TATA: Throughput-Aware TAsk Placement in Heterogeneous Stream Processing with Deep Reinforcement Learning

Xiao Huang*, Yu Jiang*, Hao Fan†, Huayun Tang†, Yiping Wang†, Jin Jin‡, Hai Wan*, and Xibin Zhao*
*Tsinghua University, Beijing, China
†ChinaBond Finance And Information Technology Co.,Ltd., Beijing, China
‡Maple Growth Research Club Institution, Toronto, Canada
*{huangx19, jiang-y20}@mails.tsinghua.edu.cn, †{fanhao, tanghy, wangyp}@chinabond.com.cn
‡Jin.jin@utoronto.ca, *{wanhai, zxb}@tsinghua.edu.cn

***Abstract*—Data Stream Processing (DSP) applications, which generate real-time analytics on continuous data flows, have become prevalent recently. For the deployment of DSP applications, task placement is an important and essential part. As determining the optimal task placement is an NP-hard problem, several efficient heuristics have been designed and Deep Reinforcement Learning (DRL) was used to train the scheduling agent. Current DRL-based approach assumes all resources including CPU, memory and networking are homogeneous. However, the available computation and network resources are heterogeneous in many scenarios. To deal with it, we devise a general DRL-based resource-aware framework, which models resources using graph embedding and attention mechanism to predict the placement. Furthermore, in order to accelerate the training process and improve the throughput, we propose an efficient throughput estimation tool, which can estimate the throughput with high accuracy. We integrated our scheduling heuristic framework into Apache Flink and conducted comprehensive testings using multiple synthetic and real DSP applications. The experimental results show that our framework increases the throughput by 64%, 42%, 29% on average respectively compared with three state-of-the-art strategies.**

***Index Terms*—Data stream processing, Task placement, Throughput, Deep reinforcement learning, Heterogeneous resources**

I. Introduction

In various industrial fields, there are many tasks using large-scale and diverse data for data-intensive decision-making. Data is produced as a stream of events, such as financial transactions, sensor measurements, etc. To extract valuable information from such a huge amount of data in a timely manner, Data Stream Processing (DSP) engines (e.g., Apache Flink [1], Apache Storm [2]) and applications have become prevalent. They aim to process data as unbounded streams continuously at any scale in a near real-time fashion.

The computational process of a DSP application is typically described through a Directed Acyclic Graph (DAG). Each vertex in DAG represents an operator that carries out a specific operation, e.g., mapping, filtering. Continuous incoming data will be manipulated by operators and transmitted through the directed edges in DAG from source to sink. In order to take full advantage of the parallelism in a DAG, DSP applications are typically deployed in distributed clusters. In such a scenario, a salient problem is to determine the computing nodes that should place and execute each operator of a DSP application, to optimize the relevant Quality of Service (QoS) attributes. This problem is known as the *operator placement problem* [3], which is a long-lasting problem because the DSP applications usually do not stop after deployment and it is difficult to make run-time adaptation without performance degradation.

However, determining the optimal operator placement is an NP-hard problem [4], [5]. Thus, many efficient heuristics have been designed to solve the placement problem within a feasible amount of time. In general, heuristics are hand-crafted according to the characteristics of a specific problem [6]. Recently, using Deep Reinforcement Learning (DRL) to train heuristics has been brought to the fore [7]–[9]. Ni [9] proposed a generalized DRL-based heuristic to solve the operator placement problem considering a fixed number of homogeneous infrastructure resources with the same computational capacity, memory and communication delay. But in fact, devices or containers which provide CPU and memory in a cluster are heterogeneous in many situations and the communication delay between resources cannot be exactly the same. Additionally, the number of available resources is constantly changing, which makes such a solution with limited merit in these scenarios.

To tackle this challenge, we propose a general DRL-based resource-aware framework considering the heterogeneity of both DSP applications and the available infrastructure resources. Firstly, we model all available resources as a graph and use graph convolutional network to encode both the DSP application's DAG and resource graph. Secondly, an RNN-based decoder is applied to predict the placement for each operator sequentially using the attention mechanism. The training process is completely offline without deployment and the model can be applied to schedule DSP applications online. All features needed by our framework can be easily estimated by users or be acquired directly in DSP engines, which makes it convenient to integrate our heuristic into DSP engines.

Throughput is one of the most critical QoS attributes for

DSP applications. However, the correlation between a placement scheme and its throughput is not intuitive. Currently, DSP applications need to be deployed in a real cluster to acquire an accurate throughput, which is very time consuming and costly. Thus, we design an efficient and accurate throughput estimation tool, which estimates throughput without real deployment, making the training process completely offline and therefore extremely fast.

Finally, we integrate our method into a modern DSP engine, Apache Flink [1], to evaluate the performance of our model and validate the throughput estimation tool in a real cluster. In particular, the main contributions of this paper are as follows:

- We present a general resource-aware framework based on graph embedding and deep reinforcement learning to solve the operator placement problem. Our model is generalizable to various DSP applications and heterogeneous resources, which is also convenient to be integrated with DSP engines, such as Flink, Storm.
- We propose an efficient algorithm to estimate throughput of a placement scheme without real deployment, which enables our framework to optimize throughput directly and speeds up the training process.
- We enhance the scheduler in Apache Flink, to support different scheduling heuristics including our method. We validate the throughput estimation tool using multiple synthetic DSP graphs and achieve 6.7% absolute deviation on average with respect to real deployment. Comparing with Flink, Storm's scheduler and Ni's method, our model achieves 64%, 42%, 29% out-performance on average on different well-known DSP topologies.

The remainder of the paper is organized as follows. Section II describes the system model and defines the task placement problem. Section III presents the resource-aware framework including the QoS estimation tool. IV provides the implementation details including training settings and Flink integration. A series of experiments are conducted in section V. We discuss the related works in section VI and conclude our work finally in section VII.

II. System Model and Problem Statement

In this section, we present the DSP model and resource model, and define the task placement problem.

A. The DSP Model

A DSP job can be represented as a DAG $G_{job} = (V_{job}, E_{job})$. Each job vertex $j \in V_{job}$ is an operator that carries out a specific operation (e.g., mapping, filtering, aggregating). Each $j \in V_{job}$ has a user-defined parallelism $j.p$, which means the number of sub tasks of j. The parallel sub tasks $t_{j,k}$ (where $k = 1, 2, \ldots, j.p$) will be created for execution.

Each edge $(j_u, j_v) \in E_{job}$ connecting j_u and j_v represents the streams flowing from j_u to j_v. We use *tuple* to describe a data item flowing in DAG. Given $(j_u, j_v) \in E_{job}$, the tasks in j_u communicates with tasks in j_v. There are two types of connections: forward connection and broadcast connection.

