 5, snk1\rangle$, which end at *snk1*, were observed to have an end-to-end latency below 117 ms (on average across 5 experimental runs). Similarly, 90 percent of all messages received along paths $\langle src1, 1, 4, snk2\rangle$ and $\langle src2, 2, 4, snk2\rangle$, which end at *snk2*, were observed to have an end-to-end latency below 69.8 ms (again on average across 5 runs). The makespan (i.e., response time) of a DAG is the maximum 90th percentile latency across all paths, which is 117 ms for DAG-1.

Based on our experiments, we made the following critical observations:

- **Impact of data rate (i.e., publishing rate):** DAG-1 in Figure 1a and DAG-2 in Figure 1b are the same, except for the publishing rate of source vertex *src2*, which generates data at 5 messages/sec in DAG-1 and at 1 message/sec in DAG-2. In DAG-1, the 90th percentile latency at sink vertex *snk1* is 117 ms and at sink vertex *snk2* is 69.8 ms. However, due to the lower publishing rate of *src2* in DAG-2, sink vertices *snk1* and *snk2* show lower 90th percentile latencies of 94.7 ms and 42.2 ms, respectively.
- **Impact of DAG Structure:** DAG-1, DAG-3 and DAG-4 in Figure 1a, Figure 1c and Figure 1d, respectively, are composed of the same set of intermediate vertices, although they are structurally different. All three DAGs show markedly different response times on account of this difference in their DAG structures. Simplifying assumptions that do not consider DAG structure imposed execution semantics, such as assuming constant execution time or sum of execution times of all co-located vertices, can respectively underestimate or overestimate a DAG's makespan. For example, if we assume constant execution time for each vertex, the latency of path $\langle src1, 1, 3, 5, snk1\rangle$ in DAG-4 will be ~35 ms, which is much less than the observed path latency of 77.2 ms. Similarly, if we assume each vertex's execution time to be the sum of execution times of all 5 co-located vertices given that we are using a single core BBB board, then the path latency would be ~150 ms, which is twice the experimentally observed path latency of 77.2 ms.
- **Performance Interference:** The latency of path $\langle src1, 1, 3, 5, snk1\rangle$ in DAG-4 (i.e., 77.2 ms) is higher than a simple sum of the execution times of all co-located operators (i.e., 50 ms), since all 5 operators are executing on the same single core BBB and the overhead of context switching is not negligible. On constrained edge devices, impact of performance interference on latency is more pronounced and simple analytical models fail to take this into account. In practice, estimating the impact of performance interference due to co-location of applications is a hard problem, for which no

good analytical models are known to exist [6, 19, 21]. This is further compounded by the heterogeneity of hardware and software components deployed in IoT environments.

Thus, to accurately estimate the impact of co-location of operators on path latencies, we have relied on using a data-driven approach which takes both data rate and DAG-based execution semantics into account.

3 PROBLEM FORMULATION AND HEURISTIC SOLUTION

In this section, we formally describe the operator placement problem by first introducing the models and assumptions. We then demonstrate the complex trade-offs between communication and interference induced costs, and show the complexity. Finally, we present a greedy heuristic solution for the problem.

3.1 Models and Assumptions

A stream processing application can be represented by a Directed Acyclic Graph (DAG) $\mathbb{G} = (\mathbb{O}, \mathbb{S})$, where the set of operators $\mathbb{O} = \{o_i\}$ form the vertices of $\mathbb{G}$ and the set of data streams $\mathbb{S} = \{s_{ij}\}$, connecting the output of an operator o_i to its downstream operator o_j, form the directed edges of $\mathbb{G}$. Source operators, $\mathbb{O}_{src} \subset \mathbb{O}$, do not have any incoming edges and publish data into $\mathbb{G}$, i.e., $\mathbb{O}_{src} = \{o_i | \nexists s_{ji} \in \mathbb{S}, o_j \in \mathbb{O}\}$. Sink operators, $\mathbb{O}_{snk} \subset \mathbb{O}$, do not have any outgoing edges and receive the final results of $\mathbb{G}$, i.e., $\mathbb{O}_{snk} = \{o_i | \nexists s_{ij} \in \mathbb{S}, o_j \in \mathbb{O}\}$. All source and sink operators are *no-op* operators, i.e., they do not perform any computation. Each intermediate operator, i.e., $\mathbb{O}_{int} = \{o_i | o_i \notin \mathbb{O}_{src}, o_i \notin \mathbb{O}_{snk}\}$, performs CPU intensive computation and is characterized by its: 1) *execution time*, $\rho(o_i)$, which defines the average time interval of processing that o_i performs on every input message, and 2) *incoming rate*, $\lambda(o_i)$, which defines the rate at which o_i receives incoming messages. An intermediate operator with multiple incoming streams follows interleaving semantics [28] for data processing and it will perform its computation whenever it receives a message on any of its incoming streams.

The problem requires finding a *placement* $\mathcal{P} : \mathbb{O}_{int} \rightarrow \mathbb{E}$ for the set of intermediate operators $\mathbb{O}_{int}$ over a cluster of homogeneous edge nodes $\mathbb{E} = \{e_j\}$, such that the *makespan* of $\mathbb{G}$, specified by its maximum end-to-end latency[1] is minimized. Formally, the makespan of a graph $\mathbb{G}$ under a placement $\mathcal{P}$ is defined as:

$$\ell_{\mathcal{P}}(\mathbb{G}) = \max_{p \in \Pi} \ell_{\mathcal{P}}(p) \tag{1}$$

where Π represents the set of all paths in $\mathbb{G}$ and $\ell_{\mathcal{P}}(p)$ represents the latency of a path $p \in \Pi$ under placement $\mathcal{P}$. Suppose the path p has n intermediate operators, i.e., $p = \langle o_s, o_1, o_2, \ldots, o_n, o_k \rangle$, where $o_s \in \mathbb{O}_{src}$, $o_k \in \mathbb{O}_{snk}$, and $o_i \in \mathbb{O}_{int}$ for $1 \leq i \leq n$. Given a placement $\mathcal{P}$, the latency of path p can be expressed as:

$$\ell_{\mathcal{P}}(p) = \sum_{i=1}^{n} \omega_{\mathcal{P}}(o_i) + \sum_{i=1}^{n-1} d_{\mathcal{P}}(o_i, o_{i+1}) \tag{2}$$

Here, $\omega_{\mathcal{P}}(o_i)$ denotes the processing delay experienced by an operator o_i under placement $\mathcal{P}$, which may be higher than the operator's isolated execution time $\rho(o_i)$ due to potential interference with other co-located operators [37, 39] (see Section 2). Typically, the more co-located operators on the same edge node, the higher the processing delay. Also, $d_{\mathcal{P}}(o_i, o_{i+1})$ denotes the communication delay between an upstream operator o_i and its downstream operator o_{i+1} in the path. If o_i and o_{i+1} are placed on the same edge node under $\mathcal{P}$, then no network delay will be incurred. Otherwise, we assume a constant network delay c between any two edge nodes:

$$d_{\mathcal{P}}(o_i, o_{i+1}) = \begin{cases} 0 & \text{if } \mathcal{P}(o_i) = \mathcal{P}(o_{i+1}) \\ c & \text{if } \mathcal{P}(o_i) \neq \mathcal{P}(o_{i+1}) \end{cases}$$

Our model makes the following assumptions:

(1) All intermediate operators perform CPU intensive computations; we do not consider other resources, such as memory, I/O, etc.
(2) All intermediate operators perform their computations on each incoming message; we do not consider window-based operations.
(3) All intermediate operators follow OR/interleaving [28] semantics for data processing; we do not consider AND semantics, which encode more complex interactions between multiple incoming data streams (e.g., join).
(4) All the edge nodes are homogeneous.
(5) Number of available edge nodes is not a constraint.
(6) Network delay between any two edge nodes is constant.

