Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

Valeria Cardellini
cardellini@ing.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

Francesco Lo Presti
lopresti@info.uniroma2.it
University of Rome Tor Vergata
Rome, Italy

ABSTRACT

Data Stream Processing (DSP) has emerged as a key enabler to develop pervasive services that require to process data in a near real-time fashion. DSP applications keep up with the high volume of produced data by scaling their execution on multiple computing nodes, so as to process the incoming data flow in parallel. Workloads variability requires to elastically adapt the application parallelism at run-time in order to avoid over-provisioning. Elasticity policies for DSP have been widely investigated, but mostly under the simplifying assumption of homogeneous infrastructures. The resulting solutions do not capture the richness and inherent complexity of modern infrastructures, where heterogeneous computing resources are available on-demand. In this paper, we formulate the problem of controlling elasticity on heterogeneous resources as a Markov Decision Process (MDP). The resulting MDP is not easily solved by traditional techniques due to state space explosion, and thus we show how linear Function Approximation and Tile Coding can be used to efficiently compute elasticity policies at run-time. In order to deal with parameters uncertainty, we integrate the proposed approach with Reinforcement Learning algorithms. Our numerical evaluation shows the efficacy of the presented solutions compared to standard methods in terms of accuracy and convergence speed.

CCS CONCEPTS

• **Information systems** → *Stream management*; • **Computing methodologies** → **Reinforcement learning**; **Markov decision processes**; • **Computer systems organization**;

KEYWORDS

Elasticity, Reinforcement Learning, Markov Decision Process, Function Approximation, Tile Coding

ACM Reference Format:
Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329506

DEBS '19, June 24–28, 2019, Darmstadt, Germany

ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329506

1 INTRODUCTION

Distributed Data Stream Processing (DSP) systems play a key role in the emerging Internet of Things and Smart City landscape, as they allow for near real-time analysis of fast incoming data streams, comprising an ever growing amount of data collected by a countless number of sensors, wearable devices, monitoring systems, etc. A DSP application can be represented as a directed acyclic graph, with data sources, operators, and final consumers as vertices, and streams as edges. Each *operator* can be seen as a black-box processing element, that continuously receives incoming streams, applies a transformation, and generates new outgoing streams.

In order to deal with the highly variable and unpredictable data rates while keeping processing latency under control, a commonly adopted stream processing optimization technique is *data parallelism*, which consists in replicating an operator into several parallel instances, so that each instance processes a portion of the incoming data flow in parallel [14]. By applying data parallelism, we can handle larger data volumes at the price of using more computing resources for execution. Unfortunately, a static parallelism configuration is likely to cause either under-provisioning or over-provisioning due to the highly variable stream data rates.

In the literature, several approaches have been proposed for the elasticity control of DSP applications, including threshold-based heuristics [7, 10], control theory [5], linear programming [3], and reinforcement learning [2, 13]. However, the proposed solutions make the simplifying assumption that DSP applications are deployed on homogeneous computing infrastructures, where elasticity policies are independent of the computing nodes that host the operator replicas. Such an assumption is too simplistic for modern deployment environments, where resource heterogeneity is a key feature (e.g., fog/edge computing), or virtualization is employed to provide fine-grained resource allocation (e.g., on-demand virtual machine (VM) or container provisioning in the cloud). In these scenarios, applications can pick different computing nodes for execution, possibly adapting their resource demand over time. In order to optimize the resulting trade-off between performance, monetary cost, and energy consumption, elasticity policies should be aware of heterogeneity, so as to guarantee the desired service level with optimal resource usage.

To overcome these limitations, in this paper we consider the problem of controlling elasticity of DSP applications deployed on heterogeneous infrastructures. As starting point, we cast the elasticity control problem for each DSP operator as an infinite-horizon Markov Decision Process (MDP). The goal is to minimize a discounted infinite-horizon cost function which accounts for the operator response time, the resource usage, and the reconfiguration

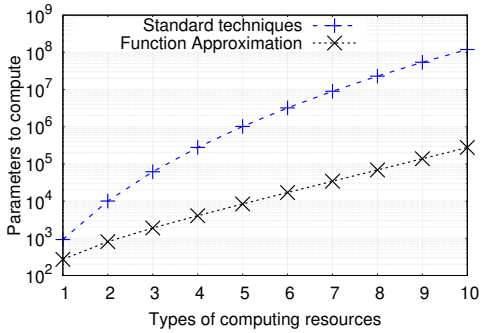


Figure 1: Number of parameters used by standard and FA-based MDP resolution algorithms for our problem.

overhead which is incurred whenever an adjustment in the deployment occurs. An MDP based approach, unfortunately, suffers from two major limitations. First and foremost, MDPs are afflicted by the curse of dimensionality, with the state space rapidly growing as the number of different type of resources grows, which translates into high memory and computational requirements. Secondly, solving MDP requires complete system knowledge, such as data streams dynamics, complete resources characterization in terms of response time as function of input rate, etc., which is not necessarily completely known at run-time. These potential severe issues *de facto* rule-out a straightforward adoption of MDP in realistic scenarios, where decisions need to be taken in a timely manner. Nevertheless, in this paper we will consider the baseline MDP approach, since the resulting optimal policy provides the benchmark against which we can compare other solutions.

To address the aforementioned problems with MDPs, in this paper we blend together Reinforcement Learning (RL) and Function Approximation techniques. RL refers to a collection of trial-and-error methods by which an agent can learn to make good decisions through a sequence of interactions with a system or environment [28], thus allowing for learning an optimal policy on-line even without knowledge of the underlying system model. RL by itself would not be sufficient in our scenario since most RL techniques suffer from slow convergence, especially as the system model size grows. Therefore, we also resort to linear Function Approximation (FA) techniques. By exploiting an approximate representation of the state space, FA techniques allow to achieve near-optimal solutions with reduced computational and memory demand. Figure 1 shows that FA-based resolution algorithms need to compute (and store) a dramatically smaller number of parameters, compared to standard approaches. A key challenge of this kind of approximate representations is feature construction, i.e., the choice of a suitable set of feature functions that result in an accurate and efficient representation of the problem state space. In this paper we resort to *tile coding* [27], where the state space is partitioned into *tiles* of variable size and shape, and a binary feature is associated with each tile. In the resulting representation, contiguous regions of the state space are aggregated in single tiles, thus enabling generalization over "similar" states.

A numerical evaluation of the proposed algorithms demonstrates the benefits of our approach, which: (i) achieves near-optimal accuracy compared to standard methods on small-medium problem instances, and (ii) allows to solve large problem instances in a reasonable amount of time, whilst standard techniques fail. We also show how in practice *model-based* MDP resolution and *model-free* RL can be combined to further improve convergence speed, only leveraging some estimates of the system dynamics.

The rest of this paper is organized as follows. We review related work in Sec. 2. In Sec. 3 we present the elasticity control problem, and we provide an MDP-based formulation of the problem in Sec. 4. In Sec. 5 we show how Function Approximation techniques can be used to deal with large problem instances. FA-based methods leverage approximate state space representations, and in Sec. 6 we present the features we defined for our problem relying on tile coding. We discuss the numerical evaluation of the proposed algorithms in Sec. 7, before concluding in Sec. 8.

2 RELATED WORK

Elasticity is a key feature for scalable DSP systems, which ingest data from sources with dynamic rates and must produce timely results under variable workload conditions. It requires to adapt the application deployment at run-time, either (i) by scaling the number of operator replicas, or (ii) by relocating some operators onto the computing resources, which are in turn elastically scaled, as surveyed in [4].

A key design decision of elastic DSP systems is the methodology employed to define how an elastic policy determines its adaptation decisions. Most approaches, in particular the first ones to address elasticity in the context of DSP, exploit best-effort threshold-based policies based on the utilization of the computing resources, e.g., [6, 7, 10]. Other works use more complex policies to make scaling decisions, exploiting optimization theory [3], control theory [5], game theory [22], queueing theory [20], or reinforcement learning [2, 13]. Heuristic policies have been also proposed, e.g., [19].

Another property that characterizes different elasticity approaches in literature is where the adaptation is controlled. Many approaches rely on a centralized adaptation planner (e.g., [3, 10, 19]), that is a single elasticity controller that manages the parallelism degree for the complete DSP application. Such a centralized component can exploit its global view of the system state; however, it can become a bottleneck and introduce communication overhead, especially in case of geo-distributed application deployment. Some solutions implement a fully decentralized approach (e.g., [13, 22]) that rely only on a local system view (usually at per-operator level), while few propose a hierarchical approach (e.g., [2]) that addresses the lack of coordination of fully decentralized controllers, e.g., by limiting the overall number of reconfigurations.

In the following, we focus our attention on RL-based elasticity policies, as they allow to deal with uncertainty regarding the system model, as model-free heuristics do, and aim at optimizing the system performance in the long term, as more complex model-based solutions can do. Furthermore, RL-based elasticity policies are amenable to the design of a decentralized elasticity controller.

