APEROL: Adaptive Parallel Edge-to-cloud Runtime Optimization for Layered Workflow Execution

Dimitrios Banelas
Technical University of Crete
dbanelas@tuc.gr

Alkis Simitsis
Athena Research Center
alkis@athenarc.gr

Nikos Giatrakos
Technical University of Crete
ngiatrakos@tuc.gr

ABSTRACT

The execution of streaming analytics workflows across large-scale IoT infrastructures poses unique challenges. Central data collection depletes the available bandwidth and leaves IoT device resources unutilized. Therefore, workflow execution should be performed in-network, assigning workflow operator execution on devices across the cloud-to-edge continuum. However, the vast scale of devices results in an exponential number of possible combinations of workflow operator assignments. On top of that, workflows are executed on dynamic environments where volatile data stream distributions and device churn may render a deployed plan inefficient and, therefore, rapid adaptation decisions are crucial. To address these challenges, we present APEROL, the first suite of parallel optimization algorithms for timely and efficient workflow execution in IoT environments. APEROL introduces a novel conceptualization of the optimization search space, coupled with a signature-based execution plan enumeration scheme, that enable scalable, parallel plan exploration. The suite includes exhaustive, heuristic, greedy, and random sampling algorithms, which are complementary in algorithm speed vs. plan quality trade-offs under different setups. The current implementation examines up to 2M candidate plans per second on commodity hardware. Experiments with 5 challenging workflows from 2 streaming benchmarks, over real and simulated networks ranging from 10s to 1000s sites show APEROL's effectiveness and timeliness.

PVLDB Reference Format:
Dimitrios Banelas, Alkis Simitsis, and Nikos Giatrakos. APEROL: Adaptive Parallel Edge-to-cloud Runtime Optimization for Layered Workflow Execution. PVLDB, 19(2): 156 - 169, 2025.
doi:10.14778/3773749.3773755

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBanelas/aperol

1 INTRODUCTION

Applications ranging from smart cities [39] and surveillance to natural disaster forecast, have become increasingly important, as they aim to significantly improve the living conditions, well-being, and safety of large parts of the population. Such applications often operate in vastly geo-distributed settings, comprised of all kinds of IoT devices, such as sensors, single-board computers (e.g. Raspberry Pi, NVIDIA Jetson, etc.), powerful workstations, and even data-centers and clouds. These devices vary in their computational capacities and may support different types of computation (e.g. GPU processing, ability to maintain state, etc.). The traditional approach of transferring all data in the cloud and perform the processing using a Data Stream Processing Engine (DSPE) such as Apache Flink [11] or Spark [6] is, however, suboptimal in such settings. This is due to excessive network latencies for transmitting the raw data and the fact that the available computational resources of devices across the cloud-to-edge continuum are not exploited. The natural alternative would be to assign some operators of an analytics workflow to be executed on various devices in the network [13, 14, 54]. Such an approach can offload some computations in the lower layers, i.e., edge and fog, reducing transmitted data volumes, minimizing network latencies, and decreasing cloud dependencies. Efficiently utilizing the computational capacity of all types of devices, results in overall more efficient data processing across the infrastructure [13, 33, 58].

Consider a real-time environmental monitoring system deployed at a large scale to detect wildfires [45]. To make the setting concrete, assume a national authority with hundreds of watch-towers and long-range drones, jointly covering a vast forest area. Each one of the watch-towers hosts: (i) a micro-controller swarm that polls hundreds of temperature, humidity, wind-speed, and other sensors which can potentially execute simple filtering operators; (ii) a quad-core ARM CPU machine that can handle relational stream operators such as selection, projection, joins, and windowed aggregations; and (iii) a modest accelerator (e.g., an embedded GPU or FPGA) reserved for heavier analytics such as burst-detection or anomaly scoring. Intra-device transfers rely on shared memory and PCIe, adding less than 0.1 ms, whereas inter-tower LoRa/5G links fluctuate between 3 ms and 180 ms. For the reasons explained above, it would be beneficial to assign some operators for execution at the edge or fog devices of the network to extract aggregative results early and only transmit those more compact data figures towards the cloud. In that, performance in terms of throughput, processing and network latency and communication can considerably improve.

However, the conditions of the involved set up are highly volatile. Many sensors can self-adapt their sampling frequency: under normal conditions temperature probes emit one reading every few seconds, but when a rapid rise is detected they switch to a 10 Hz mode to capture the evolution of a potential ignition. A sudden blast of hot wind can therefore suddenly increase the data rate of an entire cluster of sensors by orders of magnitude. If some early aggregation operators, placed for execution on a specific tower, are not re-assigned, either split to multiple neighboring towers or to the towers' accelerators, the overload propagates and end-to-end alert latency can grow significantly with severe safety implications.

Proceedings of the VLDB Endowment, Vol. 19, No. 2 ISSN 2150-8097.
doi:10.14778/3773749.3773755

In addition, network connections may deteriorate abruptly either because normal winds bend trees obscuring line-of-sight or due to sudden smoke bursts. Further volatility arises because drones depart when batteries drain or more drones may approach to inspect the area upon an alert, instantly invalidating any static operator placement. Hence, a preferable workflow execution plan can quickly become obsolete and a new one should be provided immediately.

The problem of determining an efficient assignment of workflow operators for execution on IoT devices is challenging for the following reasons: (a) the operator placement problem is NP-hard [13, 54]. The number of possible execution plans across the IoT grows exponentially with the number of devices and workflow operators. For example, a workflow composed of 6 operators potentially placed among a moderate network of 100 towers yields a search space of candidate execution plans in the order of trillions (100^6), (b) in a streaming IoT setup, any candidate optimization algorithm should make rapid decisions since volatile data distributions and device additions or departures as those mentioned above may render a delayed execution plan suboptimal as soon as it is output. In other words, an efficient execution plan should be instantly distinguished among trillions of options, (c) workflows run for protracted periods of time and volatile data streams, network or environmental conditions necessitate adaptation of the deployed plan by migrating to a new plan for maintaining real-time performance in the long run.

Additionally, each device may be capable of hosting multiple execution platforms. The idea of multi-platform options on a single network site is well established in the literature [3, 8, 37]. For instance, a simple filter operator may be preferably executed on a device using mere JavaStreams due to the fact that temporarily low stream volumes do not worth the overheads of a job- and task-manager [57], while stream bursts later on may render the DataStream API of Flink the prominent option. Modern, commercial platforms including AWS IoT Greengrass v2 [16, 17], Azure IoT Edge [20, 21], and Google Anthos at the Edge [9, 10, 19] promote multi-platform execution patterns at the edge, routinely relying on lightweight container orchestration (e.g., Docker + K3s/Kubernetes)

Breaking up the exploration of the search space to independent, parallel tasks can tremendously decrease the runtime of the optimization algorithm itself, examining better execution plans faster. Surprisingly, no prior approach [12–14, 27, 44, 47, 53, 54] proposes parallel or even parallelizable optimization algorithms. Additionally, since the need for adapting a deployed execution plan is the norm, runtime adaptation should be embodied in the very design of an optimization algorithm. Only DAG* [54] examines plan adaptation, but our study showed that it cannot scale to large networks.

To overcome these limitations, we introduce APEROL (Adaptive Parallel Edge-to-Cloud Runtime Optimization for Layered workflow execution). Our contributions are:

- We present the first inherently parallel and scalable IoT optimization algorithms to explore valid plans concurrently across edge-to-cloud deployments of any size and workflow.
- We introduce a novel conceptualization of the search space and a new signature-based plan enumeration scheme, enabling parallelization both for APEROL and future optimization algorithms.
- We design a suite of optimization algorithms shown to complement each other in speed vs. plan quality trade-offs.
- We present an elaborate experimental evaluation using two streaming benchmarks [36, 49] composed of 5 challenging workflows, across more than 10 different network setups ranging from 10s to 1000s of devices, with both homogeneous and heterogeneous device capacity in both simulated IoT environments and the real FIT IoT Lab testbed [1]. APEROL can examine up to 1M/s possible workflow execution plans using 8 threads on a laptop and over 2M/s on servers with higher CPU capacity.
- We compare APEROL against 3 state-of-the-art algorithms [13, 14, 54]. As we show, in terms of both workflow execution plan quality and algorithm execution time, APEROL variants rank higher among all competitors.

2 SETTING UP THE OPTIMIZATION SCENE

2.1 Parallelizable Plan Exploration Basics

Consider a workflow as a *Directed Acyclical Graph* $\mathcal{G}_\mathcal{W} = (V_\mathcal{W}, E_\mathcal{W})$, where the set $V_\mathcal{W}$ contains all the operators of the workflow, and the set $E_\mathcal{W}$ contains their upstream, downstream interdependencies. Each operator can be placed in one of the $V_\mathcal{N}$ network sites (i.e. processing devices) that, along with edges $E_\mathcal{N}$ comprise the network graph $\mathcal{G}_\mathcal{N}$. Each network node $v \in V_\mathcal{N}$ can support a subset of execution platforms from a predefined set $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ (such as (Mi)NiFi, NebulaStream [58], Flink, Kafka Streams). Each node v has an associated subset $\mathcal{P}(v) \subseteq \mathcal{P}$, indicating which platforms it supports. Each edge $e_{i \longmapsto j} \in E_\mathcal{N}$ models the communication between sites i and j in the network graph. The exhaustive examination of all possible plans given $\mathcal{G}_\mathcal{W}$, $\mathcal{G}_\mathcal{N}$ and $\mathcal{P}$ yields a total of $(|\mathcal{P} \times V_\mathcal{N}|)^{|V_\mathcal{W}|}$ plans, if all operators of the workflow can be placed on all network sites and get executed on any available platform.

