Monte-Carlo Tree Search and Reinforcement Learning for Reconfiguring Data Stream Processing on Edge Computing

Alexandre da Silva Veith, Marcos Dias de Assunção, Laurent Lefèvre
University of Lyon, ENS of Lyon, Claude Bernard University Lyon 1,
CNRS, Inria, Parallel Computing Lab (LIP),
F-69342, LYON Cedex 07, France
{alexandre.veith, marcos.dias.de.assuncao, laurent.lefevre}@ens-lyon.fr.

***Abstract*—Distributed Stream Processing (DSP) applications are increasingly used in new pervasive services that process enormous amounts of data in a seamless and near real-time fashion. Edge computing has emerged as a means to minimise the time to handle events by enabling processing (*i.e.*, operators) to be offloaded from the Cloud to the edges of the Internet, where the data is often generated. Deciding where to execute such operations (*i.e.*, edge or cloud) during application deployment or at runtime is not a trivial problem. In this work, we employ Reinforcement Learning (RL) and Monte-Carlo Tree Search (MCTS) to reassign operators during application runtime. Experimental results show that RL and MCTS algorithms perform better than traditional placement techniques. We also introduce an optimisation to a MCTS algorithm, called MCTS-Best-UCT, that achieves similar latency with fewer operator migrations and faster execution time. In certain scenarios, the time needed by MCTS-Best-UCT to find the best end-to-end latency is at least 62% smaller than the time required by the other algorithms.**

I. Introduction

Society is increasingly instrumented with sensors integrated to mobile phones, Internet of Things (IoT), and in systems for monitoring operational infrastructure, transportation and precision agriculture. These sources provide continuous data streams gathered and analysed by Distributed Stream Processing (DSP) applications that extract information required for decision making in near real-time.

A DSP application is often structured as 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, projection, and aggregation); and sinks (*i.e.*, queries that consume or store the data). Traditionally, DSP applications were conceived to run on clusters of homogeneous resources or/and on the cloud. 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 between data sources and cloud servers. *Edge computing* is an attractive solution for performing certain stream processing operations as many *edge devices* have non-trivial compute capacity and are often geographically closer[1] to where the data is generated.

The task of scheduling operators of a DSP application on available computing resources is generally referred to as *operator placement*; a problem that is exacerbated when considering the joint exploration of cloud and edge devices as assigning operators to heterogeneous resources has proven to be NP-hard [1]. Moving operators from cloud to edge devices is also challenging due to limitations of devices and network latency. Existing work proposed several techniques to address the operator placement problem [2], some of which consider only cloud [3], [4], and others which consider edge computing [5], [6], [7], [8], [9]. In previous work, we considered the placement of applications with feedback loops (*i.e*, data sinks/actuators at the edge) onto cloud and edge resources [10].

Although the initial placement is important, operators may need to be reconfigured during an application life-cycle due to variable load conditions or device failures. The solution search space for operator reconfiguration can be enormous depending on the number of operators, streams, resources and network links. Moreover, it is important to minimise the number of migration while improving the application end-to-end latency due to the time required to transfer and catch up the operator data. Reinforcement Learning (RL) and Monte-Carlo Tree Search (MCTS) have been used to tackle problems with large search spaces and states [11], [12], performing at human-level or better in games such as Go. In the present work, we model the operator reconfiguration problem as a Markov Decision Process (MDP) and investigate the use of RL and MCTS algorithms to devise reconfiguration plans that improve the end-to-end latency of DSP.

Hence, the main contributions of this paper are:

- We model the operator reconfiguration problem as an MDP and employ RL algorithms to devise reconfiguration plans that minimise the aggregate end-to-end latency of data events;
- a performance evaluation comparing the RL and MCTS

[1]We use geographical closeness as synonymous to latency closeness.

algorithms to traditional strategies used to compute the placement of DSP operators; and
- a domain optimisation that improves the search time of a MCTS-UCT algorithm.

The rest of this paper is organised as follows. Section II presents background information on Markov decision processes and reinforcement learning algorithms. Section III describes the system model and the optimisation problem for reconfiguring data stream processing applications. The proposed solutions are introduced in Section IV. The experimental setup and performance evaluation are presented in Section V. Section VI discusses related work, whereas Section VII concludes the paper.

II. Background

This section describes background on Markov decision process and reinforcement learning algorithms.

A. Markov Decision Process

A Markov Decision Process (MDP) provides a decision-making framework where an agent makes decisions by interacting with a simulated environment over a number of steps. An MDP often comprises a set of environment states $\mathcal{S}$ including the initial state s_0 and a terminal state $s_{|\mathcal{S}|-1}$, where each state s has a set of possible actions $\mathcal{A}(s)$ and a reward function $R(s)$. At a non-terminal state, the agent picks an available action and interacts with the simulated environment to determine the state and reward for the next step.

The goal of solving an MDP is to determine the mapping from states to actions (*i.e.*, policy), which maximises the reward. When the transition model and reward function are available, dynamic programming easily solves this task. Otherwise, the concept of an iterative approach remains the backbone of most RL algorithms. These algorithms apply greedy approaches based on MCTS and/or Temporal-Difference Tree Search (TDTS) in order to keep track of state transitions when evaluating the application using the MDP framework and by applying mathematical approaches to balance exploration of new solutions and exploitation of good and well-known ones.

