Efficient Operator Placement for Distributed Data Stream Processing Applications

Matteo Nardelli *Member, IEEE*, Valeria Cardellini, *Member, IEEE*, Vincenzo Grassi, and Francesco Lo Presti, *Member, IEEE*

Abstract—In the last few years, a large number of real-time analytics applications rely on the Data Stream Processing (DSP) so to extract, in a timely manner, valuable information from distributed sources. Moreover, to efficiently handle the increasing amount of data, recent trends exploit the emerging presence of edge/Fog computing resources so to decentralize the execution of DSP applications. Since determining the Optimal DSP Placement (for short, ODP) is an NP-hard problem, we need efficient heuristics that can identify a good application placement on the computing infrastructure in a feasible amount of time, even for large problem instances.
In this paper, we present several DSP placement heuristics that consider the heterogeneity of computing and network resources; we divide them in two main groups: model-based and model-free. The former employ different strategies for efficiently solving the ODP model. The latter implement, for the problem at hand, some of the well-known meta-heuristics, namely greedy first-fit, local search, and tabu search. By leveraging on ODP, we conduct a thorough experimental evaluation, aimed to assess the heuristics' efficiency and efficacy under different configurations of infrastructure size, application topology, and optimization objectives.

Index Terms—Distributed data stream processing, Geo-distributed systems, Heuristics, Operator placement, Quality of Service.

1 Introduction

TODAY we have access to a huge amount of data from which we would like to extract valuable information in a timely manner. The identification of customer sentiments from social network data, the prediction of health risks from wearable devices, or the optimization of public transports in response to social events are only some examples of the potentialities of intelligent analytics services. Exploiting on-the-fly computation, Data Stream Processing (DSP) applications can process unbounded streams of data in a near real-time fashion. DSP applications are typically deployed on large-scale and centralized (Cloud) data centers, which are often distant from data sources. However, in the emerging scenario, DSP applications expose strict quality requirements, thus calling for an efficient usage of the underlying processing infrastructure. Indeed, as data increase in size, pushing them towards centralized data centers could exacerbate the load on the network infrastructure and also introduce excessive delays. To improve scalability and reduce network delays, a solution lies in taking advantage of the ever increasing presence of edge/Fog computing resources [1]. They allow to decentralize the execution of DSP applications, by moving the computation towards the network edges (i.e., close to data sources). Nevertheless, the use of a geographically distributed infrastructure poses new challenges to deal with. They include network and system heterogeneity as well as non-negligible network delays among nodes processing different parts of a DSP application. This latter aspect could have a strong impact on DSP applications for latency-sensitive domains.

• *V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli are with University of Rome Tor Vergata, Italy. E-mails: {cardellini, nardelli}@ing.uniroma2.it, vincenzo.grassi@uniroma2.it, lopresti@info.uniroma2.it*

In such a distributed scenario, a relevant problem consists in determining the computing nodes that should host and execute each processing element (i.e., *operator*) of a DSP application, aiming to optimize some Quality of Service (QoS) attributes. This problem is known in literature as the *operator placement problem* (or scheduling problem) [2]. In a previous work [3], we proposed a general formulation of the Optimal DSP Placement (for short, ODP), which takes into account the heterogeneity of application requirements and infrastructural resources. Although ODP determines the optimal placement for a DSP application, we demonstrated that it solves an NP-hard problem. For this reason, we need efficient heuristics that can solve the DSP placement problem within a feasible amount of time, even for large problem instances. Several heuristics have been already proposed in literature (e.g., [4], [5], [6], [7], [8], [9]). However, most of them have been designed for a clustered environment where network delays are negligible (e.g., [10], [11], [12]). Therefore, they are not well suited for the edge/Fog environment we want to consider. Other proposals lack of flexibility and cannot easily optimize new placement goals (e.g., [7], [8]). Furthermore, a systematic analysis of the existing heuristics' performance, under different deployment configurations and with respect to the (theoretic) optimal solution, is almost always missing.

In this paper, we present several heuristics that solve the operator placement problem while considering the heterogeneity of application requirements and computing resources. Then, we assess the heuristics' quality by using the optimal DSP placement formulation (i.e., ODP). We develop the heuristics following three main guidelines: *flexibility*, *quality* of the computed placement solution, and *optimal model exploitation*. As regards *flexibility*, we have observed that many solutions are specifically crafted for optimizing specific QoS metrics and cannot be easily customized or

extended to account for new metrics (e.g., [7], [8], [13], [14]). Conversely, we aim to provide a general framework that can be easily tuned to optimize different QoS metrics (e.g., response time, availability, network usage, or a combination thereof). As regards *quality*, most of the existing heuristics usually determine best-effort solutions, meaning that they do not provide guarantees, quantitative, or qualitative information regarding their ability to compute near-optimal solutions. For example, many placement approaches rely on greedy strategies (e.g., [4], [5], [10], [12]) that, by moving through local improvements, can miss the identification of globally optimal configurations. Together with the reduced resolution time, we aim to evaluate the heuristics' ability to compute placement solutions that are as close as possible to the theoretically optimal ones. This is relevant when applications with stringent requirements run over heterogeneous and distributed infrastructures, where the inefficient utilization of resources can strongly penalize the application performance. As regards *optimal model exploitation*, we want to explore the possibility of efficiently using the ODP model, aiming to determine high-quality placement solutions (as also proposed in [15], [16]).

The main contributions of this paper are as follows.

- We design several model-based and model-free heuristics. The *model-based* heuristics revolve around the ODP model. They determine the application placement by solving ODP on a reduced set of computing resources. The *model-free* heuristics implement the greedy first-fit, local search, and tabu search approaches for the problem at hand. All the proposed heuristics rely on a special *penalty function*, which captures the cost of using any given computing resource. Such function allows to easily deal with single and multi-dimensional QoS attributes.
- We run a thorough set of numerical experiments to evaluate the proposed heuristics under different configurations of infrastructure size, application topology, and optimization objective. To assess the heuristic performance, we use ODP as benchmark. We demonstrate that there is not a *one-size-fits-all* heuristic; however, we identify the heuristic that, in general, achieves the best trade-off between resolution time and quality of the computed placement solution.
- We integrate the model-based and model-free heuristics in Apache Storm, a well-known open source DSP framework. Then, using Storm, we show how the heuristics can be used in a real setting to deploy DSP applications over geo-distributed computing resources. To this purpose, we use a DSP application that solves the DEBS 2015 Grand Challenge.

The remainder of the paper is organized as follows. In Section 2, we discuss related works. In Section 3, we present the system model and define the operator placement problem. Then, we describe the optimal placement formulation (Section 4) and present the model-based and model-free heuristics (Sections 5 and 6). In Section 7, we discuss a broad set of numerical and prototype-based experiments to assess the proposed heuristics. We conclude in Section 8. The paper is accompanied by a supplemental document which contains more details about the optimal placement model and the experimental results.

2 Related Work

The operator placement problem has been widely investigated in literature under different modeling assumptions and optimization goals, as surveyed in [2], [17]. In this section, we review the related works organizing them along three main dimensions, that capture one or more related facets of the problem: (1) placement goals, i.e., optimization objectives; (2) methodologies used to define the application placement; and (3) characteristics of the distributed computing infrastructure managed by the placement solution.

Objectives. The existing solutions aim at optimizing a diversity of objectives, such as to minimize the application response time (e.g., [5], [11], [12], [13]), the inter-node traffic (e.g., [4], [18], [19], [20], [21]), the network usage (e.g., [7], [8]), or a generic cost function that can comprise different QoS metrics (e.g., [15], [22], [23], [24]). Similarly to these latter works, the goals of our heuristics can be flexibly tuned so to encompass several QoS metrics, such as application response time, availability, network usage, or a combination thereof. The advantage of a multi-objective approach is that it allows to easily explore different trade-offs among the placement goals.

Methodologies. The operator placement problem has been addressed relying on a variety of methodologies. They include mathematical programming (e.g. [3], [8], [15], [16], [22]), graph-theoretic approaches (e.g., [9], [14], [25]), greedy approaches (e.g., [4], [5], [6], [10], [12], [20], [26]), meta-heuristics (e.g., genetic algorithms [27], local search [16], [28], tabu search and simulated annealing [16]), as well as custom heuristics (e.g., [7], [13], [18], [23], [29]).

The most popular open-source DSP frameworks (Storm, Spark Streaming, Flink, and Heron) usually adopt heuristic policies for the placement. They range from the simple round-round to the resource-aware heuristics R-Storm [29] both in Storm, to a multi-layer heuristic in Heron [30]. The latter first packs operators in containers using a round-robin or first-fit policy and then places containers on the computing infrastructure. Flink inherits from Stratosphere [31] a query optimizer that transforms and allocates the application graph, so to minimize a cost function that captures network traffic and CPU load.

The model-based heuristics we propose rely on ODP and explore different strategies for selecting a suitable subset of computing resources, so to speed-up the resolution time of ODP. With our model-free heuristics, we also consider strategies based on popular heuristics and meta-heuristics. The works most closely related to ours have been proposed in [16], [32]. Stanoi et al. [16] focus on maximizing the input data rate that the DSP system can support, acting on both the order of operators and their placement on the resources. In this respect, we do not consider operator re-ordering. Together with the (nonlinear) problem formulation, the authors propose different heuristics, based on local search, tabu search, and simulated annealing. Gu et al. [32] focus on minimizing the communication cost for DSP applications deployed on geo-distributed data centers and formulate a mixed-integer linear programming problem. To address the

computation efficiency issue, they propose a heuristic based on solving the linear relaxation of the problem.

Differently from all the above mentioned heuristics, we use the optimal placement formulation as a benchmark against which we compare the heuristics performances in terms of resolution time and solution quality. Abrams and Liu [26] compare greedy placement heuristics with the optimal in terms of placement cost; nevertheless, they only consider tree-structured application graphs.

Computing Infrastructure. Most of the existing solutions have been designed for a clustered environment, where network latencies are almost zero (e.g. [10], [11], [12]). Although interesting, these approaches might not be suitable for geo-distributed environments, where the non-negligible network latencies have a negative impact on the application performance. Although not explicitly modeling the network, some other works indirectly consider the network contribution by minimizing the amount of data exchanged between computing nodes (e.g., [4], [18], [20], [22], [25], [33]). For example, Eidenbenz et al. [22] consider a special type of DSP application topologies (i.e., serial-parallel decomposable graphs) and propose a heuristic that minimizes processing and transfer cost, but it works only on resources with uniform capacity. Fischer et al. [25] use a graph partitioning technique to optimize the amount of exchanged data. Relying on a greedy best-fit heuristic, Aniello et al. [4] and Xu et al. [20] propose centralized algorithms that minimize the inter-node traffic. Specifically, the solution in [4] co-locates operators with a higher amount of pairwise communication within a single node, whereas the proposal in [20] assigns each operator in descending order of incoming and outgoing traffic. The decentralized solution presented by Zhou et al. [21] finds the placement that minimizes the inter-node traffic while balancing the load among computing nodes.

Other works, e.g., [5], [7], [8], [13], [32], explicitly take into account network latencies, thus representing more suitable solutions to operate in a geo-distributed DSP system. Pietzuch et al. [7] and Rizou et al. [8] minimize the network usage, that is the amount of data that traverses the network at a given instant. Both these solutions propose a decentralized placement algorithm that exploits a latency space to find a good placement. To this end, Pietzuch et al. [7] recur to an equivalent representation of the DSP application as a system of springs, whereas Rizou et al. [8] exploit the mathematical properties of the network usage function. Backman et al. [5] and Chatzistergiou et al. [13] propose two different approaches to partition and assign group of operators while minimizing the application latency. Also the solution by Zhu et al. [9] explicitly takes into account the computational and communication delays; however, they assume that a resource node can host at most a single operator. We consider this hypothesis not realistic in today's DSP systems; therefore, our heuristics enable the co-location of operators on a resource node, according to its computational capacity. Similarly to this last group of works, we explicitly model the impact of network heterogeneity, both in terms of delay and availability. Our heuristics can easily take into account other QoS metrics of computing and network resources, such as cost, bandwidth, or energy capacity.

So far, only few works propose solutions specifically designed for Fog computing environments. SpanEdge [23] allows to specify which operators should be placed as close as possible to the data sources, while Arkian et al. [15] propose an integer non-linear formulation for placing IoT applications over Fog resources. To reduce resolution time, they linearize the problem; nevertheless, we show that also linear formulations suffer from scalability issues. We provide new heuristics to efficiently solve the placement problem when dealing with large problem instances.

In our placement characterization, we have not considered two important issues: the exploitation of performance-enhancing techniques (e.g., operator replication, transformations of application graph) and the run-time adaptation of the application placement. Determing the operator replication degree is often addressed as an independent and orthogonal decision with respect to the operator placement. Most works exploit operator replication at run-time to achieve elasticity, e.g., [34], [35]. In [36], we present a problem formulation that jointly optimizes the replication and placement of DSP applications, but it is not suitable for large scale problem instances. In this paper, we assume that the operator replication degree has been set at application design time; so, we target the initial operator placement.

Since changes of different kinds can occur during the application run (e.g., variability in system conditions, stream processing workload, location of data sources and sinks [17]), the placement should be updated at run-time so to maintain the desired application performance. Depending on the DSP framework, the new placement can be enacted by redeploying from scratch all the application (e.g., in Storm) or by migrating only a subset of operators, e.g. [37], [38]. A common approach to deal with placement adaptation relies on solving the placement problem at regular intervals, so to update the operator location, e.g., [20], [39]. Such approach can be realized through efficient heuristics, as those here proposed, that recompute the placement in a feasible amount of time. However, run-time adaptation costs [35], [38], [40], which arise from transferring the DSP operators state and/or freezing the application processing, should also be explicitly considered while reconfiguring the deployment, because they can significantly decrease the application performance. This would require to deal with a larger number of issues (e.g., impact of replication, load distribution, reliability) which does not *de facto* change the problem formulation structure (see [40] for the resulting optimization problem). The machinery and results presented in this paper would, with some adjustments, also apply to the adaptation problem (as discussed in Section 7.6). We postpone to future work the study of heuristics that take into account such costs as well as the determination of the replication degree. To more effectively cope with the highly changing execution environment of Fog computing (e.g., mobility of data sources and consumers), such heuristics could also exploit decentralized decision-making approaches, as in [41], [42], as well as be integrated within the adaptive selection of the placement strategy [42].