3.2 Cost Trade-off and Complexity

The optimal solution to the makespan minimization problem described above depends on delicately exploiting the trade-off between the communication costs incurred by dependent operators located on different edge nodes and the interference cost due to the co-location of multiple operators on the same edge nodes. For example, consider a linear chain of n operators, i.e., $\langle o_1, o_2, \ldots, o_n \rangle$. On the one hand, placing each operator separately on different edge nodes has zero interference, but incurs a large communication cost between each pair of consecutive operators. On the other hand, placing all operators on one edge node incurs zero communication cost, but incurs a large processing delay due to performance interference among the co-located operators.

It turns out that, to place a set of n operators that form a linear chain, an optimal solution that balances the two costs can be obtained by the following dynamic programming formulation:

$$\ell_i^* = \min_{i \leq j \leq n} \left(\omega_{i,j} + d(o_j, o_{j+1}) + \ell_{j+1}^* \right) \tag{3}$$

where ℓ_i^* denotes the optimal latency for placing the sub-chain $\langle o_i, o_{i+1}, \ldots, o_n \rangle$, and $\omega_{i,j} = \sum_{h=i}^{j} w(o_h)$ denotes the cumulative latency of all operators in the sub-chain $\langle o_i, o_{i+1}, \ldots, o_j \rangle$ when they are co-located on the same edge node.

For placing general DAGs, however, the problem is more difficult. Many prior works (e.g., [13, 15, 22]) have shown the NP-hardness of the problem when there is a limited number of edge nodes (i.e., a *resource-constraint* optimization problem). Here, we consider a model in which the number of edge nodes is unrestricted and the primary objective is to minimize the response time of the streaming service. Even in this case, the problem can be shown to be

[1] While the model is flexible to incorporate different definitions of latency, we consider the 90th percentile end-to-end latency in this paper.

NP-hard via a simple reduction from a multiprocessor scheduling problem with communication delays[2] [29, 41]. Hence, we will focus on designing a heuristic based solution in this paper.

3.3 Greedy Placement Heuristic

We now present a greedy placement heuristic for the makespan minimization problem formulated in Section 3.1. Algorithm 1 shows the pseudocode of the greedy heuristic. Specifically, the heuristic places the operators in the intermediate set $\mathbb{O}_{int}$ one after another in a Breadth-First Search (BFS) order (Line 2), which preserves the spatial locality of the dependent operators, thus reducing the communication cost. For each operator o_i to be placed, the heuristic tries two different options:

(1) Co-locate o_i with other operators that have already been placed on an existing edge node (Lines 6-17);
(2) Place o_i on a new edge node (Lines 18-23).

In both options, the latencies of all paths that go through operator o_i will be estimated. In the first option, this is done using our DAG linearization scheme and latency prediction model described in Section 4 (Line 11). In the second option, this is done by simply adding the isolated execution time $\rho(o_i)$ of the operator (Line 19), since it is not co-located with any other operators. Empirically, we found that co-locating too many operators or having too many paths on a single edge node can lead to degraded accuracy in the latency prediction (see Section 5). Hence, we restrict the maximum number of co-located operators on a node to be m and the maximum number of paths formed by these operators to be k (Line 10). In the experimental evaluation, we set $m = 12$ and $k = 4$.

Note that, when operator o_i is co-located with other operators on an edge node, the latencies of all paths that go through those co-located operators also need to be estimated due to the interference caused by the placement of o_i [20]. Then, the resulting partial makespan for the sub-graph $\mathbb{G}_i = (\mathbb{O}_i, \mathbb{S}_i)$ that contains the set of operators $\mathbb{O}_i = \{o_1, o_2, \ldots, o_i\}$ up to operator o_i and the set of associated data streams $\mathbb{S}_i = \{s_{jk} | o_j \in \mathbb{O}_i, o_k \in \mathbb{O}_i\}$ will be updated (Line 12 and Line 20). Finally, the option that results in the minimum predicted makespan for $\mathbb{G}_i$ is selected for placing operator o_i (Line 27).

Since each edge node hosts at least one operator, Algorithm 1 deploys at most n edge nodes in the end, where $n = |\mathbb{O}_{int}|$ denotes the number of intermediate operators in the graph. On each edge node, enumerating up to k paths from a DAG consisting of at most m co-located operators can be done by simple Depth-First Search (DFS) with backtracking, which has complexity $O(m^2k)$. Furthermore, the linearization of the DAG and the prediction of latencies for all paths can be done in $O(m^2 + k^2m)$ time (see Algorithm 2). As the placement of each operator examines at most n edge nodes, the overall complexity of the algorithm is therefore $O(n^2mk(m + k))$. As m and k are usually set to be constants, the heuristic essentially computes a placement solution in $O(n^2)$ time.

[2]The multiprocessor scheduling problem concerns mapping an arbitrary task graph with communication delays onto a set of m identical processors in order to minimize the makespan. The problem is NP-hard even when $m = \infty$ and all communication costs are uniform [29, 41]. This corresponds to a special case of our problem without any performance interference due to co-located operators, thus establishing the NP-hardness of the problem.

Algorithm 1: Greedy Placement Heuristic

Input: Operator graph $\mathbb{G} = (\mathbb{O}, \mathbb{S})$
Output: A placement $\mathcal{P}_{\text{greedy}}$ of the intermediate operator set $\mathbb{O}_{int} \in \mathbb{O}$ onto a set $\mathbb{E}$ of edge nodes

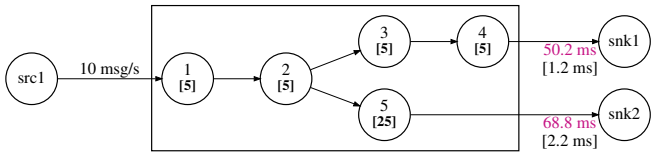
1 **begin**
2 Reorder all operators in the intermediate set $\mathbb{O}_{int} = \{o_1, o_2, \ldots, o_n\}$ in BFS order, where $n = |\mathbb{O}_{int}|$;
3 $\mathbb{E} \leftarrow \emptyset$;
4 **for** each operator o_i $(i = 1 \ldots n)$ **do**
5 $\ell^* \leftarrow \infty$ and $j^* \leftarrow 0$;
// try to place on each existing edge node
6 **for** each edge node $e_j \in \mathbb{E}$ **do**
7 $\mathcal{P}(o_i) \leftarrow e_j$;
8 $m_j \leftarrow$ getNumberOfOperators(e_j);
9 $k_j \leftarrow$ computeNumberOfPaths(e_j);
10 **if** $m_j \leq m$ *and* $k_j \leq k$ **then**
11 Predict latencies of all paths that contain operators co-located in e_j using latency prediction model (Section 4);
12 Compute partial makespan $\ell_{\mathcal{P}}(\mathbb{G}_i)$ using Equation (1);
13 **if** $\ell_{\mathcal{P}}(\mathbb{G}_i) < \ell^*$ **then**
14 $\ell^* \leftarrow \ell_{\mathcal{P}}(\mathbb{G}_i)$ and $j^* \leftarrow j$;
15 **end**
16 **end**
17 **end**
// try to place on a new edge node
18 $\mathcal{P}(o_i) \leftarrow e_{|\mathbb{E}|+1}$;
19 Update latencies of all paths that contain operator o_i by adding isolated execution time $\rho(o_i)$;
20 Compute partial makespan $\ell_{\mathcal{P}}(\mathbb{G}_i)$ using Equation (1);
21 **if** $\ell_{\mathcal{P}}(\mathbb{G}_i) < \ell^*$ **then**
22 $\ell^* \leftarrow \ell_{\mathcal{P}}(\mathbb{G}_i)$ and $j^* \leftarrow j$;
23 **end**
24 **if** $j^* = |\mathbb{E}| + 1$ **then**
25 $\mathbb{E} \leftarrow \mathbb{E} \bigcup \{e_{|\mathbb{E}|+1}\}$; // start a new edge node
26 **end**
27 $\mathcal{P}_{\text{greedy}}(o_i) \leftarrow e_{j^*}$;
28 **end**
29 **end**

4 A DATA-DRIVEN LATENCY PREDICTION MODEL

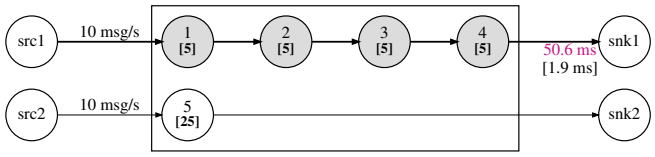
In this section, we describe the development of a data-driven latency prediction model that is used by our greedy placement heuristic (see Section 3.3). To that end, we explain our DAG linearization approach for transforming any arbitrary DAG into an approximate set of linear chains and the k-chain co-location latency prediction model which is subsequently used for predicting the latency of multiple co-located linear chains.