B. Reinforcement Learning Algorithms

Reinforcement learning algorithms such as MCTS, TDTS and Q-Learning are considered in this work for reconfiguring DSP applications. By using such algorithms, an agent (*i.e.*, the scheduler) interacts with a simulated environment using a model described later in Section III and transitions along states that maximise the reward.

1) Monte-Carlo Tree Search: is a simulation-based search mechanism consisting of running a number of simulations and building a search tree with the results [11]. Each node $n(s)$ of the search tree $\mathcal{T}$ represents a state s that has been seen during simulation. A node/state maintains a count $N(s)$ with the number of times it has been visited, an action value $Q(s,a)$ for each action $a \in A(s)$ and a count $N(s,a)$ with the number of times the action has been picked.

For each episode, the estimated value function can be updated with an incremental mean. A simulation or episode starts at the root state s_0 and is divided into two phases. First, when the state s_t is found in the search tree, a *tree policy* is employed to select an action. Otherwise in the second phase a default policy continues the simulation until a terminal state. The simplest policy is *greedy* that selects $max_a Q(s,a)$ in the first phase and random actions during the second phase. MCTS attempts to approximate value functions from experience.

MCTS with a greedy policy can lead to inefficiencies by selecting actions among a small set, avoiding other actions after a few poor outcomes. The *MCTS with Upper Confidence Bounds for Trees (MCTS-UCT)* uses an optimistic approach in the face of uncertainty by giving a bonus that represents the uncertainty in the value of a state-action (*i.e.*, exploration-exploitation dilemma [13]). The tree policy picks actions using the UCB1 algorithm [13], which maximises an Upper Confidence Bound (UCB) on the action values. The policy tree selects the action a^* that maximises the UCB value (C is a scalar exploration constant):

$$Q(s,a) = Q(s,a) + C\sqrt{\frac{2lnN(s)}{N(s,a)}} \tag{1}$$

$$a^* = max_a Q(s,a) \tag{2}$$

2) Temporal-Difference Tree Search: while MCTS needs to wait until an episode ends to update the node and action values, the basic implementation of Temporal Difference (TD), *i.e.*, $TD(0)$, waits until the next step and updates the values after transitioning to a new state s_t and receiving the reward $R(s_t)$. Since TD bases its update on an existing estimate, it is said to be a *bootstrapping* method. A TD variant that unifies TD and MCTS and allows for specifying the number of future states on which estimates are evaluated is $TD(\lambda)$ where a large value for λ will eventually result in MCTS behaviour.

TDTS-Sarsa(λ) is a TD method that combines Sarsa(λ) and UCT algorithms. The general Sarsa derives its name from how the policy evaluation algorithm is structured. A state-action (s,a) pair yields a reward R and takes the execution to a new state, s', at which the policy picks action a', and the value of this transition is evaluated to $Q(s',a')$. Sarsa(λ) considers m steps of experience [12].

3) Q-learning: is a learning algorithm where an agent tries to learn an optimal state-action transition policy based on state-action rewards that it receives by interacting with the environment [13]. The agent computes the return of state-actions, the so called *Q*-values, so that it picks actions that maximise the reward. With *Q*-values computed the value of state s is:

$$Q(s) = max_a Q(s,a) \tag{3}$$

Action values are updated when transitioning from state s to s' as follows:

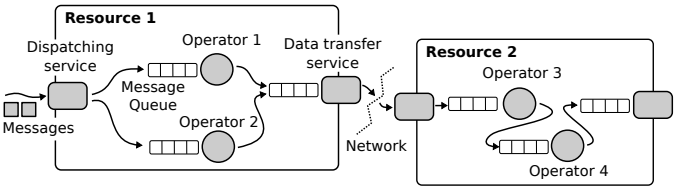


Fig. 1: Example of four operators and their respective message queues placed on two resources.

$$Q(s,a) = Q(s,a) + \alpha[R(s) + \gamma max_{a'} Q(s',a') - Q(s,a)] \quad (4)$$

where α is the learning rate and γ is the discount factor [13].

III. System Model and Problem Definition

This section describes the system model for operator placement introduced in our previous work [10] and the reconfiguration problem tackled in the present paper.

A. System Model

The infrastructure is a graph[2] $\mathcal{N} = (\mathcal{R}, \mathcal{L})$ where $\mathcal{R}$ is the set of compute resources in the cloud and/or at the edge sites and $\mathcal{L}$ comprises the logical links interconnecting the resources. A compute resource is a tuple $r_k = \langle cpu_k^r, mem_k^r \rangle$, where cpu_k^r is its CPU capability in MIPS, and mem_k^r is its memory capacity in bytes. A network link is a tuple $l_{k \leftrightarrow l} = \langle bdw_{k \leftrightarrow l}, lat_{k \leftrightarrow l} \rangle$, where $k \leftrightarrow l$ represents the interconnection of resources k and l, $bdw_{k \leftrightarrow l}$ the bandwidth in bps, and $lat_{k \leftrightarrow l}$ the latency in seconds. The latency of resource k to itself (*i.e* $lat_{k \leftrightarrow k}$) is 0.