A large number of works has exploited RL techniques to drive elasticity in the Cloud computing context, as surveyed in [1]. To

the best of our knowledge, only three works [2, 13, 21] have so far exploited RL techniques to drive auto-scaling decisions in DSP systems. Heinze et al. [13] propose a simple RL approach that learns from experience when to acquire and release computing nodes so to efficiently process the incoming workload. The adaptation goal is to keep the system utilization within a specific range, and the SARSA model-free learning algorithm [28] is responsible for learning the policy. Cardellini et al. [2] present a framework for controlling elasticity of DSP applications, where decisions are made at operator-level, and coordinated by a per-application manager. They also propose a policy for operator elasticity that leverages model-based RL, where the partial available knowledge about the system is integrated in the learning process so as to speedup convergence. Nonetheless, the adopted model-based algorithm does not scale well with large state spaces. Multi-level elasticity, that is scaling the computing infrastructure as needed in addition to the application operators, has been investigated by Lombardi et al. in [21], where a threshold-based approach is used to adjust the number of operator replicas and computing nodes at run-time. Reinforcement learning is exploited in this work as well, but instead of controlling the adaptation it is used to automate parameters tuning. In this paper, we focus on application-level elasticity, assuming that infrastructure-level elasticity is managed elsewhere.

We observe that none of the above mentioned elasticity approaches is explicitly targeted to heterogeneous computing infrastructures. Considering the trend toward fog and edge computing, that are characterized by more heterogeneous processing nodes, we believe that new elasticity policies that take into account the capabilities of the available computing nodes should be considered in order to effectively manage the elasticity in these environments. So far, heterogeneity in DSP has been investigated in the context of the application *placement* problem, that is mapping each operator to a computing node for execution at the time the application is initially deployed or every time it is re-deployed [18, 26]. We instead focus on the run-time adaptation of application parallelism, where the operator parallelism degree is adjusted over time in response to workload variations. Other works dealing with heterogeneous computing environments consider how to process data streams on heterogeneous architectures with CPU and GPU cores [16], smartphones [9], CPUs, GPUs and FPGAs in the cloud [11].

Finally, we observe that in this work we adopt an operator elasticity problem formulation similar to that used in [2], but, differently from that work, the model we present in this paper takes into account resource heterogeneity. This extended formulation leads to a significantly larger state space, forcing us to also explore more complex tools to resolve the problem, i.e., Function Approximation. Moreover, in this work we evaluate both *learning* algorithms, which compute the policy over time by experience, and *planning* algorithms, which exploit the knowledge of the system model to determine the optimal policy of the underlying MDP.

3 PROBLEM DESCRIPTION

In this paper, we consider the problem of controlling the elasticity of a DSP application deployed over a heterogeneous computing infrastructure. When deploying a DSP application, we are usually concerned with satisfying a Service Level Objective (SLO). The SLO may be specified in terms of different metrics, such as processing latency or throughput. We focus on processing latency, as it is the most relevant metric for domains that require near real-time processing (e.g., Smart City). Therefore, although our solution can be generalized to work with different metrics, in the following we will assume that the DSP application is expected to satisfy a SLO by keeping the processing latecy below a value R_{SLO}.

To guarantee an acceptable performance level under varying workload conditions, data parallelism technique [14] is applied, so that each operator is replicated into a number of parallel instances, each receiving a portion of the incoming data flow in parallel.[1] Whenever the parallelism degree of any operator is reconfigured at run-time, the data flow is rearranged so as to re-distribute the incoming streams to a larger/smaller number of replicas. To guarantee integrity and in-order processing of the data streams, during this reconfiguration process the DSP system must execute a specific protocol [12], which introduces some overhead (e.g., latency peaks). In presence of stateful operators, a similar redistribution happens for the operator internal state, so as to preserve application semantics, and the performance penalty incurred by the system due is likely more significant.

The parallel replicas of the operators must be deployed for execution on the set of computing nodes available to the DSP system. In particular, in this work we consider an infrastructure in which nodes can be acquired on-demand, and a pay-as-you-go cost model is offered by the provider. The infrastructure allows to acquire different *types* of computing nodes (e.g., corresponding to VMs/containers with different amounts of resources). Therefore, decisions must be made at run-time on (i) *how many* replicas should be allocated for each operator, (ii) *when* to change the deployment, and (iii) *which type* of computing node each replica should be deployed on, so as to consequently minimize (i) the SLO violations, (ii) the reconfiguration overhead, and (iii) the monetary cost paid for computing resources. It is easy to realize that the aforementioned objectives cannot be all optimized at the same time, and we must settle for a trade-off between performance, resource usage, and reconfiguration overhead. Moreover, because of the long-running nature of DSP applications, a good elasticity policy should pick the best trade-off taking into account the future impact of every decision.

In this paper, we focus on the problem of defining policies for controlling the operator elasticity over the considered heterogeneous infrastructures. As regards the provisioning of the computing nodes, we assume that infrastructure auto-scaling is provided at a lower level (e.g., as in [25]), so that the application resource demand is satisfied. Therefore, we assume that when a new operator replica must be deployed on a certain type of computing node, at least one node is available with enough resources to host the new replica in the proximity of the other operator instances. Our policy can be integrated with more complex placement strategies, which are out of the scope of this work though.

[1] For simplicity, and without lack of generality, we consider the ideal scenario with even partitioning of the incoming data among the operator parallel instances. In practice, good load balancing can be achieved even for stateful operators relying on appropriate stream partitioning schemes (e.g., [23]).

4 SYSTEM MODEL AND PROBLEM FORMULATION

We consider a scenario where a cloud provider offers a set T_{res} of different types of VMs, *e.g.*, $T_{res} = \{\text{small}, \text{medium}, \text{large}, \text{xlarge}\}$, each equipped with different amounts of computational resources (e.g., CPU, memory). We will denote by N_{res} the number of different types of VMs (i.e., $N_{res} = |T_{res}|$). Each $\tau \in T_{res}$ is associated with a cost c_τ, which is paid for deploying an operator replica on a node of type τ for a reference time period[2]

Following an approach similar to that proposed in [2], we consider a self-adaptive DSP system organized in a hierarchical fashion running on the aforementioned infrastructure: each operator is equipped with its elasticity controller, which makes decisions for that single operator. The resulting adaptation decisions are communicated to a per-application controller, which will possibly coordinate the decisions made at operator-level. Hereafter, we will formalize the problem of controlling the elasticity for a single DSP operator, relying on the above-mentioned hierarchical approach to integrate the resulting policies with each other.

We consider a slotted time system with fixed-length time intervals of length Δt, with the i-th time slot corresponding to the time interval $[i\Delta t, (i+1)\Delta t]$. At the beginning of time slot i, the current deployment of an operator is defined by the vector $\boldsymbol{k}_i$, with $k_{\tau,i}$ being the number of replicas deployed on computing nodes of type τ. We denote by λ_i the average input data rate measured during slot $i-1$ (i.e., the previous slot). At time i, we make a decision a_i on whether modifying the current deployment of the operator.

We formulate the operator elasticity control problem as a discrete-time infinite-horizon Markov Decision Process (MDP) [24]. A MDP is defined by a 5-tuple $\langle \mathcal{S}, \mathcal{A}, p, c, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states; $\mathcal{A}(s)$ a finite set of actions available in state s; $p(s'|s,a)$ the transition probabilities from state s to state s' given action $a \in \mathcal{A}(s)$; $c(s,a,s')$ the immediate cost when action a is executed in state s and the system enters state s'; and $\gamma \in [0,1]$ a discount factor that weights future costs.

The state of the system at time i is defined as the pair $s_i = (\boldsymbol{k}_i, \lambda_i)$, that is the current operator deployment and the input data rate. For the sake of analysis, we consider a discrete state space, that is, we discretize the arrival rate λ_i by assuming that $\lambda_i \in \{0, \bar{\lambda}, 2\bar{\lambda}, \ldots, N_\lambda \bar{\lambda}\}$ where $\bar{\lambda}$ is a quantization step size. For instance, if input rate may range between 0 and 1000 tuple/s, and it is discretized using $\bar{\lambda} = 100$ tuple/s, we have that $\lambda_i \in \{0, \bar{\lambda}, 2\bar{\lambda}, \ldots, 10\bar{\lambda}\}$

We also consider that we can deploy at most K_{max} replicas of the operator, hence at any time we have $1 \leq \sum_{\tau \in T_{res}} k_\tau \leq K_{max}$.