4.1 DAG Linearization Transformation Rules

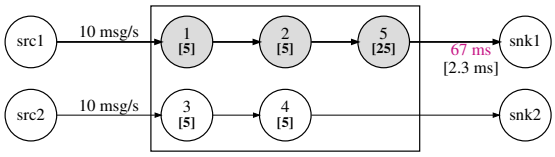
It is expensive to train a latency prediction model for arbitrary DAGs, since the structure of the DAGs need to be taken into account, which could explode the exploration space. To overcome this problem, we propose a linearization-based approach, which transforms any given DAG into multiple sets of linear chains, whose latencies will then be predicted to approximate the end-to-end latencies of paths in the original DAG. Due to the simplicity of the linear structures, the proposed approach is able to significantly reduce the space over which the latency prediction model needs to be learned. We arrived at these transformation rules based on multiple different empirical observations.



(a) Original DAG with fork operator

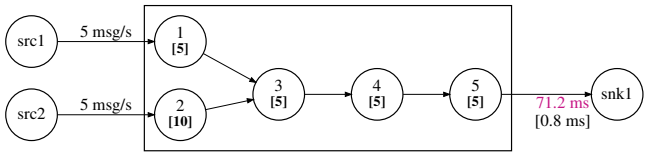


(b) Linearization for path-1

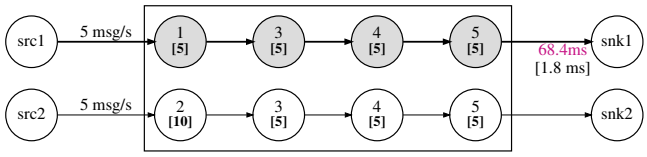


(c) Linearization for path-2

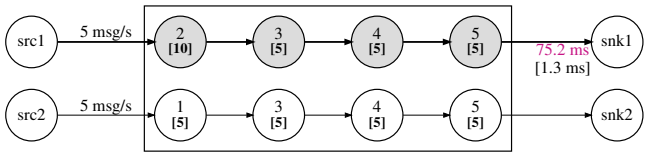
Figure 2: Linearization rule for fork operator



(a) Original DAG with join operator



(b) Linearization for path-1



(c) Linearization for path-2

Figure 3: Linearization rule for join operator

Suppose up to m operators in a connected graph[3] $\mathbb{G}'$ are all co-located on one edge node, and $\mathbb{G}'$ contains a collection $\{p_1, p_2, \ldots, p_f\}$ of f paths from its source operator(s) to its sink operator(s), where $f \leq k$. The linearization scheme first transforms graph $\mathbb{G}'$ into f sets of linear chains, denoted as $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$. For each $1 \leq x \leq f$, the set $L_x(\mathbb{G}')$ contains f linear chains in it, including a target linear chain corresponding to the path p_x in the original DAG, as well as $f-1$ auxiliary linear chains to simulate the performance interference for path p_x. The latency prediction model (Section 4.2) is then used to predict the latency of the target path p_x in each set $L_x(\mathbb{G}')$. Finally, the predicted latencies for all the paths in $\{p_1, p_2, \ldots, p_f\}$ that share the same sink operator are averaged to approximate the end-to-end latency for messages that exit that sink operator in graph $\mathbb{G}'$.

We now present our approach to transform graph $\mathbb{G}'$ into f sets of linear chains $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$. Algorithm 2 shows the pseudocode of the linearization procedure. Since an operator in the original graph $\mathbb{G}'$ may be replicated in a set $L_x(\mathbb{G}')$, we first compute the number of times each operator is replicated. To that end, we describe below the linearization rules for two types of operators in a DAG structure, namely, fork and join operators.

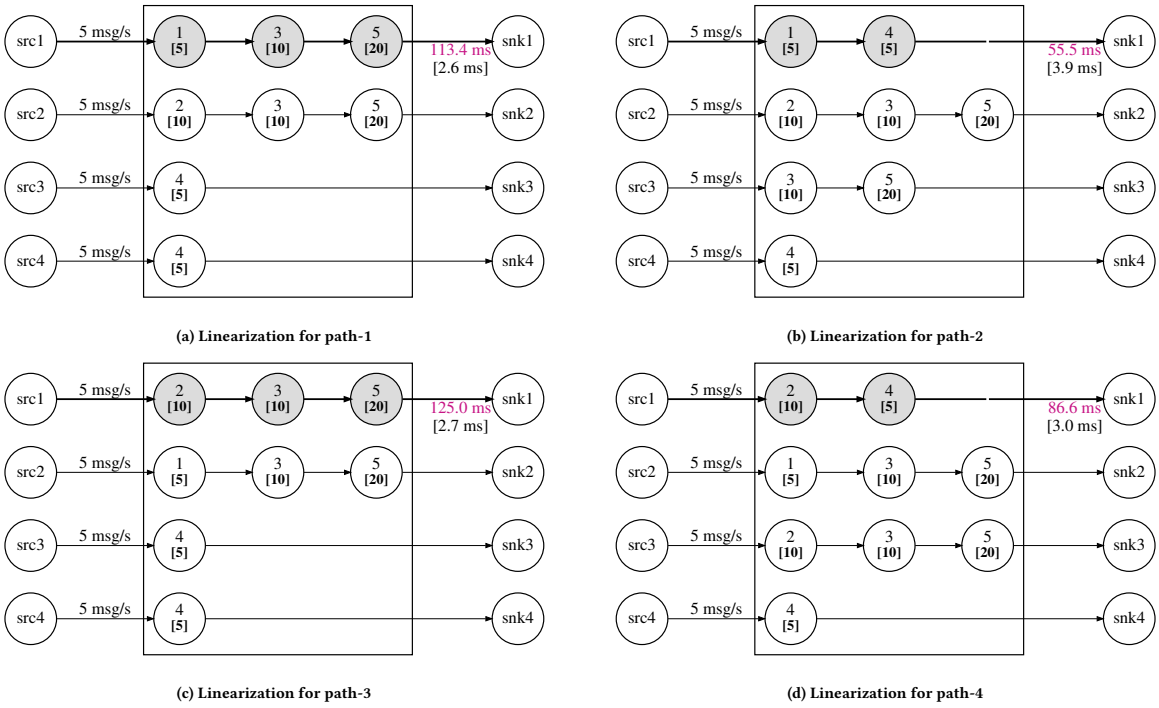
- *Fork operator*: All paths in a DAG that originate from a fork operator can be executed concurrently. Hence, we can reason about these paths as independent linear chains. The fork operator only executes once, so it is included in one of the multiple paths that originate from the fork vertex. Figure 2 illustrates this rule.
- *Join operator*: Join operators have multiple incoming edges. Therefore, we can argue that the join operator and all its downstream operators execute as many times as the number of incoming edges of the join vertex. Hence, a join operator and all downstream operators are replicated into multiple linear chains. Figure 3 illustrates this rule.