A DSP application is a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{O}, \mathcal{E})$ of *operators* $\mathcal{O}$ that execute functions over the incoming data, and *streams* $\mathcal{E}$ of data events flowing between operators. Each operator is a tuple $o_i = \langle cpu_i^o, mem_i^o, \psi_i^o, \omega_i^o \rangle$, where cpu_i^o is the CPU requirement to handle an individual event, mem_i^o is the memory in bytes to load the operator, ψ_i^o is the ratio of number of input events to output events (*i.e.*, selectivity), and ω_i^o is the ratio of the size of input events to the size of output events (*i.e.*, data compression/expansion factor). The rate at which operator i can process events at resource k is denoted by $\mu_{\langle i,k \rangle}$ and is essentially $\mu_{\langle i,k \rangle} = cpu_k^r \div cpu_i^o$. An event stream $e_{k \to l}^{\rho} \in \mathcal{E}$ connects operator k to l with a probability ρ that an output event emitted by k will flow through to l.

The rate at which operator i produces events, λ_i^{out}, is a product of its input event rate λ_i^{in} and its selectivity ψ_i^o. The output event rate of a source operator depends on the number of measurements it takes from a sensor or a monitored device. 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.

[2]The graph is directed, with each link comprising two arcs, namely for download and upload. For the sake of simplicity, however, we represent links here as bidirectional.

A compute resource can host one or more operators. Operators within the same node communicate directly whereas inter-node communication is done via a communication service that serialises events to be sent to another node, as depicted in Figure 1. Both operators and the communication service handle events in a First-Come, First-Served (FCFS) fashion and follow an M/M/1 queue model which allows for estimating the computation and communication times. The computation time $stime_{\langle o_i, r_k \rangle}$ of operator i on resource k is given by:

$$stime_{\langle i,k \rangle} = \frac{1}{\mu_{\langle i,k \rangle} - \lambda_i^{in}} \quad (5)$$

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

$$ctime_{\langle i,k \rangle \langle j,l \rangle} = \frac{1}{\left(\frac{bdw_{k \leftrightarrow l}}{\varsigma_i^{out}}\right) - \lambda_j^{in}} + l_{k \leftrightarrow l} \quad (6)$$

A mapping function $\mathcal{M} : \mathcal{O} \to \mathcal{R}$, $\mathcal{E} \to \mathcal{L}$ indicates the resource to which an operator is assigned and the link(s) to which a stream is mapped. The function $mo_{\langle i,k \rangle}$ returns 1 if operator i is placed on resource k and 0 otherwise. Likewise, the function $ms_{\langle i \to j, k \leftrightarrow l \rangle}$ 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 $p_i = o_0, o_1, \ldots, o_{n-1}, o_n$ is a sequence of n operators and $n-1$ streams, starting at a source o_0 and ending at a sink o_n. The set of all possible paths in the application graph is $\mathcal{P}$. The end-to-end latency L_{p_i} of a path p_i is 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, L_{p_i} is:

$$\begin{aligned} L_{p_i} = & \sum_{o \in \mathcal{O} r \in \mathcal{R}} mo_{\langle o,r \rangle} \times stime_{\langle o,r \rangle} \\ & + \sum_{r' \in \mathcal{R}} ms_{\langle o \to o+1, r \leftrightarrow r' \rangle} \times ctime_{\langle o,r \rangle \langle o+1,r' \rangle} \end{aligned} \quad (7)$$

B. Operator Placement and Reconfiguration Problem

Placing or scheduling a DSP application consists of finding a mapping $\mathcal{M} : \mathcal{O} \to \mathcal{R}$, $\mathcal{E} \to \mathcal{L}$ that minimises the *Aggregate End-to-End Latency (AL)* of all paths (Equation 8) that respects the resource and network constraints. A detailed description of all constraints is given in our previous work [10].

$$AL = \min \sum_{p_i \in \mathcal{P}} L_{p_i} \quad (8)$$

As DSP applications are often long-running, during their life-cycle they can experience variable load requirements that change the working conditions of operators. Unlike the cloud, edge resources are often more constrained and less reliable, with higher failure rates. To preserve the application performance within acceptable bounds it is important to adjust the initial placement and conveniently migrate operators to available resources. Migrating operators can incur a cost regarding

storing operator state, stopping an operator, migrating it and resuming it on a target resource. The present work focuses on *finding a new mapping $\mathcal{M} : \mathcal{O} \rightarrow \mathcal{R}$, $\mathcal{E} \rightarrow \mathcal{L}$ or plan that improves the AL compared to the initial placement.*

IV. Reinforcement Learning Based Reconfiguration of DSP Applications

This section first details how the reconfiguration is modelled as an MDP. After that, it presents an extension to the Upper Confidence Bounds for Trees (UCT) algorithm, and a domain optimisation to sort operators to be migrated

A. The MDP for DSP Reconfiguration

In the MDP considered for reconfiguration, a state $s \in \mathcal{S}$ contains a mapping of operator/stream onto resource/link(s) (*i.e.*, $\mathcal{M} : \mathcal{O} \rightarrow \mathcal{R}$, $\mathcal{E} \rightarrow \mathcal{L}$), where s_0 is the mapping provided by the placement algorithm. At each state there are x possible actions where x is the number of computational resources available. The set of actions $\mathcal{A}(s)$ at state s comprises the possible migrations of an operator to another resource, or maintaining the current mapping.

