Latency-Aware Placement of Data Stream Analytics on Edge Computing

Alexandre da Silva Veith(✉), Marcos Dias de Assunção, and Laurent Lefèvre

Inria, LIP, ENS Lyon, University of Lyon, 46 Allée d'Italie, 69364 Lyon, France
{alexandre.veith,marcos.dias.de.assuncao,laurent.lefevre}@ens-lyon.fr

Abstract. The interest in processing data events under stringent time constraints as they arrive has led to the emergence of architecture and engines for data stream processing. Edge computing, initially designed to minimize the latency of content delivered to mobile devices, can be used for executing certain stream processing operations. Moving operators from cloud to edge, however, is challenging as operator-placement decisions must consider the application requirements and the network capabilities. In this work, we introduce strategies to create placement configurations for data stream processing applications whose operator topologies follow series parallel graphs. We consider the operator characteristics and requirements to improve the response time of such applications. Results show that our strategies can improve the response time in up to 50% for application graphs comprising multiple forks and joins while transferring less data and better using the resources.

Keywords: Data stream processing · Edge computing
Cloud computing · Resource management · Scheduling
Series parallel graphs

1 Introduction

Today's instruments and services are producing ever-increasing amounts of data that require processing and analysis to provide insights or assist in decision making. Much of this data is received in near real-time and requires quick analysis. In the Internet of Things (IoT) [18], for instance, continuous data streams produced by multiple sources must be handled under very short delays. Under several data stream processing engines, a stream processing application is a directed graph or dataflow whose vertices are operators that execute a function over the incoming data and edges that define how data flows between the operators. A dataflow has one or multiple sources (*i.e.*, sensors, gateways or actuators), operators that perform transformations on the data (*e.g.*, filtering, and aggregation) and sinks (*i.e.*, queries that consume or store the data).

In a traditional cloud deployment, the whole application is placed on the cloud to benefit from virtually unlimited resources. However, processing all the data on the cloud can introduce latency due to data transfer, which makes near

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 215–229, 2018.
https://doi.org/10.1007/978-3-030-03596-9_14

real-time processing difficult to achieve. In contrast, *edge computing* has become an attractive solution for performing certain stream processing operations, as many *edge devices* have non-negligible compute capacity.

The deployment of data stream processing applications onto heterogeneous infrastructure, however, has proved to be NP-hard [2]. Moreover, moving operators from cloud to edge devices is challenging due to limitations of edge devices [1]. Existing work often proposes placements strategies considering user intervention [19] whereas many models do not support memory and communication constraints [6,12]. Existing work also considers all data sinks to be located in the cloud, with no feedback loop to actuators located at the edge [5,16]. There is hence a lack of solutions covering scenarios involving smart cities, precision agriculture, and smart homes comprising various heterogeneous sensors and actuators, as well as, time-constraint applications that may contain actuators often placed close to where data is collected.

In this paper, we introduce a set of strategies to place operators onto cloud and edge while considering characteristics of resources and meeting the requirements of applications. We consider analytics applications with multiple sources and sinks distributed across cloud and edge. In particular, we first decompose the application graph by identifying behaviors such as *forks* and *joins*, and then dynamically split the dataflow graph across edge and cloud. Comprehensive simulations considering multiple application settings demonstrate that our approach can improve the response time in up to 50%.

The contributions of this work are: (i) it presents a model for Distributed Stream Processing (DSP) applications in heterogeneous infrastructure (§2); (ii) it introduces placement strategies for dynamically identifying how to split the application graph across cloud and edge (§3); and (iii) it evaluates the strategies against traditional and state-of-the-art schemes (§4).

2 System Model and Problem

This section introduces preliminaries and then describes the placement problem.

2.1 System and Application Models

The network topology is a graph $\mathcal{N} = (\mathcal{R}, \mathcal{L})$ with a vertex set $\mathcal{R} = (r_1, r_2, r_3, \dots)$ of computational resources (*i.e.*, cloud servers and edge devices) and links $\mathcal{L} = (l_1, l_2, l_3, \dots)$ interconnecting the resources. Each $r_i \in \mathcal{R}$ has capabilities in terms of CPU cpu_{r_i} and memory mem_{r_i} expressed respectively in Millions of Instructions per Second (MIPS) and bytes. A network link $i \leftrightarrow j \in \mathcal{L}$ interconnecting resources i and j has bandwidth $bdw_{i \leftrightarrow j}$ and latency $lat_{i \leftrightarrow j}$ represented in bits per second (bps) and seconds respectively. We consider the latency of a resource i to itself (*i.e* $lat_{i \leftrightarrow i}$) to be 0. The network topology is known as we consider scenarios using Software Defined Network solutions [4] or discovery algorithms such as Vivaldi [7] to determine and maintain the topology information.

A DSP application is viewed as a graph $\mathcal{G} = (\mathcal{O}, \mathcal{S})$ whose vertices $\mathcal{O}$ are *operators* that perform operations on the incoming data, and edges $\mathcal{S}$ that are streams of events/messages flowing between the operators. The set of operators $\mathcal{O}$ comprises data *sources* $\mathcal{O}^{src}$, *sinks/outputs* $\mathcal{O}^{out}$ where data is stored or published, and *transformations* $\mathcal{O}^{trn}$ performed over the data.