Generalizing the rules above, for each operator o_i in $\mathbb{G}'$, we can set the number of times t_i it should appear in any set $L_x(\mathbb{G}')$ of linear chains as the total number of paths from the source operator(s) of $\mathbb{G}'$ to o_i. This can be computed by a simple breadth-first traversal of the graph (Lines 2-12), which takes $O(|\mathbb{O}'| + |\mathbb{S}'|) = O(m^2)$ time.

The algorithm then constructs the set $L_x(\mathbb{G}')$ of linear chains for each $1 \leq x \leq f$ (Lines 13-27). Specifically, it first adds the target path p_x into the set $L_x(\mathbb{G}')$, and then examines the operators from the remaining paths in sequence. If an operator o_i has already appeared t_i times from the previously examined paths, it will be removed from the current and subsequent paths. This ensures the correct number of replicas for each operator in the set. Since there are f sets of linear chains, and the construction of each set examines all f paths, each containing at most $|\mathbb{O}'|$ operators, the complexity of this part is $O(f^2|\mathbb{O}'|) = O(k^2 m)$. The overall complexity of the algorithm is therefore $O(m^2 + k^2 m)$.

Figure 4 shows the linearization results for the DAG shown in Figure 1a, which contains two source operators, two sink operators, and four different paths: $\langle 1, 3, 5\rangle$, $\langle 2, 3, 5\rangle$, $\langle 1, 4\rangle$ and $\langle 2, 4\rangle$. Therefore, four corresponding sets of linear chains are constructed as shown in Figure 4a to Figure 4d. In each set, the chain highlighted in grey represents the target path whose latency will be predicted, and the other chains represent auxiliary paths to simulate the performance interference.

[3]If a graph contains several connected components, the linearization can be done separately for each connected component.



(a) Linearization for path-1

(b) Linearization for path-2

(c) Linearization for path-3

(d) Linearization for path-4

Figure 4: Linearization and latency prediction results for the DAG shown in Figure 1a.

Algorithm 2: DAG Linearization

Input: Operator graph $\mathbb{G}' = (\mathbb{O}', \mathbb{S}')$ that contains f paths $\{p_1, p_2, \ldots, p_f\}$ from its source operator(s) to its sink operator(s).
Output: f sets of linear chains $\{L_1(\mathbb{G}'), L_2(\mathbb{G}'), \ldots, L_f(\mathbb{G}')\}$, each with a target path p_x from $\mathbb{G}'$ whose latency will be predicted ($1 \leq x \leq f$).

begin
  Identify the set $\mathbb{O}'_{src}$ of source operators of $\mathbb{G}'$;
  $t_i \leftarrow 1$ for $\forall o_i \in \mathbb{O}'_{src}$, and $t_i \leftarrow 0$ for $\forall o_i \in \mathbb{O}' \backslash \mathbb{O}'_{src}$;
  Initialize an empty queue $Q \leftarrow \emptyset$;
  $Q.enqueue(\mathbb{O}'_{src})$;
  while $Q \neq \emptyset$ **do**
    $o_i \leftarrow Q.dequeue()$;
    for each $o_j \in o_i.childen()$ **do**
      $t_j \leftarrow t_j + 1$;
      $Q.enqueue(o_j)$;
    end
  end
  for $x = 1$ to f **do**
    $p'_h \leftarrow p_h$, for $\forall 1 \leq h \leq f$;
    $t'_i \leftarrow t_i$, for $\forall o_i \in \mathbb{O}'$;
    $t'_i \leftarrow t'_i - 1$, for $\forall o_i \in p'_x$;
    for each $p'_h \in \{p'_1, p'_2, \ldots, p'_f\} \backslash \{p'_x\}$ **do**
      for each $o_i \in p'_h$ **do**
        if $t'_i = 0$ **then**
          remove o_i from p'_h;
        else
          $t'_i \leftarrow t'_i - 1$;
        end
      end
    end
    $L_x(\mathbb{G}') \leftarrow \{p'_1, p'_2, \ldots, p'_f\}$;
  end
end

4.2 Training the *k*-chain Co-location Latency Prediction Model

In this section, we describe the k-chain co-location latency prediction model we trained for predicting the latencies of k co-located linear chains. Given a set of k linear chains, our latency prediction model first employs a classification model to determine if the placement of these k linear chains at an edge device is feasible; i.e., the placement does not saturate an edge node's resources. In case of resource saturation, the observed latency values become significantly high and unpredictable. If the classification model predicts that the placement is feasible, then a regression model is used to predict each linear chain's 90th percentile latency.

Both the classification and regression models for a k-chain co-location take the same set of 7 input features. Of these 7 input features, the first 3 features characterize the foreground (target) linear chain, or the chain under observation, and the remaining 4 features characterize the background load imposed by the set of background (auxiliary) chains co-located along with the foreground chain. These input features are described below, where c_f denotes the foreground linear chain and $\mathbb{C}_B$ denotes the set of background linear chains.

- $n(c_f)$: number of operators in c_f;
- $\sum_{o \in c_f} \rho(o)$: sum of execution intervals of operators in c_f;
- $\lambda(c_f)$: incoming data rate for c_f;
- $\sum_{c_b \in \mathbb{C}_B} n(c_b)$: sum of number of operators in all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \sum_{o \in c_b} \rho(o)$: sum of execution intervals of all operators in all background chains;

- $\sum_{c_b \in \mathbb{C}_B} \lambda(c_b)$: sum of incoming data rates over all background chains;
- $\sum_{c_b \in \mathbb{C}_B} \lambda(c_b) \cdot \sum_{o \in c_b} \rho(o)$: sum of the product of $\lambda(c_b)$ and $\sum_{o \in c_b} \rho(o)$ over all background chains.

The classification model takes these input features and outputs a binary value: 0 to indicate that the placement is feasible and 1 otherwise. The regression model outputs the predicted 90th percentile latency of the foreground chain co-located with a given background load. For $k = 1$, i.e., only one chain exists, the four input features characterizing the background load are set to 0. Latency prediction models are specific to a hardware type and a separate set of prediction models will need to be learned for each new hardware type on which an operator can be placed.

We used neural networks for learning both the classification and regression models. For higher accuracy, we learned separate models for different numbers k of co-located chains. The neural networks comprise an input layer, one or more hidden layers and an output layer, where each layer is composed of nodes called neurons [12]. Neurons belonging to different layers are interconnected with each other via weighted connections. The input layer feeds the input features to the network. Neurons belonging to the intermediate and output layers sum the incoming weighted signals, apply a bias term and an activation function to produce their output for the next layer. The architecture of a neural network, i.e., the number of hidden layers and number of neurons per hidden layer, choice of the activation function, regularization factor, and the solver greatly determine the accuracy of the learned model.

Typically, learning curves [38] are plotted to understand the performance of a neural network and to guide the selection of various parameters, such as number of layers, number of neurons per layer and regularization factor. A learning curve shows the training and validation errors as functions of the training data size. If the learning curve shows that the training error is low, but the validation error is high, the model is said to suffer from high variance [38], i.e., it is over-fitting the training data and may not generalize well. If the learning curve shows high training and validation errors, the model is said to suffer from high bias [38], i.e., it fails to learn from the data or is under-fitting the data.

A neural network model for which both the training and validation errors converge to a low value is selected. Such a model neither over-fits (low variance) nor under-fits (low bias) the data and is expected to perform well. We use the learning curves for different neural network architectures for each value of k to help us select a model with low bias and variance. Section 5 shows the learning curves and accuracy results for the selected k-chain co-location classification and regression models.