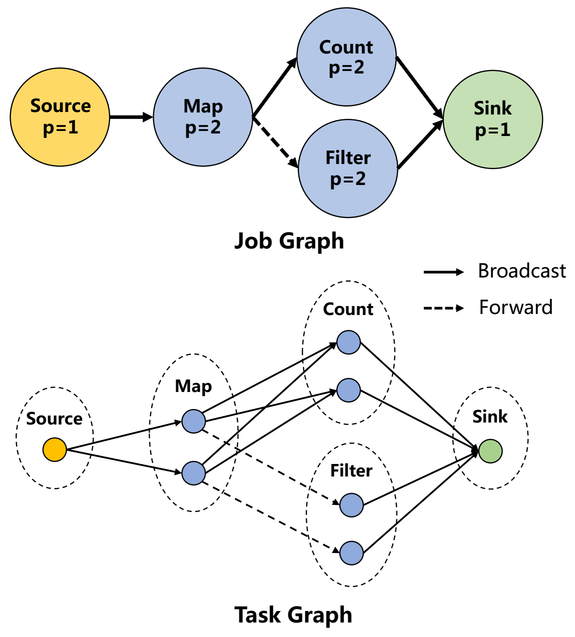


Fig. 1: Job graph and the corresponding task graph example. p in each job vertex means its parallelism.

Forward connection only occurs when j_u and j_v have the same parallelism (i.e., $j_u.p = j_v.p$) and each $t_{u,i} \in j_u$ is connected with one and only one $t_{v,i} \in j_v$. Broadcast connection means each $t_{u,i} \in j_u$ is connected with all tasks in j_v. We assume each task distributes tuples to its downstream tasks uniformly. According to the definition of parallelism and connection type, a corresponding task level graph $G_{task} = (V_{task}, E_{task})$ can be constructed based on a given job graph. An example of a job graph and the corresponding task graph is shown in Fig. 1. The map operator connects to $filter$ operator by forward connection and other job vertices are all connected by broadcast connection. Then a task graph can be obtained based on the above job graph in Fig. 1. This kind of representation is applicable in most DSP engines.

Task placement focuses on G_{task}. Each task $t \in V_{task}$, which is the smallest unit to be placed on a resource node, is characterized by two parameters: $t.cpu$ (the CPU utilization) and $t.mem$ (the required memory). CPU utilization describes the number of instructions required per tuple. The real CPU utilization value is very hard to estimate, hence relative values are used to describe it. $t.mem$ is the real value representing the maximum memory task t may consume.

B. The Resource Model

Infrastructure resources can be represented as a fully connected undirected graph $G_{res} = (V_{res}, E_{res})$. V_{res} is the set of ***slot*** which is the smallest unit to place a task. Each edge in E_{res} represents the *logical connectivity* between two slots. Each slot $s \in V_{res}$ is characterized by two parameters: $s.cpu$ (the CPU capacity) and $s.mem$ (the available memory). Similar to the CPU utilization of a task (i.e., $t.cpu$), we use relative values to characterize $s.cpu$, which is easy to estimate in DSP engines.

Each edge $(u, v) \in E_{res}$ with $s_u, s_v \in V_{res}$ is characterized by $d_{(u,v)}$, the communication delay between slot s_u and slot s_v. A slot can be assumed as a thread belonging to a process and a host contains one or more processes. Based on the locations of the slots communicating, there are 4 types of communications. Their delays are different and are list in ascending order as follows:

1) Intra-slot communication
2) Inter-slot (intra-process) communication
3) Intra-host (inter-process) communication
4) Inter-host communication

The delay values of different communication types can be configured by users according to real environments. Relative values are allowed to describe the delays. In the resource model, all required features can be easily acquired or estimated in DSP engines, which makes it convenient for the deployment of DSP applications.

C. The Task Placement Problem

Given a pair of G_{task} and G_{res}, the DSP scheduler should determine a placement scheme $\mathcal{P} : V_{task} \to V_{res}$, which is a suitable mapping between V_{task} and V_{res}. Specifically, we should place each task $t_i \in V_{task}$ on a specific slot $s_{t_i} \in V_{res}$ with the objective of optimizing some QoS metrics. In this paper, throughput is of most concern. In other words, we need to find a placement scheme which could make full advantage of the available resources and supports DSP applications with high throughput requirements.

As demonstrated in [5], task placement problem is NP-hard, which means there is no algorithm that could find the optimal solution in polynomial time. In order to deploy DSP applications efficiently, we need heuristics to predict an appropriate placement in feasible amount of time.

Tasks in task graphs in practice have different CPU utilization with a complex dependency structure. If the scheduler places all tasks on a small subset of all available slots, the communication delay is almost negligible, but the slots may be exhausted to execute so many tasks, which incurs negative impact on throughput. On the contrary, if the scheduler distributes tasks to all slots evenly, each task will have enough resources, but the communication delay can be overly large. Considering the heterogeneity of resources, even distribution is also not a good choice. Hence, an ideal scheduler must be aware of both task graphs and resource graphs to make good trade-offs between different QoS metrics and predict the appropriate placement efficiently.

III. Resource-aware Framework with DRL

In this section, we present an integrated approach to solve the task placement problem. First, we propose an encoder-decoder model to generate the placement scheme in section III-A. Then, we design the algorithm of the QoS estimation tool in section III-B, which is used to estimate the quality of a placement scheme. Finally, we introduce the DRL-based training process and the design of reward calculation based on the QoS estimation tool in section III-C.

A. Encoder-Decoder Model

We enhance the model proposed in [9] to be aware of both homogeneous and heterogeneous resources without limitation on the number of resources. Fig. 2 shows the overall architecture of the encoder-decoder model. The model first encodes G_{task} and G_{res} respectively into node embeddings. Then, an RNN-based decoder generates the placement for each task sequentially using the attention mechanism.

1) Graph Encoding: To be aware of graphs with different structures, our model achieves scalability using Graph Neural Network [10], [11], which encodes the graph information into a set of embedding vectors. Such embeddings are calculated using Graph Convolutional Network (GCN) [10], [12], [13]. Specifically, each node v's embedding is updated iteratively following GraphSAGE [14] with the customization for the task placement problem. v's initial feature is defined as f_v and its embedding at the k-th step as e_v^k. When $k = 0$, $e_v^k = f_v$. There are some differences between encoding G_{task} and G_{res} because of their different graph types.