Each operator $o_i \in \mathcal{O}$ has CPU cpu_{o_i} and memory mem_{o_i} requirements for processing incoming events, given respectively in Instructions per Second (IPS) to handle an individual event and number of bytes to load the operator in memory. The rate at which operator i can process events at reference resource k is denoted by $\mu_{\langle i,k \rangle}$ and is essentially $\mu_{\langle i,k \rangle} = cpu_{r_k} \div cpu_{o_i}$. When performing a transformation on the incoming data, an operator can, for instance, parse data or filter events hence reducing the number of events or their size. The ratio of number of input events to output events is determined by the operator's *selectivity* ψ_{o_i}. The data compression/expansion factor is the ratio of the size of input events to the size of output events, and is represented by ω_{o_i}.

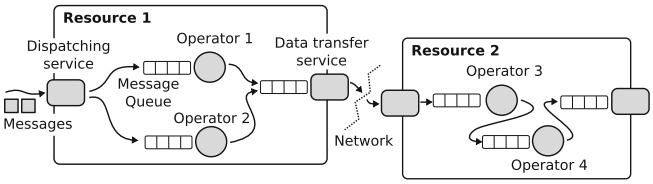

Fig. 1. Example of four operators and their respective queues placed on two resources.

Each event stream $s_{i \to j} \in \mathcal{S}$ connecting operator i to j has a probability $\rho_{i \to j}$ that an output event emitted by i will flow through to j. Here we consider that such information is obtained via profiling techniques or from previous executions of the application. Existing work has demonstrated how such information can be obtained [14]. The rate at which operator i produces events is denoted by λ_i^{out} and is a product of its input event rate λ_i^{in} and its selectivity. The output event rate of a source operator $k \in \mathcal{O}^{src}$ depends on the number of measurements it takes from a sensor or another monitored device. We can then recursively compute the input and output event rates for downstream operators j as follows:

$$\lambda_i^{in} = \lambda_k^{out} \qquad \forall s_{k \to i} \in \mathcal{S}, k \in \mathcal{O}^{src} \tag{1}$$

$$\lambda_j^{in} = \sum_{s_{i \to j} \in \mathcal{S}} \lambda_i^{in} \times \psi_{o_i} \times \rho_{s_{i \to j}} \qquad \forall i \in \mathcal{O}, i \notin \mathcal{O}^{src} \tag{2}$$

$$\lambda_j^{out} = \lambda_j^{in} \times \psi_{o_j} \qquad \forall j \in \mathcal{O}, j \notin \mathcal{O}^{out} \tag{3}$$

Likewise, we can recursively compute the average size ς_i^{in} of events that arrive at a downstream operator i and the size of events it emits ς_i^{out} by considering

the upstream operators' event sizes and their respective compression/expansion factors (*i.e.*, ω_{o_i}). In other words:

$$\varsigma_i^{in} = \varsigma_k^{out} \qquad \forall s_{k \to i} \in \mathcal{S}, k \in \mathcal{O}^{src} \tag{4}$$

$$\varsigma_j^{in} = \varsigma_i^{in} \times \omega_{o_i} \qquad \forall i \in \mathcal{O}, i \notin \mathcal{O}^{src} \tag{5}$$

$$\varsigma_j^{out} = \varsigma_j^{in} \times \omega_{o_j} \qquad \forall j \in \mathcal{O}, j \notin \mathcal{O}^{out} \tag{6}$$

When placed onto available resources, operators within a same host communicate directly whereas inter-node communication is done via a communication service as depicted in Fig. 1. If more events arrive than an operator can handle when placed at a given resource, queues will form and the overall service time will increase. Events are handled in a First-Served (FCFS) fashion both by operators and the computation service that serialises messages to be sent to another host. This guarantees the time order of events; an important requirement in many data stream processing applications. We model both operators and the communication service as M/M/1 queues which allows for estimating the waiting and service times for computation and communication. The computation or service time $stime_{\langle o_i, r_k \rangle}$ of an operator i placed on a resource k is hence given by:

$$stime_{\langle i,k \rangle} = \frac{1}{\mu_{\langle i,k \rangle} - \lambda_i^{in}} \tag{7}$$

while the communication time $ctime_{\langle i,k \rangle \langle j,l \rangle}$ for operator i placed on a resource k to send a message to operator j on a resource l is:

$$ctime_{\langle i,k \rangle \langle j,l \rangle} = \frac{1}{\left(\frac{bdw_{k \leftrightarrow l}}{\varsigma_i^{out}}\right) - \lambda_j^{in}} + l_{k \leftrightarrow l} \tag{8}$$

Furthermore, the number of events waiting to be served, being processed, or waiting to be transferred to another resource, enable us to compute the memory requirements of operators at the resources onto which they are placed. The number of events in service at an operator i at resource k is given by:

$$\varphi_{\langle i,k \rangle}^{comp} = \frac{\frac{\lambda_i^{in}}{\mu_{\langle i,k \rangle}}}{1 - \frac{\lambda_i^{in}}{\mu_{\langle i,k \rangle}}} \tag{9}$$

while the number of events waiting in the communication service to be transferred from operator i on resource k to operator j placed on resource l is:

$$\varphi_{\langle i,k \rangle \langle j,l \rangle}^{comm} = \frac{\frac{\lambda_j^{in}}{\left(\frac{bdw_{k \leftrightarrow l}}{\varsigma_i^{out}}\right)}}{1 - \frac{\lambda_j^{in}}{\left(\frac{bdw_{k \leftrightarrow l}}{\varsigma_i^{out}}\right)}} \tag{10}$$