Placement and Plans. We define the placement function that maps workflow operators on a site and a platform:

$$\pi(o) : V_\mathcal{W} \longrightarrow V_\mathcal{W} \times V_\mathcal{N} \times \mathcal{P}(v), \quad o \mapsto (o, s, p),$$

$\pi(o)$ places an operator $o \in V_\mathcal{W}$ on a site, platform pair $(s, p) \in V_\mathcal{N} \times \mathcal{P}$. A complete plan π is a plan that can be directly deployed:

$$\pi = \{ \pi(o) \mid o \in V_\mathcal{W} \} = \{ (o, s, p) \mid o \in V_\mathcal{W} \}$$

At any given time, we have a currently deployed plan π which we term as the *root plan*.

Definition of an Action. An *action* is applied to exactly one operator and changes its assignment to a new site-platform pair. Formally, the set of possible actions is:

$$\mathcal{A} = \{(s, p) \mid s \in V_\mathcal{N},\ p \in \mathcal{P}(s)\}$$

Apply Action ($\xleftarrow{\text{act}}$): An $\xleftarrow{\text{act}}$ operation on a plan π is defined as the application of an action, single change, to an operator. During runtime adaptation, a change of operator o on a new (site, platform) pair, constitutes a single action. An action a is chosen out of the set $\mathcal{A}$ of possible actions. Let π be the root plan. The result of applying the action $a = (s, p)$ to operator o of the root plan, changes π to π':

$$\pi' = \pi \xleftarrow{\text{act}} (o, a)$$

Consecutively, by applying actions to the *root plan*, we create new candidate execution plans.

Plans that undergo a single $\xleftarrow{\text{act}}$ are referred to as *1-hop* (neighbor) plans. Similarly, plans that are formed after two $\xleftarrow{\text{act}}$ from the

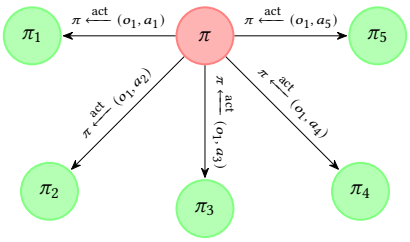

Figure 1: Graph illustrating a *1-hop* apply action alternatives on a root plan p with a single operator o after each possible action (operator o placement change).

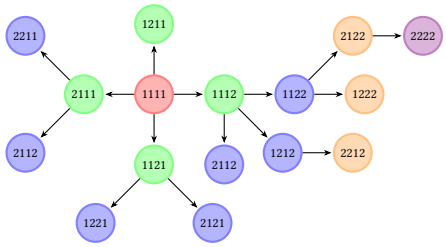

Figure 2: Graph of Plans showing the optimization search space. Each node is an execution plan π yielded from the root (red) plan by applying actions. Labels are plan signatures.

root plan are referred to as *2-hop* plans, with the pattern extending similarly for plans involving more moved operators. For instance, let us consider a simplified workflow consisting of only one operator, and actions that comprise the combinations of a single platform and five sites. Figure 1 shows how the root plan changes to producing new candidate execution plans by applying actions. Here the actions correspond to: $\{a_1, a_2, a_3, a_4, a_5\} = \{(1,1), (2,1), (3,1), (4,1), (5,1)\}$. For instance the pair $\alpha_4 = (4,1)$ corresponds to the site with an id of 4 and a platform with an id of 1. One of the resulting plans will be the root plan modeling the choice of no runtime adaptation.

The Search Space as a Graph of Plans (GoP) : As shown in Figure 1, when an action is applied to an operator of plan π, it creates a new plan, with the placement of the operator being changed to a new platform or site based on what the action dictates. The newly created plan is a direct descendant of the root plan. The search space can be modeled as a graph, with each node containing a candidate, complete execution plan, i.e., each node contains a workflow graph with a placement for each and every operator. Each edge in this structure represents the action that was applied to the parent plan, in order to generate the corresponding candidate plan, as shown in Figure 1. To improve readability, Figure 2 and henceforth, we omit the $\xleftarrow{\text{act}}$ notation. Figure 2 illustrates the search space for a workflow that consists of four operators and actions that involve the placement of the operators to a single platform and two sites in the network, i.e. the *Graph of Plans* for this setting. Note that duplicate plans are possible with the application of a series of actions in a different order starting from the same root plan. These plans are omitted from the figure for clarity. The figure illustrates that, in total, there are $2^4 = 16$ unique plans. Each possible execution plan is depicted as a node and its color indicates the number of hops (actions) required to reach this node from the root plan. Green nodes represent *1-hop* plans, which are the execution plans yielded from applying all possible actions to the root plan, one at a time. Each green node is a workflow that is produced upon applying a single unique action (change of operator placement) to the root plan. Blue nodes represent *2-hop* execution plans, i.e., plans that are *1-hop* plans of the connected green nodes and, thus *2-hop* execution plans (result by applying 2 consecutive actions) from the root plan.

Chains of Adaptation Decisions: Since streaming workflows are long running, a currently deployed root plan, should be monitored and an adaptation should happen at runtime upon performance degradation. This creates a chain of adaptation decisions, where APEROL begins from the currently deployed root plan at time t, runs some algorithm among those contained in the APEROL suite, and picks a new execution plan to deploy, which becomes the root plan at time t'. After some time, upon a new performance deficit, APEROL will again explore the search space and pick a new, better plan to deploy, which again becomes the new root plan at t'', and so on. This creates a chain of adaptation decisions, while at each decision, APEROL may choose to keep the root plan (e.g. due to high migration costs of other candidates). Each APEROL algorithm we present in this work explores the search space (Figure 2) in a different way to make an adaptation decision.

Execution Plan Cost Estimation: Each plan $\pi \in GoP$, GoP_π has an associated cost composed of performance dimensions including the communication, throughput, processing and network latency and migration cost. Inspired by [29–31, 36, 44], we define four metrics to evaluate a plan π:

- $R_o(\pi)$: operator throughput, number of tuples being processed per time unit of operator o under plan π (K tuples/s),
- $L_o(\pi)$: processing latency of operator o under plan π (ms),
- $D_e(\pi)$: edge network latency (ms) incurred when data is transferred over edge $e \in E_W$ under plan π,
- $B_e(\pi)$: bandwidth usage on edge e under plan π (Mb/s),
- $M_\tau^{root}(\pi)$: migration cost (ms) of the τ-th action in a chain of actions $\langle (o_1, a_1), \ldots, (o_T, a_T) \rangle$ to deploy plan π for given root.

Then, the four aggregate performance metrics are [31, 34, 54]:

- $Thr(\pi) = \min_{o \in V_W} R_o(\pi)$
- $Lat(\pi) = \max_{\text{source} \to \text{sink paths } P} (\sum_{o \in P} L_o(\pi) + \sum_{e \in P} D_e(\pi))$
- $Comm\_cost(\pi) = \sum_{e \in E_W} B_e(\pi)$
- $Migr\_cost(\pi, root) = \max_{\tau=1,\ldots,T} M_\tau^{root}(\pi)$

Overall Cost. The overall execution cost of a plan π, denoted $Cost(GoP_\pi)$, is computed by negating metrics of negative performance impact ($Lat(\pi)$, $Comm\_cost(\pi)$, $Migr\_cost(\pi, root)$) and computing a weighted combination of the four metrics [4, 34, 38]:

$$Cost(GoP_\pi) = w_1 \cdot Thr(\pi) + w_2 \cdot Comm\_cost(\pi) + w_3 \cdot Lat(\pi) + w_4 \cdot Migr\_cost(\pi)$$

A cost model $\mathcal{E}$ is necessary to provide estimations for the involved performance dimensions. The choice of the cost model is orthogonal to APEROL. In our experimental evaluation (Section 4), we provide the details of the specific models instantiating $\mathcal{E}$, which we derive from machine learning and parametric statistics [35].

We decided to model performance by abstracting resource constraints (e.g., node compute budgets, link bandwidth) and network topology away into aggregate throughput/latency/communication

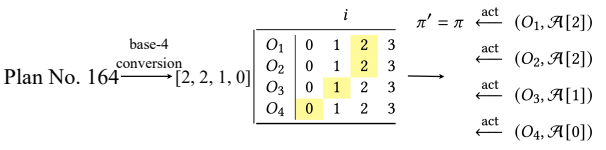

Figure 3: Base conversion and plan enumeration procedure.

terms instead of modeling them as explicit constraints. In this way, our design is general and APEROL can be utilized unaltered across IoT environments with thousands of devices with mixed and shifting capabilities. As a trade-off, search may briefly take into account plans otherwise rejected in the early stages of a more constraint-aware approach. However, in practice, as our algorithms examine up to 2M execution plans per second (see Section 4.3), this choice has little effect. In our experimental evaluation, we compare both with related works [13, 54] which employ similar, implicit constraint modeling and explicit constraint modeling [14]. We show that APEROL swiftly provides plans of better quality in terms of the performance measures involved. Enriching our techniques with declarative resource/topology constraints, would be an interesting future direction.