For a DAG G_{task}, information from v's upstream and downstream neighbors is aggregated respectively, since they have different impacts on decision making. v's upstream neighbors is denoted as $\mathcal{N}_u(v)$ and downstream neighbors as $\mathcal{N}_d(v)$. Taking $\mathcal{N}_u(v)$ for example, for each node $u \in \mathcal{N}_u(v)$, it's embedding at the k-th step is e_u^k. We compute $e_u^{(up)} = \mathrm{ReLU}(\mathbf{W_1^{(up)}} e_u^k)$. Then, all $e_u^{(up)}, \forall u \in \mathcal{N}_u(v)$ are computed, and the upstream-view embedding of v is updated using:

$$e_v^{(up)} = \mathrm{ReLU}(\mathbf{W_2^{(up)}} \left[e_v^k : \frac{\sum_{u \in \mathcal{N}_u(v)} e_u^{(up)}}{|\mathcal{N}_u(v)|} \right]). \quad (1)$$

Similarly, the downstream-view embedding of v is updated to get $e_v^{(down)}$ using parameters $\mathbf{W_1^{(down)}}$ and $\mathbf{W_2^{(down)}}$. Then, v's embedding at the $k+1$-th step is the concatenation of its two views' embeddings: $e_v^{k+1} = \left[e_v^{(up)} : e_v^{(down)} \right]$

For G_{res}, which is an undirected graph with edge features (communication delay), To be ware of the edge features, node embeddings are concatenated with edge features during the first transformation:

$$e_u^{(neighbor)} = \mathrm{ReLU}(\mathbf{W_1^{(res)}} \left[e_u^k : d_{(u,v)} \right]) \quad (2)$$

Notice that G_{res} is a fully connected undirected graph, there is no upstream and downstream neighbors. The aggregation formula can be adjusted to:

$$e_v^{k+1} = \mathrm{ReLU}(\mathbf{W_2^{(res)}} \left[e_v^k : \frac{\sum_{u \neq v} e_u^{(neighbor)}}{|V_{task}| - 1} \right]) \quad (3)$$

After K iterations, the embeddings of all nodes have been computed. The entire task graph information can be computed by feeding each task's embedding to a fully connected layer

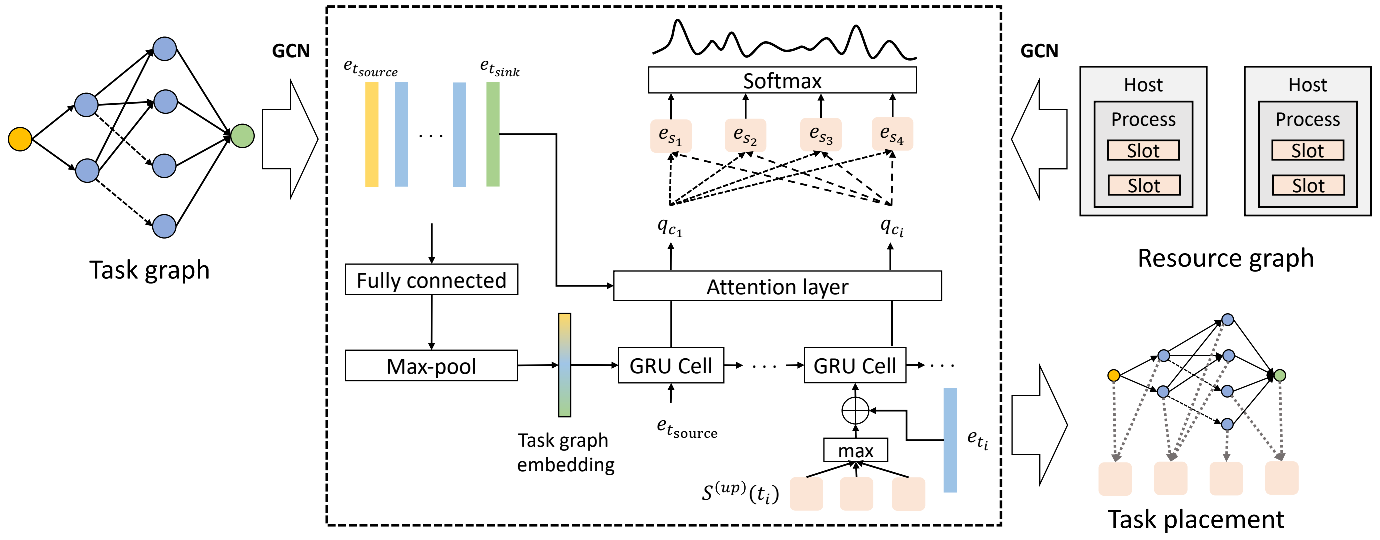


Fig. 2: Overview of the model architecture. Our model is composed of task graph and resource graph encoders and an RNN-based decoder using attention for task placement prediction.

followed by a max-pooling layer. The embedding of task t_i in G_{task} is denoted as e_{t_i} and slot s_i in G_{res} as e_{s_i} for short in the following sections.

2) Graph-Aware Decoding and Task Placement: The objective of task placement is to place each task $t_i \in V_{task}$ on a specific slot $s_{t_i} \in V_{res}$. Obviously, for a task t_i in G_{task}, the slots assigned to t_i's upstream tasks usually have a significant impact on deciding t_i's placement. Therefore, we always ensure that t_i's upstream tasks are placed before t_i by ordering all tasks via topological sort. The decoder will generate the placement for each task sequentially in topological order.

Denote the final placement scheme generated by decoder as $\mathcal{P}$, we adopt the approximation in [9] to formulate the task placement problem:

$$p(\mathcal{P}|G_{task}, G_{res}) = \prod_i p(s_{t_i}|\mathcal{S}^{(up)}(t_i), G_{task}, G_{res}) \quad (4)$$

Here, $\mathcal{S}^{(up)}(t_i)$ refers to the set of slots where all upstream tasks of t_i are placed.

In practice, GRU [15] is adopted to memorize the dependencies by learning a state representation h_{t_i}, which encodes the information associated with $\mathcal{S}^{(up)}(t_i)$ and G_{task}. At each step, the input vector e_{t_i} will be added with embeddings of slots in $\mathcal{S}^{(up)}(t_i)$ to strengthen the understanding of placements related to t_i to help its placement. Different from the formula proposed in [9], here we only consider slots related to t_i's upstream tasks excluding t_{i-1}'s placement because t_{i-1} may not be an upstream task of t_i.

$$h_{t_i} = \text{GRU\_Cell}(e_{t_{i-1}}, h_{t_{i-1}}, \mathcal{S}^{(up)}(t_i)) \quad (5)$$

To predict which slot to place t_i, we first use the temporal attention mechanism [16] to get a context vector c_i. In detail, each e_{t_j} receives an attention score at step i as $e_{ij} = h_{t_i} e_{t_j}$. Then all e_{ij}s are fed into a softmax layer to achieve the score α_{ij}. The final context vector $c_i = [h_{t_i} : \sum_j \alpha_{ij} e_{t_j}]$.

Based on the context vector c_i, the probabilities $p(s_{t_i}|c_i)$ are computed referring to the self-attention mechanism [17]. The keys and values come from the slot embeddings e_{s_j}. A single query q_{c_i} from c_i is calculated as:

$$q_{c_i} = \mathbf{W^Q} c_i, k_j = \mathbf{W^K} e_{s_j}, v_j = \mathbf{W^V} e_{s_j}. \quad (6)$$

Then $u_{(c_i)j}$ can be computed using the formula according to the self-attention mechanism. Following [18], the result will be clipped within $[-C, C]$ (C=10) using tanh.

$$u_{(c_i)j} = C \cdot \tanh\left(\frac{q_{c_i}^T k_j}{\sqrt{d_k}}\right) \quad (7)$$

Here, d_k is the query/key dimensionality and we use a single attention head to predict [19], which means d_k equals to the dimensionality of slots embeddings. The final output probability vector $\mathbf{p}$ can be computed using a softmax:

$$p_k = p_\theta(s_{t_i} = s_k|c_i) = \frac{e^{u_{(c_i)k}}}{\sum_j e^{u_{(c_i)j}}} \quad (8)$$

By predicting the placement for each task sequentially, the model will generate the final placement scheme $\mathcal{P}$.