The overall memory required by an operator i allocated on a resource k comprises the memory needed to load it as well as the memory required by in-service events, and events waiting to be serviced or waiting to be transferred to another resource:

$$mem_{\langle i,k\rangle} = \varphi^{comp}_{\langle i,k\rangle} \times \varsigma^{in}_{i} + mem_{o_i} + \sum_{\substack{j\in\mathcal{O}\\ l\in\mathcal{R}}} \varphi^{comm}_{\langle i,k\rangle\langle j,l\rangle} \times \varsigma^{in}_{i} \tag{11}$$

A mapping function $\mathcal{M} : \mathcal{O} \rightarrow \mathcal{R}$, $\mathcal{S} \rightarrow \mathcal{L}$ indicates the resource to which an operator is assigned and the link to which a stream is mapped. The function $mo_{\langle i,k\rangle}$ returns 1 if operator i is placed at resource k and 0 otherwise. Likewise, the function $ms_{\langle i\rightarrow j,k\leftrightarrow l\rangle}$ returns 1 when the stream between operators i and j has been assigned to the link between resources k and l, and 0 otherwise. A *path* in the DSP application graph is a sequence of operators from a source to a sink. A path p_i of length n is a sequence of n operators and $n-1$ streams, starting at a source and ending at a sink:

$$p_i = o_0, o_1, \ldots, o_k, o_{k+1}, \ldots, o_{n-1}, o_n \tag{12}$$

Where $o_0 \in \mathcal{O}^{src}$ and $o_n \in \mathcal{O}^{out}$. The set of all possible paths in the application graph is denoted by $\mathcal{P}$. The end-to-end latency of a path comprises the sum of the computation time of all operators along the path and the communication time required to stream events on the path. More formally, the end-to-end latency of path p_i, denoted by L_{p_i}, is:

$$L_{p_i} = \sum_{\substack{o\in p_i\\ r\in\mathcal{R}}} mo_{\langle o,r\rangle} \times stime_{\langle o,r\rangle} + \sum_{r'\in\mathcal{R}} ms_{\langle o\rightarrow o+1,r\leftrightarrow r'\rangle} \times ctime_{\langle o,r\rangle\langle o+1,r'\rangle} \tag{13}$$

2.2 Operator Placement Problem

The problem of placing a distributed stream processing application consists of finding a mapping that minimises the aggregate end-to-end latency of all application paths and that respects the resource and network constraints. In other words, find the mapping that minimises the aggregate end-to-end event latency:

$$\min \sum_{p_i\in\mathcal{P}} L_{p_i} \tag{14}$$

Subject to:

$$\lambda^{in}_{o} < \mu_{\langle o,r\rangle} \qquad \forall o \in \mathcal{O}, \forall r \in \mathcal{R} | mo_{\langle o,r\rangle} = 1 \tag{15}$$

$$\lambda^{in}_{o} < \left(\frac{bdw_{k\leftrightarrow n}}{\varsigma^{out}_{o-1}}\right) \qquad \forall o \in \mathcal{O}, \forall k \leftrightarrow n \in \mathcal{L} | mo_{\langle o,k\rangle} = 1 \tag{16}$$

$$\sum_{o\in\mathcal{O}} mo_{\langle o,r\rangle} \lambda^{in}_{o} \leq c_r \qquad \forall r \in \mathcal{R} \tag{17}$$

$$\sum_{o \in \mathcal{O}} mo_{\langle o,r \rangle} \times mem_{\langle o,r \rangle} \leq mem_r \qquad \forall r \in \mathcal{R} \tag{18}$$

$$\sum_{\substack{s_{i \rightarrow j} \in \mathcal{S} \\ k \leftrightarrow l \in \mathcal{L}}} ms_{\langle i \rightarrow j, k \leftrightarrow l \rangle} \times \varsigma_i^{out} \leq bwd_{k \leftrightarrow l} \qquad \forall k \leftrightarrow l \in \mathcal{L} \tag{19}$$

$$\sum_{r \in \mathcal{R}} mo_{\langle o,r \rangle} = 1 \qquad \forall o \in \mathcal{O} \tag{20}$$

$$\sum_{k \leftrightarrow l \in \mathcal{L}} ms_{\langle i \rightarrow j, k \leftrightarrow l \rangle} = 1 \qquad \forall s_{i \rightarrow j} \in \mathcal{S} \tag{21}$$

Constraint 15 guarantees that a resource can provide the service rate required by its hosted operators whereas Constraint 16 ensures that the links are not saturated. The CPU and memory requirements of operators on each host are ensured by Constraints 17 and 18 respectively. Constraint 19 guarantees the data requirements of streams placed on links. Constraints 20 and 21 ensure that an operator is not placed on more than a resource and that a stream is not placed on more than a network link respectively.

3 Application Placement Strategies

This section explains how patterns in the DSP application graphs are identified and then introduces strategies that employ the patterns to devise placement decisions.