For each state s, we denote the set of available actions by $\mathcal{A}(s)$, which is defined as:

$$\mathcal{A}(s) = \{\omega\} \cup \left\{(\delta, \tau) : \begin{smallmatrix} \delta \in \Delta(s) \\ \tau \in T_{res} \end{smallmatrix}\right\} \tag{1}$$

where ω is the "do nothing" action that leaves the operator deployment unchanged, while a reconfiguration action is a pair (δ, τ) that specifies the number $\delta \in \Delta(s)$ of replicas to add/remove on nodes of type τ. Without lack of generality, we assume that $\Delta(s) = \{+1, -1\}$ for all states, except for those where the operator parallelism is 1 and $\Delta(s) = \{+1\}$ (we cannot have less than a replica running), or the parallelism is equal to K_{max} and $\Delta(s) = \{-1\}$ (we cannot add instances beyond the maximum allowed level). For instance, assume, as above, that we have $T_{res} = \{\text{small}, \text{medium}, \text{large}, \text{xlarge}\}$ the input rate is discretized using $\bar{\lambda} = 100$ tuple/s and an operator may have maximum parallelism $K_{max} = 2$. When a single replica is used, deployed on a VM of type *medium*, and the input rate is about 100 tuple/s, the operator state is $s = ([0,1,0,0]^\top, 2)$. Valid actions in this state include a scale-out on the same type of node, a scale-out on a *large* VM, a do-nothing action, that is, $\mathcal{A}(s) = \{(+1, \text{medium}), (+1, \text{large}), \omega, \ldots\}$.

State transitions occur as a consequence of reconfiguration decisions and tuple arrival rate variations. Following an action a, the operator deployment $\boldsymbol{k}$ is (possibly) updated according to a into $\boldsymbol{k}'$. In the following, we will describe a deployment reconfiguration as $\boldsymbol{k}' = \boldsymbol{k} \oplus a$. Let us denote by $p(s'|s,a)$ the transition probability from state s to state s' given action a. We readily obtain:

$$\begin{aligned} p(s'|s,a) &= P[s_{i+1} = (\boldsymbol{k}', \lambda') | s_i = (\boldsymbol{k}, \lambda), a_i = a] = \\ &= \begin{cases} P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda] & \boldsymbol{k}' = \boldsymbol{k} \oplus a \\ 0 & \text{otherwhise} \end{cases} \end{aligned} \tag{2}$$

It is easy to realize that the system dynamics comprise a stochastic component due to the tuple rate variation, which we assume exogenous, captured by the transition probabilities $P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda]$, and a deterministic component due to the fact that, given action a, the new deployment configuration $\boldsymbol{k}'$ is $\boldsymbol{k} \oplus a$.

To each triple (s, a, s') we associate a cost $c(s,a,s')$, which captures the cost of operating the system in state s', after carrying out action a in state s. In particular, we consider three different costs, related to resource usage, adaptation overhead, and SLO violation.

The **resources cost c_{res}** is the cost paid for using the computing resources on which the operator is deployed throughout the next interval[3]. Formally, we have:

$$c_{res}(s,a,s') = \sum_{\tau \in T_{res}} k'_\tau c_\tau$$

where c_τ is the cost paid for deploying an operator replica on a resource of type τ for the next time interval.

The **reconfiguration penalty c_{rcf}** captures the impact of deployment adaptation (i.e., state movement, application downtime). Detailed models to evaluate the penalty incurred after reconfiguration are available in the literature (e.g., in [3]). However, the large reconfiguration overhead imposed by most DSP frameworks dominates the costs related to state movements for many applications. Therefore, for the sake of simplicity, we will assume a constant penalty paid whenever $a \neq \omega$, i.e., $c_{rcf}(a) = \mathbb{1}_{\{a \neq \omega\}}$.

The **SLO violation cost c_{SLO}** captures the penalty incurred whenever the system violates the SLO in the next interval. Considering a SLO expressed in terms of application latency, $c_{SLO}(s,a,s') = \mathbb{1}_{\{R(s') > R_{max}\}}$, where $R(s)$ is the operator response time in state s, and R_{max} is a suitable per-operator latency bound.

We combine the different costs into a single cost function using the *Simple Additive Weighting* (SAW) technique [30]. According to

[2] In some scenarios, a monetary cost may be associated with the usage of a computing node despite of what is deployed on top of it. Even in this case, the cost c_τ can be used to guide the elasticity controller towards better resource utilization, letting an infrastructure-level auto-scaling component optimize the actual monetary cost.

[3] Since we assume the action to be executed at the beginning of a time period, the operator deployment during an interval is $\boldsymbol{k} \oplus a$.

SAW, we define the cost function $c(s, a, s')$ as the weighted sum of the normalized costs:

$$c(s, a, s') = w_r \frac{c_{res}(s, a, s')}{C_{max}} + w_a c_{rcf}(a) + w_s c_{SLO}(s, a, s') \quad (3)$$

where w_r, w_a and w_s, $w_r + w_a + w_s = 1$, are non negative weights, C_{max} is a normalization parameter defined as $\max_{\tau \in T_{res}} c_\tau$, and $\mathbb{1}_{\{\cdot\}}$ is the indicator function.

5 PLANNING AND LEARNING USING FUNCTION APPROXIMATION

The ultimate goal of each operator elasticity controller, which corresponds to the role of the *decision making agent* in the MDP parlance, is to minimize the total cost incurred by its actions over time. To this end, the elasticity controller needs a good *policy*, that is a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$ to associate each state with an action to choose (i.e., a particular deployment adaptation decision).

Formally, let $V^\pi(s) : \mathcal{S} \rightarrow \mathbb{R}$ be the *value function*, which represents the expected infinite-horizon discounted cost given an initial state s and following a policy π, defined as $V^\pi(s) = E^\pi \left[\sum_{i=0}^{\infty} \gamma^i c(s_i, a_i, s_{i+1}) \middle| s_0 = s\right]$. We are interested in determining the optimal policy π^* that minimizes the expected discounted cost, which satisfies the Bellman optimality equation [24] for every state s:

$$V^{\pi^*}(s) = \min_{a \in \mathcal{A}(s)} \left\{ \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[c(s, a, s') + \gamma V^{\pi^*}(s') \right] \right\} \quad (4)$$

where the first term represents the cost associated to the current state s and decision a; the second term represents the future expected discounted cost under the optimal policy. It is also convenient to define the action-value function $Q^\pi : \mathcal{S} \times A \rightarrow \mathbb{R}$, which is the expected infinite-horizon discounted cost achieved by taking action a in state s and then following the policy π:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[c(s, a, s') + \gamma V^\pi(s') \right] \quad (5)$$

It is easy to realize that the value function V and the Q-function are closely related in that $V^\pi(s) = \min_{a \in A(s)} Q^\pi(s, a),\ \forall s \in \mathcal{S}$. More importantly, the knowledge of the Q-function directly provides the associated policy, as $\pi(s) = \arg\min_{a \in \mathcal{A}(s)} Q(s, a),\ \forall s \in \mathcal{S}$.

When the system dynamics and parameters are completely known, *planning* techniques can be used to solve the optimality equation (4) via, e.g., dynamic programming or linear programming. Conversely, in presence of uncertainty about the system dynamics (e.g., unknown response time model), the elasticity controller can *learn* π^* over time by experience, using reinforcement learning algorithms [28]. No matter which approach is adopted (i.e., either planning or learning), all the solution algorithms derive the optimal policy π^* by computing the optimal Q-function[4], and they only differ in the way they estimate it. The baseline approach relies on storing and updating the entries $Q(s, a)$ for every state-action pair (s, a) in memory (the so-called *Q table*). This tabular representation limits the scalability of traditional algorithms, because (i) its memory requirements become challenging for large state spaces, and (ii) the execution time may grow significantly as the value of every state-action pair must be computed and updated separately.

To overcome the above-mentioned issues, a widely used approach is *Function Approximation* (FA). When using FA, the Q table is replaced by a parametric function $\hat{Q}(s, a, \boldsymbol{\theta})$ that is used to estimate $Q(s, a)$. By doing so, the parameters vector $\boldsymbol{\theta}$ is the only thing we need to compute and store. Therefore, on the one hand, FA allows to dramatically reduce the computational requirements of planning and learning algorithms; on the other hand, the *approximate* Q-function representation may lead to sub-optimal policies, and also makes the planning/learning algorithms more complex.

The accuracy of the approximation strongly depends on how the function $\hat{Q}$ is defined. In this work, we focus on *linear* FA, where $\hat{Q}$ is expressed as a linear combination of the parameters $\boldsymbol{\theta}$. Non-linear function approximators (e.g., Artificial Neural Networks) have been successfully applied within a large number of domains, where linear models cannot achieve acceptable performance. However, the accuracy of more complex models often comes at the cost of long training phases, large number of hyperparameters, and higher computational demand in general. Since we target distributed environments, where control policies should be possibly computed on resource-constrained devices, we considered linear FA well suited for our problem, also encouraged by the good accuracy achieved by our solution. Therefore, we will use the following approximate value functions:

$$\hat{Q}(s, a, \boldsymbol{\theta}) = \sum_i \phi_i(s, a)\theta_i \quad (6)$$

$$\hat{V}(s, \boldsymbol{\theta}) = \min_{a \in \mathcal{A}} \hat{Q}(s, a, \boldsymbol{\theta}) \quad (7)$$

where $\boldsymbol{\phi}(s, a) = [\phi_1(s, a), \phi_2(s, a), \ldots]^\top$ is a vector of *features*, and $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots]^\top$ a vector of *parameters* (or, *weights*). Each element of $\boldsymbol{\phi}(s, a)$ is a feature, that is a function $\phi_i : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, which encodes some properties of a state-action pair. The parameters $\boldsymbol{\theta}$ weight the contribution of each feature to the estimated Q-function.