2.2 APEROL's Plan Signature Scheme

Consider the set of all actions $\mathcal{A}$ as an ordered array of size $|\mathcal{A}|$, where each element $\mathcal{A}[i]$ corresponds to the i^{th} available action. In an execution plan, the site and platform of each operator of the workflow has been dictated by an action of $\mathcal{A}$. We refer to the vectors containing $|V_W|$ indices pointing to $\mathcal{A}$ as index vectors. When all $|V_W|$ actions dictated by the index vector are applied to the root plan, a new (possibly multiple hops away from the root) plan occurs. The table in Figure 3 shows the action indices for each operator of a workflow with four operators, with yellow color marking the indices that comprise the index vector. Given an ordering for the operators (e.g. sorted by name or topologically sorted) we can construct a unique representation for each plan/placement, just by keeping the index vector. For instance, the placement created by the yellow cells in Figure 3 can be uniquely identified by the vector $[2, 2, 1, 0]$. This signature denotes that $\mathcal{A}[2]$ must be applied to O_1, $\mathcal{A}[2]$ to O_2, $\mathcal{A}[1]$ to O_3 and $\mathcal{A}[0]$ to O_4. This representation allows for the systematic enumeration of all possible plans, due to the fact that each number of the signature is an index in vector $\mathcal{A}$, and therefore a base-$|\mathcal{A}|$ number. Specifically, by counting from 0 to $(|\mathcal{P} \times V_N|)^{|V_W|} - 1$ and performing a base-$|\mathcal{A}|$ conversion to each resulting number, all possible operator placements can be efficiently generated. For instance, consider Figure 3. The exemplary plan numbered 164, converted to base-$|\mathcal{A}|$ would yield a plan representation of $[2, 2, 1, 0]$, which results in the previously discussed mapping. It is important to note that the enumeration procedure does not produce duplicate plans and produces all possible plans.

3 THE APEROL ALGORITHMIC SUITE

3.1 Exhaustive Search with a Queue (ESQ)

The ESQ algorithm utilizes a shared work queue to examine the search space in a BFS-like manner. The algorithm starts from the root plan and consecutively creates new plans by applying all actions to all operators. The new plans are reinserted into the queue for further action applications. To enable parallel plan examination, worker threads share a queue, using atomic constructs and synchronized blocks as needed.

Algorithm 1: Exhaustive search with queue (ESQ)

Input: GoP_{root} : Root plan
$\mathcal{A} = \mathcal{P} \times V_N$: Available actions
$\mathcal{E}$: Cost model
n: Number of threads
Output: G^* : Optimal plan

```
1  Shared (thread-safe) data structures
2      G* ← GoP_root
3      c* ← Cost(GoP_root)
4      planQueue ← ConcurrentQueue( GoP_root )
5      visited ← ConcurrentHashSet(Sig(GoP_root))
6  foreach worker ∈ SpawnWorkers(n) in parallel do
7      while planQueue ≠ ∅ do
8          GoP_parent ← planQueue.poll()
9          V_W ← V(GoP_parent)
10         foreach v ∈ V_W do
11             foreach (p, s) ∈ A do
12                 GoP_candidate ← Copy(GoP_parent)
13                 SetPlatform(v, p), SetSite(v, s)
14                 if Sig(GoP_candidate) ∉ visited then
15                     visited.add(Sig(GoP_candidate))
16                     UpdateCost(GoP_candidate, E)
17                     c ← Cost(GoP_candidate)
18                     if c < c* then
19                         c* ← Cost(GoP_candidate)
20                         G* ← GoP_candidate
21                     planQueue.offer(GoP_candidate)
22 return G*
```

Algorithm 1 provides the pseudocode for ESQ. Initially, the algorithm initializes the necessary shared variables. The atomic references G^* and c^* are initialized to point to the root plan GoP_{root} and its cost respectively (Lines 2–3). Furthermore, the queue that holds the plans during the search (*planQueue*) is initialized with GoP_{root} (Line 4), and the set of visited plans (Line 5) is populated with its signature, which serves as its unique representation (Section 2.2).

The main body of the algorithm is a while-loop executed concurrently by all worker threads (Lines 6–7). Each thread dequeues a plan denoted as GoP_{parent} (Lines 8–9). Then, it applies every action to each vertex of this plan (Lines 10–11). At each step of this procedure—i.e., applying a single action to an operator—a new plan, denoted as $GoP_{candidate}$, is generated (Lines 12–13). If the newly created plan has not been previously visited based on its signature *Sig* (Line 14), it is added to the visited set, its cost is calculated according to the provided cost model $\mathcal{E}$ (Line 16), and the plan is inserted into the queue (Line 21). Prior to this insertion step, the optimal plan and its cost, i.e. G^* and c^*, are updated based on $GoP_{candidate}$ (Lines 18–20). When the *planQueue* empties, all worker threads terminate their search process and G^* is returned.

Discussion: Since the algorithm is exhaustive, it returns the optimal plan. A careful inspection of the main body of the algorithm reveals the possibility of *collisions*. A *collision* is defined as having two threads applying a different action on different plans creating

the same plan or a plan that has been evaluated before, and therefore, all its derivative plans are already present in the queue. This motivates the use of the *visited* set, as a means of detecting these collisions. If a signature is present in the *visited* set, the corresponding plan is not inserted into the queue for further actions. Collisions significantly impact the algorithm's performance, as CPU cycles are wasted generating plans that have already been evaluated. While the *visited* set is highly effective in filtering out duplicate plans, it may introduce delays when the number of visited plans reaches hundreds of millions (due to cache misses, GC pressure, etc.). In such cases, the cost of set containment queries is non-negligible. To avoid such delays a Bloom Filter on the *visited* set can be used.

The space complexity of a single *GoP* plan is $\Theta(|V_{\mathcal{W}}| + |E_{\mathcal{W}}|)$. Also, the algorithm maintains two persistent structures: *i*) the *visited* set with $O(|\mathcal{A}|^{|V_{\mathcal{W}}|})$ space cost and *ii*) the *planQueue*, which may also contain $O(|\mathcal{A}|^{|V_{\mathcal{W}}|})$ distinct plan signatures. The queue only stores plan signatures, which means each entry takes $O(1)$ space. Each signature has to be materialized before being expanded and the total space complexity is $O(|\text{visited}|+|\text{planQueue}|+|G^*|) = O(|\mathcal{A}|^{|V_{\mathcal{W}}|} + |V_{\mathcal{W}}| + |E_{\mathcal{W}}|) = O(|\mathcal{A}|^{|V_{\mathcal{W}}|})$.

3.2 Exhaustive Search with Counting (ESC)

The ESC algorithm builds directly upon the plan signature representation and enumeration of Section 2.2.

Algorithm 3 provides the ESC pseudocode. The algorithm starts by initializing the necessary shared variables G^* and c^* to their respective initial values (Lines 1–3). The loop in Line 4 contains the core iterative process of the algorithm. It enumerates from 0 to $|\mathcal{A}|^{|V_{\mathcal{W}}|} - 1$ and constructs, in parallel, each candidate plan $GoP_{candidate}$ with the use of the ConstructPlan function, outlined in Algorithm 2. This function: *i*) converts the plan number to the calculated base and sets the $actionIndexVector$ (Line 2, ConstructPlan), *ii*) creates a copy of the original plan (Line 3–4, ConstructPlan), *iii*) applies the actions dictated by the resulting indices from the base change (Lines 6–9, ConstructPlan). Upon application of all actions, the function returns the newly constructed plan (Line 10, ConstructPlan). Back to Algorithm 3, the cost of $GoP_{candidate}$ is calculated (Lines 6–7) and the optimal plan/cost are updated. The optimal plan G^* is returned at Line 11.

Discussion: Parallel processing is achieved by many worker threads sharing a queue, and requesting plans to be inserted by a separate thread. At any given time, at most one plan per worker is materialized, resulting in a space complexity of $\Theta(|V_{\mathcal{W}}| + |E_{\mathcal{W}}|)$. Due to the counting scheme, no collisions can occur. Lines 5–10 are executed in parallel by each worker thread, which requires that variables G^* and c^* are protected by locks and atomic constructs to guarantee the correctness of the algorithm. Multiple threads accessing these two variables to update their value for a single plan may create contention degrading performance. One remedy for this situation is to introduce batching. Using the counting scheme, the range of *planIDs* can be partitioned into blocks of fixed size (i.e. a batch). The producing thread can create batches of plans, and have each worker thread process a batch of plans instead of a single plan. This allows each worker thread to first perform a local, lock-free aggregation to find the best plan in each batch, before updating the shared variables that require locking mechanisms. This reduces thread contention and drastically enhances ESC's performance (Section 4.3).

ESQ and ESC with timeouts can be used to limit runtime in large spaces. However, a key difference emerges: ESC begins from plan 0, while ESQ starts from the root. This leads to different results under time constraints due to divergent adaptation paths (Section 2.1).

Algorithm 2: ConstructPlan Function

Input: GoP_{root} : Root plan
$planID$: Unique integer identifying a plan
$\mathcal{A}$: Available actions
Output: $G_{candidate}$: The newly constructed plan

1 $b \leftarrow |\mathcal{A}|$
2 $actionIndexVector \leftarrow$ ConvertToBase($planID$, b, $|V(GoP_{root})|$)
3 $GoP_{candidate} \leftarrow$ Copy(GoP_{root})
4 $V_{\mathcal{W}} \leftarrow V(GoP_{candidate})$
5 $index \leftarrow 0$;
6 **foreach** $v \in V_{\mathcal{W}}$ **do**
7 $\quad actionIndex \leftarrow actionIndexVector[index++]$
8 $\quad (p, s) \leftarrow \mathcal{A}[actionIndex]$
9 $\quad$ SetPlatform(v, p); SetSite(v, s)
10 **return** $GoP_{candidate}$;

Algorithm 3: Exhaustive Search with Counting (ESC)

Input: GoP_{root} : Root plan
$\mathcal{A} = \mathcal{P} \times V_{\mathcal{N}}$: Available actions
$\mathcal{E}$: Cost model
Output: G^* : Optimal plan

1 **Shared (thread-safe) data structures**
2 $\quad G^* \leftarrow GoP_{root}$
3 $\quad c^* \leftarrow$ Cost(GoP_{root})
4 **for** *planID from* 0 *to* $(|\mathcal{A}|)^{|V_{\mathcal{W}}|} - 1$ ***in parallel*** **do**
5 $\quad GoP_{candidate} \leftarrow$ ConstructPlan(GoP_{root}, $planID$, $\mathcal{A}$)
6 $\quad$ UpdateCost($GoP_{candidate}$, $\mathcal{E}$)
7 $\quad c \leftarrow$ Cost($GoP_{candidate}$)
8 $\quad$ **if** $c < c^*$ **then**
9 $\quad\quad c^* \leftarrow c$;
10 $\quad\quad G^* \leftarrow GoP_{candidate}$
11 **return** G^*;

3.3 Heuristic Search (Heuristic)

APEROL's Heuristic search utilizes a Pareto set to find a good execution plan G^*, heuristically pruning parts of the search space. During its operation, the algorithm maintains a set of Pareto-optimal plans, across the cost estimation dimensions discussed in Section 2.1, and systematically prunes dominated solutions to ensure efficient search. The algorithm is based on the observation that execution plans that are so far optimal in at least one performance dimension, are more promising to make an even better plan with more action applications. Already dominated plans are less likely to give good candidates as the migration cost adds up after more actions.