3.1 Finding Application Patterns

As depicted in Fig. 2, a dataflow can comprise multiple patterns such as (i) forks, where messages can be replicated to multiple downstream operators or scheduled to downstream operators in a round-robin fashion, using message key hashes, or considering other criteria [16]; (ii) parallel regions that perform the same operations over different sets of messages or where each individual region executes a given set of operations over replicas of the incoming messages; and (iii) joins, which merge the outcome of parallel regions.

We consider Series-Parallel-Decomposable Graphs (SPDG) and related techniques to identify graph regions that present these patterns [8]. This information is used to build a hierarchy of region dependencies (*i.e.* downstream and upstream relations between regions) and assist on placing operators across cloud and edge resources. The streams in the graph paths that separate the operators are hereafter called the *split points*. Figure 2 illustrates the phases of the method to determine the split points (green circles), where red circles represent operators placed on edge resources whereas blue ones are on the cloud: (i) The method starts with sources and sinks whose placements are predefined by the user; (ii) split points are discovered (green circles) as well as sinks that correspond to actuators that can be placed on the edge; (iii) the branches between the existing

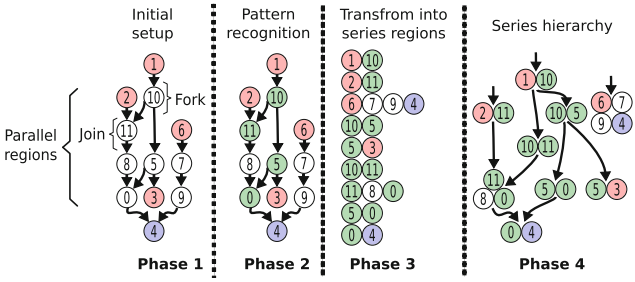

Fig. 2. Method for finding the dataflow split points, where red means placed on edge, blue represents placed on cloud, and green delimits forks and joins. (Color figure online)

patterns (green, red and blue circles) are transformed into series regions; and (iv) a hierarchy following the dependencies between regions is created.

Algorithm 1 describes the function *GetRegions* used to identify the patterns and obtain the series regions. First, the function adds two virtual vertices to the graph, one named *virt_src* connected to all data sources and another named *virt_sink* to which all sinks are connected (line 2–4). The virtual vertices allow for recognizing all paths between sources and sinks. Second, each path is iterated moving operators to a temporary vector and classifying the operators as upstream and downstream according to the number of input and output edges (lines 5–8). If the operator is a split point, the temporary vector is converted to a subset of regions set, and the temporary vector receives the current operator (lines 9–10). Third, the function removes the redundant values (line 11). At last, the region set is iterated comparing the regions by the first and the last position values (equal values represent a connection) and consequently, they are stored in the hierarchy set (lines 12–16).

3.2 Operator Placement Strategies

The region hierarchy allows us to determine the data paths and operator deployment priorities, based on which two strategies are applied: Response Time Rate (RTR) that iterates the deployment sequence and for each operator estimates the *end-to-end latency* (*i.e.* response time); and Response Time Rate with Region Patterns (RTR+RP) which uses the hierarchy to split the application graph across edge and cloud, optimizing only the response time on the edge.

Response Time Rate (RTR) is a greedy strategy that places operators incrementally by evaluating the end-to-end latency of paths (Eq. 13) while respecting the resource constraints (Eqs. 15–21). The response time of an operator in a path comprises the time taken to transfer data and to compute an event. As an operator can be in multiple paths, the RTR strategy accumulates the time taken to transfer data from multiple paths rather than evaluating each path individually.

RTR organizes the operators into deployment sequence and consecutively calculates the response time for each operator by considering the previous map-

Algorithm 1. Algorithm to detect forks and joins.

Function *GetRegions*($\mathcal{G} = (\mathcal{O}, \mathcal{S}), \mathcal{O}^{src}, \mathcal{O}^{out}$)
- $\mathcal{O} \leftarrow \mathcal{O} \cup virt\_src \cup virt\_sink$
- $\mathcal{S} \leftarrow \mathcal{S} \cup s_{virt\_src \rightarrow o}, \forall o \in \mathcal{O}^{src}$
- $\mathcal{S} \leftarrow \mathcal{S} \cup s_{o \rightarrow virt\_sink}, \forall o \in \mathcal{O}^{out}$
- **for** $p \in$ `GetAllPaths`($\mathcal{G}, virt\_src, virt\_sink$) **do**
  - **for** $o \in p$ **do**
    - $temp \leftarrow temp \cup \{o\}, \forall o \notin \{virt\_src, virt\_sink\}$
    - $ups \leftarrow |\langle *, o \rangle \subset \mathcal{S}|, downs \leftarrow |\langle o, * \rangle \subset \mathcal{S}|$
    - **if** $ups > 1$ **or** $downs > 1$ **and** $o \notin \{virt\_src, virt\_sink\}$ **then**
      - $regions \leftarrow regions \cup temp$, $temp \leftarrow \{o\}$
- Delete duplicate *regions*
- **for** $src\_series \in regions$ **do**
  - **for** $dst\_series \in regions$ **do**
    - **if** $src\_series \neq dst\_series$ **then**
      - **if** $src\_series[|src\_series| - 1] = dst\_series[0]$ **then**
        - $hierarchy \leftarrow hierarchy \cup \{src\_series, dst\_series\}$
- **return** *hierarchy*

Algorithm 2. Calculating the computational response times.