We can simulate the expected AL of new states by applying the model in Section III. The reward $R(s)$ of a state s is given by the reduction of its aggregate latency AL_s compared to the aggregate latency AL_{s_0} given by the original mapping:

$$R(s) = AL_{s_0} - AL_s \tag{9}$$

By solving the MDP one obtains a policy $\pi(s) : s \in \mathcal{S} \mapsto a \in \mathcal{A}(s)$ with the migrations needed to reconfigure an operator deployment. An *optimal policy* maximizes the reward and the migration actions that minimise the AL.

B. MCTS-Best-UCT

We propose a strategy that extends MCTS-UCT by storing the UCB value at each node and enabling a search for the best UCT node. The MDP reconfiguration model allows for making any node terminal as it contains a valid stream and operator mapping. Algorithm 1 depicts the $MCTSBestUCT$ function that receives the current mapping (s_0), initialises the search mechanism and builds the tree. $MCTSBestUCT$ is similar to MCTS-UCT, but with different function behaviours. $TreePolicy$ (line 9) returns the node with highest UCT value, whereas MCTS-UCT starts the search at the root and estimates the UCT values up to a new state $n' \notin \mathcal{T}$. $DefaultPolicy$ (lines 15-17) expands and simulates ($f(s(n), a)$) for the new node (n') taking a random action from the input node. $Backup$ (lines 19-23) includes and updates the UCB value for each node as well as the $N(s)$ count and Q value.

Algorithm 1: The MCTS-Best-UCT algorithm.

```
1  Function MCTSBestUCT(s_0)
2      create root node n_0 with state s_0
3      while within computational budget:
4          n ← TreePolicy()
5          n', Δ ← DefaultPolicy(n)
6          Backup(n', Δ)
7      return BestChild()
8  Function TreePolicy(n)
9      return BestChild()
10 Function Expand(n)
11     choose a ∈ untried actions from A(s(n)) at random
12     add a new child n' to n with s(n') = f(s(n), a)
         and a(n') = a
13     return n'
14 Function DefaultPolicy(n)
15     n' ← Expand(n)
16     Δ ← R(s(n'))
17     return n', Δ
18 Function Backup(n, Δ)
19     while n is not null:
20         N(n) ← N(n) + 1
21         Q(n) ← Q(n) + Δ
22         UCT(n) = Q(s,a)/N(s,a) + C sqrt(2 ln N(s) / N(s,a))
23         n ← parent of n
24 Function BestChild()
25     return argmax_{n' ∈ T} UCT(n')
```

C. Building a Deployment Hierarchy (DH)

The search space of the application reconfiguration can be large due to the number of available computing resources and application operators. In IoT scenarios, DSP applications can be time-sensitive and have certain data sinks placed on the cloud and other sinks on the edge, which provide information to actuators or warning systems or alert data sinks. When optimising the AL, one can assume that operators that communicate only with data sinks on the cloud have less priority than operators that send data to sinks on the edge.

We propose a domain optimisation approach that sorts operators by their potential impact on aggregate end-to-end latency when assessing the application reconfiguration. The approach builds a hierarchy of region dependencies (*i.e.* downstream and upstream relations between regions) and ignores sources and sinks as we consider that their placement is user-defined and do not change. Hereafter called *Deployment Hierarchy (DH)*, the approach identifies the *split points* used to build a deployment hierarchy of operators and determine what operators can be placed on the edge and which should be hosted in the cloud. This deployment hierarchy is used for sorting operators to build the MDP for reconfiguration, hence increasing the chance that operators with greater impact on AL, and that are likely to be moved to edge resources, are evaluated first.

V. Evaluation

This section describes the experimental setup and results.

A. Experimental Setup

We use a framework built in house atop OMNET++ [10] to model and simulate DSP applications. We resort to simulation as it provides a controllable and repeatable environment. Our

TABLE I: Operator attributes.