3 System Model and Problem Statement

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

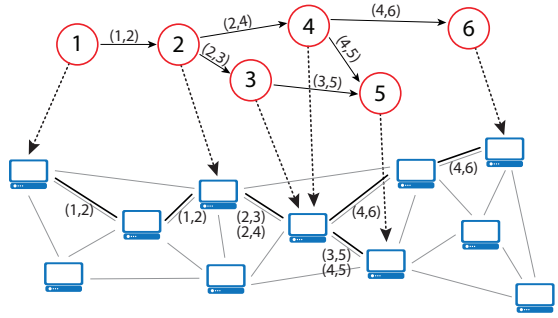


Fig. 1: Placement of the application operators on the computing and network resources.

DSP Model. A DSP application consists of a network of operators connected by streams. An operator is a self-contained processing element that carries out a specific operation (e.g., filtering, aggregation, merging). A stream is an unbounded sequence of data (e.g., packet, tuple, file chunk). A DSP application can be represented as a labeled directed acyclic graph (DAG) $G_{dsp} = (V_{dsp}, E_{dsp})$: the nodes in V_{dsp} represent the application operators as well as the data stream sources and sinks (i.e., nodes with no incoming and no outgoing link, respectively); the links in E_{dsp} represent the streams flowing between nodes. Due to the difficulties in formalizing the non-functional attributes of an abstract operator, we characterize it with the non-functional attributes of a reference implementation on a reference architecture: C_i, the amount of resources required for its execution; and R_i, the operator latency per unit of data. We characterize the stream exchanged from operator i to j, $(i,j) \in E_{dsp}$, with its average data rate $\lambda_{(i,j)}$.[1]

Resource Model. Computing and network resources can be represented as a labeled, fully connected, and directed graph $G_{res} = (V_{res}, E_{res})$: the nodes in V_{res} represent the distributed computing resources; the links in E_{res} represent the *logical connectivity* between nodes. Observe that, at this level, links represent the logical links across the network, which results by the underlying physical network paths and routing strategies. Each node $u \in V_{res}$ is characterized by: C_u, the amount of available resources; S_u, the processing speed-up on a reference processor; and A_u, its availability, i.e., the probability that u is up and running. Each link $(u,v) \in E_{res}$, with $u, v \in V_{res}$ is characterized by: $d_{(u,v)}$, the network delay between node u and v; and $A_{(u,v)}$, the link availability, i.e., the probability that the link between u and v is active. This model considers also edges of the type (u,u); they capture network connectivity between operators placed in the same node u, so they are considered as perfect links, i.e., always active with no network delay.

Operator Placement Problem. The DSP placement problem consists in determining a suitable mapping between the DSP graph G_{dsp} and the resource graph G_{res} so that all constraints are fulfilled. Figure 1 represents a simple instance of the problem. Observe that a DSP operator cannot be usually placed on every node in V_{res}, because of physical (i.e., *pinned* operator) or other motivations (e.g., security, political). This observation allows us to consider for each operator $i \in V_{dsp}$ a subset of candidate resources $V_{res}^i \subseteq V_{res}$ where it can be deployed. For example, if data stream sources and sinks ($V_{dsp}^{pinned} \subset V_{dsp}$) are external applications, their placement is fixed, that is $\forall i \in V_{dsp}^{pinned}, |V_{res}^i| = 1$.

1. We assume that the amount of resources C_i is sufficient to sustain the input rate to the operator i, i.e., $\lambda_i = \sum_{j:(j,i)\in E_{dsp}} \lambda_{(j,i)}$, on the reference architecture. R_i is the associated (average) latency.

4 The Placement Problem

In this section we formalize the Optimal DSP Placement (ODP) problem and provide an overview of the proposed heuristics. We also define the resource penalty function which plays a key role in our heuristics.

4.1 Optimal Placement Formulation

The ODP problem can be conveniently formulated as an Integer Problem (IP), where the operators placement is modeled by using binary variables $x_{i,u}$, $i \in V_{dsp}$, $u \in V_{res}^i$: $x_{i,u} = 1$ if operator i is deployed on node u and $x_{i,u} = 0$ otherwise. The ODP formulation takes the general form:

$$\min_{\boldsymbol{x}} F(\boldsymbol{x})$$

$$\textbf{subject to:} \quad \sum_{i \in V_{dsp}} C_i x_{i,u} \leq C_u \qquad \forall u \in V_{res} \qquad (1)$$

$$\sum_{u \in V_{res}^i} x_{i,u} = 1 \qquad \forall i \in V_{dsp} \qquad (2)$$

$$x_{i,u} \in \{0,1\} \qquad \forall i \in V_{dsp}, \; u \in V_{res}^i \qquad (3)$$

where $F(\boldsymbol{x})$ represents a suitable objective function (to be minimized) and $\boldsymbol{x}$ the placement vector, i.e., $\boldsymbol{x} = \langle x_{i,u} \rangle$, $\forall i \in V_{dsp}$, $\forall u \in V_{res}^i$. In the formulation, (1) represents the nodes' resources constraints, which limit the placement of operators on a node $u \in V_{res}$ according to its available resources. Equation (2) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V_{res}^i$. The objective function $F(\boldsymbol{x})$ defines the placement strategy goals. Depending on the usage scenario, a DSP placement strategy could be aimed at optimizing different, possibly conflicting, QoS attributes. In this paper, without lack of generality, we consider the application response time $R(\boldsymbol{x})$, the application availability $A(\boldsymbol{x})$, and the network usage $Z(\boldsymbol{x})$. This leads to a multi-objective optimization problem, which can be transformed into a single objective problem using the Simple Additive Weighting technique [43]. To this end, we define $F(\boldsymbol{x})$ as a weighted sum of the normalized QoS attributes of the application, as follows:

$$F(\boldsymbol{x}) = w_r \frac{R(\boldsymbol{x}) - R_{\min}}{R_{\max} - R_{\min}} + w_a \frac{\log A_{\max} - \log A(\boldsymbol{x})}{\log A_{\max} - \log A_{\min}} + w_z \frac{Z(\boldsymbol{x}) - Z_{\min}}{Z_{\max} - Z_{\min}} \qquad (4)$$

where $w_r, \; w_a, \; w_z \geq 0$, $w_r + w_a + w_z = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$), $A_{\max}$ ($A_{\min}$), and $Z_{\max}$ ($Z_{\min}$) denote respectively the maximum (minimum) value for the overall expected response time, availability, and network usage. Actually, we consider the logarithm of the availability, $\log A(\boldsymbol{x})$, so to obtain a linear expression. Observe that, after normalization, each metric ranges in the interval $[0,1]$, where the value 0 corresponding to the best and 1 to the worst metric value. As shown in Appendix A,

with this choice for the objective function $F(\boldsymbol{x})$, the ODP problem takes the form of an Integer Linear Problem (ILP)[2] which can be solved via standard techniques.

4.2 Heuristics: Overview

As demonstrated in [3], ODP is NP-hard. For supporting online operations, we develop several new heuristics for solving the operator placement problem. We present them as belonging to two main groups: model-based and model-free heuristics. All of them aim to solve the ODP problem while minimizing the objective function $F(\boldsymbol{x})$, defined in (4). To this end, the heuristics use a special *penalty function*, that defines an order relationship among resources, with respect to their ability in minimizing the objective function $F(\boldsymbol{x})$.

The model-based heuristics are named Hierarchical ODP, ODP-PS, and RES-ODP. They try to restrict the set of candidate computing resources, before solving the ODP problem. *Hierarchical ODP* represents the computing infrastructure as organized in a hierarchy of virtual data centers (VDCs). Then, starting from the hierarchy root, this strategy recursively explores subsets of VDCs, until identifying the computing resources that will execute the application operators. Instead of aggregating resources in VDCs, *ODP on a Pruned Space* (ODP-PS) works directly with computing nodes. First, it identifies a reduced set of computing nodes, i.e., the best candidates for hosting the DSP operators. Then, it solves ODP by considering only this set of candidate computing resources. *Relax, Expand and Solve ODP* (RES-ODP) exploits the linear relaxation of ODP so to identify a first set of candidates resources, which is then augmented by including some neighbor nodes of the candidates. Ultimately, RES-ODP solves the placement problem by considering only the set of candidate computing nodes. Sections from 5.1 to 5.3 present in detail the model-based heuristics.

The model-free heuristics implement some of the well-known meta-heuristics to solve the ODP problem. *Greedy First-fit* is one of the most popular approaches used to solve the bin-packing problem; it is also widely applied for solving the operator placement problem (e.g. [4], [20]). Our implementation considers the DSP operators as elements to be (greedily) allocated in bins of finite capacity, which represent the computing resources. *Local Search* is an algorithm that, starting from an initial placement, greedily moves through the configurations that reduce the objective function $F(\boldsymbol{x})$, until a stopping criterion is met (e.g., no further improvement can be achieved). Since it only accepts local improvements, it can get stuck in local optima, missing the identification of a global optimum solution. *Tabu Search* uses a simple strategy to escape from local optima and further explore the solution space, thus improving the probability of finding a globally optimal solution. Starting from an initial placement, through a set of iterations, it finds a local optimum; then, it explores the search space by selecting the best non-improving placement configuration, which can be found in the neighborhood of the local optimum. To avoid cycles back to an already visited configuration, the procedure uses a limited *tabu list* of previous moves that cannot be further explored. In [16], Stanoi et al. use local search and tabu search to efficiently solve the operator placement problem; nevertheless, their work does not provide details on key design choices. Differently from existing approaches (e.g., [4], [16]), our model-free heuristics rely on the resource penalty function δ so to improve the quality of the computed placement solutions. Sections from 6.1 to 6.3 present in detail the model-free heuristics.

The optimal placement depends on the location of data sources and sinks; all these heuristics assume that their placement is fixed a priori. If their location is not defined, we can conveniently pin them before solving the heuristics.

4.3 Resource Penalty Function

The heuristics involve, at different stages, the selection of suitable nodes and/or links to guide the placement decisions. To this end, we need a metric that enables a comparison among different alternatives; it should capture the cost (in terms of objective function $F(\boldsymbol{x})$) of using any given node/link resources for the application deployment.

The resource selection problem would be trivial in an ideal setting where we could have access to a node with infinite capacity, infinite computing speed, and 100% availability, to which also the data sources and data sinks could be pinned. In such a case, we would just place all the operators on this single node. However, in real use cases, because of the limited capacity of a single node and the data sources and sinks distribution, we need to possibly deploy the application operators on several computing nodes. This placement introduces network delays and network traffic, and can also suffer from non-ideal node/link availability. Therefore, it becomes natural to associate to any resource a penalty which captures the relative measure of degradation with respect to an ideal resource. The latter is characterized by infinity capacity, infinite computing speed, and no network delay. The lower the penalty, the better the resource.

We introduce a link penalty function $\delta(u,v) \in [0,1]$, which assigns a *penalty* to the network link $(u,v) \in E_{res}$ as a measure of degradation with respect to the ideal performance. Formally, we define the link penalty function $\delta(u,v)$ as the weighted combination of the QoS attributes of the link (u,v) and the associated upstream and downstream nodes $u,v \in V_{res}$, respectively. We have:

$$\delta(u,v) = w_r\delta_R(u,v) + w_a\delta_A(u,v) + w_z\delta_Z(u,v)$$

where $w_r, w_a, w_z \in [0,1]$ are the same weights used in $F(\boldsymbol{x})$. The terms $\delta_R(u,v)$, $\delta_A(u,v)$, and $\delta_Z(u,v)$ model the penalty with respect to the single QoS metrics, i.e., application response time, availability, and network usage, respectively. These terms are defined as follows:

$$\delta_R(u,v) = \frac{\tilde{R}(u,v) - \tilde{R}_{\min}}{\tilde{R}_{\max} - \tilde{R}_{\min}} \quad \delta_Z(u,v) = \frac{\tilde{Z}(u,v) - \tilde{Z}_{\min}}{\tilde{Z}_{\max} - \tilde{Z}_{\min}}$$
$$\delta_A(u,v) = \frac{\log \tilde{A}_{\max} - \log \tilde{A}(u,v)}{\log \tilde{A}_{\max} - \log \tilde{A}_{\min}}$$

where $\tilde{R}(u,v)$, $\log \tilde{A}(u,v)$, and $\tilde{Z}(u,v)$ capture the effects of using the link (u,v) on the application placement, in terms of the specific QoS metric. We compute these terms by considering the placement of two reference operators on

2. This requires the introduction of auxiliary variables which model the assignment of streams to links and the application response time.

u and v, respectively. The reference operators allow to neglect the application-specific contributions. The maximum and minimum values of $\tilde{R}(u,v)$, $\tilde{A}(u,v)$, and $\tilde{Z}(u,v)$ are respectively $\tilde{M}_{\max}$ and $\tilde{M}_{\min}$, with $\tilde{M} = \tilde{R}|\tilde{A}|\tilde{Z}$.

The response time $\tilde{R}(u,v)$ on the link (u,v) depends on the network delay $d_{(u,v)}$ and on the execution time of the reference operators on u and v, respectively. We have:

$$\tilde{R}(u,v) = d_{(u,v)} + \frac{R_{\mathbb{1}}}{S_u} + \frac{R_{\mathbb{1}}}{S_v}$$

where $R_{\mathbb{1}}$ is the (unitary) execution time of the reference operators and S_u and S_v are the processing speed-up of u and v, respectively. The term $\log \tilde{A}(u,v)$ gauges the probability that the link (u,v) and the nodes u and v are up and running; it is computed as:

$$\log \tilde{A}(u,v) = \begin{cases} \log A_{(u,v)} + \log A_u + \log A_v & \text{if } u \neq v \\ \log A_u & \text{if } u = v \end{cases}$$

where $A_{(u,v)}$, A_u, and A_v are the availability of (u,v), u, and v, respectively. The network usage $\tilde{Z}(u,v)$ models the amount of data that traverses the network at a given time. Given the unitary data rate $\lambda_{\mathbb{1}}$ exchanged between the reference operators, we define $\tilde{Z}(u,v)$ as follows:

$$\tilde{Z}(u,v) = \lambda_{\mathbb{1}} d_{(u,v)}$$

5 Model-based Heuristics

In this section, we present in detail the model-based heuristics, namely Hierarchical ODP, ODP-PS, and RES-ODP.