Function *EstimateResponseTimes*($\mathcal{N} = (\mathcal{R}, \mathcal{L}), \mathcal{G} = (\mathcal{O}, \mathcal{S}), o$)
- **for** $child \in \langle o, * \rangle \subset \mathcal{S}$ **do**
  - $upstreams \leftarrow \langle child, r \rangle, \forall r \in \mathcal{R}$ ***and*** $mo_{\langle child, r \rangle} = 1$
- **for** $r \in \mathcal{R}$ **do**
  - $comm \leftarrow 0$
  - **for** $mapping \in upstreams$ **do**
    - **if** `GetHost`($mapping$) $\neq r$ **then**
      - $com \leftarrow comm + ctime_{\langle mapping \rangle \langle o, r \rangle}$
  - **if** `MeetConstraints` **then**
    - $rt \leftarrow rt \cup \langle r, stime_{\langle o, r \rangle} + comm \rangle$
- **return** *rt*

pings, resource capabilities, and operator requirements. The approach initially obtains the region hierarchy and then establishes the deployment sequence employing a breadth-first search traversal algorithm [17] to give priority to upstream operators. Each operator of the deployment sequence has its response time estimated for non-constrained computational resources (Algorithm 2). After that, the resources are sorted in ascending manner by their response times and the host with the shortest response time is picked, and the host's residual capabilities are updated.

Response Time Rate with Region Patterns (RTR+RP) is a strategy that handles complex dataflows that contain multiple paths from sources to sinks. It explores the operator patterns (split points) and the sink placement (cloud or edge) respecting the environment constraints (Eqs. 15–21). Based on the region hierarchy (Fig. 3), the operators are classified and allocated. Operator 5, for instance, was reallocated since the edge does not respect the resource constraints. RTR+RP aims to allocate operators across edge and cloud meeting the response time rate only for operators located in the edge, in contrast to the RTR strategy that evaluates the response time rate for all operators.

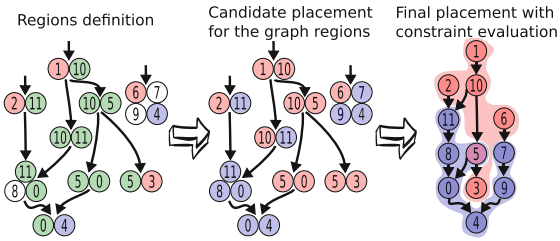

Fig. 3. Blue circles are operator candidates to be deployed on cloud whereas red circles are candidates for edge. The right-hand graph shows the final deployment. (Color figure online)

RTR+RP defines the deployment sequence similar to RTR, but it builds upon the classification of the operators considering the served sink infrastructure (candidate infrastructure). The classification is (i) *cloud-only* if the operator serves only sinks placed on the cloud, and (ii) *edge* if the operator shares paths with sinks located on edge. Each operator on the deployment sequence has its candidate infrastructure evaluated. Edge candidates have their response time estimated for non-constrained edge devices where the device with the shortest response time is picked. On the other hand, cloud candidates do not have their response time estimated, hence, the cloud hosts its operator candidates and those that do not meet the constraints on edge. At last, after the operator mapping, the resources have their residual capabilities updated.

4 Evaluation

In this section, we first describe the experimental setup and performance metrics and then discuss experimental results.

4.1 Experimental Setup

We built a framework atop OMNET++[1] to model and simulate distributed stream processing applications. A computational resource is an entity with CPU,

[1] Visit http://www.omnetpp.org/ for further details on OMNET++.

memory and bandwidth capabilities whereas operators comprise waiting queues and transformation operations that pose demands in terms of CPU, memory and bandwidth.

We model our edge devices as Raspberry PI's 2 (RPi) (*i.e.*, 4,74 MIPS[2] at 1 GHz and 1 GB of RAM), and the cloud as AMD RYZEN 7 1800x (*i.e.*, 304,51 MIPS[3] at 3.6 GHz and 1 TB of memory). The infrastructure comprises two cloudlets [13] with edge computing nodes (*Cloudlet 1* and *Cloudlet 2*) and a *Cloud*. Each cloudlet has 20 RPi's, whereas the cloud consists of 2 servers. A gateway interfaces each cloudlet's LAN and the external WAN [11] (the Internet). The LAN has a latency drawn from a uniform distribution between 0.015 and 0.8 ms and a bandwidth of 100Mbps. The WAN has latency drawn uniformly between 65 and 85 ms, and bandwidth of 1 Gbps [13].

As stream processing applications exist in multiple domains with diverse topologies (*e.g.*, face recognition, speech recognition, weather sensing), where sensors/actuators ingest a variety of events (*e.g.*, text, video, pictures, voice record) in the system, we aim to capture this diversity my modeling and simulating two scenarios with various application workloads:

Microbenchmarks: As in previous work [13], we first perform a controlled evaluation using a set of message sizes (10 bytes, 50 KB, and 200 KB) corresponding to multiple data types such as text, pictures/objects, and voice records. Each application, depicted in Fig. 4, has three input event rates (Table 1), a set of CPU requirements according to the message sizes (10 bytes - 3.7952 IPS, 50 KB -18976 IPS, and 200 KB - 75904 IPS) and a configuration of fork/join operators to explore the path sizes. The operators have multiple selectivity and data compression rates (100, 75, 50 and 25%). Sources ingest messages from sensors and sinks act as actuators on cloudlets and databases/message brokers on cloud.