Parameter	Value	Unit
cpu	1-100	MIPS
Data compression rate	10-100	%
mem	100-7500	bytes
Input event size	100-2500	bits/second
Selectivity	10-100	%
Input event rate	1000-10000	Number of messages

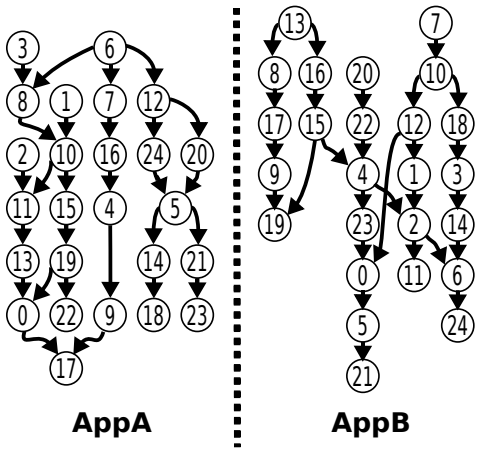


Fig. 2: Evaluated applications.

edge devices are modelled as Raspberry PI's 2 (RPi) (*i.e.*, 4,74 MIPS 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 edge sites with 20 RPi's each and a *Cloud* with 2 servers. A gateway interfaces each edge site's LAN and the external WAN [14] (the Internet). The LAN latency is uniformly distributed between 0.015 and 0.8 ms with a bandwidth of 100 Mbps. The WAN latency is drawn uniformly between 65 and 85 ms, and bandwidth of 1 Gbps. The latency values are consisted with measurements carried out in previous work [15].

We consider two applications with multiple data paths, shown as *AppA* and *AppB* in Figure 2. The graphs were crafted using a Python library[4] and their orders are based on the size of RIoTBench [16] applications – a Realtime IoT Benchmark suite. The operator behaviours vary with their parameters uniformly drawn from the values in Table I. Edge devices host sources (3, 6 for AppA and 7, 13 for AppB) and sinks (18, 23 for AppA and 11, 19, 24 for AppB), except for the sink on the critical path (17 for AppA and 21 for AppB), which is hosted on the cloud.

The reinforcement learning algorithms consider an execution budget of 10000 simulations. The considered metrics used to evaluate the algorithms performance are:

- **latency improvement** in (%), which represents the best percentage of latency improvement under the given execution budget of the MCTS algorithms and Q-Learning;
- **algorithm execution time** (in seconds), the time required by each algorithm to complete the execution budget;
- **time to best latency** (in seconds), the time required by the algorithm to find the best latency achievable under the allotted execution budget;
- **number of operator migrations** needed by the devised plan to achieve the latency improvement; and
- **minimum AL** (in ms) achieved when the simulations of the allotted budget finish.

B. Performance Evaluation

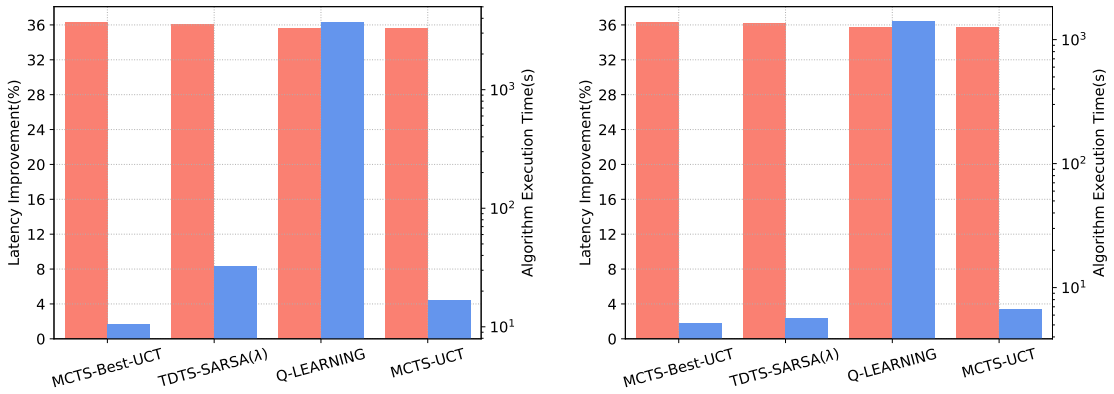
The entire application graph, except for sinks outside the critical path and data sources, are initially placed on the cloud. This is called the *cloud-only* placement. The algorithms are then evaluated under the following scenarios:

- **Scenario 1**: The MCTS algorithms, Q-Learning and MCTS-Best-UCT receive the cloud-only placement and run during the assigned execution budget in order to devise a reconfiguration plan that improves the AL. The algorithms are evaluated under two distinct cases, namely without DH and with DH.
- **Scenario 2**: Consists of evaluating the minimum AL achieved the proposed solution and by state-of-the-art algorithms: cloud-only, Taneja's [17], and RTR and RTR-RP that we have proposed in previous work [10]). We also evaluate the baseline approaches in MCTS-UCT, TDTS-Sarsa(λ), and Q-Learning – described in Section II. *Taneja*'s algorithm iterates each item of the application graph, organises the computational resources and gets the middle term resource considering the CPU capacity. *RTR* is a greedy algorithm that places operators incrementally by evaluating the end-to-end latency of paths, and *RTR-RP* explores application graph patterns and the location where the sink is assigned to optimise placement. RTR-RP is essentially RTR with DH.

1) Scenario 1: Figures 3 and 4 summarise the results on latency improvement and execution time of each algorithm for AppA and AppB, respectively. All algorithms substantially improve the AL, with improvements above 30% for AppA and over 45% for AppB. MCTS-Best-UCT achieves improvements that are similar to those obtained by the other algorithms. However, it requires less time to achieve such improvements, specially under the case without DH. MCTS-Best-UCT performs better because it needs to simulate only the next state, s' for obtaining the Q-value for bootstrapping, unlike the other algorithms that need to iterate until a terminal state. Moreover, the introduction of DH in the other algorithms improves their execution times, but they still remain higher than those achieved by MCTS-Best-UCT.