5.1 Hierarchical ODP

Hierarchical ODP represents the underlying infrastructure as organized in a limited number of entities, named *virtual data centers*[3] (VDCs). A VDC abstracts a group of computing nodes and related network links, which are exposed as an aggregated and more powerful computing element. If the computing infrastructure contains a very large number of resources, grouping them in VDCs may still result in a large number of VDCs. To further reduce their number, the heuristic can further aggregate the VDCs in a higher level of VDCs. This process results in a hierarchical representation of the computing infrastructure, where the number of entities (i.e., resources or VDCs) decreases from bottom up. Hierarchical ODP exploits this structural property to iteratively solve ODP and filter out groups of resources not suitable for running the DSP operators. In its last step, Hierarchical ODP computes the placement on a limited number of resources.

Specifically, Hierarchical ODP determines the application placement as presented in Algorithm 1 (see the HierarchicalODP function). First, it represents the computing infrastructure as a hierarchy of VDCs (line 5). The hierarchy is stored in a tree of height L, referred as H_{res}, where the tree nodes are VDCs and the leaves are the computing resources (stored at level L). A VDC of level $l \in [0, L-1]$ represents a group of VDCs/resources of level $l+1$. Moreover, the VDCs of the l-th level, $H_{res}[l]$, are interconnected by logical links which result by the physical network links of level L. Then, the heuristic exploits the infrastructure representation by navigating the hierarchy from the root down to the leaves. At level l, it determines the application placement on the available VDCs in $H_{res}[l]$ using ODP (line 8). The computed placement solution allows to identify the only VDCs/resources of level $l+1$ that will be used in the next resolution round (line 12). When $l = L$, the heuristic runs ODP on the resource graph and returns the application placement on the computing nodes (line 9).

3. We use the term *virtual data center* without any correlation with the concept of physical data centers managed by service providers.

Algorithm 1 Hierarchical ODP

```
1: function HierarchicalODP(G_dsp, G_res, g)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     Input: g, grouping factor
5:     H_res ← CreateHierarchy(G_res, g)
6:     L ← height of the hierarchy H_res
7:     for l in [0, ..., L] do
8:         S ← solve ODP (G_dsp, H_res[l])
9:         If l = L return S end if
10:        H^S_res[l] ← VDCs used in the placement solution S
11:        T_res ← extract from H^S_res[l] resources of level l + 1
12:        keep in H_res[l + 1] only the resources in T_res
13:    end for
14: end function
15: function CreateHierarchy(G_res, g)
16:    l ← ⌊log_g(|G_res|)⌉ − 1
17:    H_res[l] ← G_res                    ▷ H_res is an array of graphs
18:    while l ≥ 0 do
19:        k ← g^l
20:        C ← createClustersUsingKMeans(k, H_res[l])
21:        l ← l − 1
22:        H_res[l] ← createVDCfromClusters(C);
23:    end while
24:    return H_res
25: end function
```

Observe that the hierarchical representation of the computing infrastructure makes the exploration of the solution space faster, thus improving the heuristic scalability. Indeed, although ODP is solved multiple times, each problem instance includes a limited number of computing resources.

On Resource Aggregation. CreateHierarchy creates the hierarchical representation of the infrastructure (see Algorithm 1). First of all, it identifies the number of hierarchical levels l to be created: it considers the number of computing resources $|G_{res}|$ and a *groupingFactor* g. The latter is a parameter that allows to trade-off the hierarchy height and the number of entities within each VDC: see line 16; $\lfloor a \rceil$ indicates the rounding of $a \in \mathbb{R}$ to the closest integer in $\mathbb{N}$. Afterwards, the heuristic creates the hierarchical structure by proceeding in a bottom-up manner (lines 18–23). At level l, it uses a clustering algorithm (e.g., k-Means [44]) to determine $k = g^l$ groups of nodes, so as to minimize the average link penalty $\delta(u,v)$ between every pair of nodes u and v within the same group. Using the penalty function allows to identify computing nodes as well as network links with limited cost in term of $F(\boldsymbol{x})$. Each group of resources is then used to create a new VDC of the l-th level of the hierarchy (line 22).

The non-functional attributes of a new VDC are computed as follows. Let $\mathcal{C}_\alpha$ be the set of entities of level $l+1$ grouped around the same centroid α identified by

Algorithm 2 ODP-PS

1: **function** ODP-PS(G_{dsp}, G_{res})
2: **Input:** G_{dsp}, DSP application graph
3: **Input:** G_{res}, computing resource graph
4: $H_{res} \leftarrow$ create tree-like structure of subsets of resources
5: $P \leftarrow$ resources hosting the pinned operators of G_{dsp}
6: $T_{res} \leftarrow$ smallest set in H_{res} that strictly contains P
7: **do**
8: $S \leftarrow$ solve ODP (G_{dsp}, T_{res})
9: **if** placement S not found **and** T_{res} equals G_{res} **then**
10: **return** NOT_FEASIBLE
11: **end if**
12: $T_{res} \leftarrow$ smallest set in H_{res} that strictly contains T_{res}
13: **while** (placement S not found)
14: **return** S
15: **end function**

Algorithm 3 RES-ODP

1: **function** RES-ODP(G_{dsp}, G_{res}, k)
2: **Input:** G_{dsp}, DSP application graph
3: **Input:** G_{res}, computing resource graph
4: **Input:** k, number of new neighbors to include
5: $S \leftarrow$ solve ODP (G_{dsp}, G_{res}), no integrality constraints
6: $\mathcal{G}_{res} \leftarrow$ candidate resources used in S
7: $\mathcal{G}_{res} \leftarrow$ add k neighbor for each resource in $\mathcal{G}_{res}$
8: $S \leftarrow$ solve ODP (G_{dsp}, $\mathcal{G}_{res}$), with integrality constraints
9: **return** S
10: **end function**

the clustering algorithm (e.g., k-Means), and let $\mathcal{U}$ be the VDC of level l to created. The availability of $\mathcal{U}$, $A_{\mathcal{U}}$, and the processing speed-up of $\mathcal{U}$, $S_{\mathcal{U}}$, are defined as the average availability and speed-up of the computing resources in $\mathcal{C}_\alpha$, respectively. The amount of available resources, $Res_{\mathcal{U}}$, is the overall number of resources available in $\mathcal{C}_\alpha$. We have:

$$A_{\mathcal{U}} = \frac{\sum_{u \in \mathcal{C}_\alpha} A_u}{|\mathcal{C}_\alpha|}, \; S_{\mathcal{U}} = \frac{\sum_{u \in \mathcal{C}_\alpha} S_u}{|\mathcal{C}_\alpha|}, \; Res_{\mathcal{U}} = \sum_{u \in \mathcal{C}_\alpha} Res_u$$

At level l, the VDCs are interconnected by logical links that result from the connectivity between the computing resources at level $(l+1)$ of the hierarchy. We define $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta$ as the set of links that connect an element in $\mathcal{C}_\alpha$ to an element in $\mathcal{C}_\beta$, where $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$ are two groups identified by k-Means: $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta \doteq \{(u,v) | \, u \in \mathcal{C}_\alpha, v \in \mathcal{C}_\beta\}$. Let $\mathcal{U}$ and $\mathcal{V}$ be the VDCs created by $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$, respectively. The logical link $(\mathcal{U}, \mathcal{V})$ has availability $A_{(\mathcal{U},\mathcal{V})}$ and network delay $d_{(\mathcal{U},\mathcal{V})}$ defined as the average availability and the average network delay of the links in $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta$, respectively. We have:

$$A_{(\mathcal{U},\mathcal{V})} = \frac{\sum_{(u,v) \in \mathcal{C}_\alpha \bowtie \mathcal{C}_\beta} A_{(u,v)}}{|\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta|}, \; d_{(\mathcal{U},\mathcal{V})} = \frac{\sum_{(u,v) \in \mathcal{C}_\alpha \bowtie \mathcal{C}_\beta} d_{(u,v)}}{|\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta|}$$

5.2 ODP-PS: ODP on a Pruned Space

ODP on a Pruned Space (ODP-PS) computes the operator placement as presented in Algorithm 2. First, it selects a subset of computing resources that can possibly host the application operators (line 4). Then, it executes ODP only on the candidate resources so to determine the operators placement (line 8). If there is no feasible solution, the heuristic tries to expand this set before solving again ODP (line 12).

To identify a set of resources that can be quickly expanded as needed (line 4), the heuristic logically groups resources in a hierarchy of sets (see Figure 2). The smallest sets contain pairs of resources that minimize the penalty function $\delta(u,v)$. These sets are pairwise combined in larger sets, so that the summation of the penalty function over every pair of resources within the same set is minimized. According to this representation, referred as H_{res}, a set contains all the computing resources of its inner subsets.

To define the application placement (line 8), ODP-PS first solves ODP on the smallest set of resources T_{res} that host the pinned operators (i.e., data sources and sinks). If the placement is not found, the heuristic considers the smallest set in H_{res} that strictly contains T_{res} (line 12) and solves again ODP (line 8). The heuristic terminates when the placement is found or when the infrastructure does not contain enough computing resources (line 9). Differently from Hierarchical-ODP, ODP-PS computes the application placement on sets of computing resources, whose cardinality can increase, in the worst case, up to the whole infrastructure size.

5.3 RES-ODP: Relax, Expand, and Solve ODP

To identify a limited set of candidate resources for ODP, *Relax, Expand, and Solve ODP* (for short, RES-ODP) uses the linear relaxation of the ILP formulation of ODP. Linear relaxation is a consolidate technique in operational research [45].

As summarized in Algorithm 3, RES-ODP proceeds in three steps. First, it solves ODP by relaxing the integrality constraint for the placement variables $\boldsymbol{x}$ (line 5). Then, it uses the computed placement solution to identify an initial set of candidates nodes (referred as $\mathcal{G}_{res}$ in line 6). It may happen that the relaxed placement assigns an operator to multiple computing resources, i.e., for $i \in V_{dsp}, \exists u, v \in V_{res}, u \neq v$ such that $x_{i,u}, x_{i,v} > 0$. In such a case, RES-ODP selects one of them as candidate node with a uniform probability. Afterwards, RES-ODP expands the set of candidates by adding k neighbors for each candidate (line 7). The best neighbors are identified using the link penalty function and, to increase diversification, they are selected in a probabilistic manner: the lower the link penalty, the higher the probability of being selected. We extend the set of candidate resources, because the linear relaxation of ODP can use a reduced number of resources: indeed, by neglecting the integrality constraints, it can assign fractions of operators to resources, thus reducing resource wastage and fragmentation. Nevertheless, in our setting, a placement solution cannot fragment the operators. Finally, RES-ODP determines the application placement by solving ODP (with integrality constraints) on the extended set of candidate resources (line 8).

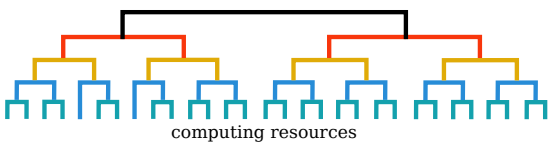


Fig. 2: ODP-PS groups the computing resources in subsets, so that the penalty function within each set is minimized.

Algorithm 4 Local Search

```
1: function LocalSearch(G_dsp, G_res)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     P ← resources hosting the pinned operators of G_dsp
5:     L ← resources of G_res, sorted by the cumulative
6:         link penalty with respect to nodes in P
7:     S ← solve GreedyFirst-fit(G_dsp, L)
8:     do                                          ▷ local search
9:         F ← value of the objective function for S
10:        S ← improve S by co-locating operators
11:        S ← improve S by swapping resources
12:        S ← improve S by relocating a single operator
13:        F' ← value of the objective function for S
14:    while F' < F                ▷ placement solution is improved
15:    return S
16: end function
```

Algorithm 5 Tabu Search

```
1: function TabuSearch(G_dsp, G_res)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     S* ← NOT_DEFINED                          ▷ best placement
5:     F* ← ∞
6:     S' ← LocalSearch(G_dsp, G_res)            ▷ local optimum
7:     F' ← objective function value for S'
8:     S ← S'
9:     tl ← create new tabu list and append S
10:    do
11:        improvement ← false
12:        S ← local search from S, excluding solutions in tl
13:        F ← objective function value for S
14:        if F = F* and S ∉ tl then tl.append(S) end if
15:        if F < F* and S ∉ tl then
16:            S* ← S; F* ← F
17:            tl.append(S)
18:            improvement ← true
19:        end if
20:        limit tl to the latest tl_max placement configurations
21:    while (improvement)
22:    if F' < F* then S* ← S' end if
23:    return S*
24: end function
```

6 Model-free Heuristics

In this section, we present in detail the model-free heuristics, namely Greedy First-fit, Local Search, and Tabu Search.

6.1 Greedy First-fit

The *Greedy First-fit* heuristic determines the placement solution using a greedy approach [45]. For each DSP operator, this heuristic selects the computing resource from a sorted list L in a first-fit manner. Specifically, let P be the set of computing nodes that host the pinned operators (e.g., data sources, sinks). The heuristic adds to a list L the available resources $u \in V_{res}$, and sorts them in ascending order of overall link penalty; the latter is the summation of the link penalty between the node u in L and every resource in P, i.e., $\sum_{v \in P} \delta(u, v)$. Then, by navigating G_{dsp} in a breadth-first manner, the heuristic defines the placement of every DSP operator on the computing resources, which are selected from L in a first-fit manner.

6.2 Local Search

Local Search explores the solution space by moving from a placement configuration to the next one using a greedy approach. We summarize its behavior in Algorithm 4.

As first step, the heuristic creates L, a list of computing resources sorted in ascending order of cumulative penalty with respect to nodes hosting the pinned operators (see Section 6.1). Then, the heuristic computes the initial application placement using Greedy First-fit (line 7) and starts the local search. Specifically, it iterates to discover new placement configurations with lower value of the objective function F, until no further improvement can be achieved (lines 8–14). At each iteration, neighbor configurations of the current placement are explored, and the best one is chosen as current configuration. Three exploration strategies are used, namely co-locate operators (line 10), swap resources (line 11), and move single operator (line 12).

Co-locate operators tries to assign two communicating operators on the same computing resource. Considering the initial configuration where $i \in V_{dsp}$ runs on $u \in V_{res}$ and $j \in V_{dsp}$ on $v \in V_{res}$, this strategy tries to co-locate i and j either on u or on v. *Swap resources* replaces an active computing resource u with a new one v from L; as a consequence, all the operators hosted on u are relocated on v. *Move single operator* relocates a single operator i from its location u to a new computing resource v, selected from L. Differently from the previous strategy, all other operators in u are not relocated. Observe that we run the local search until no further improvement can be found (line 14); however, a more stringent stopping condition can be used so as to limit the resolution time.