Table 1. Input event rate.

App	10 bytes	50 KB	200 KB
App1	124999, 624999, 1249999	24, 124, 249	6, 31, 62
App2	124999, 374999, 624999	24, 74, 124	6, 19, 31
App3	124999, 218749, 300000	24, 43, 62	6, 10, 15
App4	124999, 137499, 150000	24, 27, 30	6, 7

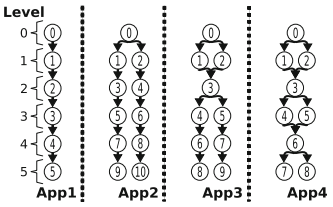

Fig. 4. Six-hop applications.

More Complex Applications: This scenario presents multiple operator behaviors and larger numbers of operators. We crafted a set of application graphs (Fig. 5) using a Python library[4] and varying the parameters of the operators using a uniform distribution with the ranges presented in Table 2. The cloudlets

[2] https://hackaday.com/2015/02/05/benchmarking-the-raspberry-pi-2/.
[3] https://reddit.com/r/BOINC/comments/5xog5v/boinc_performance_on_amd_ryzen.
[4] https://gist.github.com/bwbaugh/4602818.

Table 2. Operator attributes.

Parameter	Value
cpu	1–100
Data compression rate	10%–100%
mem	100–7500
Input event size	100–2500
Selectivity	10%–100%
Input event rate	1000–10000

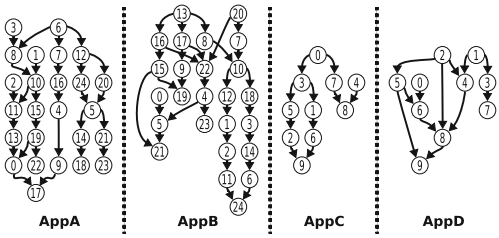

Fig. 5. Complex applications.

host the sink and source placements, except for the sink on the critical path which will be hosted on the cloud. We generated 1160 graphs randomly applying multiple selectivities, data compression rates, sink and source locations, input event sizes and rates, memory, and CPU requirements. Inspired on the size of RIoT-Bench [20] applications, a Realtime IoT Benchmark suite, we proposed two sets of applications, namely: (i) **large** (AppA and AppB) containing 25 operators; and (ii) **small** (AppC and AppD) holding 10 operators.

Metrics: The main performance metric is the *application response time*, which is the end-to-end latency from the time events are generated to the time they are processed by the sinks. To demonstrate the gains obtained by our approach, we compared the proposed strategies against a traditional approach (cloud-only) and a solution from the state-of-the-art [21] (LB). *Cloud-only* deploys all operators in the cloud, apart from operators provided in the initial placement. *Taneja et. al. (LB)* iterates a vector containing the application operators, gets the middle host of the computational vector and evaluates CPU, memory, and bandwidth constraints to obtain the operator placement.

4.2 Performance Evaluation

Figure 6 summarizes the response times for all microbenchmarks. For App1 we carried out 432 experiments (4 selectivities, 4 data compression rates, 3 input event rates, 3 sink locations and 3 input event sizes) for each solution with a pipeline application that may have messages with text, video, pictures, and voice record. Each experiment ran for 300 s in simulation time. RTR and RTR+RP have shown to be over 95% more efficient than cloud-only approach and LB. Initially, LB had its performance comparable to cloud-only, but LB lost performance afterward due to its specific modeling (*i.e.*, health care, and latency-critical gaming) and method (computational ordering).

Cloud-only achieved 5% better results (when the blue line crosses the red at ≈200 ms) when handling voice records (200 KB), selectivity, and data compression rate equal to 1 (without reducing the size of the messages and discarding events) and when the sink was placed on Cloudlet 2 and the source was located on Cloudlet 1 (traverse WAN). For the scenario mentioned before, the operators were CPU-intensive where Cloudlet 1 or 2 can host only one operator per edge device at a time, which increases the communication costs. Moreover, RTR+RP

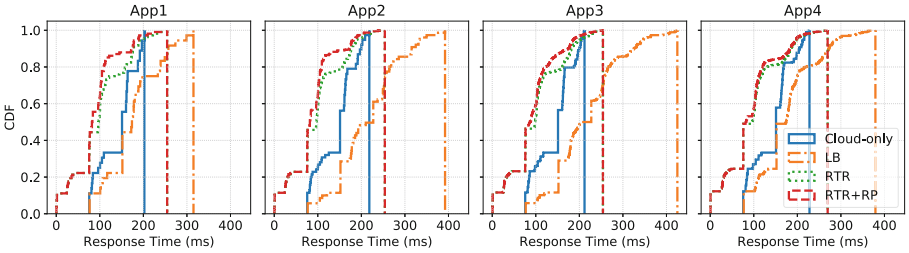

Fig. 6. CDF of response time for microbenchmarks. (Color figure online)

outperformed RTR for sinks placed on cloud, mainly without message discarding and no reduction on message sizes. Even further, to investigate the impacts generated by the split points, we launched App2, App3, and App4 and observed a gradual performance loss (decreasing on the distance between green and red line - ≈100 ms) according to the position between the split points and sinks, and the location of sinks. When sinks and sources require events to traverse the WAN and there is a low number of hops between the split point and sink, the proposed strategies cannot define a reasonable dataflow split because of the assumption to prioritize the sinks on the edge.