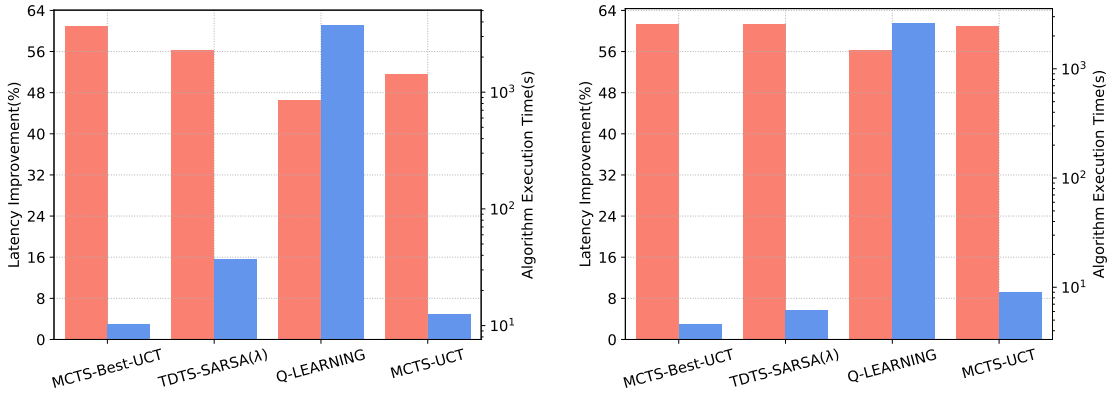
Figure 5 shows the time required by each algorithm to find the best latency achievable under the allotted execution budget. MCTS-Best-UCT again provides the shortest time to best latency compared to the other algorithms. For AppA, the time needed by MCTS-Best-UCT is at least 62% smaller than the time required by MCTS-UCT – the best of the remaining algorithms without DH – and over 85% smaller than MCTS-UCT with DH. The introduction of DH improves the time to best latency of all algorithms, except for Q-learning.

[3]https://reddit.com/r/BOINC/comments/5xog5v/boinc_performance_on_amd_ryzen
[4]https://gist.github.com/bwbaugh/4602818



(a) Without DH (b) With DH

Fig. 3: Latency improvement and the algorithms execution time for AppA.



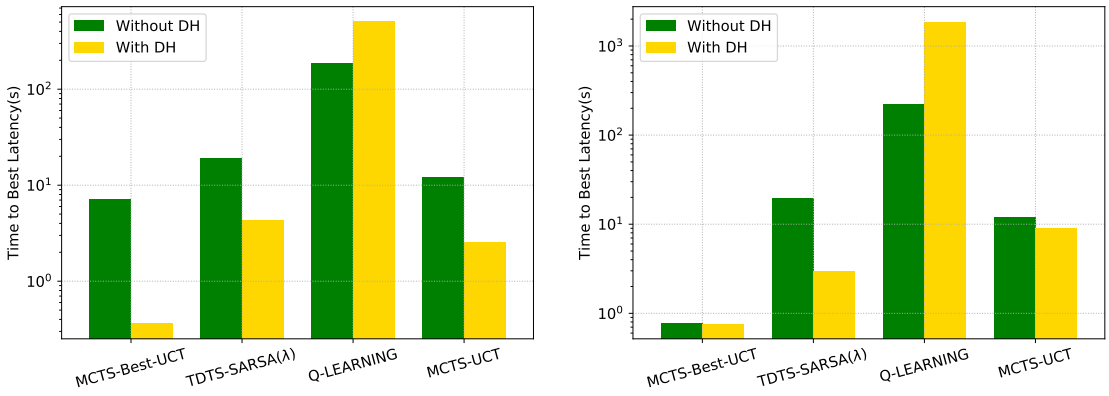
(a) Without DH (b) With DH

Fig. 4: Latency improvement and the algorithms execution time for AppB.

The overhead posed by computing the deployment hierarchy is detrimental to Q-learning, which is already compute and memory intensive as it needs to maintain a large table of Q-values. DH does not improve the time to best latency of MCTS-Best-UCT for AppB by much because the application has its latency improved by migrating most operators to edge resources (see Figure 6), which is discovered early on by the algorithms. From this point onwards, the gain brought by DH becomes negligible.