6.3 Tabu Search

The drawback of methods with local improvements (i.e., Greedy First-fit, Local Search) is that they might only find local optima, which nevertheless depend on the initial configuration, and miss the identification of global optima. Tabu Search increases the chances of finding a global optimum by moving, if needed, through non-improving placement configurations.

Algorithm 5 describes Tabu Search. It starts from an initial placement configuration, which is determined using Greedy First-fit. Then, it computes the neighbor configurations using the exploration strategies presented in Section 6.2 (i.e., *co-locate operators*, *swap resources*, and *move single operator*) and accepts the best improving placement (line 6). As soon as a local optimum is found, the heuristic continues to explore the search space by selecting the best non-improving configuration found in the neighborhood of the local optimum (line 12). This process increases the possibility of escaping from the local optimum and finding a new configuration that further decreases the objective function F (lines 15 and 22). To improve exploration and avoid cycles, the heuristic uses a tabu list (referred as tl), which contains the latest $tl_{\max}$ visited solutions which cannot be further explored. The heuristic terminates when no further improvement can be achieved (line 21). When the tabu search ends, the heuristic returns the overall best solution (line 22). Similarly to Local Search, although we run

TABLE 1: Parameters of the experimental setup.

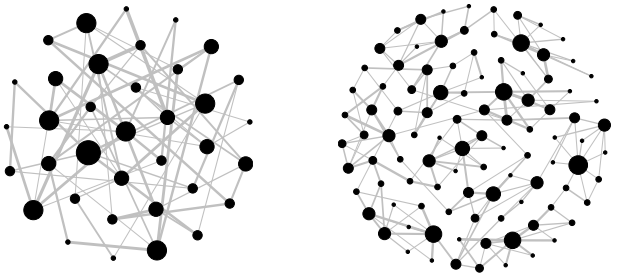
Infrastructure		Application	
$\lvert V_{res}\rvert$	$\{36, 49, 64, 81, 100\}$	$\lvert V_{dsp}\rvert$	20
A_u	$\mathcal{U}(97\%, 99.99999\%)$	C_i	1
C_u	2	R_i	3 ms
S_u	1.0	$\lambda_{(i,j)}$	100 tuples/s
$A_{(u,v)}$	100%		
avg $d_{(u,v)}$	17 ms		

BRITE's parameters used to generate the infrastructure network

Resource	Random Graph	Latency Space
AS	Waxman (α: 0.15; β 0.20)	HS: 1000; LS: 100
Routers in AS	Waxman (α: 0.15; β 0.20)	HS: 10000; LS: 1000

Normalization factors for the ODP objective function

Diamond Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	74 ms	410 ms	132.2 KB	1409.2 KB
Sequential Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	114 ms	3098 ms	8.4 KB	303.8 KB
Replicated Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	49 ms	247 ms	52.0 KB	446.4 KB



(a) Smallest network, $|V_{res}| = 36$ (b) Largest network, $|V_{res}| = 100$

Fig. 3: BRITE-generated reference networks. The size of nodes and links is proportional to their connectivity degree and network delay, respectively.

the heuristic until no further improvement can be found, a more stringent stopping condition can be used.

7 Experimental Results

Relying on ODP as benchmark, we evaluate the efficacy and efficiency of the proposed heuristics under different utilization scenarios. We describe the numerical experimental setup in Section 7.1. In Section 7.2, we analyze the heuristics performance for different application topologies and computing infrastructures (a more detailed analysis is in Appendix C.1). In Section 7.3, we briefly discuss the impact of different objective functions on the heuristics performance (further details in Appendix C.2). In Section 7.4, we summarize the results and identify the heuristic that achieves, on average, a good trade-off between resolution time and quality of the placement solution. Finally, in Section 7.5, we evaluate the heuristics using a Storm-based prototype.

7.1 Experimental Setup

We run the experiments on a virtual machine with 4 vCPU and 8 GB RAM; CPLEX© (version 12.6.3), the state-of-the-art solver for ILP problems, is used to resolve ODP.

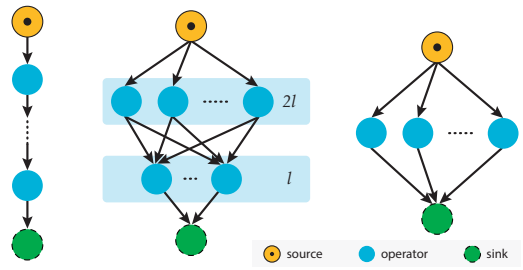


Fig. 4: Sequential, replicated, and diamond applications.

We consider several geographically distributed infrastructures, where computing nodes are interconnected with non-negligible network delays. We use BRITE [46] to generate infrastructures with $n^2 = \{36, 49, 64, 81, 100\}$ computing nodes, where the latter are interconnected in a two-layered top-down network: in the top-level, n autonomous systems (AS) communicate with high-speed links, whereas, within each AS, routers use slower links[4]. Each level is generated as a Waxman random graph. The QoS attributes of computing and network resources, together with the random graph generation parameters, are summarized in Table 1. Figure 3 proposes a graphical representation of the smallest and the largest computing infrastructure. The size of nodes is proportional to their connectivity degree, whereas the size of (physical) links is proportional to their network delay. As presented in Section 3, a *logical link* $(u, v) \in E_{res}$ between any $u, v \in V_{res}$ always exists.

We consider three alternatives of layered topology, representing *sequential*, *replicated*, and *diamond* DSP applications, where each layer has one or more operators. The first and last layer contain the sources and sinks of the application, respectively. The DAGs of sequential, replicated, and diamond applications are shown in Figure 4. The replicated application has one operator in the first and in the last layer, $2l$ operators in the second layer, and l in the third one. All the topologies contain the same number of operators. We assume that source and sink are pinned on the same node.

The baseline scenario involves applications, computing and network resources with homogeneous characteristics. This represents the worst-case scenario for the CPLEX solver that, using a branch-and-cut resolution strategy, has to explore the whole solution space in order to find and certificate the optimum. Applications and resources are parametrized as reported in Table 1. The latter also reports the normalization factors used by ODP, which have been computed using ODP with different optimization objectives[5].

Together with the model-based and model-free heuristics presented in Sections 5 and 6, we consider two additional baseline approaches: ODP+T and Greedy First-fit (no δ). *ODP+T* limits the time interval granted to CPLEX for solving ODP through a timeout, that we set to 300 s: if the

4. In BRITE, HS and LS specify the dimensions of the plane that will contain the generated topology. The plane is a square with side HS and it is internally subdivided into smaller squares with side LS.

5. Different normalization factors should be used for each combination of application and network topologies. However, in our experimental setting, the different network topologies have a limited impact on the value of normalization factors; therefore, we only consider different normalization factors for the different application topologies.

TABLE 2: Heuristics comparison when the application response time is minimized. For each heuristic, we report the resolution time speed-up (sp) and the performance degradation (pd), for diamond (DA), sequential (SA), and replicated (RA) applications. Each column reports the average value of performance metrics obtained by considering the different size of computing infrastructure. The last column reports the metrics obtained by considering all the evaluated scenarios.

Policy		**DA**	**SA**	**RA**	**Overall**
ODP	rt 36 (s)	0.1	41.4	915.2	**Average**
	rt 100 (s)	0.8	2174.8	32193.9	**Value**
Hierarchical	sp	4.40	448.39	602.77	266.30
ODP	pd	17%	3%	11%	10%
ODP-PS	sp	3.45	127.17	104.71	121.17
	pd	7%	0%	2%	2%
RES-ODP	sp	2.93	6.77	73.80	11.10
	pd	0%	0%	0%	0%
Local Search	sp	0.68	150.54	353.07	215.81
	pd	0%	1%	4%	1%
Tabu Search	sp	0.31	65.91	64.93	83.53
	pd	0%	1%	4%	1%
ODP+T	sp	1.88	2.32	40.75	38.08
	pd	0%	0%	49%	6%
Greedy First-fit	sp	454.40	$56 \cdot 10^4$	$12 \cdot 10^6$	$11 \cdot 10^6$
	pd	0%	7%	5%	11%
Greedy First-fit	sp	454.40	$56 \cdot 10^4$	$12 \cdot 10^6$	$11 \cdot 10^6$
(no δ)	pd	34%	7%	24%	19%

optimal solution has not been identified within this time interval, ODP+T returns the best solution it has computed. *Greedy First-fit (no δ)* determines the placement by assigning DSP operators to the computing resources using a first-fit approach; differently from Greedy First-fit, this heuristic does not rely on the penalty function δ to sort resources in L (as other solutions in literature usually do, e.g., [4], [20]).

We parametrize the different heuristics after preliminary experiments, aimed to minimize the quality degradation of the computed placement solutions: Hierarchical ODP uses a grouping factor $g = 2$; RES-ODP includes at most $k = 5$ neighbors for each candidate node; and Tabu Search limits the tabu list to $tl_{\max} = 1000$ configurations. Furthermore, every heuristic uses a timeout on the resolution time equal to 24 hours (value defined for practical reasons).

We compare the proposed heuristics against ODP in terms of resolution time and performance degradation. The *resolution time* (*rt*) represents the time needed to compute the placement solution. We define the *speed-up* (*sp*) as the ratio between the resolution time of ODP and that of the heuristic h, i.e., $sp_h = rt_{\text{ODP}}/rt_h$. To quantify how far is the placement solution by the heuristic h to the optimal placement, we define the *performance degradation* as $pd_h = (F_h - F_{\text{ODP}})/(1 - F_{\text{ODP}})$, where F_h is the objective function value by h and F_{ODP} is the optimal value by ODP. By definition, F_h ranges between the best value F_{ODP} and the worst value 1; in turn, pd_h ranges between the best value 0 and the worst value 1. Table 2 summarizes the experimental results that we will discuss in this section. We have evaluated 540 different configurations, each of which has been executed 5 times.

7.2 Application Topologies and Network Size

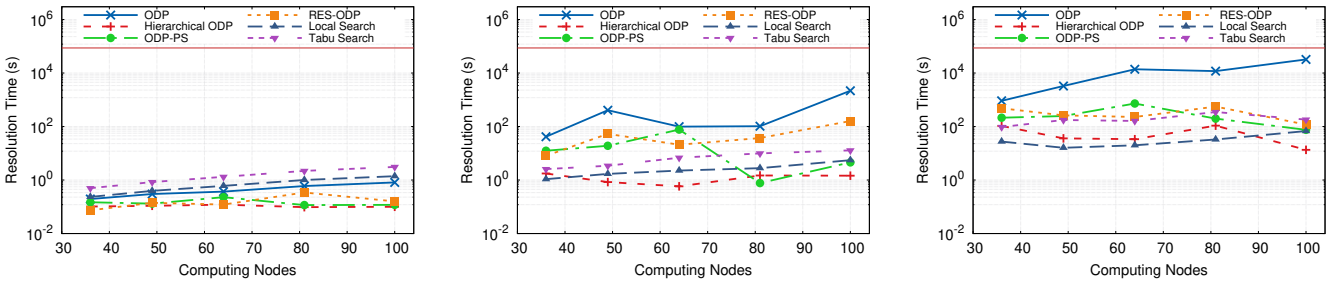
In this experiment, we compare the performance of the heuristics against ODP, when the optimization objective is the minimization of the application response time R. This corresponds to set the weights $w_r = 1$, $w_a = w_z = 0$ to the objective function F (Equation 4). We evaluate the heuristics for different application topologies and different size of the computing infrastructure. Figure 5 reports the heuristic resolution time as the number of computing resources increases, and, as a horizontal red line, the timeout (set at 24 h). For sake of clarity, we do not represent ODP+T, whose resolution time is like the one by ODP limited to 300 s, and the Greedy First-fit approaches, whose resolution time is at most 1 ms. By aggregating performance over the different configurations of infrastructure size, Figure 6 shows the average speed-up on resolution time and the average performance degradation of the heuristics for the different application topologies. Due to space limitations, we here discuss only the main outcomes of these experiments and postpone their detailed analysis to Appendix C.1.

In these experiments, the heuristics achieved different trade-offs between speed-up and performance degradation. ODP+T turned out not to be a good resolution approach: in case of replicated application, it obtained a performance degradation higher than Greedy First-fit (no δ). In most of the cases, Local Search and Tabu Search improved the placement solution with respect to Greedy First-fit. ODP-PS and RES-ODP behaved similarly to Local Search and Tabu Search, determining placement solutions with a limited performance degradation. The fastest model-based heuristic is Hierarchical ODP, which obtained, in the worst case, a rather high 17% of performance degradation (which is still lower than the one by Greedy First-fit (no δ)).

We summarize the outcomes of these experiments as follows. First, the application topology strongly influences the complexity of computing the optimal placement: a diamond application is less demanding than a sequential one, which, in turn, is less demanding than a replicated application (see Figure 5). Solving the ODP model is feasible for diamond applications (it takes at most 0.8 s); nevertheless, this does not hold true for sequential and replicated applications, which lead to an increment of resolution time up to 9 hours. Second, the infrastructure size increases the resolution time of ODP; nevertheless, working on a subset of resources, the heuristics are less prone to increase their resolution time. In these experiments, when the infrastructure size increased from 36 to 100 resources, the resolution time of ODP has grown up to 35 times, whereas the resolution time of the model-based heuristics at most up to 20 times and of the other heuristics at most up to 6 times. Third, the penalty function δ, presented in Section 4.3, helps to improve the quality of placement solutions (see Figure 6). Greedy First-fit is the fastest heuristic and, independently from the application topology, determines a placement solution in 1 ms. When the penalty function δ is used, Greedy First-fit has strongly reduced the performance degradation of the computed solutions.

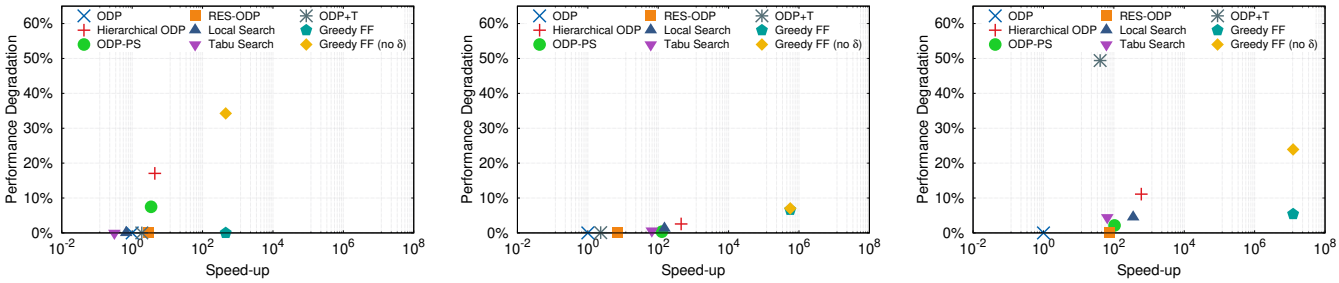
7.3 Optimization Objectives

In this set of experiments, we investigate the impact of different optimization functions on the heuristics performance. Due to space limitations, we here discuss only the main outcomes of these experiments and postpone their detailed analysis to Appendix C.2.