The complex application scenario investigates the outcomes for generic and multiple path applications using various dataflow configurations. We launched each experiment during 60 s of simulation time, and the sources and sinks were distributed uniformly and randomly across the infrastructure, except for operator 17 on AppA, operator 24 on AppB, operator 9 on AppC, and operator 9 on AppD placed in cloud due to the critical path. Figure 7 shows the CDF of response times. Even under large applications RTR+RP was able to reduce the response time by applying the region pattern identifications and recursively discovering the operator dependencies with a given sink placement onto dataflows with various paths.

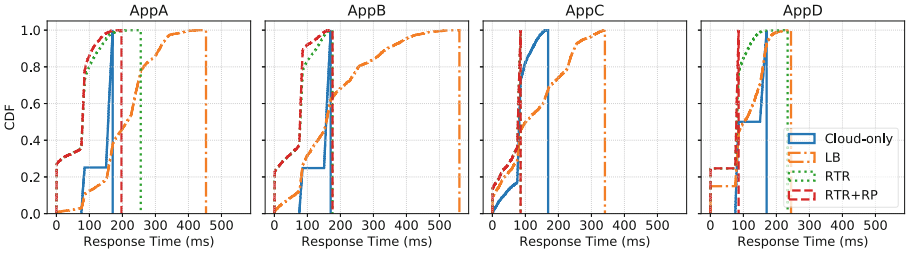

Fig. 7. CDF of response times for complex applications.

Our strategies outperformed cloud-only in over 6% and 50% under small and large applications, respectively. Cloud-only poses high communication overhead when the sink is located on cloudlets due to messages having to traverse the internet at least twice. Similarly, we improve the response times in over

23% (small) and 57% (large applications) compared to the LB approach. This occurs because LB does not estimate the communication overhead and assumes a shorter response time on cloudlets.

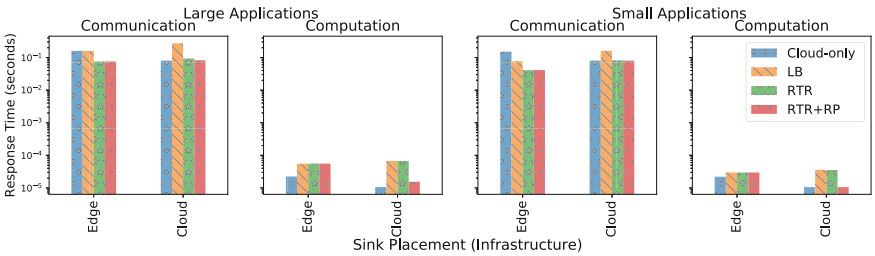

Fig. 8. Communication and computation time for sinks placed on cloud and cloudlets.

Figure 8 shows the communication latency which comprehends the total time to transfer a message between the resources, and the computation that corresponds to the total time to compute all operators. The communication cost for sinks placed on cloudlets at cloud-only was about 160 ms, and RTR+RP was 76 ms. Our solution outperformed cloud-only in up to 52% by putting operators closer to cloudlet sinks, but sinks on the cloud. RTR+RP had a slight performance loss of 3%. Hence, our approach is effective in reducing the communication cost, and, by doing so, it compensates the edge limitations and reaches good results in minimizing the total response time.

5 Related Work

IoT services are increasingly being employed on environments that span multiple data centers or on the edges of the Internet (*i.e.*, edge and fog computing). Existing work proposes architecture that places certain stream processing elements on micro data centers located closer to where the data is generated [3] or employs mobile devices for stream processing [9,15]. The problem of placing DSP applications onto heterogeneous hardware is at least NP-Hard as shown by Benoit *et al.* [2]. To simplify the placement problem, communication is often neglected [6], although it is a relevant cost in geo-distributed infrastructure [13]. Likewise, the operator behavior and requirements are oversimplified using static splitting decisions as proposed by Sajjad *et al.* [19].

Meanwhile, many efforts have been made on modeling the placement problem of DSP application on heterogeneous infrastructure [16] using Petri nets to formalize the application regions and the multiple response times that they produce. On the other hand, Eidenbenz *et al.* [8] evaluated SPDG from its parallelism degree to decompose the application graph and by an approximation algorithm to determine the placement. Cloud and edge have been explored to supply application requirements. For instance, Ghosh *et al.* [10] proposed a model

evaluating dynamically the time taken to process an operator into an exclusive computational resource as well as the communication times. Similarly, Taneja et. al. [21] offer a naive approach deploying the application graph across cloud and edge using a constraint sensitive approach.