B. Estimating QoS Attributes

To estimate some critical but complicated QoS attributes of a placement scheme $\mathcal{P}$, such as throughput, the most ideal way is to deploy G_{task} to a cluster based on G_{res} following $\mathcal{P}$ and acquire run-time metrics. Obviously, it is too costly and time-consuming to make it feasible. Thus, we propose a QoS estimation tool to estimate delay and throughput without deployment.

Algorithm 1 Calculate throughput and delay

```
Require: $G_{task},\ G_{res},\ \mathcal{P}$
 1: for all $t_i \in V_{task}$ do
 2:     $t_i.max \leftarrow 0$                         ▷ Maximum throughput of $t_i$
 3:     $t_i.cur \leftarrow 0$                         ▷ Current throughput
 4:     $t_i.back \leftarrow 0$                        ▷ Throughput after back pressure
 5:     $t_i.delay \leftarrow 0$                       ▷ Delay from source to $t_i$
 6: $t_{source}.cur \leftarrow \infty$
 7: repeat
 8:     $throughput \leftarrow t_{source}.cur$
 9:     for $s_i \in V_{res}$ do
10:         for $t_j \in \mathcal{T}(s_i)$ do           ▷ Recycle CPU resources
11:             $recycle \leftarrow t_j.max - t_j.cur$
12:             $s_i.cpu \leftarrow s_i.cpu + recycle * t_j.cpu$
13:             $t_j.max \leftarrow t_j.max - recycle$
14:         for $t_j \in \mathcal{T}(s_i)$ do           ▷ Allocate CPU resources
15:             $t_j.max \leftarrow t_j.max + s_i.cpu / \sum_{t_k \in \mathcal{T}(s_i)} t_k.cpu$
16:         $s_i.cpu \leftarrow 0$
17:     for $t_i \in \mathrm{topologicalSort}(G_{task})$ do
18:         if $t_i \neq t_{source}$ then
19:             $t_i.cur \leftarrow \min(t_i.max,\ \sum_{t_j \in \mathcal{N}_u(t_i)} \frac{t_j.cur}{|\mathcal{N}_d(t_j)|})$
20:     for $t_i \in \mathrm{reversed}(\mathrm{topologicalSort}(G_{task}))$ do
21:         for $t_j \in \mathcal{N}_u(t_i)$ do          ▷ Back pressure
22:             $t_j.back \leftarrow t_j.back + \frac{t_i.cur * t_j.cur}{|\mathcal{N}_d(t_i)| \sum_{t_k \in \mathcal{N}_u(t_i)} \frac{t_k.cur}{|\mathcal{N}_d(t_k)|}}$
23:     for $t_i \in \mathrm{topologicalSort}(G_{task})$ do
24:         $t_i.cur \leftarrow t_i.back$
25:         $t_i.back \leftarrow 0$
26: until $throughput - t_{source}.cur <$ threshold
27: for $t_i \in \mathrm{topologicalSort}(G_{task})$ do     ▷ Calculate delay
28:     for $t_j \in \mathcal{N}_d(t_i)$ do
29:         $t_j.delay \leftarrow t_j.delay + \frac{t_i.cur * (t_i.delay + d_{(i,j)})}{|\mathcal{N}_d(t_i)| * t_j.cur}$
30: return $throughput, t_{sink}.delay$
```

1) Delay: For a tuple, delay is defined as the sum of the communication delays between all slots passing from source to sink [1]. There might be multiple different paths from source to sink, so we calculate the final delay as the average delay on all possible paths. This definition of delay is intuitive and it is easy to estimate from a task placement scheme $\mathcal{P}$.

2) Throughput: Throughput represents the number of tuples processed per second for a specific DSP job. For most DSP applications, throughput is the most significant QoS attribute. However, different from delay, it is difficult to estimate throughput without deploying the job.

Here, we estimate the ***theoretical maximum throughput*** (sustainable throughput) that resources can support for a job's placement scheme. Assuming the source sends tuples as quickly as possible at the very beginning, the ***back pressure*** mechanism in DSP engines ensures that if the tuple consuming rate of t_i is slower than the total tuple producing rate of all $t_j \in \mathcal{N}_u(t_i)$, t_i will notify t_js to reduce the producing rate until it matches the consuming rate of t_i. Back pressure will propagate from downstream tasks to upstream tasks until the source to decrease its sending rate. Thus, the final sending rate of the source is defined as the theoretical maximum throughput.

3) Estimation Algorithm: Algorithm 1 shows the logic of throughput and delay calculation. Throughput can be calculated by recycling and allocating resources and applying back pressure mechanism iteratively. Here, $\mathcal{T}(s_i)$ is denoted as all tasks placed on s_i according to $\mathcal{P}$.

First, the source sends tuples as quickly as possible (line 6). For each slot $s_i \in V_{res}$, it's remaining CPU computational capacity are allocated to each $t_j \in \mathcal{T}(s_i)$, then each t_j's maximum throughput can be calculated (line 14-15). However, the throughput of t_i cannot exceed the sum of t_i's upstream tasks' sending rate for tuples sending to t_i. Therefore, we can calculate the current throughput for each $t_i \in V_{task}$ in topological order (line 17-19).

Then, according to the back pressure mechanism, each $t_i \in V_{task}$ should check the back pressure status and propagate to its upstream tasks. All $t_i \in V_{task}$ are traversed in the reversed topological order to recycle the computational capacity (line 20-22) and reallocate them (line 10-15). The above process will be repeated until the throughput of t_{source} become stable.

After the throughput calculation, each task's delay can be calculated from source to sink in topological order (line 27-29).

The QoS estimation tool can not only speed up the training process but also guide the deployment. In practice, the encoder-decoder model can generate multiple placement schemes, then the estimation tool can be applied to select the best placement scheme for the final deployment.

C. Training with Reinforcement Learning

A policy gradient algorithm is used for training. The main idea is to learn by performing gradient descent on the neural network parameters using the rewards observed during the training. For conciseness, denote all parameters as θ and π_θ as the distribution over all possible placement schemes. Our model aims to maximize the following objective:

$$J(\theta) = \sum_{\mathcal{P}} \pi_\theta(\mathcal{P}) r(\mathcal{P}) \tag{9}$$

Then, the REINFORCE algorithm [20] is used to to compute the policy gradients and learn the network parameters θ:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log \pi_\theta(\mathcal{P}_n) \left[r(\mathcal{P}_n) - b \right] \tag{10}$$

Here, N is the fixed batch size and b is the average reward of the N samples, which is used to reduce the variance of policy gradients [21]. $\nabla_\theta \log \pi_\theta(\mathcal{P}_n)$ provides a direction in the

[1]Here, we only consider the communication delay between tasks excluding the processing time at each processing unit. The end-to-end tuple processing time is another QoS attribute which is hard to estimate.

parameter space to increase the probability of choosing $\mathcal{P}_n$. Denote learning rate as α, the parameters θ can be updated by $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$, which aims to increase the probability of choosing a placement scheme $\mathcal{P}$ that has a better-than-average reward.