(a) Diamond application (b) Sequential application (c) Replicated application

Fig. 5: Computational cost of different approaches in defining the application placement with minimum response time R, when several application topologies and size of the computing infrastructure are considered.



(a) Diamond application (b) Sequential application (c) Replicated application

Fig. 6: Heuristics performance to compute the application placement that minimizes the response time R, when different application topologies are considered. Each point reports the average performance on the different infrastructure settings.

This set of experiments extends the results of Section 7.2. Specifically, we can see that, together with the application topology and infrastructure size, also the optimization goal impacts on the heuristics performance. In general, we have empirically observed that some single-objective functions can be more easily optimized (e.g., availability) than others (e.g., network usage). In the worst case, i.e., minimize the network usage for a replicated application, the resolution time of ODP grows from 169 s to 24 h, whereas the resolution time of Hierarchical ODP, the fastest heuristics excluding Greedy First-fit, grows from 3.2 s to 13 min. Interestingly, when we are interested in maximizing the application availability, the well-know meta-heuristics, i.e., Local Search and Tabu Search, are rather slow. Conversely, Greedy First-fit is tremendously fast but computes low quality solutions (up to 29% of performance degradation).

Determining an application placement that optimizes the multi-objective function (i.e., when $w_a = w_r = w_z = 0.33$ in F) represents the hardest case. Here, although Hierarchical ODP performs well in terms of speed-up and performance degradation (at most, 14%), Local Search shows even better performances (resolution time in the order of minutes and at most 3% of performance degradation).

All these experiments clearly show that it is not easy to identify *the best* heuristic: in the different scenarios, the heuristics achieve a different trade-off between speed-up and performance degradation. Nevertheless, these experiments provide us enough data to discuss which are the heuristics that, on average, show a good behavior.

7.4 Heuristics Overall Performance

In this section, we analyze the heuristics performance obtained during the whole experimental evaluation, aiming to identify their average behavior. To this end, in Table 2, we report the average values of speed-up and performance degradation obtained for each combination of infrastructure size, application topology, and optimization objective. Figure 7 also reports the distribution of these two performance indexes (i.e., speed-up and performance degradation): each boxplot reports the minimum value, the 5th percentile, the median, the 95th percentile, and the maximum value. We summarize the experimental results as follows.

The diamond application presents a topology whose complexity can be efficiently handled by ODP and the model-based heuristics. Interestingly, in this case, Tabu Search and Local Search perform poorly. With different application topologies, we need to use heuristics to efficiently deploy the application. This is especially true when complex objective functions should be optimized (e.g., multi-objective functions).

The Greedy First-fit heuristic is the fastest one, although it obtains the solution with the lowest quality (19% of average performance degradation). However, when equipped with our penalty function δ, the quality of this heuristic increases: it decreases the quality degradation of the computed placement solution from 19% to 11%, on average. These results empirically show the benefits of our penalty function δ, which is adopted also by the other heuristics.

In several configurations, ODP has a prohibitively high

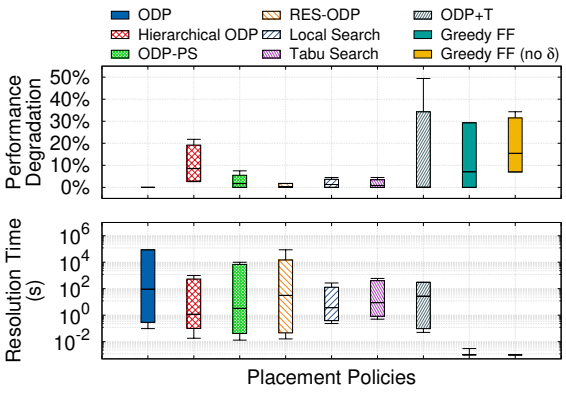


Fig. 7: Performance distribution of ODP and of the heuristics, considering their speed-up and performance degradation obtained throughout the whole experimental session.

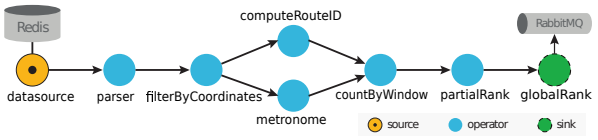


Fig. 8: Reference DSP application

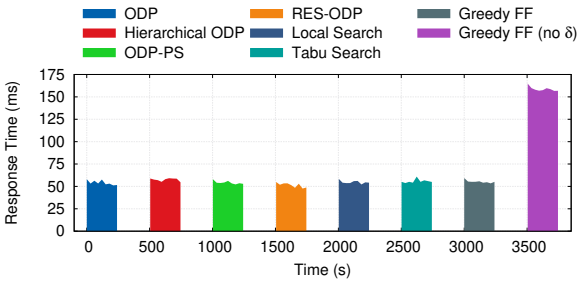


Fig. 9: Response time of the DEBS 2015 Grand Challenge application when deployed using the different heuristics.

resolution time. We have shown that the idea of applying a stringent timeout to the CPLEX solver (as ODP+T does) does not always work fine. We have shown that, in some cases, ODP+T computes low quality solutions, obtaining a performance degradation higher than Greedy First-fit (no δ). Figure 7 shows that the 95th percentile and the maximum value of performance degradation by ODP+T are the highest achieved in our experiments.

The heuristic ODP-PS, RES-ODP, Local Search, and Tabu Search have a very good trade-off between speed-up and performance degradation. The latter is always below 10%, whereas the resolution time of these heuristics is distributed on a wide range. We observe that RES-ODP shows an interesting behavior, indeed it always computes near-optimal solutions (with at most 2% of performance degradation), by reducing the resolution time of ODP by 11 times on average. Nevertheless, in the worst case, it takes several hours to compute the placement solution. As regards the other heuristics (i.e., ODP-PS, Local Search, and Tabu Search), there is not a net supremacy of one on the others. However, we consider Local Search to be the one having the best performance trade-off: it shows an average speed-up of 216 times and performance degradation of 1% with respect to ODP. In the worst case, Local Search took 269.8 s to compute the placement solution. Furthermore, this heuristic can easily be extended to further limit the resolution time by using time-based stopping criteria, e.g., by setting a timeout on the exploration phase or by limiting the number of neighbor configurations to evaluate (see Section 6.2).

We also observe that Hierarchical ODP is very fast (average speed-up of 266 and 1.2 s as median value of resolution time) and shows an average performance degradation of 10%. On average, it determines higher quality solutions with respect to those achieved by Greedy First-fit heuristics.

To conclude, we have shown that the combination of application topology, optimization function, and infrastructure size can deeply change the complexity of the placement problem to be solved. Moreover, we have shown that is not easy to identify a single heuristic that always achieves the best performance: e.g., see Local Search for diamond applications in Table 4 (Appendix C.2). By analyzing the overall behavior in our experiments, we have concluded that Local Search achieves the best performance trade-off between efficiency (i.e., speed-up) and efficacy (i.e., performance degradation).

7.5 Evaluating the Storm-based Heuristics Prototype

This section aims to evaluate the heuristics behavior when employed in a real setting. To this end, we integrate them in Distributed Storm, our extension of the popular DSP framework Apache Storm. We postpone to Appendix B the details about the heuristic integration in Storm.

We perform the experiments using 32 worker nodes and one further node for Nimbus and ZooKeeper. The worker nodes are distributed across 7 different data centers: our university cluster and 6 regions of the Google Cloud Platform (i.e., *europe-** and *us-east1*). Each worker node has 1 vCPU and 1.7 GB of RAM (as the *g1-small* instances of Google). Hence, we consider $S_u = 1, \forall u \in V_{res}$. To avoid overloading the computing resources, each worker node can host at most 2 DSP operators, i.e., $C_u = 2$. The average inter-data center network delay is 58 ms, and the intra-data center network delay is negligible (further details in Appendix D).

We use the reference application that solves a query of the DEBS 2015 Grand Challenge [47]; it finds the top-10 most frequent routes of New York taxis for the last 30 minutes. The application consists of 8 operators (see Figure 8, whose description is in Appendix E) and we feed it with sample data emitted with a rate of 10 tuples/s. This allows to run the application in steady traffic conditions, and more easily study the initial placement. We pin the data source and sink of the application on a node located in the university cluster.

We set the heuristics to minimize the application response time R. We use the same parameters of Section 7.1 and normalization factor $R_{\max} = 750$ ms. Figure 9 reports the application performance in terms of response time.

ODP identifies the optimal placement that spans the DSP application over two different data centers: the university cluster and the Google data center closest to our university (i.e., Frankfurt with an average network delay of 22 ms). The model-based heuristics efficiently exploit ODP

and successfully identify the optimal placement solution. As we can readily see from Figure 9, Hierarchical ODP, ODP-PS, and RES-ODP let the application experience the best performance with an average response time of 55 ms. The model-free heuristics that exploit the resource penalty function δ exhibit a very good behavior: in this case, Greedy First-fit, Local Search, and Tabu Search determine a best application placement, which exploits the same data centers of the optimal solution. Furthermore, Figure 9 clearly shows the benefits of the performance penalty function δ. Indeed, Greedy First-fit (no δ) identifies a sub-optimal placement that scatters the application operators over 4 different data centers. As a result, a higher response time is experienced (159 ms on average), which corresponds to a performance degradation of 15%.

7.6 Considerations on Run-time Adaptation

We conclude this section by discussing how the presented heuristics can be modified to support run-time adaptation, that is to adjust the application placement as the execution conditions change. Local Search represents the most promising approach for run-time adaptation, since it can start searching directly from the current placement. Indeed, by construction, it quickly identifies solutions with relatively small placement adjustments, thus indirectly minimizing the adaptation costs. All the other heuristics require some adjustment to be efficiently used to update the application deployment at run-time while considering the reconfiguration costs. For example, if periodically executed, the Greedy First-fit heuristics would recompute the placement from scratch; as such, it may lead a completely new application deployment with a high adaptation cost. Conversely, the model-based heuristics can be easily extended to explicitly incorporate the adaptation costs; to this end, instead of solving ODP, they should consider the elastic placement problem formulation, like the one we presented in [40].

8 Conclusion

In this paper, we presented several heuristics for computing the placement of DSP applications over geo-distributed infrastructures. Relying on a penalty function, these heuristics explicitly take into account the heterogeneity of optimization goals and computing and network resources. Some of them are built around the ODP model (i.e., Hierarchical ODP, ODP-PS, and RES-ODP), whereas others implement well-known meta-heuristics for the problem at hand (i.e., Greedy First-fit, Local Search, and Tabu Search).

We conducted a thorough evaluation by means of numerical and prototype-based experiments. We investigated the heuristics' performance under different configurations of application topology, computing infrastructure size, and deployment optimization objective. To this end, we used ODP as benchmark to determine the heuristics speed-up on resolution time and the quality of the computed placement solution (i.e., closeness to the optimal placement solution). The experimental results showed that the heuristics achieve different speed-up and performance degradation for the different combinations of application topology, infrastructure size, and optimization objective. There is not a *one-size-fits-all* heuristic, and we discussed how the different approaches behave under different deployment configurations. By aggregating the results over the evaluated configurations, we identified Local Search as the heuristic that achieves the best trade-off between speed-up and performance degradation.

As future work, we want to explore the design of more sophisticated heuristics that can combine the strengths of model-based and model-free solutions. We also plan to investigate heuristics for the run-time adaptation of placement solutions. They should support the execution of applications with desirable QoS, even in face of varying input workloads and changing environment conditions. To this end, we plan to extend our heuristics to support run-time reconfigurations taking into account their cost. We will also consider the more general setting where a geo-distributed infrastructure hosts multiple concurrent DSP applications, each arriving and departing over time with unforeseen requirements and characteristics.

References

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

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

[3] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proc. ACM DEBS '16*, 2016, pp. 69–80.

[4] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm," in *Proc. ACM DEBS '13*, 2013, pp. 207–218.

[5] N. Backman, R. Fonseca, and U. Çetintemel, "Managing parallelism for stream processing in the cloud," in *Proc. HotCDP '12*. ACM, 2012, pp. 1:1–1:5.

[6] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *Proc. 2015 IEEE Int'l Conf. on Big Data*, 2015, pp. 333–338.

[7] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos *et al.*, "Network-aware operator placement for stream-processing systems," in *Proc. IEEE ICDE '06*, 2006.

[8] S. Rizou, F. Durr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *Proc. ICCCN '10*, 2010.

[9] Q. Zhu and G. Agrawal, "Resource allocation for distributed streaming applications," in *Proc. IEEE ICPP '08*, 2008, pp. 414–421.

[10] B. Gedik, H. Özsema, and O. Öztürk, "Pipelined fission for stream programs with dynamic selectivity and partitioned state," *J. Parallel Distrib. Comput.*, vol. 96, pp. 106–120, 2016.

[11] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement of replicated tasks for distributed stream processing systems," in *Proc. ACM DEBS '10*, 2010, pp. 128–139.

[12] M. Rychly, P. Koda, and P. Mr, "Scheduling decisions in stream processing on heterogeneous clusters," in *Proc. 8th Int'l Conf. Complex, Intelligent and Software Intensive Systems*, 2014.

[13] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proc. ACM CIKM '14*, 2014, pp. 1579–1588.

[14] J. Li, A. Deshpande, and S. Khuller, "Minimizing communication cost in distributed multi-query processing," in *Proc. IEEE ICDE '09*, 2009, pp. 772–783.

[15] H. R. Arkian, A. Diyanat, and A. Pourkhalili, "MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications," *J. Parallel Distrib. Comput.*, vol. 82, pp. 152–165, 2017.

[16] I. Stanoi, G. Mihaila, T. Palpanas, and C. Lang, "WhiteWater: Distributed processing of fast streams," *IEEE Trans. Softw. Eng.*, vol. 19, no. 9, pp. 1214–1226, 2007.