Algorithm 4 provides the pseudocode for the Heuristic algorithm. The algorithm begins by initializing: *i*) $\mathcal{X}^*$ as the set of Pareto-optimal plans, initially containing only the given root plan GoP_{root} (Line 2), *ii*) *planQueue*, a queue that stores plans to be evaluated, also initialized with GoP_{root} (Line 3), *iii*) *visited*, a set storing the signatures of already explored plans to prevent redundant evaluations (Line 4) and *iv*) *removed*, an empty set that will

Algorithm 4: Pareto-Guided Heuristic Search (HEURISTIC)

Input: GoP_{root} : Root plan
$\mathcal{A} = \mathcal{P} \times V_{\mathcal{N}}$: Available actions
$\mathcal{E}$: Cost model
n : Number of threads
Output: G^* : Best plan

1 **Shared (thread-safe) data structures**
2 $\mathcal{X}^* \leftarrow \{ GoP_{\text{root}} \}$; // current Pareto frontier
3 $planQueue \leftarrow$ CONCURRENTQUEUE(GoP_{root})
4 $visited \leftarrow$ CONCURRENTSET(Sig(GoP_{root}))
5 $removed \leftarrow$ CONCURRENTSET()
6 **foreach** *worker* $\in$ SPAWNWORKERS(n) ***in parallel*** **do**
7 **while** *true* **do**
8 $GoP_{parent} \leftarrow planQueue$.POLL() ; // returns null if empty
9 **if** GoP_{parent} = null **then** // Search concludes
10 **break**
11 $V_{\mathcal{W}} \leftarrow V(GoP_{parent})$; // operators of the workflow
12 **foreach** $v \in V_{\mathcal{W}}$ **do**
13 **foreach** $(p, s) \in \mathcal{A}$ **do**
14 **if** Sig(GoP_{parent}) $\in$ *removed* **then**
15 **break**
16 $GoP_{candidate} \leftarrow$ DEEPCOPY(GoP_{parent})
17 SETPLATFORM(v, p); SETSITE(v, s)
18 **if** Sig($GoP_{candidate}$) $\in$ *visited* **then**
19 **continue**
20 *visited*.ADD(Sig($GoP_{candidate}$))
21 UPDATECOST($GoP_{candidate}, \mathcal{E}$)
22 **lock** $\mathcal{X}^*$; // Thread safe updates on the Pareto
23 **if** $\exists G' \in \mathcal{X}^* : G' \prec GoP_{candidate}$ **then**
24 **unlock** $\mathcal{X}^*$
25 **continue**
26 $dominated \leftarrow \{ G' \in \mathcal{X}^* \mid GoP_{candidate} \prec G' \}$
27 $\mathcal{X}^* \leftarrow \mathcal{X}^* \cup \{GoP_{candidate}\} \setminus dominated$
28 **unlock** $\mathcal{X}^*$
29 *removed*.ADDALL(*dominated*)
30 *planQueue*.REMOVEALL(*dominated*)
31 *planQueue*.OFFER($GoP_{candidate}$)
32 **return** $\arg\min_{G \in \mathcal{X}^*} \text{COST}(G)$

later store dominated, therefore obsolete, plans (Line 5). Multiple worker threads are utilized so as to retrieve and process plans from the *planQueue* in parallel. The main body of each thread's plan processing routine starts by dequeueing a plan from the queue, namely GoP_{parent}, and extracting its operators $V_{\mathcal{W}}$ (Lines 7–11).

The algorithm iterates over each operator v in the workflow graph (Line 12), considering all possible actions, as device-platform pairs (Line 13). Before proceeding any further, a check must be made against the *removed* set, in order to ensure that the parent plan GoP_{parent} of which all other plans will stem, has not been rendered obsolete by another thread. If GoP_{parent} has been removed from the Pareto set by another thread, there is no need to examine its derivative plans further (Lines 14–15). Performing this check inside the nested loop might seem redundant at first, but the algorithm's parallel execution means that another thread could add GoP_{parent} to the *removed* set at any moment. Detecting this as soon as possible, enables us to abort any further expansion of this plan, thus saving needless computation. As long as the parent plan is not found in the *removed* set, the algorithm proceeds by creating the candidate plan $GoP_{candidate}$ as a copy of GoP_{parent} and applies the

Algorithm 5: GREEDY Search w/ Progressive Global Optima

Input: GoP_{root} : Root plan
$\mathcal{A} = \mathcal{P} \times V_{\mathcal{N}}$: Available actions
$\mathcal{E}$: Cost model
Output: G^* : Best plan

1 **Shared (thread-safe) variables** : c^*, τ^*
2 $GoP_{\text{state}} \leftarrow$ DEEPCOPY(GoP_{root})
3 $V_{\text{rem}} \leftarrow V(GoP_{\text{root}})$
4 $G^* \leftarrow GoP_{\text{root}}$ // progressive optimum
5 **while** $V_{rem} \neq \emptyset$ **do** // until every operator is placed
6 $c^* \leftarrow +\infty$ // Best plan cost of the iteration
7 $\tau^* \leftarrow \langle \perp, \perp, \perp \rangle$ // Best operator, platform, site triple
8 **for** $v \in V_{\mathcal{W}}$ ***[in parallel]*** **do** // Operator level par/lism
9 **for** $(p, s) \in \mathcal{A}$ ***[in parallel]*** **do** // Action level par/lism
10 $GoP_{\text{tmp}} \leftarrow$ DEEPCOPY(GoP_{state})
11 SETPLATFORM(v, p); SETSITE(v, s)
12 UPDATECOST($G_{\text{tmp}}, \mathcal{E}$)
13 $c \leftarrow$ COST(G_{tmp})
14 **lock** c^*, τ^*
15 **if** $c < c^*$ **then**
16 $c^* \leftarrow c$
17 $\tau^* \leftarrow \langle v, p, s \rangle$
18 **unlock** c^*, τ^*
19 $\langle v^*, p^*, s^* \rangle \leftarrow \tau^*$
20 APPLYACTION($GoP_{\text{state}}, v^*, (p^*, s^*)$)
21 UPDATECOST($GoP_{\text{state}}, \mathcal{E}$)
22 $G^* \leftarrow GoP_{\text{state}}$
23 V_{rem}.REMOVE(v^*)
24 **return** G^*

corresponding action to the corresponding operator (Lines 16–17). At this point, another check is made to ensure that $GoP_{candidate}$ has not been examined before. Then, the signature of $GoP_{candidate}$ is added to *visited* (Line 20) and the cost model is applied to the candidate plan in order to calculate its cost (Line 21). If any existing plan in $\mathcal{X}^*$ dominates $GoP_{candidate}$, the plan is discarded as non-Pareto optimal (Lines 23–25). In the case that $GoP_{candidate}$ is not pruned, it is inserted to $\mathcal{X}^*$, while the plans dominated by it are removed (Lines 26–28). In this section of the algorithm, we explicitly show the locking of $\mathcal{X}^*$ (Lines 22 & 28), to highlight the concurrent nature of its underlying data structure and underscore that this synchronization is essential for the algorithm's correctness. Continuing, the *removed* set is updated with the newly dominated plans (Line 29) and each of these plans is also removed from the *planQueue* to avoid being retrieved by other threads (Line 30). Lastly, $GoP_{candidate}$ is inserted into *planQueue* to be further examined, by applying more actions (Line 31). The search is terminated when the queue is empty, meaning that no other plans managed to enter $\mathcal{X}^*$. The returned plan, is the one with the best cost (Section 2.1), out of $\mathcal{X}^*$ (Line 32).

Discussion: HEURISTIC's execution time may vary depending on runtime conditions such as the size of the Pareto set at any given time. For deriving the space complexity, one can easily notice that it is tightly coupled with (i) the size of the Pareto front, $|\mathcal{X}^*|$, at any given time and (ii) the number of operators and edges in a workflow $|V_{\mathcal{W}}| + |E_{\mathcal{W}}|$ which determines the space needed to materialize these $|\mathcal{X}^*|$ plans. Therefore, $O((|V_{\mathcal{W}}| + |E_{\mathcal{W}}|)|\mathcal{X}^*|)$ space complexity. According to Theorem 3 and Theorem 4 from Shang et al [48], for our search space of $|\mathcal{A}|^{|V_{\mathcal{W}}|}$ cardinality, in the worst case $|\mathcal{X}^*| = |\mathcal{A}|^{|V_{\mathcal{W}}|}$, i.e., no plan will dominate another and all plans

will stand in the Pareto front, irrespectively of the number of cost dimensions (Section 2.1). For the worst case runtime complexity of the algorithm, we consider that each of the, at most, $O(|\mathcal{A}|^{|V_\mathcal{W}|})$ points/plans will be pairwise compared with at most $O(|\mathcal{A}|^{|V_\mathcal{W}|})$ other points before getting added in the Pareto front. Therefore, the worst case runtime complexity of the HEURISTIC is $O(|\mathcal{A}|^{2|V_\mathcal{W}|})$. However, these are extreme, worst case complexities. In practice, HEURISTIC prunes not only 1-hop, but also multi-hop offsprings of plans that do not enter the Pareto front at each iteration of the algorithm and prunes entire subgraphs of the search space.