The reward $r(\mathcal{P})$ is determined by both throughput and delay. The proposed estimation algorithm in section III-B is used to efficiently estimate throughput and delay of $\mathcal{P}$.

Notice that the delay and throughput calculated by our algorithm are relative values because of the relative initial features f_v and $d_{(u,v)}$. Meanwhile, QoS attributes are not comparable for different training samples. Thus, we define the reward means the improvement of $\mathcal{P}$ over some predefined baseline heuristics. Denote the placement scheme generated by our model and a baseline approach are $\mathcal{P}$ and $\mathcal{Q}$ respectively. The delay and throughput of $\mathcal{P}$ and $\mathcal{Q}$ are $\text{delay}_\mathcal{P}, \text{throughput}_\mathcal{P}$ and $\text{delay}_\mathcal{Q}, \text{throughput}_\mathcal{Q}$ respectively. The reward of delay and throughput can be calculated as below:

$$r_{\text{delay}}(\mathcal{P}) = \frac{\text{delay}_\mathcal{Q} - \text{delay}_\mathcal{P}}{\text{delay}_\mathcal{Q}} \tag{11}$$

$$r_{\text{throughput}}(\mathcal{P}) = \frac{\text{throughput}_\mathcal{P} - \text{throughput}_\mathcal{Q}}{\text{throughput}_\mathcal{Q}} \tag{12}$$

The final reward is defined as the linear combination of r_{delay} and $r_{\text{throughput}}$:

$$r(\mathcal{P}) = \lambda * r_{\text{throughput}}(\mathcal{P}) + (1-\lambda) * r_{\text{delay}}(\mathcal{P}) \tag{13}$$

Here, λ is configurable. The reward setting is highly flexible and scalable, which can be easily extended to other QoS attributes. In practice, we find that setting $\lambda = 0.8$ can optimize throughput with a small impact on delay. In practice, $\text{delay}_\mathcal{Q}$ and $\text{throughput}_\mathcal{Q}$ are calculated by averaging over different handcrafted heuristics to avoid the potential bias as the model tends to just improve the weaknesses of a single heuristic instead of optimizing the objective according to λ.

Notice that some placement schemes may exceed the memory limit of one or more slots in G_{res}. For such cases, we set the reward $r(\mathcal{P})$ to a large negative constant, which is called the failing signal [22] to avoid placing too many tasks on a single slot.

IV. Implementation

We train our encoder-decoder model and implement it as a pluggable scheduling service that provide task placement schemes for Flink. In section IV-A, we describe the construction of our data set and hyperparameters. Next, we describe the integration with Flink in section IV-B. Our code and data set are available at GitHub[2].

[2]https://github.com/Shawn-Hx/TATA

A. Training Settings

1) Data Set Construction: A data set including 400 task graphs and 112 resource graphs are created. Initial features of each graph vertex are assigned randomly. Task graphs are constructed according to the DSP model in section II-A. The parallelism of each job vertex ranges from 1 to 10 and the longest length from source to sink ranges from 2 to 6. The number of tasks in a task graph varies from 3 to 36. 30% of job graphs have the same parallelism for all job vertices and 40% of two adjacent job vertices have the same parallelism. The construction of task graphs considers the characteristics of real DSP jobs in production environments while also takes the diversity of data into account. The set of resource graphs contains both homogeneous and heterogeneous slots and for 72% of task graphs, their slots are heterogeneous. The number of slots in a task graph ranges from 2 to 15. Four levels' communication delays are set to 1, 1.5, 2 and 4 respectively.

Each training and test sample is composed of a task graph and a resource graph. The training and test samples are constructed through the combination of task graphs and resource graphs. In practice, 1500 samples are generated for training and 500 samples for test.

2) Hyperparameters: We set the hyperparameters as follows. The number of iterations K in graph embedding for task graph and resource graph are set to 2 (the same as GraphSAGE [14]). The length of task embeddings and slot embeddings are set to 128. The batch size N is set to 20. The learning rate α is 0.0001 and Adam optimizer [23] is used for gradient descent. The network is trained for at least 2000 epochs for different parameter λ.

B. Flink Integration

A Flink cluster [3] typically consists of three main components: *JobManager*, *ResourceManager* and *TaskManager*.

- **JobManager** is a JVM process which has two components: *slotPool* and *Scheduler*. The *slotPool* serves slot requests issued by the *Scheduler*. It will attempt to acquire new slots from the *ResourceManager* when it cannot serve a slot request. The *Scheduler* generates the placement scheme for each task when a DSP job is submitted to the cluster and deploys DSP tasks accordingly. The *Scheduler* applies Flink's default scheduling method, which is described in section V-C2, to compute a placement scheme.
- **ResourceManager** manages all resources in a Flink cluster. In detail, the *slotManager* component is responsible for maintaining a view on all registered *TaskManager* slots and their allocation.
- **TaskManager** (also called worker) is a JVM process which contains several *slots*. There must always be at least one *TaskManager* in the cluster. *TaskManagers* execute the tasks and exchange the datastreams.

[3]We discuss Flink's standalone deployment mode here (https://ci.apache.org/projects/flink/flink-docs-release-1.11/ops/deployment/cluster_setup.html). YARN-based deployments can use our model in principle, but require modifying both Flink and YARN.

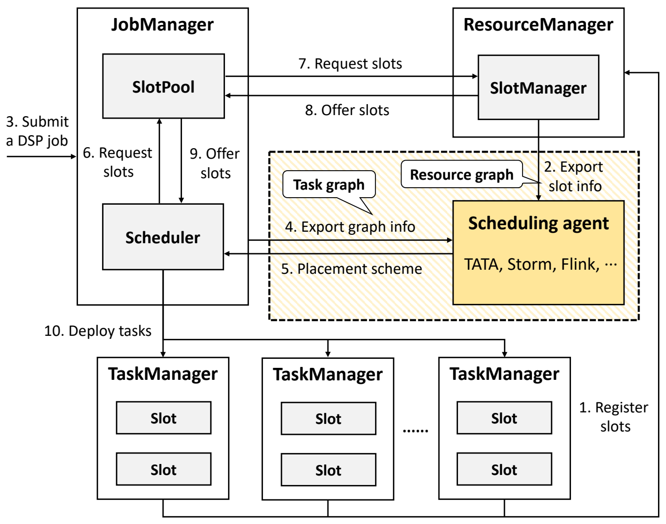


Fig. 3: Flink standalone cluster architecture with our task placement agent. The labeled directed edges illustrate the process to compute a placement scheme and deploy a DSP job.

Standalone cluster's architecture of current Flink version is illustrated in Fig. 3 (excluding the dotted box), which does not support the integration of user-defined scheduling strategies. In order to integrate our scheduling strategy to Flink, we implement a plugin mechanism, which is illustrated as the dotted box in Fig. 3. *Scheduling agent* is abstracted to serve this purpose. Informally speaking, a *scheduling agent* has three interfaces: (1) the first one for query the resource graph from *ResourceManager*, (2) the second one is called by *Scheduler* to pass the DSP graph information to *scheduling agent*, (3) the third one is used to return the placement scheme to *Scheduler*.