The feature functions $\boldsymbol{\phi}(s, a)$ should extract relevant information from states and actions, and constructing an appropriate vector of features is a critical yet difficult task for achieving good accuracy. Several approaches exist to guide feature engineering in the context of MDPs and RL tasks [17, 28]. In Section 6, we will describe the set of features we adopt, and we will explain how we constructed it leveraging on the Tile Coding technique.

Given a set of features and a collection of examples of the true Q-function values, planning (or learning) algorithms should compute the weights $\boldsymbol{\theta}$ so as to well approximate the true function. The true Q-function values are collected in a different way in planning and learning: they are produced by a generative model in planning, whilst they come from experience in the learning case. For linear FA, a simple semi-gradient approach can be used to update the weights [28]:

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i + \alpha \boldsymbol{\phi}(s, a) \left(\hat{q} - \hat{Q}(s, a, \boldsymbol{\theta}_i) \right) \quad (8)$$

where $\hat{q}$ is the new sampled value of the Q-function, and $\alpha \in (0, 1)$ is a step size parameter.

In the next sections, we will show how FA can be integrated in a planning algorithm (Section 5.1), and in the well-known Q-learning algorithm to deal with model uncertainty (Section 5.2).

[4] In the following we will focus on the Q-function, although most of the concepts apply to the value function with minor adaptations.

5.1 Planning: Trajectory-Based Value Iteration

Planning algorithms determine the optimal value function (hence the optimal policy) given an MDP instance. A popular planning algorithm is Value Iteration (VI) [28], based on dynamic programming. When dealing with large problem instances, VI cannot be adopted for two main reasons: (i) it expects to be able to store and retrieve a value $Q(s, a)$ for every state-action pair (s, a); (ii) it loops over all the state-action space to update the value of every single entry, which may be computationally intractable. The *Trajectory Based Value Iteration* (TBVI) [8] is an extension of VI that overcomes both these limitations. To reduce the memory requirement, the Q table is replaced with the FA-based estimates. To tackle the second issue, instead of scanning the whole state-action space at every iteration, TBVI relies on *trajectory sampling* [28]. According to this technique, state-action pairs are updated following a *trajectory*: we pick a random state, and then (i) we select actions according to the currently available policy, and (ii) continue the trajectory with the resulting next state. By doing so, states that are rarely encountered by following a reasonable policy are less likely to be updated.

We rely on TBVI in this work, and we slightly modify it to achieve better performance in our context. The resulting algorithm is shown in Algorithm 1. The weights $\boldsymbol{\theta}$ can be initialized arbitrarily (line 1), and they are then updated until the available time expires (line 2). Trajectories are initialized by randomly choosing an initial state (line 3). Differently from the algorithm presented in [8], we do not use a single trajectory, and, to speedup the exploration of states "far" from the current one, we instead reinitialize a new trajectory after following one for ξ steps (line 4). In each state, we use ϵ-greedy action selection (line 5): with probability $\epsilon \ll 1$ we select a random action to be explored, otherwise we pick the most promising actions according to the current Q estimates. A new estimate $\hat{q}$ of $Q(s, a)$ is computed at each step using the Bellman equation (line 6), and it is used to update the weights applying (8) (line 7). The next state in the trajectory is sampled among the possible states encountered after making action a in state s (we need sampling because of the non-deterministic input rate variations).

Algorithm 1 Trajectory Based Value Iteration algorithm.

1: $\boldsymbol{\theta}_0 \leftarrow$ initialize arbitrarily
2: **while** time left **do**
3: $\quad s \leftarrow$ random state
4: $\quad$ **for** $j = 1, 2, \ldots, \xi$ **do**
5: $\quad\quad a \leftarrow \epsilon$-greedy action selection
6: $\quad\quad \hat{q} \leftarrow \sum_{s' \in \mathcal{S}} p(s'|s, a) \left(c(s, a, s') + \gamma \hat{V}(s', \boldsymbol{\theta}_i) \right)$
7: $\quad\quad \boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i + \alpha \boldsymbol{\phi}(s, a) \left(\hat{q} - \hat{Q}(s, a, \boldsymbol{\theta}_i) \right)$
8: $\quad\quad s \leftarrow$ sample next state following (s, a)
9: $\quad$ **end for**
10: **end while**

5.2 Learning: Q-learning

When a model of the system is not available, planning algorithms cannot be applied. Reinforcement learning is a collection of techniques to deal with this kind of scenario, where the underlying idea is that we can learn what the optimal actions are by experience (i.e., by making actions and getting feedback from the environment). The simplest and most popular RL algorithm is *Q-learning* [29], which does not require any estimate of the system dynamics and learns the optimal policy from scratch. Q-learning updates its estimates of the Q-function using a simple rule. After observing a transition from state s to s', with action a being taken and a cost c_i paid, the algorithm updates the estimates as follows:

$$\hat{q} \leftarrow c_i + \gamma \min_{a' \in \mathcal{A}(s')} Q_i(s', a') \tag{9}$$

$$Q_{i+1}(s, a) \leftarrow (1 - \alpha) Q_i(s, a) + \alpha \hat{q} \tag{10}$$

Therefore, in its simplest form, Q-learning relies on a tabular representation of the Q-function, suffering from the scalability issues described in the previous sections. The main drawback of this model-free learning algorithm is the long time it often requires to converge to the optimal policy. Indeed, as a single Q table entry is updated at each step, the learning process may need a very large number of steps. Moreover, Q-learning is characterized by the *exploration-exploitation* [28] dilemma: at each step, the algorithm cannot simply *exploit* the most promising action according to its Q estimates, but sometimes it has to *explore* (e.g., choosing the action randomly) to update other parts of the Q table. Exploration clearly has an additional negative impact on the agent behavior during the learning phase.

FA can be integrated in Q-learning, thus removing the necessity of a Q table. Consequently, the update rule (10) is simply replaced by (8). By doing so, we do not just reduce the memory requirement of the algorithm, but also improve its *generalization* capability. Indeed, by using an approximate state space representation, each update of the weights is likely to impact more than a single state-action pair, accelerating the convergence to the optimal Q-function (hence, to the optimal policy).

6 TILE CODING FOR THE ELASTICITY CONTROL PROBLEM

When using FA, we replace the tabular representation of the Q-function with a parametric function $\hat{Q}(s, a, \boldsymbol{\theta})$, as defined in (6). In the previous sections, we have shown how planning and learning algorithms can deal with this kind of representation, and compute $\boldsymbol{\theta}$. We now turn our attention to the problem of constructing the vector $\boldsymbol{\phi}(s, a)$, i.e., the features.

The process of selecting features is critical for achieving an accurate state space representation, hence a near-optimal MDP solution. The chosen features must capture all the relevant aspects of states and actions so as to well approximate the Q-function, at the same time favoring generalization in presence of states and actions with similar impact on the value functions. Therefore, the set of features should be large enough so as to well characterize the state-action space, but having too many features we could suffer from the same scalability issues of tabular methods.

In order to simplify the construction of a suitable set of features, a common approach is focusing on the state space representation and isolating the impact of actions. Specifically, a vector of features $\tilde{\boldsymbol{\phi}}(s) = [\phi_1(s), \phi_2(s), \ldots,]^\top$ is used to gather relevant information about states, and this vector is then replicated for each possible action (i.e., the same feature will get a different weight depending on the associated action). By doing so, we must only focus on state

representation, but the overall number of features may increase significantly. We seek for a better solution, and observe that in our model: (i) the impact of actions on state transitions is deterministic (i.e., given a state and an action, it is straightforward to compute the new deployment configuration); (ii) according to (3), the only cost term that directly depends on the chosen action is the reconfiguration cost, which in turn does not depend on the state.

Given the observations above, we can exploit the concept of *post-decision state* (PDS) (also known as *afterstate [28]*). A PDS represents the state of the system *after* an action is executed, and *before* the unknown dynamics take place (i.e., the input rate variation). Formally, we denote the PDS as $\tilde{s}$, and being $s = (\boldsymbol{k}, \lambda)$, we have $\tilde{s} = (\boldsymbol{k} \oplus a, \lambda)$. Any state transition $s \rightarrow s'$ can be accordingly split into two "virtual" transitions: $s \rightarrow \tilde{s}$, followed by $\tilde{s} \rightarrow s'$, with the former capturing the known impact of actions on state and paid costs. Relying on the concept of PDS, we can construct $\boldsymbol{\phi}(s, a)$ by picking two types of features:

(1) features $\boldsymbol{\phi_A}(a)$ that only depend on the current action and capture the reconfiguration-related impact;
(2) features $\boldsymbol{\phi_S}(\tilde{s})$ that only depend on the PDS and capture the dynamics related to resource usage and performance.