3.4 Greedy Search (GREEDY)

The GREEDY algorithm of APEROL explores the search space in a BFS fashion, but at each level of the BFS, it keeps only one (best found so far) plan for further $\xleftarrow{\text{act}}$ applications. Algorithm 5 provides GREEDY's pseudocode. GREEDY initializes the optimal plan reference, copies the provided starting plan to create the state graph GoP_{state}, and creates a list of the operators of the workflow, i.e. V_{rem} (Lines 2–4). V_{rem} tracks which operators have been modified. When it empties, the search concludes. GoP_{state} tracks the operator changes made at each level of the BFS, one, best action per level.

A while loop continues until V_{rem} becomes empty (Line 5). At each iteration the variables c^* and τ^* are reset: the former stores the best cost observed during the current iteration, the latter the operator-platform-site triplet with that best cost. Then, for every remaining operator in V_{rem} and every action in $\mathcal{A}$, the algorithm applies the action to the operator (Lines 10–11) creating a new plan implicitly on GoP_{tmp} and evaluates the cost of the said plan (Line 12–13). If this cost is better than c^*, both c^* and τ^* are updated (Lines 16–17). Updates to these variables are guarded by a lock in order to guarantee correctness when one of the surrounding loops is executed in a parallel manner (Lines 14 & 18). Once all combinations have been examined, the triple stored in τ^* is known to be the best choice discovered during the iteration (Line 19). The algorithm therefore applies the chosen action to the operator v^* of GoP_{state}, evaluates the cost of the modified plan, records the new plan as the current progressive optimum G^*, and removes v^* from V_{rem} (Lines 20–23). Because each pass permanently fixes one operator and removes it from the V_{rem}, the loop is guaranteed to finish after $|V_\mathcal{W}|$ iterations. When all operators have been placed, V_{rem} is empty, G^* is returned as the best found plan.

Discussion: A key aspect of this algorithm is that each operator is modified only once. This makes the search progressive and limits the number of examined plans. The GREEDY algorithm will examine exactly $\mathcal{T} = |\mathcal{A}| \cdot \frac{|V_\mathcal{W}| \cdot (|V_\mathcal{W}|+1)}{2}$ plans. In each iteration of the main loop (Line 5), one operator will be placed, while all actions will be applied on all available operators. Thus, the 1^{st} iteration will apply the actions to $|V_\mathcal{W}|$ operators, 2^{nd} iteration will apply actions to $|V_\mathcal{W}| - 1$ operators and so on. In detail: $\mathcal{T} = |\mathcal{A}| \cdot \left(|V_\mathcal{W}| + \sum_{i=1}^{|V_\mathcal{W}|}(|V_\mathcal{W}| - i)\right) = |\mathcal{A}| \cdot \left(|V_\mathcal{W}| + \frac{|V_\mathcal{W}|^2 - |V_\mathcal{W}|}{2}\right) = |\mathcal{A}| \cdot \frac{|V_\mathcal{W}| \cdot (|V_\mathcal{W}|+1)}{2}$. Thus, the GREEDY algorithm runs in time $O(|\mathcal{A}| \cdot |V_\mathcal{W}|^2)$, i.e., quadratic in the number of workflow operators. Regarding the space complexity of the algorithm, one can notice that the only persistent structures throughout GREEDY's execution are: V_{rem} ($O(|V_\mathcal{W}|)$) and G^*, G_{state} and G_{tmp} which all use $\Theta(|V_\mathcal{W}| + |E_\mathcal{W}|)$ space. Therefore the total space complexity is $\Theta(|V_\mathcal{W}| + |E_\mathcal{W}|)$. The algorithm can be parallelized, either on the operator level or the action level (Lines 8 & 9).

Algorithm 6: Random Sampling Search Algorithm (RSS)

Input: GoP_{root} : Root plan
$\mathcal{A} = \mathcal{P} \times V_\mathcal{N}$: Available actions
$\mathcal{E}$: Cost model
n: Sample size
Output: G^* : Best plan

```
1  Shared (thread-safe) data structures
2      G* ← GoP_root
3      c* ← Cost(GoP_root)
4      V_W ← V(GoP_root)
5      visited ← ConcurrentSet()
6  for 0 to n − 1 in parallel do
7      planID ← Random(0, |A|^|V_W|)
8      if planID ∈ visited then
9          continue;
10     visited.add(planID)
11     GoP_candidate ← ConstructPlan(GoP_root, planID, A)
12     UpdateCost(GoP_candidate, E)
13     c ← Cost(GoP_candidate)
14     if c < c* then
15         c* ← c;
16         G* ← GoP_candidate
17 return G*;
```

3.5 Random Sampling Search (RSS)

Both HEURISTIC and GREEDY prune parts of the search space that are considered unpromising. However, since they narrow down their search based on the current Pareto optimal plans or the best plan found so far, they may also get trapped to local optima. Motivated by the above observation, we propose RSS which explores the search space in a randomized fashion. The RSS algorithm is very similar in its structure with the ESC algorithm, as it also uses the plan signature and enumeration technique in order to produce candidate plans. The main difference is that, instead of enumerating all possible plans, it visits only a subset of the search space. The RSS algorithm is provided with a parameter termed n, which denotes the total number of plans that the algorithm will examine. Sample size n is usually set as a percentage of the total plans of the search space, i.e. $n = k \cdot |\mathcal{A}|^{|V_\mathcal{W}|}$, $k \in (0, 1]$, but can alternatively be specified as a numerical value (e.g. 5K, 10K plans). The main body of the algorithm closely resembles the ESC algorithm and therefore attains the same space and time complexities. Here, the plan numbers are generated randomly, using a thread safe random number generator. Since randomly generating plan numbers entails the risk of duplicate numbers, a variable to hold the generated plan numbers is necessary, i.e. the *visited* set. The algorithm begins by performing the same initialization steps as previously described for the ESC (Lines 1–5). The main loop of the algorithm (Line 6) generates n random plan numbers (Line 7) that are evaluated in parallel by the worker threads. For each plan, RSS checks whether it has been evaluated before (Line 8–9). If not, it adds it to the visited set (Line 10). Lastly, the best plan reference and cost are updated based on the recently evaluated plan (Lines 11–16).

Discussion: The algorithm has fixed execution time for given sample size and it is less susceptible to local optima. Sampling the

search space can be performed either uniformly or via Sobol sequences [52] for even search space examination. Combining virtues of Heuristic, Greedy with RSS, one can come up with hybrid algorithms. For instance, Greedy can be followed up by RSS, restricted in sampling from up to $k - hop$ neighbors of the plan suggested by Greedy, to heal Greedy's susceptibility to local optima.

4 EXPERIMENTAL EVALUATION

In our GitHub repository, we provide a detailed guide on how to run our experiments and our results are fully reproducible.

Experimental Objectives & Metrics: We compare the APEROL algorithms against two performance criteria, namely Plan Cost Improvement and Average Algorithm Execution Time per Adaptation Decision. Plan Cost Improvement corresponds to how much one or more adaptation decisions are improving the root plan as each workflow is executed, i.e., $GoP_{\pi} - GoP_{root}$. The Average Algorithm Execution Time shows how fast can each algorithm devise a new plan upon an adaptation decision.

Workload: To evaluate APEROL we employ two well known benchmarks, namely the Yahoo Benchmark [15, 36] and the RIoT benchmark [50]. The Yahoo Benchmark is a standard stream analytics benchmark composing a workflow of selection, projection, join, windowing and aggregation operators (8 operators in total).

RIoT [50] provides four real-world workflows from the smart cities domain [50] – termed TRAIN, PRED, STATS, ETL – with 8, 9, 11, 11 operators, respectively. The RIoT workflows involve a machine learning training, predictive analytics, higher order statistic extraction and extract-transform-load task, respectively.

Real Testbed: We reserve all the 194 available devices at the Grenoble site of the publicly available FIT IoT-LAB [1] testbed, consisting of Raspberry Pi 3 boards and A8 devices. The A8 board is based on an ARM Cortex-A8 micro-processor. On this Real Testbed, Raspberry Pi 3 nodes can execute functional Python code and JavaStreams and can even host a lightweight Apache Flink deployment with a single Task Manager instance. The resource-constrained A8 nodes are limited to running only functional Python or JavaStreams, due to limited hardware capabilities. We monitor setups composed of $|V_N|$ = {7, 15, 31, 127, 194} sites in the Real Testbed. This Real Testbed can only execute the Yahoo Benchmark workflow due to resource constraints, but it is used in our experiments for validating the trends seen in simulated networks (see below) and, ultimately, for comparing the performance of APEROL against DAG* [54], NEMO [13] and Governor [14] on an actual network, besides simulated setups.

For our simulated network setups of sizes $|V_N|$ = {7, 15, 31, 127, 1023, 2047} sites, we use the highly cited iFogSim tool [29, 41] to simulate RIoT workflows' execution over these networks.