[17] F. Starks, V. Goebel, S. Kristiansen, and T. Plagemann, "Mobile distributed complex event processing—Ubi sumus? Quo vadimus?" in *Mobile Big Data: A Roadmap from Models to Technologies*. Springer, 2018, pp. 147–180.

[18] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "T3-Scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster," *Future Gener. Comput. Syst.*, vol. 89, pp. 617–632, 2018.

[19] J. Ghaderi, S. Shakkottai, and R. Srikant, "Scheduling storms and streams in the cloud," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 4, pp. 14:1–14:28, 2016.

[20] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-aware online scheduling in Storm," in *Proc. IEEE ICDCS '14*, 2014, pp. 535–544.

[21] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," in *On the Move to Meaningful Internet Systems 2006*, ser. LNCS. Springer, 2006, vol. 4275, pp. 54–71.

[22] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proc. IEEE INFOCOM '16*, 2016.

[23] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "SpanEdge: Towards unifying stream processing over central and near-the-edge data centers," in *Proc. IEEE/ACM SEC '16*, 2016, pp. 168–178.

[24] L. Tian and K. M. Chandy, "Resource allocation in streaming environments," in *Proc. 7th IEEE/ACM Int'l Conf. Grid Computing*, 2006, pp. 270–277.

[25] L. Fischer, T. Scharrenbach, and A. Bernstein, "Scalable linked data stream processing via network-aware workload scheduling," in *Proc. 9th Int'l Workshop Scalable Semantic Web Knowledge Base Systems*, 2013.

[26] Z. Abrams and J. Liu, "Greedy is good: On service tree placement for in-network stream processing," in *Proc. IEEE ICDCS '06*, 2006.

[27] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-aware scheduling of streaming applications using genetic algorithm," *Procedia Computer Science*, vol. 108, pp. 2240–2249, 2017.

[28] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate latency estimation in a distributed event processing system," in *Proc. IEEE ICDE '11*, 2011, pp. 255–266.

[29] B. Peng, M. Hosseini, Z. Hong, R. Farivar *et al.*, "R-Storm: Resource-aware scheduling in Storm," in *Proc. of Middleware '15*. ACM, 2015, pp. 149–161.

[30] M. Fu, A. Agrawal, A. Floratou, B. Graham *et al.*, "Twitter Heron: Towards extensible streaming engines," in *Proc. IEEE ICDE '17*, 2017, pp. 1165–1172.

[31] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag *et al.*, "The Stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[32] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu, "A general communication cost optimization framework for big data stream processing in geo-distributed data centers," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 19–29, 2016.

[33] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu, "StroMAX: Partitioning-based scheduler for real-time stream processing system," in *Proc. DASFAA '17*. Springer, 2017, pp. 269–288.

[34] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, 2014.

[35] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 572–585, 2018.

[36] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 4, 2017.

[37] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3553–3569, 2017.

[38] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong *et al.*, "MCEP: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 6:1–6:24, 2014.

[39] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch, "SQPR: Stream query planning with reuse," in *Proc. IEEE ICDE '11*, 2011, pp. 840–851.

[40] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurr. Comput.: Pract. Exper.*, vol. 30, no. 9, 2018.

[41] ——, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[42] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi *et al.*, "TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proc. ACM DEBS '18*, 2018, pp. 136–147.

[43] K. P. Yoon and C.-L. Hwang, *Multiple Attribute Decision Making: an Introduction*. Sage Pubns, 1995, vol. 104.

[44] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[45] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: A survey," in *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., 1997, pp. 46–93.

[46] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *Proc. IEEE MASCOTS '01*, 2001, pp. 346–353.

[47] Z. Jerzak and H. Ziekow, "The DEBS 2015 grand challenge," in *Proc. ACM DEBS '15*, 2015, pp. 266–268.

Matteo Nardelli is research associate at the University of Rome Tor Vergata. He received the Doctorate degree in computer science from the University of Rome Tor Vergata in 2018. His research interests are in the field of distributed computing systems, with a special emphasis on data stream processing systems. He has more than 20 publications in international conferences and journals.

Valeria Cardellini is associate professor of computer science at the University of Rome Tor Vergata. She received the Doctorate degree in computer science from the University of Rome Tor Vergata in 2001. Her research interests are in the field of distributed computing systems, with a focus on Web and Cloud systems and services. She has more than 90 publications in international conferences and journals. She is TPC co-chair of IEEE/ACM UCC 2018, has served as TPC member of conferences on performance and Web and as frequent reviewer for well-known international journals.

Vincenzo Grassi is full professor of computer science at the University of Rome Tor Vergata. His general research interests are in the field of methods and tools for performance and dependability modeling and analysis of computing and communication systems. Within this general framework, he has recently focused on mobile computing systems and on geographically distributed service-oriented systems. He has more than 80 publications in international conferences and journals and he has served and is currently serving in the program committees of conferences.

Francesco Lo Presti is associate professor of computer science at the University of Rome Tor Vergata. He received the Doctorate degree in computer science from the University of Rome Tor Vergata in 1997. His research interests include measurements, modeling and performance evaluation of computer and communications networks. He has more than 90 publications in international conferences and journals. He has served as TPC member of conferences on networking and performance areas, and as reviewer for various international journals.

Supplemental Materials

Matteo Nardelli, *Member, IEEE*, Valeria Cardellini, *Member, IEEE,* Vincenzo Grassi, and Francesco Lo Presti, *Member, IEEE*

This document includes the supplemental materials for the paper entitled "Efficient Operator Placement for Distributed Data Stream Processing Applications" (doi: 10.1109/TPDS.2019.2896115).

APPENDIX A
OPTIMAL PLACEMENT MODEL

In this appendix, we derive the expression for the QoS metrics of interest and we present the ODP problem formulation in detail. For the sake of clarity, in Table 3 we summarize the main used notation.

TABLE 3: Main notation adopted in the paper.

Symbol	Description
V_{dsp}	Set of DSP application operators
E_{dsp}	Set of DSP application streams
$\lambda_{(i,j)}$	Average data rate exchanged on $(i,j) \in E_{dsp}$
V_{res}	Set of computing nodes
E_{res}	Set of logical links
S_u	Processing speed-up of $u \in V_{res}$
A_u	Availability of node $u \in V_{res}$
$d_{(u,v)}$	Network delay on $(u,v) \in E_{res}$
$A_{(u,v)}$	Availability of $(u,v) \in E_{res}$

A.1 ODP Variables

We model the ODP problem with binary variables $x_{i,u}$, $i \in V_{dsp}$, $u \in V_{res}^i$: $x_{i,u} = 1$ if operator i is deployed on node u and $x_{i,u} = 0$ otherwise. A correct placement must deploy an operator on one and only one computing node. For the problem formulation, we also find convenient to consider binary variables associated to links, namely $y_{(i,j),(u,v)}$, $(i,j) \in E_{dsp}$, $(u,v) \in E_{res}$, which denote whether the data stream flowing from operator i to operator j traverses the network path from node u to node v. By definition, we have $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$. For short, in the following we denote by $\boldsymbol{x}$ and $\boldsymbol{y}$ the placement vectors for nodes and edges, respectively, where $\boldsymbol{x} = \langle x_{i,u} \rangle$, $\forall i \in V_{dsp}$, $\forall u \in V_{res}^i$ and $\boldsymbol{y} = \langle y_{(i,j),(u,v)} \rangle$, $\forall x_{i,u}, x_{j,v} \in \boldsymbol{x}$.

- *V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli are with University of Rome Tor Vergata, Italy. E-mails: {cardellini, nardelli}@ing.uniroma2.it, vincenzo.grassi@uniroma2.it, lopresti@info.uniroma2.it*

A.2 QoS Metrics

Response Time. For a DSP application, with data flowing from several sources to several destinations, there is no unique definition of response time. For any placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$), we consider as response time the *critical path average delay* $R(\boldsymbol{x}) = R'(\boldsymbol{x}, \boldsymbol{y})$. We define the critical path of the DSP application as the set of nodes and edges, forming a path from a data source to a sink, for which the sum of the operator computational latency and network delays is maximal. Hence, the critical path average delay is the expected traversal time of the critical path. Formally, we have:

$$R(\boldsymbol{x}) = R'(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R'_\pi(\boldsymbol{x}, \boldsymbol{y}) \tag{5}$$

where $R'_\pi(\boldsymbol{x}, \boldsymbol{y})$ is the end-to-end delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. For any path $\pi = (i_1, i_2, \ldots, i_{n_\pi}) \in \Pi_{dsp}$, where i_p and n_π denote the p^{th} operator and the number of operators in the path π, respectively, we obtain:

$$R'_\pi(\boldsymbol{x}, \boldsymbol{y}) = \sum_{p=1}^{n_\pi} R'_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi - 1} D'_{(i_p, i_{p+1})}(\boldsymbol{y}) \tag{6}$$

where

$$R'_i(\boldsymbol{x}) = \sum_{u \in V_{res}^i} \frac{R_i}{S_u} x_{i,u} \tag{7}$$

$$D'_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j} d_{(u,v)} y_{(i,j),(u,v)} \tag{8}$$

denote respectively the execution time of operator i when mapped on node u and the network delay for transferring data from i to j when mapped on the path from u to v, where $i, j \in V_{dsp}$ and $u, v \in V_{res}$.

Availability. We define the application availability A as the availability of all the nodes and paths involved in the processing and transmission of the application data streams. We have $A(\boldsymbol{x}) = A'(\boldsymbol{x}, \boldsymbol{y})$ where:

$$A'(\boldsymbol{x}, \boldsymbol{y}) = \prod_{i \in V_{dsp}} A'_i(\boldsymbol{x}) \cdot \prod_{(i,j) \in E_{dsp}} A'_{(i,j)}(\boldsymbol{y}) \tag{9}$$

where

$$A'_i(\boldsymbol{x}) = \sum_{u \in V_{res}^i} A_u x_{i,u} \tag{10}$$

$$A'_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j} A_{(u,v)} y_{(i,j),(u,v)} \tag{11}$$

denote respectively the availability of the operator $i \in V_{dsp}$ and of the data stream from i to j, $(i,j) \in E_{dsp}$. To obtain a linear expression, we consider the logarithm of the availability, obtaining:

$$\log A'(\boldsymbol{x},\boldsymbol{y}) = \sum_{i\in V_{dsp}} \sum_{u\in V_{res}^i} a_u x_{i,u} + \\ + \sum_{(i,j)\in E_{dsp}} \sum_{(u,v)\in V_{res}^i \times V_{res}^j} a_{(u,v)} y_{(i,j),(u,v)} \tag{12}$$

where $a_u = \log A_u$ and $a_{(u,v)} = \log A_{(u,v)}$. Expression (12) deserves some comments. Let us focus on the first term and observe that the logarithm of the first factor of $A'(\boldsymbol{x},\boldsymbol{y})$, that is $\prod_{i\in V_{dsp}} A'_i(\boldsymbol{x}) = \prod_{i\in V_{dsp}} (\sum_{u\in V_{res}^i} A_u x_{i,u})$, is actually $\sum_{i\in V_{dsp}} \log(\sum_{u\in V_{res}^i} A_u x_{i,u})$ and in general $\log(\sum_{u\in V_{res}^i} A_u x_{i,u}) \neq \sum_{u\in V_{res}^i} (\log A_u) x_{i,u}$. However, since only one term of the sum in the expression $\log(\sum_{u\in V_{res}^i} A_u x_{i,u})$ can be different from zero (an operator is assigned to exactly one node), it follows that only one variable in the set $\{x_{i,u}\}_{u\in V_{res}^i}$ is equal to 1. Thefore, for any application placement $\boldsymbol{x}$, $\log(\sum_{u\in V_{res}^i} A_u x_{i,u}) = \sum_{u\in V_{res}^i} (\log A_u) x_{i,u}$, from which the first term in (12) directly follows. Similar arguments apply to the second term.

Network Usage. We define the *network usage* Z as the amount of data that traverses the network at a given time. We have $Z(\boldsymbol{x}) = Z'(\boldsymbol{y})$:

$$Z'(\boldsymbol{y}) = \sum_{(i,j)\in E_{dsp}} Z'_{(i,j)}(\boldsymbol{y}) \tag{13}$$

where the stream $(i,j) \in E_{dsp}$ imposes a load expressed by:

$$Z'_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v)\in V_{res}^i \times V_{res}^j : u\neq v} \lambda_{(i,j)} d_{(u,v)} y_{(i,j),(u,v)} \tag{14}$$

where $d_{(u,v)}$ is the network delay among nodes $u, v \in V_{res}$, with $u \neq v$.

A.3 Optimal Placement Formulation

Depending on the usage scenario, a DSP placement strategy could be aimed at optimizing different, possibly conflicting, QoS attributes. Leveraging on the Simple Additive Weighting technique [43], we define the ODP optimization function $F(\boldsymbol{x}) = F'(\boldsymbol{x},\boldsymbol{y})$ as a weighted sum of the normalized application QoS attributes:

$$F'(\boldsymbol{x},\boldsymbol{y}) = w_r \frac{R'(\boldsymbol{x},\boldsymbol{y}) - R_{\min}}{R_{\max} - R_{\min}} \\ + w_a \frac{\log A_{\max} - \log A'(\boldsymbol{x},\boldsymbol{y})}{\log A_{\max} - \log A_{\min}} \\ + w_z \frac{Z'(\boldsymbol{y}) - Z_{\min}}{Z_{\max} - Z_{\min}} \tag{15}$$

where $w_r,\ w_a,\ w_z \geq 0$, $w_r + w_a + w_z = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$), $A_{\max}$ ($A_{\min}$), and $Z_{\max}$ ($Z_{\min}$) denote respectively the maximum (minimum) value for the overall expected response time, availability, and network usage. Observe that after normalization, each metric ranges in the interval $[0,1]$, where the value 0 corresponding to the best possible case and 1 to the worst case.