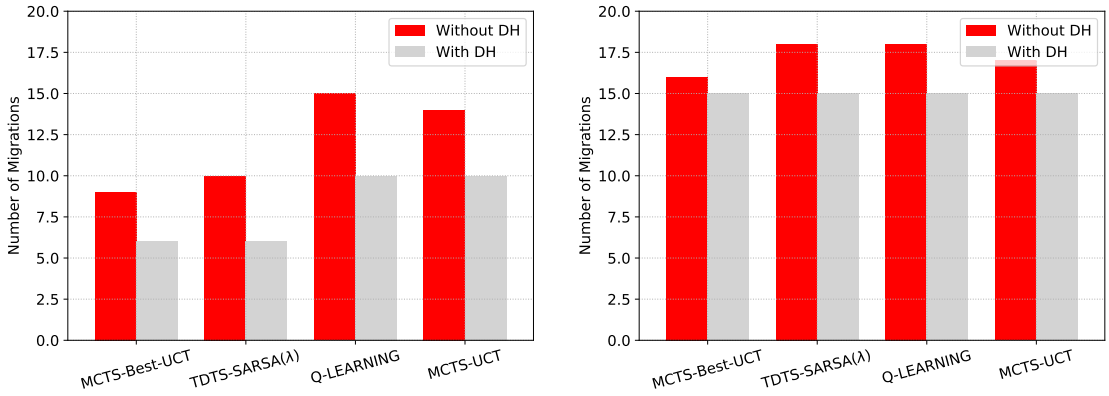
Figure 6 summarises the number of operator migrations for AppA and AppB without DH and with DH. Without DH, MCTS-Best-UCT requires fewer migrations than the other algorithms. This is because it finds the operators that have the greatest impact on AL and moves them to edge resources, mainly operators that are selective, which in turn reduces the amount of data transferred over the WAN. The other algorithms require the support of DH to perform similarly to MCTS-Best-UCT. However, as shown earlier, they have a higher execution overhead.

2) Scenario 2: Figure 7 summarises the results on minimum AL with and without DH. The RTR and Cloud-only are evaluated without applying DH because they consider the whole application dataflow. RTR-RP, as mentioned earlier, builds the DH as the algorithm optimises the number of operators to be reassigned. The results show that the RL approaches can improve and provide more stable operator reconfigurations regarding AL than the state-of-the-art. The reason is that RL algorithms balance exploration and exploitation of solutions. Also considering AL, MCTS-Best-UCT outperforms Taneja's algorithm and Cloud-only by over 48%, and RTR by over 20% without DH. With DH our proposed algorithm still outperforms RTR-RP by over 5%. When compared to the other RL algorithms, on average MCTS-Best-UCT reduces the latency by ≈4%. This improvement seems small, but needs to be considered together with the substantial reduction in algorithm execution time that it achieves as discussed in Scenario 1.



(a) AppA (b) AppB

Fig. 5: Time to achieve the best latency for AppA and AppB.



(a) AppA (b) AppB

Fig. 6: Number of operator migrations for AppA and AppB.

VI. Related Work

Existing work proposes architecture to place certain stream processing elements on resources located closer to where the data is generated [18], [7] or employs mobile devices for stream processing [19], [20]. The problem of placing DSP applications onto heterogeneous hardware is at least NP-Hard as shown by Benoit *et al.* [1]. To simplify the placement problem, communication is often neglected [7]. Likewise, the operator behaviour and requirements are oversimplified using static splitting decisions as proposed by Sajjad *et al.* [5].

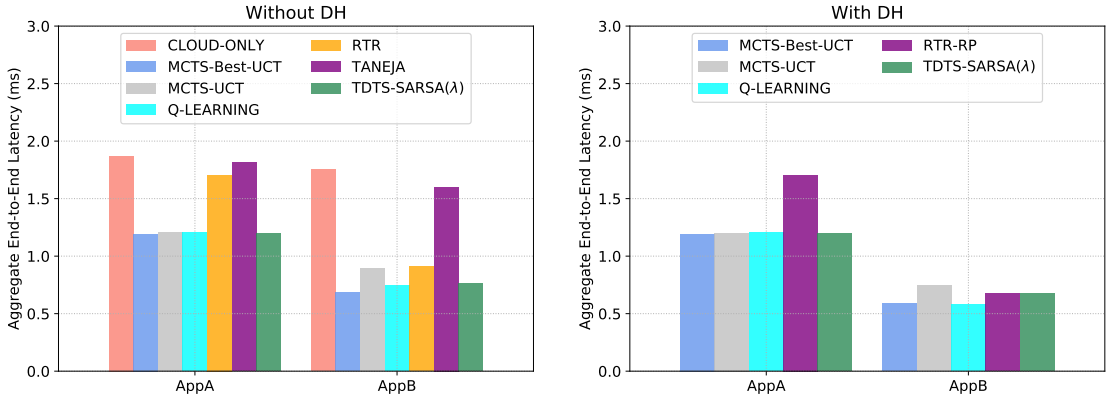
Effort has been made on modelling the placement problem of DSP applications on heterogeneous infrastructure [9] using techniques such as Petri nets. Eidenbenz *et al.* [21] evaluated series-parallel-decomposable graphs to decompose the application graph and use an approximation algorithm to determine the placement. Taneja *et. al.* [17] offer a naive approach for deploying an application graph across cloud and edge while respecting a set of constraints.

Reinforcement learning [13] and MCTS have demonstrated great potential in addressing problems with large search spaces, such as the game of Go [11]. It has been used in addressing scheduling problems [22], [23] and elasticity of DSP applications [24]. The present work considers RL and MCTS for addressing the problem of reconfiguring DSP applications on cloud-edge environments while minimising the aggregate end-to-end latency of processing graphs.