Simulated Heterogeneous Network Setup: The devices of this setup consist of a mix of resource-constrained edge nodes such as ESP32-based sensors, Raspberry Pi 4, Jetson Nano boards, and high-end edge servers. Execution engines match to device capabilities: MiNiFi C++ runs on light-weight devices and MiNiFi Java runs on Raspberry Pi-class devices, while Kafka Streams and Apache Flink run on more powerful nodes like Jetson Xavier or Intel NUCs. This setup is useful to judge the effectiveness of each APEROL algorithm in cases when some actions can make an important (positive or negative) difference compared to others and, therefore, algorithms susceptible to local optima may underperform.

Simulated Homogeneous Network Setup: In this setup, all nodes are Raspberry Pi-class devices. A single runtime engine across all nodes for processing and edge-to-cloud data forwarding is deployed. Due to the homogeneous setup, algorithms (Heuristic, Greedy) that are more susceptible to local optima are less likely to underperform in the quality of the returned plan.

Given the number of operators per workflow cited previously, this means that the cardinality of the search spaces we experiment with starts from 8^7 and exceeds 11^{2047} candidate execution plans.

Instantiating the Cost Model $\mathcal{E}$. APEROL is agnostic to the chosen cost model $\mathcal{E}$. In our evaluation we instantiate $\mathcal{E}$ (Section 2.1) both with supervised ML models and parametric statistics via large-scale simulations and actual jobs running on our Real Testbed. For each network setup and admissible workflow we run 100,000 placements (chosen randomly) on the largest network. We augmented these with 20,000 unseen placements from all remaining networks, for testing the developed models. Each placement yields estimations for the four performance dimensions of every plan π: Thr, Lat, $Comm\_cost$, $Migr\_cost$.

For the homogeneous network setup we found that the ML models, detailed hereafter, provide more accurate estimations of the ground truth performance, while for the heterogeneous setup and the Real Testbed, ground truth was more accurately described by parametric models explained afterwards.

Learning Models. For the homogeneous setup, we train XGBoost regressors per (workflow, network setup) pair, one for each performance dimension. The hyperparameters of the regressors are included in our Github repo directory `costs/models/`. Feature vectors include operator-level statistics (such as throughput, processing latency and migration cost per operator, device etc.) and network characteristics (such as hop counts, link latencies).

Distribution fitting. For the heterogeneous setup and the Real Testbed, plan costs are better modeled by parametric distributions, with the best fitting distribution among {Normal, Rayleigh, Gamma, LogNormal} chosen by maximum likelihood estimation.

Runtime cost model use. Given a candidate plan π (and the root plan), $\mathcal{E}$ (either ML-based or parametric as we describe above) returns estimations per operator and network link by querying the corresponding models. These are aggregated, as detailed in Section 2.1, to compute Thr, Lat, $Comm\_cost$, $Migr\_cost$ which, in turn, are used to compute $Cost(GoP_{\pi})$. For the total cost estimation $Cost(GoP_{\pi})$ of Section 2.1, we assign equal weight to each performance dimension, but Section 4.4 performs a sensitivity analysis on weights on performance dimensions. All model artifacts are released with our code (directory `/costs`) to ensure reproducibility.

Algorithmic Setup & Hardware: The APEROL algorithms run with a default parallelism of 4, which is a sweet spot (see Section 4.3, where we also vary parallelism). We run all our experiments on a MacBook Air M2 (8 Cores/16GB RAM), but in Section 4.3, we also report on APEROL's performance on a server Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz, 20 cores/ 40 threads, and 256Gb RAM. We set a timeout for ESQ and ESC equal to the average execution time of RSS, Heuristic, Greedy. After the timeout, ESQ and ESC output the best plan found so far. For RSS we set a default sample size of 3K execution plans and we perform Sobol sequence-based sampling. We term this version as RSS-3. For Greedy, the introduction of

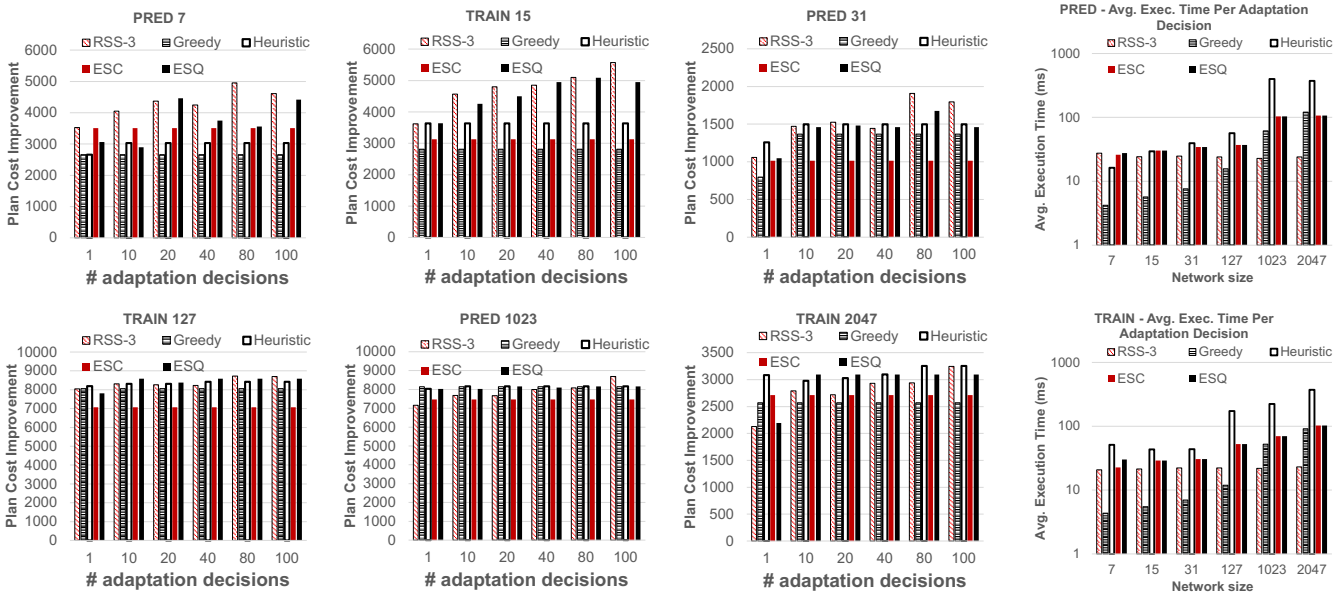

Figure 4: Heterogeneous Setup - PRED, TRAIN Workflows: APEROL Plan Cost Improvement (the higher the better) across Network Sizes vs Number of Adaptation Decisions (Less to More Volatile Conditions).

Figure 5: PRED & TRAIN: Average time per adaptation decision.

thread safe data structures and synchronization mechanisms results in overheads, which deteriorate the otherwise minimal Algorithm Execution Time. Therefore, we set the parallelism of Greedy to 1.

Main findings: ① The APEROL algorithms are complementary in their utility in different settings ② In heterogeneous setups, for networks up to 31 or 127 devices, RSS provides the best Plan Cost Improvement vs Algorithm Execution Time trade-offs ③ For heterogeneous networks of more than 127 sites, the time-capped version of ESQ is the most preferable candidate ④ In homogeneous setups, Greedy is less affected by local optima and provides the best Plan Cost Improvement vs Algorithm Execution Time trade-off ⑤ Experiments on the Real Testbed validate the trends observed in simulated setups, upon comparing APEROL vs [13, 14, 54] ⑥ In latency-based optimization vs competitors [13, 14, 54], Heuristic and Greedy clearly overtake the third Governor, with Heuristic providing 3× better Plan Cost vs Algorithm Execution Time trade-off ⑦ Hybrid RSS+Greedy often remedies local optima providing better plans compared to individual RSS, Greedy ⑧ ESC is the least dependent on shared variables and by imposing batching (Section 3.2), it can scale up to 2.1M plans per second on commodity hardware ⑨ Heuristic is the least sensitive on cost dimension weightings.

4.1 Evaluation on Simulated Setups

Heterogeneous Setup: The plots in Figure 4 illustrate the Plan Cost Improvement of the plan devised by each APEROL algorithm, across networks of various sizes, for PRED and TRAIN workflows in the heterogeneous setup. The trends for ETL and STATS are similar; we omit them due to space constraints. The vertical axes in the figures represent the Plan Cost Improvement of the output plan for each algorithm compared to the root plan (the higher the better). The horizontal axes represent the number of adaptation decisions within the lifetime of the workflow. In that, we provide results across networks and workflows which show conditions of various volatilities. Under rarely (respectively frequently) changing conditions, the number of adaptation decisions (i.e., the length of the chains of adaptation decisions discussed in Section 2.1) is low (respectively high). We couple the description of these figures with Figure 5, which shows the Average Execution Time of each algorithm per optimization decision. Hence, we can judge the plan quality vs algorithm execution time trade-offs of APEROL.

For networks up to 31 sites, RSS is the best option because it consistently provides the highest Plan Cost Improvement across various numbers of optimization decisions (Figure 4) and simultaneously provides an average algorithm execution time of 30ms (Figure 5). For networks of 127 and up to 1023 sites, Heuristic progressively overtakes RSS and for 2047 sites Heuristic mostly provides the best Plan Cost Improvement.

In the majority of the cited cases in Figure 4, ESC provides a cost improvement comparable to or worse than Greedy. Upon an adaptation decision, ESC starts its exploration from plan 0 (Section 3.2). It can therefore be more detached from the current situation of the deployed execution plan. ESQ is the second best alternative across network sizes in Plan Cost Improvement and, combined with its capped execution time in Figure 5, it is the best choice for networks of thousands of sites across the number of adaptation decisions.