We formulate the ODP problem as an ILP model as follows:

$$\min_{\boldsymbol{x},\boldsymbol{y},r} F''(\boldsymbol{x},\boldsymbol{y},r)$$

subject to:

$$r \geq \sum_{p=1}^{n_\pi} \sum_{u\in V_{res}^{i_p}} \frac{R_{i_p}}{S_u} x_{i_p,u} + \\ \sum_{p=1}^{n_\pi - 1} \sum_{\substack{(u,v)\in \\ V_{res}^{i_p}\times V_{res}^{i_{p+1}}}} d_{(u,v)} y_{(i_p,i_{p+1}),(u,v)} \qquad \forall \pi \in \Pi_{dsp} \tag{16}$$

$$C_u \geq \sum_{i\in V_{dsp}} C_i x_{i,u} \qquad \forall u \in V_{res} \tag{17}$$

$$\sum_{u\in V_{res}^i} x_{i,u} = 1 \qquad \forall i \in V_{dsp} \tag{18}$$

$$x_{i,u} = \sum_{v\in V_{res}^j} y_{(i,j),(u,v)} \qquad \begin{matrix}\forall (i,j)\in E_{dsp},\\ u\in V_{res}^i\end{matrix} \tag{19}$$

$$x_{j,v} = \sum_{u\in V_{res}^i} y_{(i,j),(u,v)} \qquad \begin{matrix}\forall (i,j)\in E_{dsp},\\ v\in V_{res}^j\end{matrix} \tag{20}$$

$$x_{i,u} \in \{0,1\} \qquad \begin{matrix}\forall i\in V_{dsp},\\ u\in V_{res}^i\end{matrix}$$

$$y_{(i,j),(u,v)} \in \{0,1\} \qquad \begin{matrix}\forall (i,j)\in E_{dsp},\\ (u,v)\in V_{res}^i \times V_{res}^j\end{matrix}$$

In the problem formulation we replaced the objective function $F'(\boldsymbol{x},\boldsymbol{y})$, which is not a linear in $\boldsymbol{x}$ and $\boldsymbol{y}$ because of term $R'(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi\in\Pi_{dsp}} R'_\pi(\boldsymbol{x},\boldsymbol{y})$, with the linear function $F''(\boldsymbol{x},\boldsymbol{y},r)$. The latter is obtained from $F'(\boldsymbol{x},\boldsymbol{y})$ by replacing the response time $R'(\boldsymbol{x},\boldsymbol{y})$ with the auxiliary variable r. Indeed, if we introduce r and substitute the expressions (12) and (14) in (15), we readily obtain:

$$F''(\boldsymbol{x},\boldsymbol{y},r) = w_r \frac{r - R_{\min}}{R_{\max} - R_{\min}} \\ + w_a \frac{\log A_{\max} - \log A'(\boldsymbol{x},\boldsymbol{y})}{\log A_{\max} - \log A_{\min}} \\ + w_z \frac{Z'(\boldsymbol{y}) - Z_{\min}}{Z_{\max} - Z_{\min}} \\ = \frac{w_r}{R_{\max} - R_{\min}} r \\ + \frac{w_a}{\log A_{\min} - \log A_{\max}} \sum_{i\in V_{dsp}} \sum_{u\in V_{res}^i} a_u x_{i,u} \\ + \frac{w_a}{\log A_{\min} - \log A_{\max}} \sum_{\substack{(u,v)\in \\ V_{res}^i \times V_{res}^j \\ u\neq v}} a_{(u,v)} y_{(i,j),(u,v)} \\ + \frac{w_z}{Z_{\max} - Z_{\min}} \sum_{\substack{(u,v)\in \\ V_{res}^i \times V_{res}^j \\ u\neq v}} \lambda_{(i,j)} d_{(u,v)} y_{(i,j),(u,v)} \\ + w_r \frac{R_{\min}}{R_{\min} - R_{\max}} + w_a \frac{\log A_{\max}}{\log A_{\max} - \log A_{\min}} \\ + w_z \frac{Z_{\min}}{Z_{\min} - Z_{\max}} \tag{21, 22}$$

which is linear in in r, $\boldsymbol{x}$, and $\boldsymbol{y}$.

In the formulation, Equation (16) follows from (5)–(6). Since r must be larger or equal than the response time of

any path and, at the optimum, r is minimized, therefore $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) = R(\boldsymbol{x}, \boldsymbol{y})$. The constraint (17) limits the placement of operators on a node $u \in V_{res}$ according to its available resources. Equation (18) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V^i_{res}$. Finally, constraints (19)–(20) model the logical AND between the placement variables, that is, $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$.

APPENDIX B
STORM INTEGRATION

To use the heuristics in a real DSP framework, we have integrated them as custom schedulers for Apache Storm. We first briefly describe the main features of Storm and how it represents and executes DSP applications. Then, we present the prototype design in details.

B.1 Apache Storm

Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It manages the execution of DSP applications over a set of worker nodes interconnected in an overlay network. A *worker node* is a generic computing resource (i.e., physical or virtual machine).

In Storm, we can distinguish between an abstract application model and an execution application model. In the abstract model, a DSP application is represented by its *topology*, which is a DAG with spouts and bolts as vertices and streams as edges. A *spout* is a data source that feeds data into the system through one or more streams. A *bolt* is either a processing element, which generates new outgoing streams, or a final information consumer. A *stream* is an unbounded sequence of *tuples*, which are key-value pairs. We refer to spouts and bolts as operators.

In the execution model, Storm transforms the topology by replacing each operator with its tasks. A *task* is an instance of an application operator (i.e., spout or bolt), and it is in charge of a share of the incoming operator stream. For the execution, one or more tasks of the *same* operator are grouped into executors, implemented as threads. An *executor*, which is the smallest schedulable unit, can execute one or more tasks related to the same operator. The framework also introduces the *worker process*, that is basically a Java process acting as a container for a subset of the executors of the *same* topology. To summarize the execution model of Storm, we can say that a group of tasks runs sequentially in the executor, which is a thread within the worker process, that in its turn serves as container on the worker node.

Besides the computing resources (i.e., the worker nodes), the architecture of Storm includes two additional components: Nimbus and ZooKeeper. *Nimbus* is a centralized component that manages the topology execution. It uses its *scheduler* to define the placement of the application operators on the available worker nodes. The assignment plan determined by the scheduler is communicated to the worker nodes through *ZooKeeper*[1], that is a shared in-memory service for managing configuration information and enabling distributed coordination. Since each worker node can execute one or more worker processes, a *Supervisor* component, running on each node, starts or terminates worker processes according to the Nimbus assignments. A worker node can concurrently run a limited number of worker processes, based on the number of available *worker slots*.

B.2 Heuristics Prototype

We develop new custom schedulers for Storm, whose core are the model-based and model-free heuristics presented in Sections 5 and 6. To design these schedulers, we have to address two issues: (1) to adapt the DSP model to the specific execution entities of Storm, and (2) to instantiate the heuristics with the proper QoS information about computing and networking resources.

As regards the first issue, we have to model the fact that the Storm scheduler places the application executors on the available worker slots, considering that at most $EPS_{\max}$ executors can be co-located on the same slot. Hence, the heuristics consider $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of executors and E_{dsp} as the set of streams exchanged between the executors. In Storm, an executor is considered as a black box element; therefore, we conveniently assume unitary its attributes, i.e., $C_i = 1$ and $R_i = 1$, $\forall i \in V_{dsp}$. The resource model $G_{res} = (V_{res}, E_{res})$ must take into account that a worker node $u \in V_{res}$ offers some worker slots $WS(u)$, and each worker slot can host at most $EPS_{\max}$ executors[2]. For simplicity, the heuristics consider the amount of available resources C_u on a worker node $u \in V_{res}$ equals to the maximum number of executors it can host, i.e., $C_u = WS(u) \times EPS_{\max}$. Each worker node u is also characterized by processing speed-up S_u, which captures the node speed-up with respect to a reference processing node; this parameter can be computed by means of preliminary experiments or exploiting the equivalent compute units proposed by cloud providers (e.g., the EC2 Compute Unit (ECU) used by Amazon Web Services)[3].

As regards the second issue, Storm allows to easily develop new centralized schedulers with the pluggable scheduler APIs. However, Storm is not aware of the QoS attributes of its networking and computing resources, except for the number of available worker slots. Since we need to know these QoS attributes to execute the heuristics (and solve ODP), we rely on Distributed Storm[4], our extension of Storm [48]. It enables the QoS awareness of the scheduling system by providing intra-node (i.e., availability) and inter-node (i.e., network delay and exchanged data rate) information. This extension estimates network latencies using a network coordinate system, which is built through the Vivaldi algorithm [49], a decentralized algorithm having linear complexity with respect to the number of network locations. The heuristics retrieve, from the monitoring components of the extended Storm, the information needed to parametrize the nodes and edges in G_{dsp} and G_{res}. Specifically, it considers: the node availability ($A_u, u \in V_{res}$), and

1. http://zookeeper.apache.org/

2. The number of worker slot per worker node can be freely configured by the user; nevertheless, Storm suggests to set it as proportional to the number of CPU core available on the worker node.

3. https://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it

4. Source code available at http://bit.ly/extstorm

the network latencies ($d_{(u,v)}, \forall u, v \in V_{res}$). Further details on Distributed Storm and its monitoring components can be found in [48]. The model-based prototypes rely on CPLEX© (version 12.6.3) for solving the placement problem.

APPENDIX C
EXPERIMENTAL RESULTS

In this appendix, we present in detail the experiments concerning the impact of the application topologies and of the objective functions on the heuristics performance (respectively in Sections C.1 and C.2). Table 4, which extends Table 2, reports the heuristics performance for each evaluated configuration.

C.1 Application Topologies and Network Size

In this experiment, we compare the performance of the heuristics against ODP, when the optimization objective is the minimization of the application response time R. This corresponds to set the weights $w_r = 1$, $w_a = w_z = 0$ to the objective function F (Equation 4). We evaluate the heuristics for different application topologies (i.e., diamond, sequential, replicated) and different size of the computing infrastructure (i.e., when the number of nodes grows from 36 to 100).

Diamond Application. From Figure 5, we readily see that the application topology strongly influences the overall behavior of ODP and the heuristics. The diamond application has the lowest computational demand, which leads to a resolution time always below 10 s (see Figure 5a). Conversely, the replicated application is the most demanding one and, when ODP is used, the resolution time reaches 3×10^4 s, which is 3 orders of magnitude higher than the one experienced for the diamond application (see Figure 5c). When the diamond application has to be deployed, ODP and the model-based heuristics are very competitive and can quickly compute the placement solution in less than 1 s, even when the infrastructure includes 100 computing nodes. In this case, Local Search and Tabu Search perform worse than the others: they register a resolution time that grows up to 19 s on the largest infrastructure. Interestingly, these heuristics are slower then ODP (they obtain a speed-up factor lower than 1). Greedy First-fit, with and without the penalty function δ, computes the placement solution in 1 ms, independently from the infrastructure size.

Considering the performance degradation reported in Figure 6a, we can identify two main groups of heuristics. In the first one, we find the approaches that compute lower quality placement solutions (i.e., with not negligible performance degradation), namely ODP-PS, Hierarchial ODP, and Greedy First-fit (no δ). As expected, since Greedy First-fit (no δ) does not consider the QoS attributes of computing resources, it determines placement solutions having a very limited quality. Hierarchical ODP, the fastest model-based heuristic, shows a performance degradation of about 17%, whereas ODP-PS has a degradation of 7%. All the other heuristics identify the optimal placement. Interestingly, besides being very fast (with a speed-up of 454 times), Greedy First-fit also identifies the optimal placement solution. As a consequence, also Local Search and Tabu Search identify the optimum; nevertheless, they are even slower than ODP.

Sequential Application. When the sequential application has to be deployed on the computing infrastructure, the benefits of the heuristics are clearer: they can reduce the resolution time up to 3 orders of magnitude with respect to ODP. From Figure 5b, we can observe a non-monotonic trend on the resolution time, especially for ODP. It depends on the different network topologies that, being randomly generated, do not preserve exactly the same connectivity as the network size increases. Differently from the case of diamond applications, here all the heuristics have a resolution time smaller than the one by ODP, and they present only a limited performance degradation (see Figure 6b). More precisely, Tabu Search, ODP-PS, Local Search, and Hierarchical ODP have speed-up from 66 to 448 times, respectively, with a performance degradation always below 3% (see Table 2). Observe that, in this case, Local Search and Tabu Search are effective, because they improve the sub-optimal placement solutions identified by Greedy First-fit.

Replicated Application. The replicated application is characterized by a higher number of streams exchanged between operators; this makes the optimization function harder to minimize. In general, all the placement policies have a resolution time greater by one order of magnitude than the case of sequential application. From Figure 5c, we observe an exponential growth of the ODP resolution time when the number of computing resources increases. The model-based heuristics successfully restrict the solution space and present a resolution time that increases slowly as the infrastructure size grows. Considering the case of 100 resources, the slowest model-based heuristic, i.e., RES-ODP, has a resolution time in the order of 10^2 seconds. Local Search and Tabu Search obtain similar performance. As appears from Figure 6c, ODP-PS and RES-ODP are as fast as Tabu Search, but they compute a better placement solution (performance degradation of 2%, 0%, and 4%, respectively). In this case, Hierarchical ODP is faster than Tabu Search and Local Search with a speed-up of 603 times (instead of 353 and 65, respectively), however it has a higher performance degradation, i.e., 11%. The resolution time of ODP is prohibitively high; surprisingly, ODP+T performs very badly in this case, reporting a 49% of performance degradation. Greedy First-fit is very beneficial for replicated applications, because of its limited resolution time (1 ms); moreover, the penalty function further improves this heuristic, by reducing performance degradation from 24% to 5%.

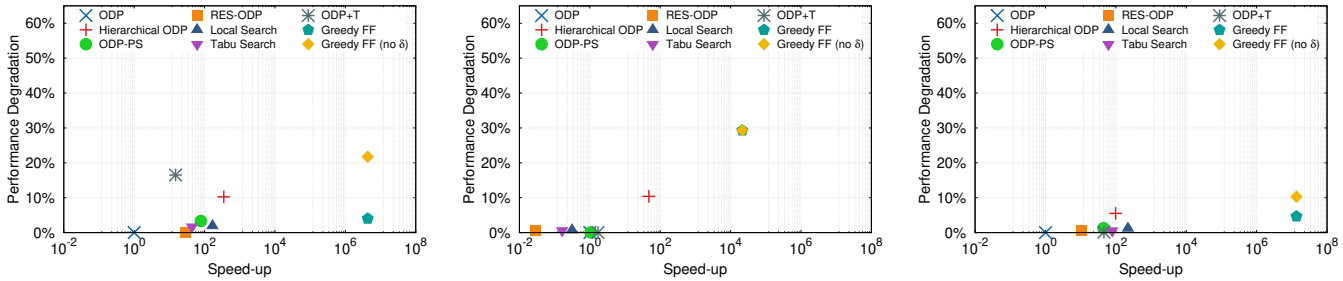
C.2 Optimization Objectives

In Section 7.2, we investigated the heuristics behavior when they minimize the application response time. In this appendix, we first consider the other single-objective optimization functions (i.e., maximization of the application availability and minimization of network usage), and then we focus on a multi-objective function. In particular, we describe the heuristics behavior aggregated by optimization objective: for each one, we consider the average value of speed-up and performance degradation which have been experienced for all the application topologies.