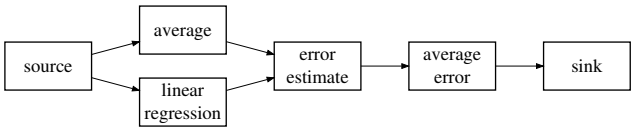
5 EXPERIMENTAL VALIDATION

In this section, we present experimental results to validate our DAG linearization-based approach for predicting the latency of arbitrary DAG structures and to evaluate our greedy placement heuristic for minimizing the DAG makespan, which relies on the latency prediction approach. We describe our experiment testbed first, followed by the accuracy results for the learned k-chain co-location latency prediction models and performance results for our greedy placement heuristic.

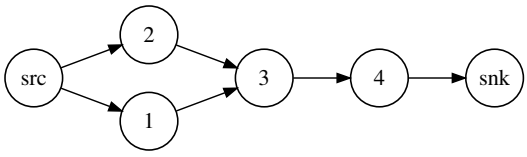
5.1 Experiment Testbed and Setup

Our experiment testbed comprises 8 Beagle Bone Black (BBB) boards running Ubuntu 18.04, where each board has one AM335x 1GHz ARM processor, 512 MB RAM and 1 Gb/s network interface card. Intermediate vertices are hosted on BBB boards whereas the source and sink vertices are hosted on a separate quad-core 2.83 GHz Intel Q9550 server with 8GB RAM and 1Gb/s network interface card, also running Ubuntu 18.04. We assume a constant network delay of 10 ms between any two BBBs and also between each BBB and the server hosting source/sink vertices. We used RTI DDS [10], a peer-to-peer topic-based publish-subscribe messaging system to model the directed edges interconnecting the vertices. Each edge is implemented as a DDS topic over which messages are sent and received by the upstream and downstream vertices, respectively.

Similar to some prior work on operator placement [27, 28], we have also used synthetic DAGs for testing our operator placement algorithm. We generated our synthetic test DAGs[4] using Fan-In-Fan-Out [18] method for task-graph generation. Since we observed that the out-degree of operators in some real world applications like extract, transform and load (ETL), statistical summarization, predictive analytics and model training, as benchmarked by RIOTBench [48] is between 1 and 3, we set the maximum out-degree for our Fan-In-Fan-Out algorithm to 2. Figure 5a shows an example application DAG on predictive analytics adapted from RIOTBench. Here, data from a source vertex is forked to linear regression and average operators, which predict a numerical attribute value and calculate the moving average of observed attribute value respectively. Subsequently, the error estimation operator calculates the error residue between predicted and observed values. Finally, average prediction error is reported to the sink. Figure 5b shows a synthetic DAG generated by our Fan-In-Fan-Out algorithm which is similar in structure to the example application DAG.



(a) Example predictive analytics DAG (adapted from RIOTBench [48])



(b) Synthetic DAG

Figure 5: Example application and synthetic DAG

RIOTBench [48] has also micro-benchmarked several categories of stream processing operators, such as those used for parsing, filtering, statistical processing, prediction and IO operations, by using real world workloads and applications. Their micro-benchmarks show that several operators have latencies up to 10 ms. Similarly, micro-benchmarks conducted by StreamBench [36] show commonly used state-less CPU intensive operators such as projection,

[4] All test DAGs used in our experiments can be viewed at http://bit.ly/sampledags

sampling, grep, etc. have latency up to 10 ms. Therefore, to generate our synthetic workload for testing, the execution interval (ρ) of intermediate vertices is randomly chosen to be 1 ms, 5 ms, 10 ms, 15 ms or 20 ms. An intermediate vertex processes a message by performing recursive fibonacci computations for the randomly chosen execution interval. Source vertices send 64 Byte time-stamped messages periodically at a configurable publishing rate λ up to 20 messages/second. Due to limited processing capability of BBB device, we kept the publishing rates low. Sink vertices log the time-stamp of reception of processed messages to compute the end-to-end latencies. To ensure the fidelity of experimental results, a DAG is executed for two minutes. Some initial end-to-end latency values logged by a sink vertex are ignored while computing the 90th percentile latency value, since they are observed to be high on account of initialization and connection setup.

5.2 Validating the *k*-chain Co-location Latency Prediction Model

As discussed in Section 4, to predict the end-to-end path latency of k co-located linear chains, we rely on two prediction models: k-chain co-location classification and k-chain co-location regression models. First, the classification model is used to assess if the placement of given k linear chains is feasible. If the classification model predicts that the placement is feasible, then the k-chain co-location regression model is used to predict each linear chain's 90th percentile latency.

Empirically, we observed that a BBB's CPU gets saturated if more than 12 vertices are placed on the node. Therefore, to create the training dataset for k-chain co-location models, the number of vertices in each linear chain is randomly chosen such that the sum of number of vertices across all k chains is not more than 12 (i.e., $m = 12$). The execution interval ρ for each vertex is uniformly randomly chosen from the set {1ms, 5ms, 10ms, 15ms, 20ms} and the incoming data rate for each chain is uniformly randomly chosen in the range [1, 20] messages/sec. We learned k-chain co-location models for k up to 4. As k increases further, the range of possible values for the input features increases and more training data is needed to get good prediction accuracy. We ran 600, 1500, 1950 and 1950 experiments for $k = 1, 2, 3$ and 4, respectively. Additionally, a separate validation dataset was created by running 50 experiments for each k. Each experiment took ~3 minutes to execute. Therefore, it took ~13 days to collect the training and validation dataset.

When an experiment for k-chain co-location is run, we get k latency data-points/samples, one corresponding to each linear chain. Therefore, the training dataset size becomes k times the number of experiments, i.e., 600, 3000, 5850 and 7800 samples for $k = 1, 2, 3$ and 4, respectively, as shown in Table 1. To learn the classification model, data-points for which the observed 90th percentile latency is greater than twice the sum of execution intervals of all vertices in all k chains are categorized as infeasible placements. While the entire dataset comprising both feasible and infeasible data-points is used for training the classification model, the regression model is trained only over the feasible subset of data-points. For example, in case of $k = 1$, all 600 data-points are used for training the classification model. Out of these, 186 data-points were categorized as infeasible placements and the remaining 416 data-points as feasible placements. To learn the regression model for $k = 1$, we therefore used the 416 feasible data-points as shown in Table 2.

Table 1: Accuracy of *k*-chain co-location classification model

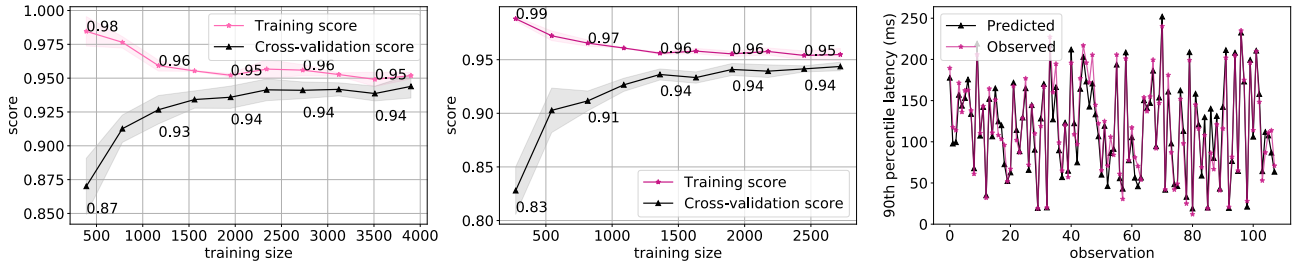
k	#samples (train)	accuracy (train)	accuracy (test)	#samples (validation)	accuracy (validation)
1	600	.98	.97	50	.98
2	3000	.98	.96	100	.98
3	5850	.96	.96	150	.97
4	7800	.96	.95	200	.94

Table 2: Accuracy of *k*=chain co-location regression model

k	#samples (train)	accuracy (train)	accuracy (test)	#samples (validation)	accuracy (validation)
1	416	.99	.99	38	.99
2	2268	.98	.98	84	.96
3	4083	.96	.95	108	.94
4	5376	.95	.94	128	.92

We tested different neural network architectures for k-chain co-location classification and regression models. For classification, we found that a neural network with one hidden layer composed of 50 neurons performed well for $k = 1$ and $k = 2$, while a neural network with one hidden layer composed of 100 neurons performed well for $k = 3$ and $k = 4$. For regression, we found that a neural network with one hidden layer composed of 50 neurons performed well for $k = 1$. However, for $k = 2, 3$ and 4, a neural network regressor with three hidden layers composed of 50 neurons each gave good accuracy. For all the models, Rectified Linear Units (ReLu) was used as the activation function, limited memory Broyden-Fletcher-Goldfarb-Shanno (lbfgs) was used as the solver and L2 regularization factor was set to 0.1. Figure 6a and Figure 6b show the learning curves for $k = 3$ classification and $k = 3$ regression models, respectively. Here, we see that the chosen neural network architectures have low bias and variance since the training and cross-validation errors converge to a reasonably low value that is at most 8%. Therefore, the models neither over-fit nor under-fit the training data and are expected to generalize well.