Formally, we will have $\boldsymbol{\phi}(s, a) = [\boldsymbol{\phi_A}(a) | \boldsymbol{\phi_S}(\tilde{s})]^\top$. In the next two sections we will show how to define, respectively, the action-dependent features, and the state-dependent features.

6.1 Action Space Representation

The features $\boldsymbol{\phi_A}(a)$ should provide significant information to represent the dynamics related to the reconfiguration cost. We recall that in our problem the reconfiguration cost is paid whenever the deployment is changed, independently on the specific action. Thus, we can simply define a single action-dependent feature as follows:

$$\phi_A(a) = \mathbb{1}_{\{a \neq \omega\}} \tag{11}$$

where ω is the "do nothing" action.

6.2 State Space Representation

State features could be constructed just leveraging insight and domain expertise, as we did for actions. Nevertheless, as in our model the state space is larger and more complex than the action space, following some guidelines is necessary. For example, a simple approach is *Fixed Sparse Representation* (FSR) [8], where each component of the state vector is considered separately, and a binary feature $\phi_{d,j}(s)$ is associated with every possible value j of the d-th state component. By doing so, on the one hand, FSR produces a reduced number of features even in presence of highly dimensional state spaces; on the other hand, it does not capture the correlation between state dimensions, which usually exists.

A more advanced coding technique on which we focus our attention is *Tile Coding* [27]. According to this approach, the state space is partitioned into a certain number of *tiles*, each covering a contiguous portion of the space. In the resulting state space partition (usually called *tiling*), the i-th tile is associated with a binary feature $\phi_i(s)$, whose value is 1 if s is covered by tile i, and zero otherwise. By doing so, a single feature is "activated" by several contiguous states, thus enabling generalization. Tiles may have different size,

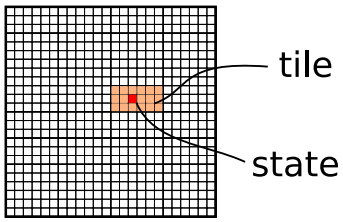


Figure 2: Example of a 2D tiling.

which impacts on representation fineness, or different shape, leading to generalization along different dimensions. An example of a tiling is given in Fig. 2, where rectangles are used to partition a bi-dimensional state space.

In order to apply tile coding to our problem, we would need a $(N_{res} + 1)$-dimensional tiling (i.e., a dimension for the number of replicas for each node type, and one more dimension for the input rate). To simplify the state space representation, we decide to combine domain expertise and tile coding, thus constructing the features in a two-phase process. In the first phase, we map the $(N_{res} + 1)$-dimensional state space onto a 3-dimensional space, by leveraging our knowledge of the system model. In particular, we consider the following three dimensions to characterize every state: (i) the input data rate, (ii) the overall operator parallelism degree, and (iii) the set of computing node types used in the given state.

In the second phase, we leverage tile coding to construct an efficient representation of the reduced state space. In particular, we consider two types of tiling for our problem, based on the shape of the tiles, rectangles or diagonal stripes. In **rectangle-based** tilings (left part of Fig. 3), tiles group states with similar parallelism and input rate, in which the same set of computing nodes is used. In particular, n_p tiles are used along the parallelism dimension, n_λ tiles along the input rate dimension, and $2^{N_{res}}$ tiles along the resource types dimension. In **stripe-based** tilings (right part of Fig. 3), tiles group states based on the input rate-per-replica ratio and the set of used computing nodes. We use n_{str} diagonal stripes to cover regions of the state space where the input rate-per-replica ratio is similar, replicated on each of the $2^{N_{res}}$ levels along the node types dimension.

During preliminary experiments, we verified that none of the two types of tiling allows to achieve good performance in any situation when used alone, whereas they work well when coupled. In particular, we identified a reference configuration in which two partially overlapping rectangle-based tilings are used, in addition to one stripe-based tiling. As we have shown in Fig. 1, the overall state-action space representation we use allows us to dramatically reduce the number of parameters used for MDP resolution. We will further discuss in Sec. 7 how we can pick a different trade-off between representation accuracy and complexity by varying the size of the tiles (hence the number of features).

7 EVALUATION

We perform a numerical evaluation of the proposed planning and learning algorithms to assess both their efficacy in driving elasticity and their efficiency in terms of required computational resources. To this end, we extract a realistic application workload from the

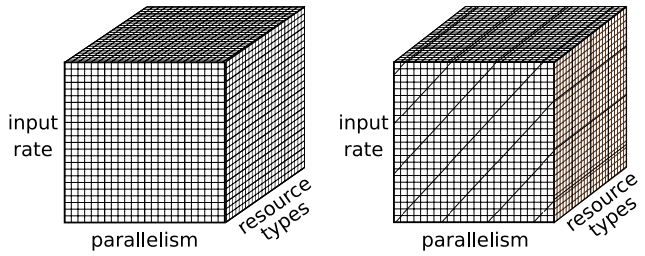


Figure 3: Rectangle-based and stripe-based tilings we use for our problem instances.

dataset[5] used in the *DEBS 2015 Grand Challenge* [15], which contains information related to taxi trips in New York City throughout one year. The taxi service utilization presents a daily pattern, which also characterizes the number of events collected in the dataset, as shown in Fig. 4. Therefore, elasticity would be a critical requirement for a system in charge of real-time data analysis. We evaluate the adaptation policies by simulating the first 400, 000 minutes in the dataset[6] (about 9 months), with a reconfiguration decision being made at the beginning of every one-minute time slot.

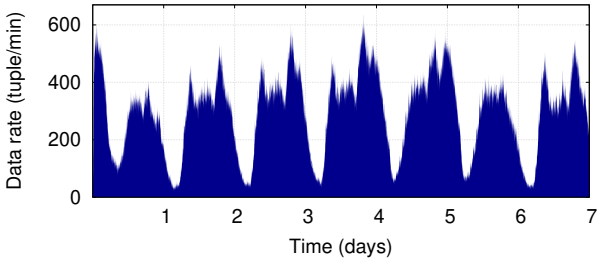


Figure 4: First part of the workload used in the experiments.

As regards the computing infrastructure, we consider several configurations in which we vary both (i) the number of node types N_{res} from 3 to 10, and (ii) the performance among different node types. Specifically, we assume that computing nodes of type τ can be characterized in terms of a single numerical speedup σ_τ (e.g., CPU speedup with respect to a reference processor), although our solution can work in more complex situations as well. Considering this speedup factor, we define two sets of node types as follows:

- Scenario A: 0.2, 0.3, 0.4, 0.5, 0.8, 1, 1.15, 1.2, 1.3, 1.5
- Scenario B: 0.1, 0.7, 0.8, 1.0, 1.2, 2, 3, 5, 10, 30

where the second configuration clearly differs in more heterogeneous nodes. Inspired by Cloud pricing models, we assume that the cost c_τ of each type of node is proportional to the associated computational capacity, and we set $c_\tau = \sigma_\tau / \sigma_{max}$, where $\sigma_{max} = \max_{\tau'} \sigma_{\tau'}$ is the maximum available speedup.

For the FA-based solutions, as a reference configuration, we consider tilings with $n_p = K_{max}$, $n_\lambda = \frac{4}{10} N_\lambda$, and $n_{str} = K_{max}$. In the following, we will also discuss the impact of using more (or less) fine-grained partitioning. The other constant parameters used in the experiments, which were tuned during preliminary evaluations, are as follows: the discount factor γ is 0.99, the number of discretized input rate levels N_λ is 30, the maximum operator parallelism K_{max} is 10, the cost function weights are $w_s = w_a = 0.4$, $w_r = 0.2$. For TBVI, we use $\xi = 500$, $\alpha = 0.1$, $\epsilon = 0.1$. We run the experiments using Amazon EC2 `c5.xlarge` instances. We first evaluate the performance achieved when the operator elasticity is tackled using approximate FA-based solutions, compared to tabular methods and simple threshold-based scaling heuristics (Sec. 7.1). Then, we show how the proposed solution can be applied to control the deployment of whole DSP applications (Sec. 7.2).

7.1 Operator Elasticity

In order to solve the operator elasticity control problem, we consider both the *planning* case, where the system model is known and the elasticity policy can be computed off-line, and the *learning* case, where the system dynamics are not completely known, hence the elasticity policy must be learned on-line.