A typical data exchange process among *JobManager*, *ResourceManager*, and *scheduling agent* is shown as follows (in Fig. 3):

1) After all slots in cluster being registered to *Resource-Manager*, *slotManager* exports slot information to the *scheduling agent* (step 2).
2) When a DSP job is submitted to the cluster, *JobManager* exports task graph information to the *scheduling agent* (step 4).
3) Then the *scheduling agent* calculates a placement scheme and return it to the *Scheduler* (step 5).
4) Finally, the *Scheduler* deploys DSP tasks to their corresponding slots according to the placement scheme.

Currently, all data exchanged among *JobManager*, *ResourceManager*, and the *scheduling agent* uses JSON files.

The conception of *scheduling agent* decouples task placement from task deployment. Compared with the current scheduler of Flink, the *scheduling agent* is aware of all available resources in cluster, which makes it possible to support user-defined scheduling strategies. Specifically, the task placement heuristics trained by TATA is implemented as an instance of *scheduling agent*.

Additionally, since Flink prohibits two subtasks belonging to one job vertex from being placed on the same slot, we have removed the restriction to make the scheduler more flexible.

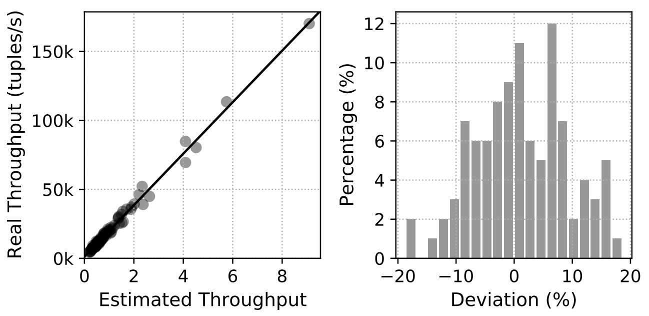


Fig. 4: Relationship and deviation between the estimated and real throughput. deviation is calculated after linear regression.

V. Experiments

Our DRL-based framework is evaluated on generated data and also on a real Flink cluster in order to answer the following questions:

1) Since the scheduling model is trained based on throughput estimation algorithm, is the algorithm accurate for real DSP applications? (section V-B)
2) Is our method applicable to heterogeneous resources? (section V-A)
3) Does our method optimize application's throughput without a (serious) negative impact on its delay and how does our method perform compared to widely-used heuristics? (section V-C)

A. Flink Cluster

A standalone cluster running Flink v1.11.2 is deployed. The cluster consists of 4 VMs. *JobManager* runs on an instance (1 CPU, 2GB RAM), three *TaskManager*s runs on three instances with 1, 2, 4 CPU cores (2.5GHz, 2.0GHz, 2.6GHz) and 2, 4, 8 GB RAM respectively. Each *TaskManager* has two slots, which means slots belonging to different *TaskManager*s are heterogeneous.

B. Throughput Estimation

In order to verify that our throughput estimation algorithm is suitable for heterogeneous DSP applications and resources, 100 job graphs are generated with the random number of job vertices ranges from 1 to 10 and random parallelism of each job vertex ranges from 1 to 6. The job graphs are deployed on a random subset of all slots in the cluster with each task being placed randomly.

The number of loops is used as a measure of different tasks' CPU utilization, which can be controlled in user-defined functions using Flink's DataStream API.

Notice that the throughput predicted by our estimation tool only reflects the relative magnitude. Fig. 4 shows the relationship and deviation between throughput calculated by our tool and the real value. As shown in the first graph, there is a clear linear relationship between the throughput calculated by our estimation tool and the real value. The relative throughput value is transformed into real value by linear regression, then the deviation can be calculated. The second graph shows that, for 78% test cases, the absolute deviations are less than 10%. For all test cases, the average absolute deviation is 6.7%, which means our estimation tool is feasible to estimate the throughput without deployment.

C. Performance of Real Case Applications

1) DSP Topologies: We conduct our model evaluation using five well-known DSP topologies, which are illustrated in Fig. 5 and described below.

- **Throughput Test Topology:** This is a topology with one source, one operator and one sink. The source repeatedly generates random strings of a fixed size of 10K bytes as tuples. The identity operator emits tuples to sink without any change. The sink increments a counter every time a tuple has been received.
- **Word Count Topology:** Word count is a well-known application which counts each word's number of appearances in one or more files. The source sends random length word list one line at a time (the length ranges is set from 1 to 1000). The split operator splits each line into words and the count operator increases the counter based on input words and send results to an empty sink.
- **Top Speed Windowing Topology:** Top speed windowing is an application to calculate the top speed of each car every x meters elapsed for the last y seconds. The source fetches events from cars and the time operator extracts timestamp from each event. The window operator groups cars by id, triggers calculation and sends results to sink. This is an application running on numerical data.
- **Log Stream Processing Topology: [24]** Log stream processing presents a real-world use case. The source sends log records one line each time. The rule operator performs rule-based analysis and emits log entries. The log entry is then sent to two operators which perform indexing and counting actions, respectively.
- **Synthetic Communication Topology [25]:** This is a synthetic topology. The source generates a random string at a time and each communication operator doubles the received string and passes them to the next operator. This is a typical communication intensive workload.

These topologies are implemented in Flink without any source and sink connectors to ensure the performance not be affected by external frameworks [26]. In detail, sources generate and send in-memory random data repeatedly and sinks receive tuples and do nothing or just increment a counter. To find an excellent placement scheme, multiple placement schemes generated by our model are sampled and the best is selected before deployment based on the QoS estimation tool.

2) Baselines: We adopt the DSP engines' task placement approaches (e.g., Storm and Flink) and Ni's method [9] as baselines.

- **Improved Storm's default scheduling**. Storm includes a default scheduler, which assigns tasks of a topology evenly into slots across the cluster using a round-robin algorithm. We improve the default scheduler by sorting all hosts in descending order of slots' number. Storm will always use all available slots in a cluster, so it is suitable for high throughput DSP applications.
- **Flink's default scheduling**. Flink's default scheduler prohibits tasks belonging to one job vertex being placed on the same slot, in order to achieve load balance to some extent. Flink also considers the dependencies of tasks and try to place non-conflict tasks to the same slot, which means each slot can run a pipeline of tasks.
- **Ni's method [9]**. Ni proposes a DRL-based method to optimize throughput with a fixed number of resources. A single model trained by Ni's method is only suitable for a fixed number of resources. Here, we apply the model architecture of Ni's method with our throughput estimation tool to train the model as a baseline.

The baselines will be simply labeled and called "*I-Storm*", "*Flink*" and "*Ni*" respectively in the following sections.