Table 1 and Table 2 show the performance of trained k-chain co-location classification and regression models on training, test and validation datasets. We used 90% of data-points for training and the remaining 10% for testing. We observed that all learned models have an accuracy of at least 92%. Figure 6c shows the performance of k-chain co-location regression model on the validation dataset for $k = 3$. We see that the predicted latencies track the experimentally observed values closely and the average difference between the predicted and observed latencies over all 108 validation data-points is 10.8 ms.

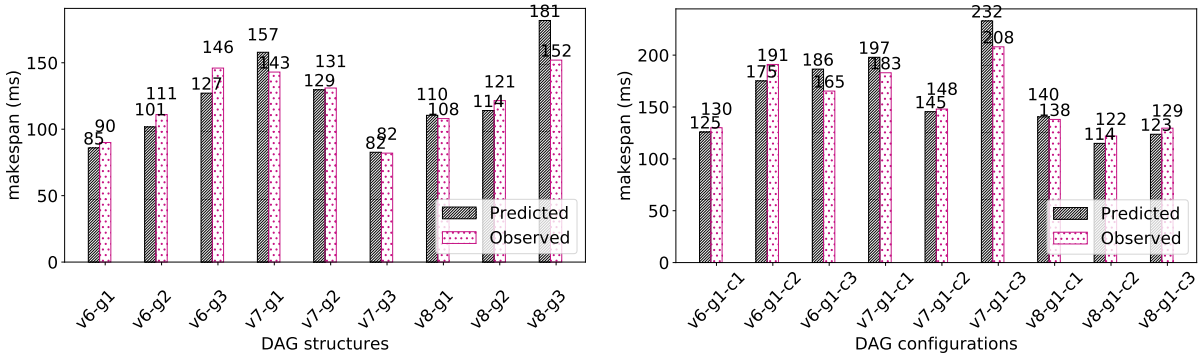


(a) Classification learning curve for $k = 3$

(b) Regression learning curve for $k = 3$

(c) Validation results for $k = 3$

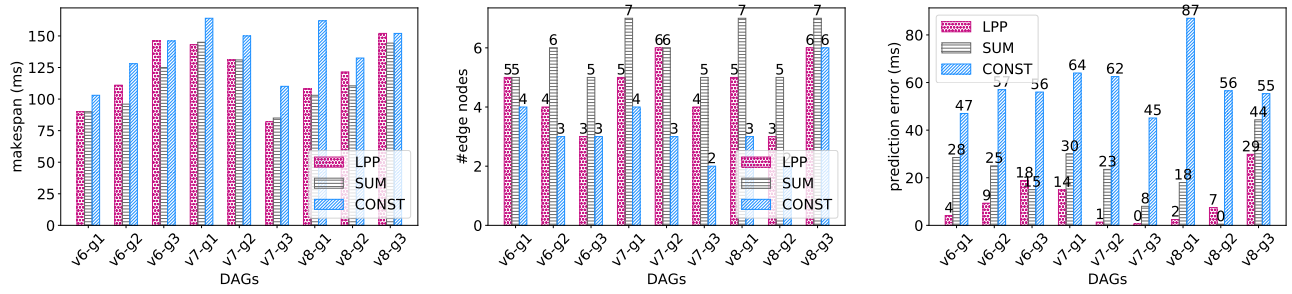
Figure 6: Performance of k-chain co-location latency prediction model for $k = 3$



(a) LPP predicted vs. observed makespan for different DAG structures

(b) LPP predicted vs. observed makespan for different DAG configurations

Figure 7: LPP makespan prediction accuracy (for various randomly generated DAGs)



(a) Observed makespan

(b) Number of edge nodes

(c) Prediction error

Figure 8: Comparison of LPP with SUM and CONST approaches (for various randomly generated DAGs)

5.3 Performance Evaluation of the LPP Approach

To assess the performance of our Linearize, Predict and Place (LPP) operator placement solution for makespan minimization, we generated nine random test DAGs with three different structures per intermediate vertex count of 6, 7 and 8 vertices. We refer to the DAG structure 1 with 6 intermediate vertices as `v6-g1`, which also exemplifies the naming convention used to identify these test DAGs.

LPP Prediction Results: Figure 7a compares the makespan predicted by LPP with experimentally observed makespan upon DAG execution, for the same test DAG structure, parameter configuration (data rate and execution intervals of all vertices) and operator placement. We see that the LPP approach based on DAG linearization is able to make a fairly good prediction for the makespan of all 9 test DAGs with an average prediction error of 9.8 ms. For the same DAG structure, different DAG parameter configurations, such as incoming data rate and execution intervals of the constituent vertices, also impact DAG latency. Figure 7b shows the variation in the makespan of the same DAG structure across three different DAG configurations *c1*, *c2* and *c3* (which are also randomly generated). The LPP approach incorporates these differences in DAG parameter configurations while predicting a DAG's makespan. As seen in Figure 7b, LPP is able to make good predictions for different parameter configurations with a mean prediction error of 11 ms across these 9 DAGs.

LPP Placement Results: We compared LPP with two solution variations: (1) SUM: Similar to LPP, this approach uses the concept of DAG linearization to approximate arbitrary DAG structures and the k-chain co-location classification model to assess if the placement of k linear chains is feasible. However, unlike the LPP approach, which uses k-chain co-location regression model to predict the path latencies, the SUM approach makes the simplifying assumption that a vertex's execution time becomes the sum of all co-located vertices' execution times; and (2) CONST: Similar to LPP, this approach uses the concept of DAG linearization to approximate arbitrary DAG structures and the k-chain co-location classification model to assess if the placement of k linear chains is feasible, however, to estimate path latencies, this approach makes the simplifying assumption that a vertex's execution time remains unchanged despite co-location.

Figure 8a compares the makespan of the placements produced by the LPP, SUM and CONST approaches. Figure 8b compares the number of edge nodes used in the placements produced by the LPP, SUM and CONST approaches. Figure 8c shows the error in predicting the makespan of a DAG by the LPP, SUM and CONST approaches. We see that, in many cases, the CONST approach underestimates the path latencies and inaccurately favors co-locating the vertices, which results in a higher makespan in comparison to LPP as seen in Figure 8a. The placement produced by CONST uses fewer edge nodes than LPP as seen in Figure 8b since CONST inaccurately favors co-location due to underestimation of path latencies. The makespan predicted by CONST is also much lower than the observed makespan upon DAG execution, which results in high prediction errors as seen in Figure 8c.