7.1.1 Planning Algorithms. We compare the standard Value Iteration algorithm (denoted as **VI**) and the FA-based Trajectory Based Value Iteration (**TBVI**). Granted that optimal policies can be computed by planning algorithms when allowed to execute as long as needed, it is not obvious that when deploying an application in a new environment there is enough lead time to perform planning until convergence. Therefore, we take the perspective on how to compute the elasticity policy assuming we do not have time to plan in advance, and we evaluate the different algorithms by limiting the planning execution time[7]. In particular, depending on the number of computing node types considered, we allow the planning algorithms to run for at most 1, 1.5, 2, and 5 minutes when N_{res} is equal to, respectively, 3, 5, 7 or 10. As a benchmark, we also consider a simple threshold-based scaling policy that increases (or decreases) the operator parallelism level based on the observed CPU utilization (see, e.g., [2]). We combine the threshold-based policy with two heuristics to choose the type of nodes to add (or remove) in case of reconfiguration: the first heuristic (denoted as **TH-cost**) always picks the cheapest available node, whereas the second one picks the resource with the highest speedup (**TH-speedup**).

We consider a DSP operator whose parallel replicas can be modeled as M/G/1 queues, with service rate $\mu = 180$ tuple/s and service time variance $\sigma^2 = \frac{1}{2\mu^2}$. According to the SLO, the operator is required to keep its response time below 12 ms.

In Table 1 we report the results of this experiment, showing the average cost paid by the algorithms (computed according to (3)), the percentage of time in which the SLO is violated, the percentage of time in which the application is reconfigured, and the monetary cost associated to resource usage. First of all, we observe that the simple threshold-based policy is not able to guarantee optimal performance in every situation. Indeed, the TH-cost heuristic, which always picks the cheapest nodes, is not able to satisfy the SLO requirement. Conversely, when using the most powerful nodes with

[5] `http://chriswhong.com/open-data/foil_nyc_taxi/`

[6] In order to have a more realistic resource demand, we amplify the number of events per minute by 60 times, so as to have in a second the amount of input data we would have in one minute.

[7] If the time limit occurs before algorithm termination/convergence (this might occur when, for instance, we try to solve the MDP with the baseline tabular approaches), we just retain the current policy computed by the algorithm.

Table 1: Results achieved by VI, TBVI, and the two threshold-based heuristics (TH-cost, TH-speedup). We report the number of parameters used to represent the Q-function (Memory), the average cost, the percentage of time in which the SLO is violated (Viol.) and the operator deployment is reconfigured (Rcf.), and the average cost due to resource usage (Res.).

		Planning		Scenario A				Scenario B			
N_{res}	Algorithm	Time (s)	Memory	Cost	Viol. (%)	Rcf. (%)	Res.	Cost	Viol. (%)	Rcf. (%)	Res.
3	TH-cost	-	-	0.4311	100.00	1.86	**1.8**	0.4082	100.00	1.88	**1.0**
3	TH-speedup	-	-	**0.0534**	**0.11**	0.51	3.8	0.0207	0.12	**0.00**	30.3
3	VI	60	$6 \cdot 10^4$	0.0545	0.63	**0.28**	3.8	**0.0040**	**0.00**	**0.00**	6.0
3	TBVI	60	$2 \cdot 10^3$	0.0539	0.68	0.31	3.7	0.0059	0.59	0.06	5.0
5	TH-cost	-	-	0.4311	100.00	1.86	**1.8**	0.4082	100.00	1.88	**1.0**
5	TH-speedup	-	-	**0.0534**	**0.11**	0.51	3.8	0.0207	0.12	**0.00**	30.3
5	VI	90	$1 \cdot 10^6$	0.0801	0.03	**0.00**	6.0	**0.0040**	**0.00**	**0.00**	6.0
5	TBVI	90	$9 \cdot 10^3$	0.0539	0.69	0.44	3.7	0.0069	0.88	0.08	4.5
7	TH-cost	-	-	0.4311	100.00	1.86	**1.8**	0.4082	100.00	1.88	**1.0**
7	TH-speedup	-	-	**0.0534**	**0.11**	0.51	3.8	0.0207	**0.12**	**0.00**	30.3
7	VI	120	$9 \cdot 10^6$	0.4267	100.00	**0.00**	2.0	0.4007	100.00	**0.00**	1.0
7	TBVI	120	$3 \cdot 10^4$	0.0561	0.75	0.94	3.7	**0.0086**	1.47	**0.00**	4.0
10	TH-cost	-	-	0.4311	100.00	1.86	**1.8**	0.4082	100.00	1.88	**1.0**
10	TH-speedup	-	-	**0.0534**	**0.11**	0.51	3.8	0.0207	**0.12**	**0.00**	30.3
10	VI	300	$1 \cdot 10^8$	0.4267	100.0	**0.0**	2.0	0.4007	100.0	**0.00**	1.0
10	TBVI	300	$3 \cdot 10^5$	0.0541	0.68	3.8	2.5	**0.0167**	3.27	**0.00**	5.4

the TH-speedup heuristic, the system performance is very good, with less than 1% of SLO violations in both the considered heterogeneity scenarios. However, the cost due to resource usage is much larger, especially in Scenario B, where the variability between nodes capacity is higher. Although we could tweak our heuristics and obtain a better behavior, these results demonstrate that manually tuned policies cannot guarantee good performance in every working scenario. Our MDP-based problem formulation aims instead at determining good policies in every situation.

When up to 5 node types are considered, the results show that, in both scenarios, the FA-based TBVI does not lead to evident performance degradation compared to VI. Both the algorithms keep the number of SLO violations and the deployment reconfigurations below 1% of the time, and the cost paid for acquiring computing resources is reduced with respect to the threshold-based heuristics, especially in Scenario B. As the number of available node types increases to 7 or 10, TBVI continues achieving good performance, whereas the standard VI is not able to perform a sufficient number of iterations to determine an acceptable policy. In fact, at the end of the planning period, VI outputs a policy which never reconfigures the deployment, and it is not able to satisfy the SLO. These results demonstrate the scalability advantage of FA-based methods, which rely on a Q-function representation made of 10^5 parameters in the case $N_{res} = 10$, compared to 10^8 entries used by tabular methods. By using such a reduced state space representation, FA allows for problem resolution on nodes with limited resources as well.

We also evaluate the performance of TBVI using different sets of features. In particular, we consider Fixed Sparse Representation (FSR) as a baseline, and compare it to tile coding.

As regards the tiling parameters, in addition to the reference configuration described above, we also consider "coarser" tilings where $n_\lambda = \frac{N_\lambda}{6}$ and $n_{str} = \frac{K_{max}}{2}$; and "finer" tilings where $n_\lambda = N_\lambda$ and $n_{str} = K_{max}$. In all these configurations, we employ two partially overlapping rectangle-based tilings, and one stripe-based tiling. During preliminary experiments, we also verified whether any benefits could be achieved by adding more overlapping tilings, or removing either the rectangle-based ones or the stripe-based ones. However, we observed that, despite of slight improvements in some scenarios, no evident advantage could be identified in general.

Table 2: Results achieved with the TBVI algorithm using different sets of features. The average cost in parentheses corresponds to experiments with reduced planning time.

N_{res}	Features	Cost	Memory
3	Fixed Sparse Representation	0.2442 (0.1950)	**32**
3	Tile Coding (coarser)	0.0564 (0.0557)	$1.2 \cdot 10^3$
3	Tile Coding	0.0539 (**0.0536**)	$2.4 \cdot 10^3$
3	Tile Coding (finer)	**0.0537** (0.0555)	$5.7 \cdot 10^3$
10	Fixed Sparse Representation	0.3744 (0.3451)	**102**
10	Tile Coding (coarser)	0.0587 (**0.1626**)	$1.3 \cdot 10^5$
10	Tile Coding	**0.0541** (0.2245)	$2.8 \cdot 10^5$
10	Tile Coding (finer)	0.0699 (0.3123)	$6.5 \cdot 10^5$

In Table 2 we show the impact of using different sets of features. We consider configurations with 3 and 10 available types of computing nodes, and we also vary the planning execution time: 10 and 60 seconds for $N_{res} = 3$; 2 and 5 minutes for $N_{res} = 10$. As expected, we observe that FSR, which uses the smallest number of features, does not provide a good representation of the state space in any configuration, leading up to a 400% cost increase compared to tile coding. We also observe that tile coding allows to pick a different trade-off between accuracy and efficiency by adjusting its fineness. Indeed, while no significant performance difference is noticeable when the allowed planning execution time is large enough, and hence there is no advantage in adding more features to our reference configuration, if a shorter planning phase is allowed, coarse-grained tilings lead to better results, because a smaller number of parameters need to be computed. The capability of reducing

the tiling resolution could be especially useful in application scenarios where a very limited amount of computational resources is available for running the elasticity controller.