Greedy exhibits important Plan Cost Improvement between network sizes of 127 and 1023, but (a) it gives an improvement that comparable RSS and (b) it is in those networks where its execution time (Figure 5) starts to match the execution time of RSS (for network size 127) and that of ESQ, ESC afterwards.

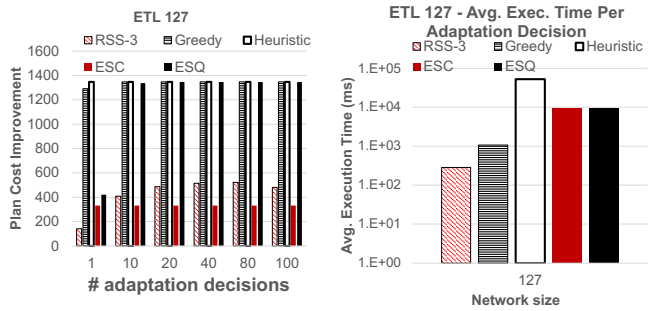

Figure 6: Homogeneous-ETL: Plan Cost & Execution Time.

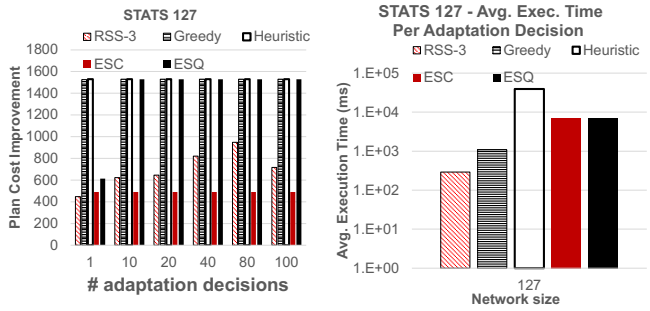

Figure 7: Homogeneous-STATS: Plan Cost & Execution Time.

Homogeneous Setup: In homogeneous setups, the previous picture is reverted. In Figure 6 and Figure 7, we plot pairs of graphs for Plan Cost Improvement, Average Algorithm Execution Time in the ETL and STATS workflows. We provide results for the middle case of 127 network size, but our observations hold across network sizes in the homogeneous setup case. Heuristic, Greedy provide the best Plan Costs Improvements, while ESQ is ranked third. Greedy simultaneously provides the lowest Average Execution Time of up to one second, making it the best choice in these setups.

4.2 Real Testbed Validation and Comparisons

We now compare APEROL against DAG* [54], Spring Relaxation (Spring-Relax) used in NEMO [13]; and Governor [14]. We compare these algorithms on our MacBook Air M2, both on the Real Testbed and on the simulation setups. DAG* and Spring-Relax operate on optimizing latency (the lower plan cost, the better for the experiments of this section) and they cannot be trivially extended to host the other performance dimensions considered in APEROL. For instance, DAG* optimality is not guaranteed for diverse performance dimensions that are non monotonic (increasing or decreasing the score as the algorithm proceeds). Similarly, Spring-Relax by design is based on physics and uses network latency values as "forces" to drag an operator on one or the other side of the network. Therefore, for a fair experimental comparison, we zero out the weight of the rest of the dimensions in the formula of Section 2.1 for the compared candidates and for APEROL. As discussed in Section 2.1, the migration cost is a latency value which accounts for the time it takes to transfer the state, upon runtime adaptation, without letting the workflow ingest and process data. Since DAG* does not account for the migration cost upon exploring the search space, we add that migration cost to the output, optimal plan of DAG*.

In Figure 8, Figure 9 (comparison on the simulated setup) and in Figure 10 (comparison on the Real Testbed), we show pairs of Plan Latency, Execution time for DAG*, Spring-Relax, Governor and Greedy, Heuristic and RSS. We omit ESQ and ESC to improve readability as the trends are similar to our previous experiments.

Cross-benchmark consistency. On matched network sizes (7, 15, 31, 127) on both the Yahoo Benchmark on the Real Testbed and RIoT on the simulated setups, the qualitative ordering of algorithms and the trends are similar: Heuristic and Greedy consistently yield the lowest plan latencies (Heuristic: 2.52–4.14 ms on RIoT, 1.43–4.13 ms on Yahoo; Greedy: 2.52–11.19 ms on RIoT, 1.47–7.46 ms on Yahoo). DAG* ties this front-runner pair on small graphs (2.52–4.14 ms on RIoT, 1.53–6.70 ms on Yahoo) but its execution time, while tiny on Yahoo (10–181 ms), already rises on RIoT at 127 devices (3–6,413 ms) and fails to complete beyond that scale. Governor, Greedy offer the fastest execution but with a higher Plan Latency (Governor: 8–15 ms on RIoT, 18–43 ms on Yahoo; Greedy: 3–26 ms on RIoT, 21–53 ms on Yahoo). But Governor has a clearly higher Plan Latency (2–7× vs. Heuristic on RIoT; 2.31× on Yahoo). RSS maintains a nearly constant execution time (67.6–79.2 ms on RIoT; 82–105 ms on Yahoo), while its plan quality degrades as the search space grows (e.g., 5.58–19.93 ms on RIoT; 2.60–15.80 ms on Yahoo). Spring-Relax constitutes a worse Plan Latency–Execution Time trade-off (9.03–28.76 ms and 42–176 ms on RIoT; 5.17–7.77 ms and 54–134 ms on Yahoo). Thus, the trends of the simulated setup are mirrored in the Real Testbed.

For a more explicit ranking of the algorithms we rank them by (i) Plan Latency and (ii) Execution Time separately. We take the respective performance values of each algorithm for each (workflow, network size) point, (separately for Plan Latency and Average Execution time, respectively), averaging each algorithm's performance across networks and benchmarks. We then compute a combined score via the Harmonic Mean of Plan Latency and Execution time.

Ranking on Average Plan Latency (lower is better):
Heuristic (4.42 ms) < DAG* (4.76 ms, when it completes) < Greedy (8.46 ms) < RSS (18.22 ms) < Spring-Relax (20.47 ms) < Governor (21.29 ms)

Ranking on Average Execution Time (lower is better):
Governor (29.35 ms) < Greedy (73.47 ms) < RSS (93.16 ms) < Spring-Relax (2,195.24 ms) < Heuristic (2,640.24 ms) < DAG* (146,379.81 ms, when it completes)

Combined Trade-off (Harmonic Mean of Plan Latency, Execution Time, lower is better):
Heuristic < Greedy < Governor < RSS < DAG* < Spring-Relax
Heuristic provides a 3× better harmonic mean than Governor.

4.3 Scalability – Batching – Hybrid Algorithms

Scalability & Batching: In Section 3.2, we argued about reducing the effect of parallelization barriers by introducing batching. In Figure 11, we study the effect of parallelism coupled with batching. Batching reduces contention among shared variables and affects the rate of examined plans. We present results for ESC, but the increasing trends are similar for all APEROL algorithms. The black

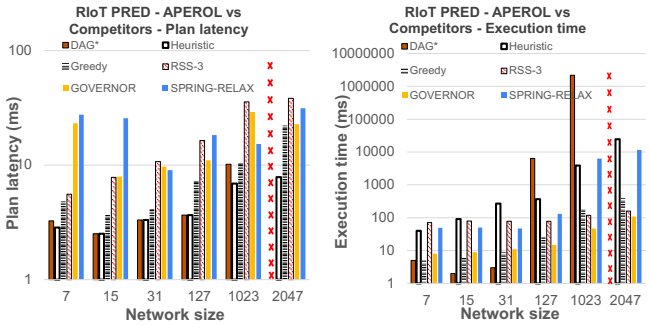

Figure 8: PRED - APEROLvsCompetitors - Plan Latency (the lower the better) and Avg. Exec.Time in Heterogeneous setup.

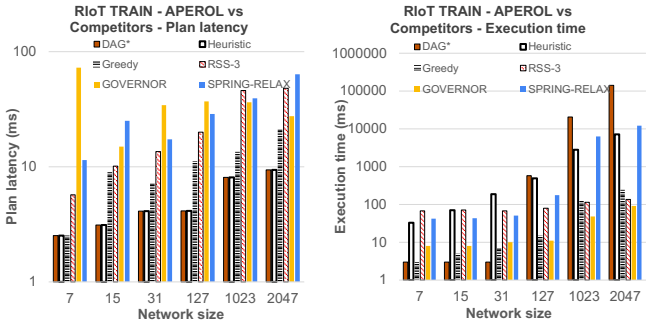

Figure 9: TRAIN - APEROLvsCompetitors - Plan Latency (the lower the better) and Avg Exec.Time in Heterogeneous setup.

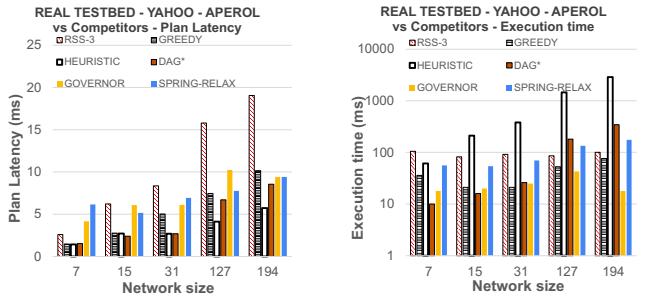

Figure 10: YAHOO - APEROL vs Competitors Plan Latency (the lower the better) and Exec. time on the Real Testbed.