The SUM approach, on the other hand, overestimates path latencies and inaccurately favors distributed placement of operators in many cases, which results in more edge nodes being used in comparison to LPP, as seen in Figure 8b. Due to such overestimation, the error in makespan prediction by the SUM approach is higher than that of the LPP approach, which uses a latency prediction model, as seen in Figure 8c. The observed makespans produced by LPP and SUM are similar as seen in Figure 8a, but LPP achieves this with less amount of edge resources. Overall, these results show that LPP makes a more accurate prediction of the path latencies and thereby, a more effective placement of operators than the other approaches that make simplifying (either overestimating or underestimating) assumptions to estimate the cost of interference.

6 RELATED WORK

In this section, we compare our work to the related work along several key dimensions, including operator placement for makespan reduction, graph transformations, operator placement at the edge, and degenerate forms of DAG placement at the edge, all of which are key considerations in our work.

Operator Placement for DAG Makespan Minimization: The operator placement problem has been studied extensively in the literature [32]. Existing solutions have varied objectives, such as minimizing network use [43, 45], minimizing inter-node traffic [53], minimizing the makespan or response time of an operator graph [13, 27, 28, 52]. Although similar in spirit to these works, the minimization of response time in our case refers to finding an appropriate operator placement that will satisfy an application's service level objective for the response time of the longest path in the DAG. More importantly, however, to the best of our knowledge, existing works on makespan minimization do not consider the impact of operator co-location and hence the interference effects on response time, while our solution expressly considers such an impact.

Operator Graph Transformation: In [16], authors leverage the technique of operator replication to provide better performance for processing incoming data streams. Apart from replication, the authors also propose an algorithm for placement of these operators on the runtime platform. Similar graph transformation using operator replication strategy has been applied in [33]. In contrast to these works, we use DAG transformations as a means to simplify model learning.

In [23], the authors decompose a series-parallel-decomposable (SPD) graph into a SPD tree structure. Here each leaf of the SPD tree corresponds to the nodes of the SPD graph. Moreover, each internal node of the tree corresponds to the serial or parallel composition. The authors further provided a theoretical analysis to the issue of resource assignment to the application graph. The focus of our work is different from this work in that we do not consider any series-parallel decomposition but rather a workflow of operators that sequentially process data from one stage to the next in a pipeline.

Edge-Based Operator Placement: A generic formulation of the operator placement problem has been presented in [15]. The authors show how their formulation can be used for optimizing different QoS parameters, such as response time, availability, network use, etc. They formulate the problem as an integer linear optimization problem and use the CPLEX solver to find an optimal placement. However, they do not consider the impact of co-location on an operator's execution time. In their formulation, an operator i is assumed to take r_i seconds to process a data sample irrespective of whether it is co-located with other operators.

In [13], the authors have proposed a greedy algorithm for operator placement with the objective of minimizing the end-to-end response time of operator graphs/DAGs. They have used a queueing theory-based model for estimating the response time of paths in a DAG. However, their model also does not consider the impact of co-location. Although we also propose a greedy heuristic, our work has not utilized queuing theory but instead uses a data-driven trained model for prediction. We believe that a data-driven model captures real-world behaviors unlike a queuing model, which is analytical and can ignore many of the system-level effects. Yet, for future work we plan to use analytical models to prove the correctness of our graph transformation algorithms.

In [17], authors implemented a distributed QoS-aware scheduler by extending the default Storm application, which has a centralized scheduler. It allows to dynamically reconfigure the operator placement under uncertain changes in environmental conditions such as network dynamics. Our work currently does not incorporate uncertainty quantification which remains a dimension of future work; however, our work is tailored to edge computing scenarios.

A heuristic-based solution for operator placement across a 3-tier edge-fog-cloud heterogeneous infrastructure has been proposed in [26]. The heuristics allows enforcing operator placement constraints on the available nodes, and also enforces the co-locations of certain operators on the same nodes. This allows for minimizing placement costs and maximizing the resource utilization. In

comparison to this work, we do not consider heterogeneous resources and it will be important to extend our work to cover the range of edge-fog-cloud resources. Unlike this work, our goal is to place a DAG on available resources with the aim of makespan minimization.

In [7], authors present an optimization framework that formulates the placement of operators across cloud and edge resources using constraint satisfaction and a system model. The goal of the optimization problem is to minimize the end to end latency. The evaluation of the proposed scheme is, however, conducted through a simulation study implemented using OMNET++ simulator. In contrast, we have validated our research on an actual IoT testbed.

DROPLET [24] presented an operator placement problem using the shortest path problem, in which the operators are placed in such a fashion that it minimizes the total completion time of the graph. Although this goal is similar to ours, we believe that this work and several of the other works outlined above have not considered the performance interference issue which can result due to resource contention happening among the participating operators.

Latency Minimization for Publish/Subscribe Systems: A publish-subscribe system which involves some processing at an intermediate broker is a degenerate form of a DAG that we consider in our work. There are some recent works that consider minimizing end-to-end latencies for such degenerate DAG topologies. For instance, in [31], the authors present an approach to minimize end-to-end latencies for competing publish-process-subscribe chains and balancing the topic loads on intermediate brokers hosted in edge computing environments. Unlike this work which focuses on only a 3-node chain and focuses more on balancing the load on the brokers, our work focuses on the placement of competing, arbitrary DAGs comprising a workflow of stream processing operators on edge resources. MultiPub [25] is an effort to find the optimal placement of topics across geographically distributed datacenters for ensuring per-topic 90th percentile latency of data delivery. Although we are also interested in 90th percentile latencies, this work considers only inter-datacenter network latencies and does not consider edge networks.

7 CONCLUSIONS

With the growing importance of the Internet of Things (IoT) paradigm, there is increasing focus on realizing a range of smart applications that must ingest the continuous streams of generated data from different sources, process the data within the stream processing application which is often structured as a directed acyclic graph (DAG) of operators, and make informed decisions in near real-time. The low latency response time requirements of these applications require that the computations of the DAG operators be performed closer to the data sources, i.e., on the IoT edge devices. However, resource constraints on these devices require careful considerations for multi-tenancy, i.e., co-location of operators on the edge devices, which must be done in a way that minimizes the DAG makespan, i.e., the end-to-end latency for the longest path in the DAG.

To that end, this paper presents an optimization problem formulation for DAG makespan minimization. The NP-hardness of the general problem and the need to realize an efficient, runtime deployment decision engine inform the design of an efficient greedy heuristic. Our heuristic-based solution makes its operator placement decisions by using a look-ahead scheme wherein the algorithm predicts the potential impact on a DAG's makespan if a specific operator co-location decision is made. To aid in this look-ahead phase of our placement algorithm, we present a data-driven latency prediction model, which is a machine learning model that is trained using empirical data generated from conducting a variety of operator co-location experiments on an edge computing testbed. A novel trait of the prediction model is the use of linearized chains of operators created using a transformation algorithm to closely approximate the performance of the original DAG structures. In summary, we presented a novel *Linearize-Predict-Place (LPP)* approach for DAG makespan minimization. Our empirical results comparing LPP with two separate baselines called SUM and CONST reveal that LPP makes a more accurate prediction of path latencies and thereby a more effective placement of operators than the SUM and CONST approaches, which otherwise make simplifying assumptions to estimate the cost of interference caused due to co-location.

There are many dimensions along which additional work will need to be performed as informed by the assumptions we made and the limitations of our experiment test-bed:

- In the current work, we have focused primarily on CPU computations alone when making the co-location decisions. Other resources, such as memory and I/O, also need to be accounted for as has been presented in earlier studies [9]. Additionally, the linearization transformation for join operators currently assumes interleaving, i.e., OR semantics. The rules need to be extended when the join operators require AND semantics.
- We have not modeled the number of available edge devices as a constraint in our optimization problem and have assumed constant network delays between edge nodes. Since in real world IoT deployments, the number of available edge devices may be limited and networks may not be stable, it is important to consider these aspects in our problem formulation.
- Our edge computing testbed is made up of homogeneous resource types (i.e., BBB boards with a single core), but IoT can illustrate significant heterogeneity in resource type, and it will be important to explore this dimension.
- The scale of our experiments and the size of the DAGs considered in this work is limited, and moreover, the applications used were synthetic. Using larger-scale and real-world IoT applications and applying easier deployment mechanisms such as those presented in UPSARA [8] and Stratum [11] is part of our future work.