Response Time. In Section 7.2, we considered the minimization of the response time as optimization objective (i.e., $w_r = 1$, $w_a = w_z = 0$). Figure 10a reports the

TABLE 4: Heuristics comparison. For each heuristic, the first row reports the resolution time speed-up (sp), whereas the second one represents the performance degradation (pd). Each column reports the average value of speed-up and performance degradation obtained by considering the different size of computing infrastructure. The last column reports the average value of the performance metrics obtained by considering the performance of all the experiments.

Policy		**Diamond Application**				**Sequential Application**				**Replicated Application**				**Average**
		w_a	w_r	w_z	$*$	w_a	w_r	w_z	$*$	w_a	w_r	w_z	$*$	
ODP	rt 36 (s)	0.1	0.1	0.1	0.7	0.7	41.4	25.2	31.1	8.2	915.2	19682.8	86404.5	
	rt 100 (s)	0.7	0.8	0.7	2.6	4.9	2174.8	388.1	5225.4	168.6	32193.9	86407.0	86411.6	
Hierarchical ODP	sp	17.19	4.40	14.46	6.26	74.25	448.39	170.30	1425.05	52.21	602.77	110.65	269.64	266.30
	pd	5%	17%	9%	14%	7%	3%	3%	8%	22%	11%	4%	11%	10%
ODP-PS	sp	0.88	3.45	10.79	3.29	1.28	127.17	71.07	896.99	1.02	104.71	52.44	180.92	121.17
	pd	0%	7%	2%	4%	0%	0%	0%	3%	0%	2%	2%	2%	2%
RES-ODP	sp	0.01	2.93	4.95	1.93	0.05	6.77	9.96	6.27	0.03	73.80	16.05	10.51	11.10
	pd	2%	0%	0%	1%	0%	0%	0%	0%	0%	0%	2%	0%	0%
Local Search	sp	0.09	0.68	0.45	1.64	0.47	150.54	72.47	333.99	0.38	353.07	587.85	1088.04	215.81
	pd	2%	0%	0%	1%	0%	1%	1%	2%	0%	4%	2%	3%	1%
Tabu Search	sp	0.07	0.31	0.21	0.84	0.26	65.91	30.64	151.26	0.16	64.93	207.63	480.08	83.53
	pd	2%	0%	0%	1%	0%	1%	1%	2%	0%	4%	1%	3%	1%
ODP+T	sp	1.74	1.88	1.79	2.22	1.98	2.32	0.96	5.90	1.42	40.75	134.63	261.35	38.08
	pd	0%	0%	0%	1%	0%	0%	0%	1%	0%	49%	0%	22%	6%
Greedy First-fit	sp	354.60	454.40	338.80	1949.32	2821.80	$56 \cdot 10^4$	$19 \cdot 10^4$	$168 \cdot 10^4$	$6 \cdot 10^4$	$12 \cdot 10^6$	$40 \cdot 10^6$	$78 \cdot 10^6$	$11 \cdot 10^6$
	pd	29%	0%	0%	2%	29%	7%	7%	12%	29%	5%	7%	8%	11%
Greedy First-fit (no δ)	sp	354.60	454.40	338.80	1949.32	2821.80	$56 \cdot 10^4$	$19 \cdot 10^4$	$168 \cdot 10^4$	$6 \cdot 10^4$	$12 \cdot 10^6$	$40 \cdot 10^6$	$78 \cdot 10^6$	$11 \cdot 10^6$
	pd	29%	34%	9%	15%	29%	7%	7%	10%	29%	24%	15%	16%	19%



(a) Minimization of the response time R (b) Maximization of the availability A (c) Minimization of the network usage N

Fig. 10: Heuristics performance to compute the application placement, when different single-objective optimization functions are considered. Each point reports the average performance on different infrastructure settings and application topologies.

heuristics performance when this objective function is considered (note that, in this figure, we aggregate results on the different application topologies). Figures 10b and 10c report the heuristic performance when the placement goal is the maximization of the application availability (i.e., $w_a = 1$, $w_r = w_z = 0$) and the minimization of network usage (i.e., $w_z = 1$, $w_a = w_r = 0$), respectively. From Figure 10, we can easily observe that the different optimization objectives lead to different performance of the heuristics. The minimization of response time and network usage result in similar trade-offs between speed-up and performance degradation. However, the maximization of availability deeply changes the heuristics behavior.

Availability. In the evaluated settings, the maximization of the application availability imposes a limited computational demand. Even in the most complex configuration, i.e., ODP deploying a replicated application on an infrastructure with 100 nodes, the resolution time is at most 169 s (see Table 4). Interestingly, Local Search, Tabu Search, and RES-ODP perform worse than ODP (they have speed-up lower than 1). Being ODP very fast, ODP-PS does not significantly reduce its resolution time, obtaining a limited speed-up (see Figure 10b). We can also observe that most of the heuristics (i.e., all but Hierarchical ODP and Greedy First-fit) determine near-optimal placement solutions, introducing at most 2% of performance degradation. In particular, Table 4 reports that this is especially true for sequential and replicated applications, where the performance degradation is reduced to 0%. Similarly to the previous scenario, Hierarchical ODP has a rather high speed-up (up to 2 orders of magnitude), albeit it degrades the quality of the computed solution (10% on average). For this optimization function, Greedy First-fit is not very effective, even when it is equipped with the penalty function δ. With and without the penalty function, the performance degradation of the computed solution is close to 30%; this result clearly shows the benefit of performing a local or a tabu search so as to escape from the local optimum and improve the solution quality.

Network Usage. As we can see from Figure 10c, when network usage is optimized, the heuristics behave similarly to the case of response time minimization. Most of the heuristics have a speed-up of 2 order of magnitude and achieve a very limited performance degradation (always below 9%, except for Greedy First-fit (no δ)). As reported

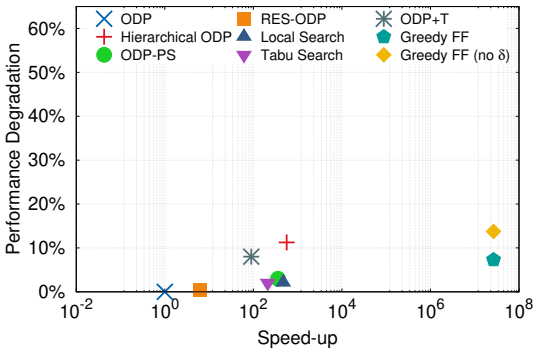


Fig. 11: Heuristics performance to compute the application placement, when a multi-objective optimization function is considered. The optimized QoS metrics are equally weighted. Each point reports the average performance on different infrastructure settings and application topologies.

in Table 4, the resolution time changes widely with respect to the application topology. In general, ODP and the other heuristics can determine the placement of the sequential and diamond applications rather quickly (at worst, ODP takes 388 s). As regards the sequential application, optimizing the network usage is apparently less computational demanding than optimizing response time. Nevertheless, when the replicated application is considered, ODP has a very high resolution time and the need of heuristics is very well motivated. More precisely, when the infrastructure contains 100 computing resources, ODP always reaches the timeout and returns the best feasible solution (i.e., it requires more than 24 h to identify the optimal solution). Albeit not the optimum, the computed placement solutions after 24 h are better than the ones by the other heuristics. As shown in Figure 10c, the Greedy First-fit heuristics are still the fastest ones, computing the placement in 1 ms. Excluding the case of sequential applications, the penalty function δ improves the placement quality by reducing the solution performance degradation from 9% to 0%, for diamond applications, and from 15% to 7%, for replicated applications. Also in this setting, Local Search and Tabu Search are beneficial for improving the solution quality; they obtain a performance degradation of 2% and 1%, respectively, for the replicated application (i.e., the most demanding one). Figure 10c also shows that most of the heuristics achieve a speed-up of 2 orders of magnitude with a very limited performance degradation. Hierarchical ODP has slightly higher performance degradation. A very good trade-off is obtained by Local Search with a speed-up of 588 times (i.e., in the worst case, it takes up to 3 minutes to compute the placement solution) and only 2% of performance degradation. ODP+T has overall good performance: with replicated applications, where the resolution time is very high, the early stop due to the timeout does not compromise the solution quality. This happens because, even though CPLEX has found the best solution within the first 300 ms, it needs to further explore the solution space so to certify the solution optimality. By taking up to 1.5 hours to compute the placement, RES-ODP is not well suited to be applied in online DSP systems.

Multi-objective Optimization. We now consider the case of multi-objective optimization function: the application requires a placement solution that minimizes response time and network usage and, at the same time, maximizes the availability. This corresponds to assign the weights $w_a = w_r = w_z = 0.33$ to the optimization function F. Figure 11 summarizes the experimental results. From Table 4 we can see that this is the most challenging scenario: the resolution time of ODP is higher than all the other configurations of objective functions. ODP takes at most 2.6 s to determine the placement of the diamond application, meaning that, in this case, ODP is very competitive (only Tabu Search has, on average, longer resolution time—its speed-up is 0.84). Conversely, for sequential applications, ODP shows its scalability issues, by requiring about 1.5 h to compute the placement. The case of replicated applications is even worse: in most of the cases (even with the 36 computing resources), ODP reaches the timeout at 24 h and does not certify the computed best solution as the optimal one. By observing Figure 11, we can classify the heuristics in four main groups, according to their performance. The first group contains RES-ODP, which has a very limited speed-up: on average, it is one order of magnitude faster than ODP (i.e., in case of replicated application, RES-ODP is prohibitively slow—it requires 2.3 h to compute the placement solution). The second group contains ODP-PS, Tabu Search, and Local Search. They achieve a very good trade-off between resolution time and solution quality, having speed-up from 2 to 3 orders of magnitude with respect to ODP and performance degradation at most of 4%. In the most challenging setting (i.e., replicated application), ODP-PS and Local Search compute the placement in 8 and 1.3 minutes, respectively. The third group comprises ODP+T and Hierarchical ODP. Their speed-up is very similar to the one by the previous group of heuristics; nevertheless, their performance degradation is slightly higher: the average performance degradation by Hierarchical ODP and ODP+T is around 10%, which, in the worst case, grows up to 14% and 22%, respectively. The fourth group comprises the Greedy First-fit heuristic, which is characterized by a very high speed-up and a rather limited performance degradation (it is always below 15%). The penalty function δ improves the computed solution quality in, basically, all the experiments. Interestingly, for sequential applications, there is an inversion of tendency and δ reduces the application quality by 2%: this is an outlier behavior, which could be caused by the computing infrastructure topology[5] or by the complexity of the objective function.

Appendix D
Prototype-based Evaluation: Details on the Experimental Setup

In Section 7.5, we execute a prototype-based evaluation of the proposed heuristics. In this appendix, we provide further details regarding the used computing infrastructure.

To perform the experiment, we use 32 worker nodes. Three of them are co-located in our university cluster placed in Rome, Italy; the remaining ones are distributed across

5. Our results present this anomaly only for the infrastructure with 49 and 64 computing nodes.

TABLE 5: Inter-data center network delays. We label our university cluster as *uniroma2*; the other regions are labeled as by the Google Cloud Platform.

Region pair	Average delay (ms)
uniroma2 ⟷ europe-west1	28
uniroma2 ⟷ europe-west2	34
uniroma2 ⟷ europe-west3	22
uniroma2 ⟷ europe-west4	28
uniroma2 ⟷ europe-north1	53
uniroma2 ⟷ us-east1	120
europe-west1 ⟷ europe-west2	7
europe-west1 ⟷ europe-west3	8
europe-west1 ⟷ europe-west4	8
europe-west1 ⟷ europe-north1	33
europe-west1 ⟷ us-east1	93
europe-west2 ⟷ europe-west3	13
europe-west2 ⟷ europe-west4	11
europe-west2 ⟷ europe-north1	38
europe-west2 ⟷ us-east1	88
europe-west3 ⟷ europe-west4	7
europe-west3 ⟷ europe-north1	32
europe-west3 ⟷ us-east1	98
europe-west4 ⟷ europe-north1	31
europe-west4 ⟷ us-east1	98
europe-north1 ⟷ us-east1	125

6 regions of the Google Cloud Platform (i.e., *europe-west1*, *europe-west2*, *europe-west3*, *europe-west4*, *europe-north1*, and *us-east1*). Each worker node has 1 vCPU and 1.7 GB of RAM (as the g1-small instances of Google). Therefore, we assign equal speed-up value S_u to each computing node, i.e., $S_u = 1$, $\forall u \in V_{res}$. To avoid overloading the computing resources, each worker node can host at most 2 DSP operators, i.e., $C_u = 2$. As regards network delay $d_{(u,v)}$, $\forall u, v \in V_{res}$, we rely on real measurements carried out on the geo-distributed computing infrastructure. Table 5 reports the (average) inter-data center network delay, as measured by the Distributed Storm monitoring system; we do not report the intra-data center network delays since they are negligible (and thus set to 0 in the optimization model).

We should observe that the inter-data center network delays play a key role in determining the operator placement solution. Indeed, in fog environments, they significantly contribute to the overall end-to-end application latency [50].

APPENDIX E
DEBS 2015 GRAND CHALLENGE APPLICATION

To evaluate the heuristics prototype in Storm, we use the reference application that solves a query of the DEBS 2015 Grand Challenge [47]. The DSP application processes data streams originated from the New York City taxis, and finds the top-10 most frequent routes during the last 30 minutes. This application includes 8 operators, which work as follows. *Data source* reads the dataset from Redis; *parser* filters out irrelevant and invalid data. Then, *filterByCoordinates* forwards only the events related to a specific area to *computeRouteID*, which identifies the routes covered by taxis. So, *countByWindow* computes the route frequency in the last 30 minutes, supported by *metronome* that defines the passing of time. Finally, *partialRank* and *globalRank* compute the top-10 most frequent routes.

REFERENCES

[48] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed QoS-aware scheduling in Storm. In *Proc. ACM DEBS '15*, pages 344–347, 2015.

[49] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[50] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. Latency-aware application module management for fog computing environments. *ACM Trans. Internet Technol.*, 19(1):9:1–9:21, November 2018.