3) Experiment Results: Fig. 6 compares the throughput of task placement decided by our model versus that of *Flink*, *I-Storm* and *Ni* for five DSP topologies respectively. Table I illustrates the four strategies' end-to-end delay[4] with different quantiles. Our method achieves superior throughput comparing with other three methods and makes a good trade-off between two QoS metrics. We use "tps" as the abbreviation for "tuples/s" in this section and the content in parentheses indicates the average throughput value.

For throughput test topology, *Flink* has similar throughput to *Ni* (6.1K). our model (9.0K) provides 8.9%, 47% and 47% higher throughput than *I-Storm* (8.3K), *Flink* (6.1K) and *Ni* (6.1K) respectively.

For word count topology. Our model supports 4.3K tps on average, the improvements over *I-Storm*, *Ni* and *Flink* are 10%, 13% and 63% respectively.

For top speed windowing topology, our model (217K) and *Ni* (224K) have the similar throughput and beat *Flink* (191K) and *I-Storm* (160K). but compared to *Ni*, our model has better performance in delay.

For log stream processing strategy, which has a complex task graph, our model provides best throughput (66K). The throughput of *Ni*, *Flink* and *I-Storm* are 64K, 62K and 50K respectively. Notice that our model provides the best throughput and delay simultaneously.

For synthetic communication topology, our model provides superior 33K tps throughput. The improvement over *Ni*, *I-Storm* and *Flink* are 28%, 40% and 63% respectively. Even

[4]https://ci.apache.org/projects/flink/flink-docs-release-1.11/monitoring/metrics.html

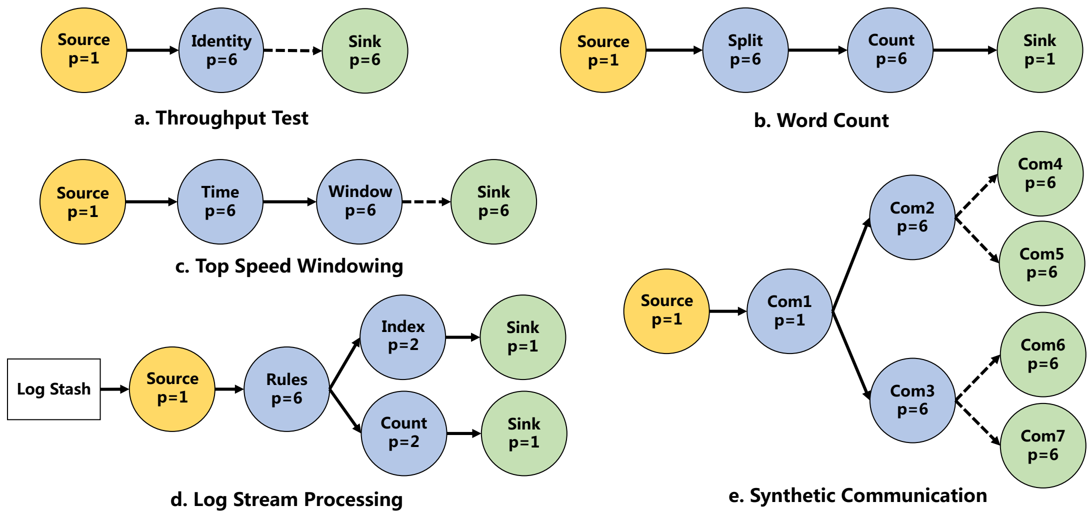
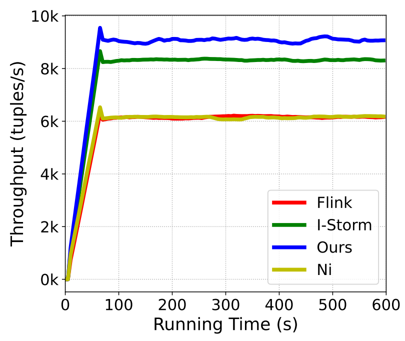
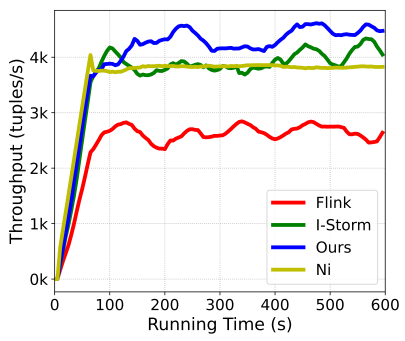


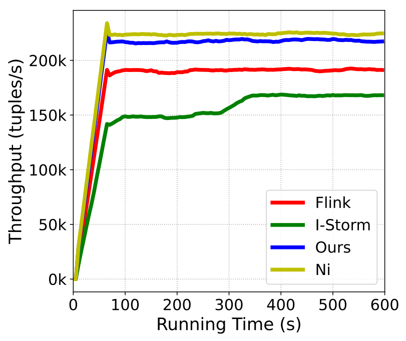
Fig. 5: Job graphs of evaluated DSP topologies. *p* in each job vertex means its parallelism.



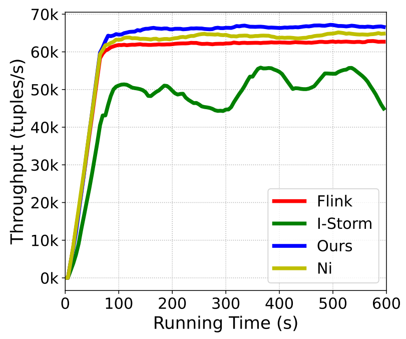
(a) Throughput Test



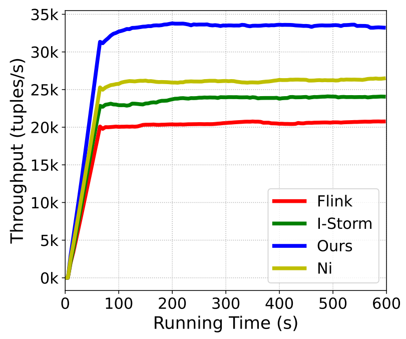
(b) Word Count



(c) Top Speed Windowing



(d) Log Stream Processing



(e) Synthetic Communication

Fig. 6: Evaluation of maximum throughput on five DSP topologies. Each topology is deployed and run for 10 minutes. The number of tuples sent from source is used as a measure of throughput.

at the expense of latency in order to improve throughput, our model provides much lower delay than *Ni*.

4) Discussion: In general, *I-Storm* distributes DSP tasks evenly to all hosts in cluster without considering tasks' dependencies, so its performance is not stable. *Flink* places a pipeline of related tasks into a single slot, which achieves better performance in delay for DSP topologies with sequential job graph (throughput test, word count and top speed windowing), but resource heterogeneity is not be considered, resulting in poor throughput performance. *Ni* focuses on throughput optimization. Due to lack of consideration of heterogeneity and delay, it only works well for jobs that don't pay much attention to delay in a homogeneous cluster. Comparing with these strategies, our framework considers all information and learn to place tasks close to source on more powerful slots and place simple and related tasks together on a less powerful

TABLE I: Evaluation of delay on five DSP topologies.