7.1.2 Learning Algorithms. As in practice the system dynamics are unlikely to be completely known, we repeat the above described experiments using reinforcement learning algorithms to determine an elasticity policy on-line. Specifically, we consider the Q-learning algorithm, described in Section 5.2, with and without Function Approximation (denoted as **Q-learning** and **Tab. Q-learning**, respectively). In addition, we also consider a solution that combines planning and learning (denoted as **Q-learning+PL**), where we use Q-learning with FA, but, instead of starting the learning process from scratch, we first perform planning using an approximate system model to initialize the Q-function. We include this solution in our experiments as in practice it is unlikely that we have no idea about the system. We consider the case in which the unknown system dynamics can be estimated, e.g., from historical data. As an example configuration, we assume that: (i) the model of the input rate variation is extracted from a one-month portion of the dataset not used in the evaluation; and (ii) the operator service time model is estimated with limited accuracy (i.e., we over-estimate the service time mean by 20%, and under-estimate the variance by 50%). We report the results of this set of experiments in Table 3. We only show the results for the first heterogeneity scenario defined above due to space limitations; nonetheless, no significant impact is noticeable by changing the set of available nodes.

Table 3: Results achieved by Tabular Q-learning, Q-learning enhanced with FA, and Q-learning initialized using an approximate system model (denoted as Q-learning+PL).

N_{res}	Algorithm	Cost	Viol. (%)	Rcf. (%)	Res.
3	Tab. Q-learning	0.5624	57.62	60.35	6.8
3	Q-learning	0.0746	2.24	0.92	4.7
3	Q-learning+PL	**0.0620**	**0.38**	**0.91**	**4.3**
5	Tab. Q-learning	0.7676	96.43	79.22	4.9
5	Q-learning	0.1102	10.73	1.99	4.5
5	Q-learning+PL	**0.0627**	**0.42**	**1.03**	**4.3**
7	Tab. Q-learning	0.7523	98.34	72.09	5.3
7	Q-learning	0.2189	36.00	4.53	4.3
7	Q-learning+PL	**0.0629**	**0.43**	**1.20**	**4.2**
10	Tab. Q-learning	0.7483	94.01	68.48	7.4
10	Q-learning	0.4263	78.94	9.43	5.5
10	Q-learning+PL	**0.1563**	**20.49**	**3.11**	**4.6**

Our experiments show that tabular Q-learning, which is guaranteed to learn the optimal policy given a sufficient amount of experience, is not able to converge to a satisfactory behavior during the simulated 9 months period. Even in the simplest scenario with 3 types of nodes, it violates the SLO for more than 50% of the experiment, reconfiguring the deployment for 60% of the time. Unfortunately, the performance gets even worse when the state space gets larger, making this simple model-free algorithm not usable in practice for our problem.

Q-learning equipped with the FA-based state space representation achieves better performance. When $N_{res} = 3$, it limits SLO violations to 2% of the experiments, with less than 1% of reconfigurations, and resource usage comparable to that of planning algorithms. Unfortunately, the performance gap with respect to the planning algorithms becomes larger as N_{res} increases, and so the parameters to be learned. The SLO violations increase up to 10% with 5 types of nodes, 36% with 7 types, and 78% with 10, which is of course not satisfactory.

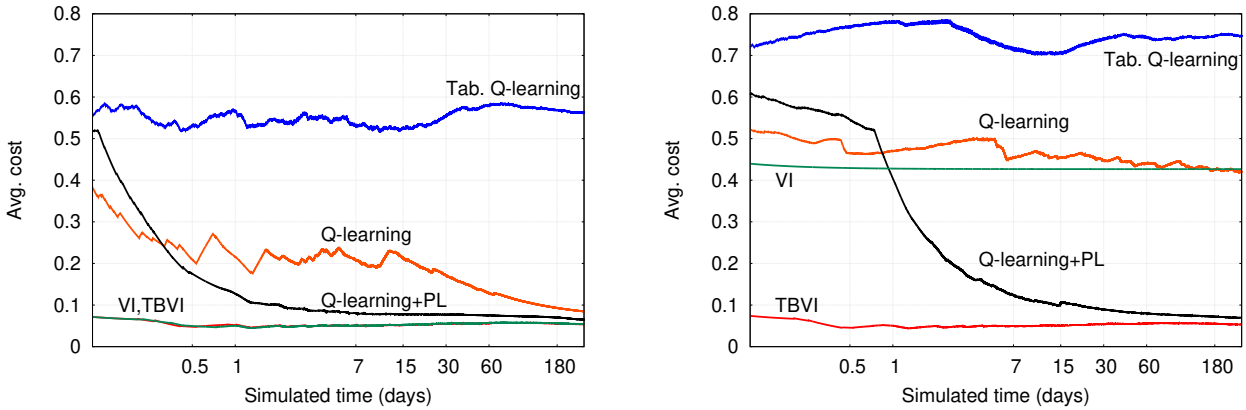
The combined Q-learning+PL solution leads to even better results, with less than 1% SLO violations for $N_{res} \leq 7$, and the average cost being comparable to the planning case. When $N_{res} = 10$, the SLO violations rise up to 20% of the time, which is anyway a fourth with respect to plain learning. Overall, the results show how the availability of partial knowledge of the system model allows to dramatically boost the process of learning an elasticity policy. This fact is further documented by Fig. 5, where we plot the average cost incurred by each algorithm during a single experiment, considering configurations with $N_{res} \in \{3, 10\}$.

Observing Fig. 5a, which shows the simpler case with $N_{res} = 3$, we can note that tabular Q-learning is far from converging to the optimal policy even after about 400,000 updates. Q-learning and Q-learning+PL instead converge towards the optimal solution determined by both VI and TBVI during the evaluation period. However, Q-learning+PL benefits from the initial planning phase, and its average cost approaches the optimal one after a single simulated day, whilst plain Q-learning shows a lower convergence velocity. We remark that the average cost paid by planning algorithms (i.e., VI and TBVI) does not decrease over time, as those algorithms compute the policy off-line during the planning phase, instead of learning it by experience. In Fig. 5b, we can observe how the performance gap between the different algorithms gets more evident as the state space becomes larger with $N_{res} = 10$. First of all, as we noted above, the VI algorithm in this scenario is not able to determine a good policy within the available planning time, and hence incurs a much higher cost compared to TBVI. In addition to Tabular Q-learning, in this configuration even the FA-based plain Q-learning is still far from convergence at the end of the simulated 9 months. Conversely, Q-learning+PL guarantees acceptable performance, approaching the best solution determined by TBVI after the first two weeks.

7.2 Application Elasticity

We have shown how FA allows to solve the operator elasticity control problem in a scalable manner. We conclude the evaluation of the proposed approaches by showing how the elasticity of a whole DSP application can be controlled in a decentralized fashion by letting each operator solve its own elasticity control problem. For these experiments, we consider a simple DSP application composed of a source, a consumer, and 6 operators, as depicted in Fig. 6. As for the previous experiments, we assume that the operator replicas can be modeled as M/G/1 queues, with service rate $\mu \in [185, 200]$ tuple/s, and service time variance $\sigma^2 \in [0, 1.5\frac{1}{\mu^2}]$.

We require the application to satisfy a SLO by keeping the source-to-consumer processing latency below $R_{SLO} = 65$ ms. In order to use our elasticity control algorithms, we first need to determine the SLO of each operator. In this paper, for the sake of simplicity, for each source-consumer path, we simply apportion an equal fraction of the end-to-end processing latency to each operator in the



(a) $N_{res} = 3$ **(b)** $N_{res} = 10$

Figure 5: Average cost over time during a single run using different algorithms.

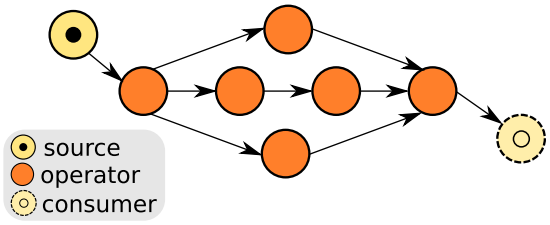


Figure 6: DSP topology used for experiments.

sequence. In other words, if a source-consumer path comprises N operators, we set $R_{max} = R_{SLO}/N$ for each operator in the path (if an operator appears in more than one path we set R_{max} as the smallest of such values). We defer to future work the investigation of more complex strategies, so as to better coordinate the different operator policies. As regards the computing infrastructure, we consider configurations with $N_{res} = 3$ and $N_{res} = 10$ types of nodes, for both the sets of heterogeneous nodes defined above.

Table 4 reports the results of these experiments, in which we consider the threshold-based heuristics, the planning algorithms, and the learning algorithms. As for the single operator case, the TH-cost heuristic does not guarantee acceptable performance, with the SLO being violated all the time. TH-speedup, which selects nodes by maximizing the speedup, instead leads to better performance, but its average cost is far from optimal in the second heterogeneity scenario due to expensive resource usage. The behavior of these simple policies is clearly not affected by adding more types of nodes (as the least costing ones, and the most powerful ones are already included in the initial set).

As expected, by exploiting the knowledge of the system model, planning algorithms achieve the best results. With both VI and TBVI, in the configuration with $N_{res} = 3$ the SLO is violated less than 0.1% of the time, with a very low number of reconfigurations (below 2%). When $N_{res} = 10$, we are not able to run the experiments using the tabular VI, as it would require too much memory to store the Q table for every operator, and we have already shown in the single operator scenario how the standard VI is not able to determine a good policy in a reasonable amount of time. Conversely, FA-based planning can be still used in this configuration, where it achieves only a slightly larger number of SLO violations (2.6%) in Scenario A, whilst they stay below 0.1% in Scenario B.