VII. Conclusions and Future Work

This paper explored the reconfiguration of data stream processing applications. It proposed the MCTS-Best-UCT algorithm to reconfigure the applications in order to minimise the aggregate end-to-end latency while reducing the number of required iterations in the simulations to create an episode. We optimise the number of operators to be reassigned by applying a method named Deployment Hierarchy, which prioritises certain operators and avoids reassigning those that only process and forward events to sinks on the cloud. We also compared the performance of Monte-Carlo Tree Search and Reinforcement Learning algorithms.

Our solution was evaluated considering applications with generic application behaviours. We simulated reassigning ap-



(a) Without DH

(b) With DH

Fig. 7: Minimal AL of state-of-the-art against RL approaches with/without DH.

plication operators using our approach and compared it against state-of-the-art MCTS algorithms and Q-Learning. The results showed that our approach is capable of achieving similar or better latency improvement while being faster and requiring fewer operator migrations. In certain scenarios, the time needed by MCTS-Best-UCT to find the best aggregate end-to-end latency under a given execution budget is at least 62% smaller than the time required by the other evaluated algorithms.

As future work we aim to investigate machine learning techniques to optimise the energy consumption, and the monetary and migration costs of reconfiguring data stream processing applications a real cloud-edge infrastructure.

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] A. Benoit *et al.*, "Scheduling linear chain streaming applications on heterogeneous systems with failures," *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1140–1151, Jul. 2013.

[2] G. T. Lakshmanan *et al.*, "Placement strategies for internet-scale data stream systems," *IEEE Internet Computing*, vol. 12, no. 6, pp. 50–60, November 2008.

[3] J. Xu *et al.*, "T-Storm: Traffic-aware online scheduling in storm," in *IEEE 34th Int. Conf. on Distributed Computing Systems (ICDCS)*, June 2014, pp. 535–544.

[4] T. Li *et al.*, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proc. VLDB Endow.*, vol. 11, no. 6, pp. 705–718, Feb. 2018. [Online]. Available: https://doi.org/10.14778/3199517.3199521

[5] H. P. Sajjad *et al.*, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing*, Oct 2016, pp. 168–178.

[6] C. Hochreiner *et al.*, "VISP: An ecosystem for elastic data stream processing for the Internet of things," in *IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC)*, September 2016, pp. 1–11.

[7] B. Cheng *et al.*, "Geelytics: Enabling on-demand edge analytics over scoped data sources," in *IEEE BigData*, 2016, pp. 101–108.

[8] V. Cardellini *et al.*, "Distributed QoS-aware scheduling in Storm," in *9th ACM DEBS*, New York, USA, 2015, pp. 344–347.

[9] L. Ni *et al.*, "Resource allocation strategy in fog computing based on priced timed petri nets," *IEEE IoT Journal*, vol. PP, pp. 1–1, 2017.

[10] A. Veith *et al.*, "Latency-aware placement of data stream analytics on edge computing," in *ICSOC 2018*, Nov 2018, pp. 215–229.

[11] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[12] T. Vodopivec *et al.*, "On monte carlo tree search and reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 60, pp. 881–936, 2017.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An introduction*. MIT press, 2018.

[14] K. Ha *et al.*, "The impact of mobile multimedia applications on data center consolidation," in *IEEE International Conference on Cloud Engineering (IC2E)*, March 2013, pp. 166–176.

[15] W. Hu *et al.*, "Quantifying the impact of edge computing on mobile applications," in *7th ACM SIGOPS Asia-Pacific Workshop on Systems*. New York, USA: ACM, 2016, pp. 5:1–5:8.

[16] A. Shukla *et al.*, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[17] M. Taneja and A. Davy, "Resource aware placement of iot application modules in fog-cloud computing paradigm," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2017, pp. 1222–1228.

[18] T. Buddhika and S. Pallickara, "Neptune: Real time stream processing for internet of things and sensing environments," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*, May 2016, pp. 1143–1152.

[19] J. Morales *et al.*, "Symbiosis: Sharing Mobile Resources for Stream Processing," in *IEEE Symposium on Computers and Communication – Workshops*, June 2014, pp. 1–6.

[20] M. S. Elbamby *et al.*, "Proactive edge computing in latency-constrained fog networks," in *2017 European Conference on Networks and Communications*, June 2017, pp. 1–6.

[21] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *IEEE 35th Annual International Conference on Computer Communications (INFOCOM 2016)*, April 2016, pp. 1–9.

[22] A. I. Orhean *et al.*, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 292–302, 2018.

[23] L. Mai *et al.*, "Real-time task assignment approach leveraging reinforcement learning with evolution strategies for long-term latency minimization in fog computing," *Sensors*, vol. 18, no. 9, p. 2830, 2018.

[24] G. R. Russo *et al.*, "Multi-level elasticity for wide-area data streaming systems: A reinforcement learning approach," *Algorithms*, vol. 11, no. 9, p. 134, 2018.