	Throughput Test			Word Count			Top Speed Windowing			Log Stream Processing			Synthetic Communication		
quantile	99	98	95	99	98	95	99	98	95	99	98	95	99	98	95
Flink	10.0	7.8	5.5	389	348	281	743	724	687	1113	1062	1003	71.1	55.7	40.8
I-Storm	11.3	8.6	6.2	827	751	608	535	506	479	839	749	690	48.3	39.2	32.8
Ni	20.6	14.5	9.3	504	475	400	861	813	764	840	817	777	266	249	221
Ours	18.6	13.8	9.2	572	507	450	808	751	696	795	769	731	147	97.1	79.2

slot. The scalable reward design enables our model to optimize throughput without a serious impact on delay.

Our framework solves the task placement problem assuming that the parallelism of each job vertex has been set by users. Since changes of system conditions and workload can occur during execution, the placement scheme should be adjusted at run-time. run-time adaptation costs because of DSP operators state but does not change the placement problem definition. Thus, our framework can be applied to run-time adaptation with some adjustments. However, our model is currently not suitable for DSP applications with very large-scale task graphs (thousands of tasks) because we generate the complete placement scheme using an RNN-based architecture. As part of the future work, we consider to segment the large graph into groups and make our model support multiple DSP job graphs.

VI. Related Works

In this section, we review the approaches from prior works and organize them along two dimensions: objectives and methodologies.

A. Objectives

The existing solutions optimize diversity of objectives. Pietzuch [27] aims to minimize the network usage while Chatzistergiou [28] aims to minimize the application response time. Nardelli [6] proposes a cost function which comprises different QoS attributes including response time, availability and network usage. These QoS attributes can be estimated by some well-defined algorithms without deployment.

Some prior works optimize the QoS attributes which are difficult to calculate or estimate directly. Li [8] aims to minimize the average end-to-end tuple processing time without calculating it directly depending on online deep reinforcement learning. Wu presents TurboStream [29] to improve the end-to-end latency of intra-node IPC with a coarse-grained scheduler. Ni [9] adopts a simulator CEPSim [30] to evaluate and optimize throughput. CEPSim extends CloudSim [31] and estimates throughput based on the simulation ticks. However, it is not efficient to make CEPSim reflect the maximum throughput and it is also not flexible for different deployment scenarios.

B. Methodologies

The task placement problem has been addressed mainly relying on three methods: handcrafted heuristics, mathematical programming and deep reinforcement learning.

The most popular open-source DSP engines (e.g. Flink, Storm) usually adopt efficient handcrafted heuristics. Storm supports a simple round-robin algorithm and a resource-aware heuristic R-Storm [32]. R-Storm presents a heuristic which determines the candidate node by finding the closest Euclidian distance in 3-d space. Flink proposes an elaborate heuristic which is easy to configure and can achieve load balance to some extent. Nardelli [6] formulates the problem as an integer problem (IP), and develops several model-based and model-free heuristics to solve the problem.

DRL-based methods have been used to solve scheduling problems in recent years. Mirhoseini [22] works on learning device placement in a TensorFlow [33] graph using a sequence-to-sequence model, which relies on recurrent neural networks to get state embedding rather than a graph neural network. Their objective is to schedule a single TensorFlow job with one fixed graph well. Mao introduces Decima [7] to learn scheduling algorithms for continuous batch processing jobs, which converts job graphs to vectors by graph embedding and uses reinforcement learning to minimize average job completion time. Different from other applications, for DSP applications, Li [8] introduces an actor-critic approach aiming to minimize the average tuple processing time by learning to control DSP jobs from experience rather than using an accurate and mathematically solvable system model. Ni [9] formulates the task placement problem as a graph partition problem and propose a generalizable encoder-decoder model considering a fixed number of homogeneous resources.

VII. Conclusion

In this paper, a generic DRL-based resource-aware framework is presented to solve the task placement problem for DSP applications. We use graph convolutional network to utilize the information both in DSP task graphs and resource graphs. Additionally, a throughput estimation tool is proposed to optimize throughput and speed up the training process without deployment. Experiments' results show that the tool can match the real environment in most scenarios and our method achieves superior throughput without a serious nagative impact on delay in a heterogeneous Flink cluster.

Acknowledgement

This work was supported in part by Key-Area Research and Development Program of Guangdong Province under Grant 2020B010164001, National Key Research and Development Project under Grant 2018YFB1703404, and NSFC Program under Grant U19A2062 and Grant U1801263.

Appendix

The list of mathematical symbols in the paper is summarized in Table II.

TABLE II: Notations

Notation	**Description**
G_{job}	DSP job graph
G_{task}	DSP task graph
G_{res}	resource graph
j	a job vertex in job graph
$j.p$	parallelism of a job vertex
t	a task vertex in task graph
$t_{j,k}$	the k-th parallel sub task of the job vertex j
s	a slot in resource graph
s_{t_i}	slot where the i-th task is placed
$d_{(i,j)}$	delay between the i-th and j-th slot
e_v^k	embedding of node v at the k-th step in GCN
e_{t_i}	embedding of the i-th task
e_{s_i}	embedding of the i-th slot
$\mathcal{P}$	task placement scheme
$\mathcal{N}_u(t_i)$	set of all upstream tasks of the i-th task
$\mathcal{N}_d(t_i)$	set of all downstream tasks of the i-th task
$\mathcal{T}(s_i)$	set of all tasks placed on the i-th slot
$\mathcal{S}^{(up)}(t_i)$	set of slots where the i-th task's upstream tasks are placed

References

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

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[3] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement strategies for internet-scale data stream systems," *IEEE Internet Computing*, vol. 12, no. 6, pp. 50–60, 2008.

[4] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4. IEEE, 2000, pp. 101–104.

[5] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 69–80.

[6] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1753–1767, 2019.

[7] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[8] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *arXiv preprint arXiv:1803.01016*, 2018.

[9] X. Ni, J. Li, M. Yu, W. Zhou, and K.-L. Wu, "Generalizable resource allocation in stream processing via deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 857–864.

[10] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[11] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.

[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[13] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in neural information processing systems*, 2017, pp. 6348–6358.

[14] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[15] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[16] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[18] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[19] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[20] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[21] L. Weaver and N. Tao, "The optimal reward baseline for gradient-based reinforcement learning," *arXiv preprint arXiv:1301.2315*, 2013.

[22] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *arXiv preprint arXiv:1706.04972*, 2017.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] Q. Anderson, *Storm real-time processing cookbook*. PACKET Publishing, 2013.

[25] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu, "Stromax: Partitioning-based scheduler for real-time stream processing system," in *International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 269–288.

[26] J. Grier, "Extending the yahoo! streaming benchmark," Feb. 2016. [Online]. Available: https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark

[27] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 49–49.

[28] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, 2014, pp. 1579–1588.

[29] S. Wu, M. Liu, S. Ibrahim, H. Jin, L. Gu, F. Chen, and Z. Liu, "Turbostream: Towards low-latency data stream processing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 983–993.

[30] W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "Cepsim: Modelling and simulation of complex event processing systems in cloud environments," *Future Generation Computer Systems*, vol. 65, pp. 122–139, 2016.

[31] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[32] 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, pp. 149–161.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.