All contributions to the problem of placing DSP applications evaluate edge devices for improving the network efficiency by treating the infrastructure and application constraints to unleash the full potential of applications triggered by IoT and smart scenarios. The present work considers edge and cloud, and the restrictions created by their interactions. The target scenario includes real-time analytics which comprises multiple forks and joins building multiple paths between data sources and sinks. Our solution exploits the construction of the paths to optimize the end-to-end application latency by decomposing the dataflow and defining the operator's placement dynamically.

6 Conclusions and Future Work

In this paper, we modeled the problem of placing DSP applications onto heterogeneous computational and network resources. We proposed two strategies to minimize the application response time by splitting the application graph dynamically and distributing the operators across cloud and edge. Our solutions were evaluated considering key aspects to identify application behaviors. The RTR strategy estimates the response time for each operator in all computational resources while RTR+RP strategy splits the dataflow graph using region patterns and then calculates the response time only for operators that are candidates to be deployed on edge.

We simulated the strategies' behavior and compared them against the state-of-the-art. The results show that our strategies are capable of achieving 50% better response time than cloud-only deployment when applications have multiple forks and joins. For future work, we intend to investigate further techniques to deal with CPU-intensive operators and their energy consumption.

Acknowledgements. This work was performed within the framework of the LABEX MILYON (ANR-10-LABX-0070) of the University of Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007).

References

1. de Assunção, M.D., da Silva Veith, A., Buyya, R.: Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J. Netw. Comput. Appl. **103**, 1–17 (2018)
2. Benoit, A., Dobrila, A., Nicod, J.M., Philippe, L.: Scheduling linear chain streaming applications on heterogeneous systems with failures. Future Gener. Comput. Syst. **29**(5), 1140–1151 (2013)
3. Buddhika, T., Pallickara, S.: Neptune: real time stream processing for internet of things and sensing environments. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1143–1152, May 2016

4. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Optimal operator placement for distributed stream processing applications. In: 10th ACM International Conference on Distributed Event-Based Systems, pp. 69–80. ACM, New York (2016)
5. Cardellini, V., Grassi, V., Presti, F.L., Nardelli, M.: Distributed QoS-aware scheduling in Storm. In: 9th ACM International Conference on Distributed Event-Based Systems, DEBS 2015, pp. 344–347. ACM, New York (2015)
6. Cheng, B., Papageorgiou, A., Bauer, M.: Geelytics: enabling on-demand edge analytics over scoped data sources. In: IEEE International Congress on BigData, pp. 101–108 (2016)
7. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. SIGCOMM Comput. Commun. Rev. **34**(4), 15–26 (2004)
8. Eidenbenz, R., Locher, T.: Task allocation for distributed stream processing. In: IEEE INFOCOM 2016, pp. 1–9, April 2016
9. Elbamby, M.S., Bennis, M., Saad, W.: Proactive edge computing in latency-constrained fog networks. In: European Conference on Networks and Communications, pp. 1–6, June 2017. https://doi.org/10.1109/EuCNC.2017.7980678
10. Ghosh, R., Simmhan, Y.: Distributed scheduling of event analytics across edge and cloud. ACM Trans. Cyber-Phys. Syst. **2**(4), 24 (2017, to Appear)
11. Ha, K., et al.: The impact of mobile multimedia applications on data center consolidation. In: IEEE International Conference on Cloud Engineering (IC2E), pp. 166–176, March 2013
12. Hochreiner, C., Vogler, M., Waibel, P., Dustdar, S.: VISP: an ecosystem for elastic data stream processing for the internet of things. In: 20th IEEE International Enterprise Distributed Object Computing Conference, pp. 1–11, September 2016
13. Hu, W., et al.: Quantifying the impact of edge computing on mobile applications. In: 7th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2016, pp. 5:1–5:8. ACM, New York (2016)
14. Kaur, N., Sood, S.K.: Efficient resource management system based on 4VS of big data streams. Big Data Res. **9**, 98–106 (2017)
15. Morales, J., Rosas, E., Hidalgo, N.: Symbiosis: sharing mobile resources for stream processing. In: IEEE Symposium on Computers and Communications Workshop, pp. 1–6, June 2014
16. Ni, L., Zhang, J., Jiang, C., Yan, C., Yu, K.: Resource allocation strategy in fog computing based on priced timed petri nets. IEEE IoT J. **4**(5), 1216–1228 (2017)
17. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: resource-aware scheduling in storm. In: 16th Annual Middleware Conference, Middleware 2015, pp. 149–161. ACM, New York (2015)
18. Ravindra, P., Khochare, A., Reddy, S.P., Sharma, S., Varshney, P., Simmhan, Y.: Echo: an adaptive orchestration platform for hybrid dataflows across cloud and edge. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-69035-3_28
19. Sajjad, H.P., Danniswara, K., Al-Shishtawy, A., Vlassov, V.: Spanedge: towards unifying stream processing over central and near-the-edge data centers. In: 2016 IEEE/ACM Symposium on Edge Computing, pp. 168–178, October 2016
20. Shukla, A., Chaturvedi, S., Simmhan, Y.: Riotbench: an IoT benchmark for distributed stream processing systems. Concurr. Comput.: Pract. Exp. **29**(21), e4257 (2017)
21. Taneja, M., Davy, A.: Resource aware placement of IoT application modules in fog-cloud computing paradigm. In: IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 1222–1228, May 2017