As regards the learning algorithms, tabular Q-learning does not achieve acceptable performance, with 90% and 60% of SLO violations in the two scenarios. As for the single operator case, Q-learning enhanced with FA leads to better performance when $N_{res} = 3$, and it is able to limit the SLO violations to less than 10%, and the reconfigurations to less 5% in the worst case. But in the more challenging configuration with 10 types of nodes, the algorithm, which is initialized with no *a priori* knowledge about the system, is not able to converge to a near-optimal policy during the experiment (e.g., in Scenario A it violates the SLO for more than 90% of time).

When using the approximate system model for initialization, Q-learning+PL achieves much better results, reducing the SLO violations from 9% to 0.3% in Scenario A, and from 2% to 0.1% in Scenario B, when $N_{res} = 3$. When 10 types of nodes are considered, it limits the SLO violations in Scenario A from 90% to 7%, from 22% to 0.06% in Scenario B. The number of deployment adaptations is analogously reduced in all the considered configurations, thus leading to an overall better service level. Therefore, the presented solutions can be effectively used in practice, by (i) exploiting FA to perform planning with an approximate system model, and (ii) improving the elasticity policy on-line using RL techniques.

8 CONCLUSIONS

In this paper we have studied the problem of controlling the elastic deployment of DSP applications over a heterogeneous set of computing nodes. We formulated the elasticity control problem for DSP operators as an infinite-horizon Markov Decision Process, and investigated scalable resolution techniques based on Function Approximation and tile coding. In order to compute an elasticity policy even in presence of uncertainty about the system dynamics and parameters, we also integrated FA in reinforcement learning techniques. Our numerical evaluation reveals that using FA we

Table 4: Results of the experiments in which a full DSP topology is considered. When 10 types of computing nodes are available, we do not use the tabular algorithms as they would require enough memory to store the Q table for every operator.

			Scenario A			Scenario B		
N_{res}	Algorithm	Memory	Viol. (%)	Rcf. (%)	Res.	Viol. (%)	Rcf. (%)	Res.
3	TH-cost	-	100.00	4.60	**5.3**	100.00	3.31	**2.8**
3	TH-speedup	-	0.08	1.35	10.7	0.12	0.01	90.6
3	VI	$6 \cdot 10^4$	**0.01**	**1.08**	10.1	**0.00**	**0.00**	12.0
3	TBVI	$2 \cdot 10^3$	0.05	1.68	9.5	**0.00**	0.30	11.4
3	Tab. Q-learning	$6 \cdot 10^4$	90.77	98.68	20.0	68.37	74.48	114.1
3	Q-learning	$2 \cdot 10^3$	9.00	3.07	11.7	2.22	0.77	35.0
3	Q-learning+PL	$2 \cdot 10^3$	0.26	3.45	10.8	0.11	1.03	16.3
10	TH-cost	-	100.00	4.59	**5.3**	100.00	4.11	**2.8**
10	TH-speedup	-	**0.08**	**1.36**	10.7	0.12	**0.01**	90.6
10	VI	$1 \cdot 10^8$	-	-	-	-	-	-
10	TBVI	$3 \cdot 10^5$	2.61	1.79	10.7	**0.10**	0.03	17.7
10	Tab. Q-learning	$1 \cdot 10^8$	-	-	-	-	-	-
10	Q-learning	$3 \cdot 10^5$	90.88	29.34	16.6	22.48	3.75	60.7
10	Q-learning+PL	$3 \cdot 10^5$	7.02	4.48	11.3	0.06	0.55	29.0

can overcome the scalability issues of standard MDP resolution algorithms, and also improve the convergence speed of the popular Q-learning algorithm. Moreover, our experiments show that, by combining model-based planning and model-free learning, we can obtain even better results just relying on estimates of the unknown parameters. These results demonstrate that our approach can be effective in practice, where an approximate system model can be constructed leveraging, e.g., historical data.

For future work, we plan to investigate adaptive tile coding techniques, which can build an efficient state space representation by progressively refining the current set of features during policy computation/learning, without depending on handcrafted tilings. We will also consider the opportunity of comparing linear FA-based solutions presented in this paper to non-linear approaches, including Artificial Neural Networks, and to heuristic search-based techniques, which provide an alternative set of tools to face large state spaces. In the meanwhile, we plan to further evaluate our elasticity policies by implementing them in a DSP framework.

REFERENCES

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. 2018. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Trans. Serv. Comput.* 11 (2018), 430–447.

[2] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. 2018. Decentralized Self-Adaptation for Elastic Data Stream Processing. *Future Gener. Comput. Syst.* 87 (2018), 171–185.

[3] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. 2018. Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing. *Concurr. Comput.: Pract. Exper.* 30, 9 (2018), e4334.

[4] M.D. de Assunção, A. da Silva Veith, and R. Buyya. 2018. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Netw. Comput. Appl.* 103 (2018), 1–17.

[5] T. De Matteis and G. Mencagli. 2017. Proactive Elasticity and Energy Awareness in Data Stream Processing. *J. Syst. Softw.* 127 (2017), 302–319.

[6] R.C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proc. ACM SIGMOD '13*. 725–736.

[7] B. Gedik, S. Schneider, M Hirzel, and K. Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463.

[8] A. Geramifard, T.J. Walsh, S. Tellex, G. Chowdhary, N. Roy, J.P. How, et al. 2013. A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning. *Found. Trends in Mach. Learn.* 6, 4 (2013), 375–451.

[9] P. Graubner, C. Thelen, M. Körber, A. Sterz, G. Salvaneschi, et al. 2018. Multimodal Complex Event Processing on Mobile Devices. In *Proc. ACM DEBS '18*. 112–123.

[10] V. Gulisano, R. Jiménez-Peris, M. Patiño Martinez, C. Soriente, and P. Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (2012), 2351–2365.

[11] J. He, Y. Chen, T. Z. J. Fu, X. Long, M. Winslett, L. You, and Z. Zhang. 2018. HaaS: Cloud-Based Real-Time Data Analytics with Heterogeneity-Aware Scheduling. In *Proc. IEEE ICDCS '18*. 1017–1028.

[12] T. Heinze, L. Aniello, L. Querzoni, and J. Zbigniew. 2014. Cloud-based Data Stream Processing. In *Proc. ACM DEBS '14*. 238–245.

[13] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. 2014. Auto-scaling Techniques for Elastic Data Stream Processing. In *Proc. IEEE ICDEW '14*. 296–302.

[14] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4 (2014), 46:1–46:34.

[15] Z. Jerzak and H. Ziekow. 2015. The DEBS 2015 Grand Challenge. In *Proc. ACM DEBS '15*. ACM, 266–268.

[16] A. Koliousis, M. Weidlich, R. Castro Fernandez, A.L. Wolf, P. Costa, and P. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proc. ACM SIGMOD '16*. 555–569.

[17] R. M. Kretchmar and C. W. Anderson. 1997. Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In *Proc. ICNN '97*, Vol. 2. 834–837.

[18] G. T. Lakshmanan, Y. Li, and R. Strom. 2008. Placement Strategies for Internet-scale Data Stream Systems. *IEEE Internet Comput.* 12, 6 (2008), 50–60.

[19] X. Liu, A.V. Dastjerdi, R.N. Calheiros, C. Qu, and R. Buyya. 2018. A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4 (2018), 24:1–24:33.

[20] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *Proc. IEEE ICDCS '15*. 399–410.

[21] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. 2018. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 3 (2018), 572–585.

[22] G. Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2 (2016), 13:1–13:34.

[23] M.A.U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. 2015. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proc. IEEE ICDE '15*. 137–148.

[24] M.L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

[25] G. Russo Russo, M. Nardelli, V. Cardellini, and F. Lo Presti. 2018. Multi-Level Elasticity for Wide-Area Data Streaming Systems: A Reinforcement Learning Approach. *Algorithms* 11, 9 (2018), 134. https://doi.org/10.3390/a11090134

[26] F. Starks, V. Goebel, S. Kristiansen, and T. Plagemann. 2018. Mobile Distributed Complex Event Processing—Ubi Sumus? Quo Vadimus? In *Mobile Big Data: A Roadmap from Models to Technologies*. Springer, 147–180.

[27] R.S. Sutton. 1995. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In *Proc. NIPS '95*. MIT Press, 1038–1044.

[28] R.S. Sutton and A.G. Barto. 1998. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.

[29] C. Watkins and P. Dayan. 1992. Q-learning. *Machine Learning* 8, 3-4 (1992), 279–292.

[30] K.P. Yoon and C.-L. Hwang. 1995. *Multiple Attribute Decision Making: an Introduction*. Sage Pubs.