ACKNOWLEDGMENTS

This work is supported in part by NSF US Ignite CNS 1531079 and AFOSR DDDAS FA9550-18-1-0126. 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 NSF or AFOSR.

REFERENCES

[1] [n. d.]. Apache Edgent. http://edgent.apache.org/.

[2] [n. d.]. AWS IoT Greengrass. https://aws.amazon.com/greengrass/.

[3] [n. d.]. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/.

[4] [n. d.]. Beagle Bone Black. https://beagleboard.org/black.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*. 734–746.

[6] Y. Amannejad, D. Krishnamurthy, and B. Far. 2015. Managing Performance Interference in Cloud-Based Web Services. *IEEE Transactions on Network and Service Management* 12, 3 (Sep. 2015), 320–333. https://doi.org/10.1109/TNSM.2015.2456172

[7] Gayashan Amarasinghe, Marcos D de Assunçao, Aaron Harwood, and Shanika Karunasekera. 2018. A Data Stream Processing Optimisation Framework for Edge Computing Applications. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 91–98.

[8] Yogesh Barve, Shashank Shekhar, Shweta Khare, Anirban Bhattacharjee, and Aniruddha Gokhale. 2018. UPSARA: A Model-driven Approach for Performance Analysis of Cloud-hosted Applications. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 1–10.

[9] Y. D. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, and A. Gokhale. 2019. FECBench: A Holistic Interference-aware Approach for Application Performance Modeling. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 211–221. https://doi.org/10.1109/IC2E.2019.00035

[10] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini. 2013. Data Distribution Service (DDS): A performance comparison of OpenSplice and RTI implementations. In *2013 IEEE Symposium on Computers and Communications (ISCC)*. 000377–000383. https://doi.org/10.1109/ISCC.2013.6754976

[11] Anirban Bhattacharjee, Yogesh Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. 2019. Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*. 59–61.

[12] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[13] Xinchen Cai, Hongyu Kuang, Hao Hu, Wei Song, and Jian Lü. 2018. Response Time Aware Operator Placement for Complex Event Processing in Edge Computing. In *International Conference on Service-Oriented Computing*. Springer, 264–278.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[15] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/2933267.2933312

[16] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review* 44, 4 (2017), 11–22.

[17] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2015. On QoS-aware scheduling of data stream applications over fog computing infrastructures. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*. IEEE, 271–276.

[18] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. 2010. Random graph generation for scheduling simulations. *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques* (2010). https://doi.org/10.4108/ICST.SIMUTOOLS2010.8667

[19] C. Delimitrou and C. Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 23–33. https://doi.org/10.1109/IISWC.2013.6704667

[20] Christina Delimitrou and Christos Kozyrakis. 2013. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 23–33.

[21] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[22] R. Eidenbenz and T. Locher. 2016. Task allocation for distributed stream processing. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524433

[23] Raphael Eidenbenz and Thomas Locher. 2016. Task allocation for distributed stream processing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[24] Tarek Elgamal, Atul Sandur, Phuong Nguyen, Klara Nahrstedt, and Gul Agha. 2018. DROPLET: Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 1–8.

[25] J. Gascon-Samson, J. Kienzle, and B. Kemme. 2017. MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2075–2082.

[26] Julien Gedeon, Michael Stein, Lin Wang, and Max Muehlhaeuser. 2018. On scalable in-network operator placement for edge computing. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[27] Rajrup Ghosh, Siva Prakash Reddy Komma, and Yogesh L. Simmhan. 2018. Adaptive Energy-Aware Scheduling of Dynamic Event Analytics Across Edge and Cloud Resources. *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2018), 72–82.

[28] Rajrup Ghosh and Yogesh Simmhan. 2018. Distributed Scheduling of Event Analytics Across Edge and Cloud. *ACM Trans. Cyber-Phys. Syst.* 2, 4, Article 24 (July 2018), 28 pages. https://doi.org/10.1145/3140256

[29] J.A. Hoogeveen, J.K. Lenstra, and B. Veltman. 1994. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters* 16, 3 (1994), 129–137.

[30] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Victor Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters.

[31] Shweta Khare, Hongyang Sun, Kaiwen Zhang, Julien Gascom-Samson, Aniruddha Gokhale, and Xenofon Koutsoukos. 2018. Scalable Edge Computing Architectures for Low Latency Data Dissemination in Topic-based Publish/Subscribe. In *3rd ACM/IEEE Symposium on Edge Computing (SEC)*. Bellevue, WA, USA, 214–227.

[32] G. T. Lakshmanan, Y. Li, and R. Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* 12, 6 (Nov 2008), 50–60. https://doi.org/10.1109/MIC.2008.129

[33] Geetika T Lakshmanan, Ying Li, and Rob Strom. 2010. Placement of replicated tasks for distributed stream processing systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, 128–139.

[34] T. Li, J. Tang, and J. Xu. 2016. Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing. *IEEE Transactions on Big Data* 2, 4 (Dec 2016), 353–364. https://doi.org/10.1109/TBDATA.2016.2616148

[35] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free Control for Distributed Stream Data Processing Using Deep Reinforcement Learning. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 705–718. https://doi.org/10.14778/3199517.3199521

[36] R. Lu, G. Wu, B. Xie, and J. Hu. 2014. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 69–78. https://doi.org/10.1109/UCC.2014.15

[37] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 237–250.

[38] Andrew Ng. [n. d.]. Learning Curves. https://www.coursera.org/lecture/machine-learning/learning-curves-Kont7.

[39] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 219–230. http://dl.acm.org/citation.cfm?id=2535461.2535489

[40] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. 2018. Frontier: resilient edge processing for the internet of things. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1178–1191.

[41] Christos H. Papadimitriou and Mihalis Yannakakis. 1990. Towards an Architecture-independent Analysis of Parallel Algorithms. *SIAM J. Comput.* 19, 2 (1990), 322–328.

[42] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 149–161. https://doi.org/10.1145/2814576.2814808

[43] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*. 49–49. https://doi.org/10.1109/ICDE.2006.105

[44] E. G. Renart, J. Diaz-Montes, and M. Parashar. 2017. Data-Driven Stream Processing at the Edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. 31–40. https://doi.org/10.1109/ICFEC.2017.18

[45] S. Rizou, F. Durr, and K. Rothermel. 2010. Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. 1–6. https://doi.org/10.1109/ICCCN.2010.5560127

[46] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (Jan 2017), 30–39. https://doi.org/10.1109/MC.2017.9

[47] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (Oct 2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[48] Anshu Shukla, Shilpa Chaturvedi, and Yogesh L. Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29 (2017).

[49] Anshu Shukla and Yogesh Simmhan. 2017. Benchmarking Distributed Stream Processing Platforms for IoT Applications. In *Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 90–106.

[50] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna. 2013. Cloud-Based Software Platform for Big Data Analytics in Smart Grids. *Computing in Science Engineering* 15, 4 (July 2013), 38–47. https://doi.org/10.1109/MCSE.2013.39

[51] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[52] Xiaohui Wei, Xun Wei, Hongliang Li, Yuan Zhuang, and Hengshan Yue. 2018. Topology-Aware Task Allocation for Distributed Stream Processing with Latency Guarantee. In *Proceedings of the 2Nd International Conference on Advances in Image Processing (ICAIP '18)*. ACM, New York, NY, USA, 245–251. https://doi.org/10.1145/3239576.3239621

[53] J. Xu, Z. Chen, J. Tang, and S. Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*. 535–544. https://doi.org/10.1109/ICDCS.2014.61

[54] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737