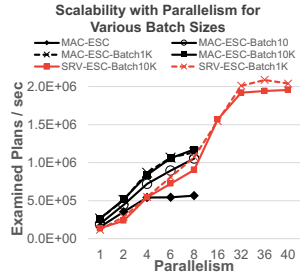

Figure 11: Scaling with Parallelism for various Batches and Hardware Setups

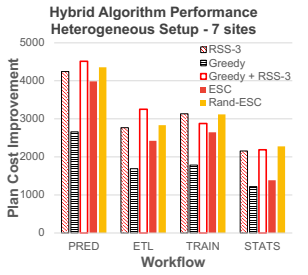

Figure 12: Effect of Hybrid Greedy + RSS and Randomized ESC

lines (prefixed with MAC) in Figure 11, plot ESC on the MacBook Air M2 (up to 8 cores). ESC without batching increases the rate of examined plans linearly up to a parallelism of 4, but then a plateau appears due to barriers. This is the reason we chose a default parallelism of 4 in our previous experiments. Batches of 10, 1K and 10K size, increase the rate of examined plans linearly with the full range of parallelism. ESC-Batch10 has the major effect, almost doubling the examined plans per second for a parallelism of 8. ESC-Batch1K improves ESC-Batch10 by up to +100K plans per second. Further increasing the batch size to ESC-Batch10K has little effect, which is because threads are overutilized. In Figure 11, up to 1.2M plans/sec are examined by tuning the batch size.

We also test scalability on a server with Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz, 20 cores/ 40 threads, and 256Gb RAM (SRV lines in Figure 11). ESC examines more than 2M plans per second on par with batching. Hence, upon deploying APEROL on more powerful hardware, the rate of examined plans keeps scaling before reaching a plateau (on 36 – 40 threads), due to contention on shared variables. Note that for the same parallelism (e.g. 4, 8) and batch size, MAC has higher throughput due to faster CPU clock speed.

Hybrid Variants: In Section 3.5, we discussed the ability to create hybrid approaches combining Heuristic or Greedy with RSS to help the former avoid local optima. The idea is that we let one of the former algorithms run and then receive their output plan (without deploying it yet) and use it as the root plan for RSS. In Figure 12, we apply this concept on a hybrid Greedy+RSS variation for all workflows, on a network of size 7, for the heterogeneous setup, where Greedy shows poor performance. Greedy+RSS provides plans that considerably surpass the Plan Cost Improvement of both Greedy and RSS in 2/4 workflows, in one workflow it equalizes the best (RSS), while only in the TRAIN workflow it gives a plan of lower quality than RSS. Notice that Greedy+RSS always improves Greedy, validating the claim that it helps in avoiding local optima.

We also experiment with another hybrid approach, termed "Rand-ESC", to alleviate the fact that ESC starts from plan numbered 0 and its time-capped version may examine plans that are away from the current root, therefore do not capture the conditions of the currently deployed plan. The Rand-ESC enhancement we introduce randomly shuffles plan ID batches and assign these shuffled batches (instead of sequential plan IDs) to threads. This is expected to improve resilience and exploration diversity in limited-time budgets for ESC. As Figure 12 shows, Rand-ESC does considerably improve ESC, but mostly ties the performance of the fully randomized search of RSS.

4.4 Sensitivity to Performance Weightings

To examine APEROL's sensitivity to different weightings on the four cost dimensions, we varied w_1–w_4 used in the $Cost(GoP_\pi)$ (Section 2.1) over 16 settings (single-dimension dominance, paired balances, and mixed 'random' weights), for Greedy, Heuristic and RSS. We perform this experiment on the Real Testbed using the Yahoo Benchmark workflow. For each set of weights, Table 1 reports the normalized $Cost(GoP_\pi)$ of score of the plan suggested

Table 1: Sensitivity Analysis: *Balanced Baseline* *Single weight emphasis* *Pairwise trade-offs* *Random weights*

weight	Greedy	Heuristic		RSS
	Norm. score	[thrpt, lat, comm, migr]	Norm. Score	Norm. Score
[0.25, 0.25, 0.25, 025]	0.692	[325.7K, 0.81, 7.81MB, 22.00]	0.798	0.629
[0.70, 0.10, 0.10, 0.10]	0.316	[325.7K, 0.81, 7.81MB, 22.00]	0.606	0.320
[0.10, 0.70, 0.10, 0.10]	0.804	[325.7K, 0.81, 7.81MB, 22.00]	0.912	0.75
[0.10, 0.10, 0.70, 0.10]	0.865	[325.7K, 0.81, 7.81MB, 22.00]	0.887	0.831
[0.10, 0.10, 0.10, 0.70]	0.862	[11.2K,, 25.75, 3.13MB, 0.00]	0.862	0.851
[0.40, 0.40, 0.10, 0.10]	0.56	[325.7K, 0.81, 7.81MB, 22.00]	0.759	0.53
[0.40, 0.10, 0.40, 0.10]	0.571	[326.6K, 0.81, 7.84MB, 22.80]	0.746	0.539
[0.40, 0.10, 0.10, 0.40]	0.561	[325.7K, 12.42, 32.00MB, 9.23]	0.715	0.551
[0.10, 0.40, 0.40, 0.10]	0.838	[325.7K, 0.81, 7.81MB 22.00]	0.900	0.795
[0.10, 0.40, 0.10, 0.40]	0.811	[326.6K, 3.90, 15.68MB, 15.97]	0.852	0.746
[0.10, 0.10, 0.40, 0.40]	0.856	[11.2K, 25.75, 3.13MB, 0.00]	0.856	0.812
[0.03, 0.43, 0.31, 0.23]	0.89	[107.8K, 0.03, 0.00MB, 21.20]	0.926	0.802
[0.07, 0.21, 0.59, 0.13]	0.882	[107.8K, 0.03, 0.00MB, 21.20]	0.912	0.837
[0.15, 0.34, 0.10, 0.41]	0.767	[326.6K, 3.90, 15.68MB, 15.97]	0.828	0.717
[0.22, 0.06, 0.51, 0.21]	0.749	[325.7K, 0.81, 7.81MB, 22.00]	0.811	0.702
[0.44, 0.11, 0.29, 0.16]	0.526	[325.7K, 0.81, 7.81MB, 22.00]	0.718	0.487

by each algorithm to ease comparisons. For Heuristic, we also provide the individual performance dimensions of the suggested plan. As shown in the table, Heuristic (Pareto-based) shows the smallest variation in aggregate quality as its normalized score ranges from 0.606 to 0.926 (Δ=0.320). Greedy spans 0.316–0.890 (Δ=0.574) and RSS 0.320–0.851 (Δ=0.531). Thus, Heuristic is the least sensitive to weight changes, because it does not guide its search based on weights, but only applies weights at its final iteration to pick only one plan out of those standing on the Pareto front. In contrast, Greedy immediately applies the weighting to find the locally optimal plan and examines only the neighbors of that plan.

In Table 1, high weight on individual dimensions (weights highlighted in red) does not change the performance of each dimension ([325.7K/s, 0.81 ms, 7.81MB/s, 22.00 ms]) of the chosen plan, but it does alter $Cost(GoP_\pi)$ due to the different weights on each dimension. But, high weight on migration cost (row [0.10, 0.10, 0.10, 0.70]) drives to 0.00 ms migration cost, which implies Heuristic chooses to keep the root plan. Across the six pairwise weight changes in Table 1 (highlighted in green), the per dimension effects show that Heuristic consistently keeps latency low and lets the dimensions with higher weights rise moderately. Again, setting the weight on migration cost to 0.40 affects the rest of the dimensions of the chosen plan, the most. Finally, in the random weight assignments (weights highlighted in yellow), Heuristic reaches its maximum score (0.926, row [0.03, 0.43, 0.31, 0.23]) with very low latency (0.03 ms) and zero communication cost. Heuristic in this case (and the [0.07, 0.21, 0.59, 0.13] row below) places all operators on a RPi.

5 RELATED WORK

The first distributed stream processing systems (DSPEs) emerged with frameworks such as Flux [47] and Borealis [32]. These, as well as modern DSPEs including Apache Flink [11], Storm [18] and Spark [6], are well suited for large clusters and cloud environments, but their heavyweight runtimes make them unsuitable for IoT networks, which may rely on resource constrained devices [58]. The optimizers of such systems do not consider network-related metrics, but focus on optimizing the workflow's performance by sophisticated task scheduling [43] or by trying to optimally set parallelism and balance the workflow load [33].

Several works optimize the execution of streaming workflows using powerful Big Data platforms [2, 12, 24, 56]. In addition to optimizing execution, another goal is to unify data analytics. Cross-platform optimization [4, 22, 23, 27, 28, 51] addresses workflow optimization in cloud-based setups but neglects the unique characteristics of streaming environments. Frameworks such as [4, 22, 28] typically optimize workflows at a single cloud hosting multiple Big Data frameworks, overlooking the placement of operators in distant machines and the necessity for incremental re-optimization.

Frameworks, such as Medusa [7] and SQPR [34], explicitly address network-related metrics, aiming to efficiently distribute workloads across networked hosts to minimize resource consumption. SBON [44] focuses on network resource efficiency for operator placement and it was later extended by Rizou et al. [46] to handle multiple operators. These methods mainly consider network latency metrics, neglecting the metrics mentioned in Section 2.1. SBON and NEMO [13] on NebulaStream both employ the Spring Relaxation algorithm for placing operators. Recently, Tzortzi et al.[55] introduced machine learning approaches to estimate operator and network costs. Governor [14], relies on manually crafted heuristics (e.g., filtering at the edge and relational processing in the cloud).

Governor is also used by COSTREAM [30] to provide an initial placement of a workflow on a never before seen network, besides utilizing GNNs to learn the cost of operators on devices with different computational capacities. DAG* [54] introduces a novel A*-like algorithm with a new admissible heuristic and plan expansion rules that drastically prunes the explored search space, guaranteeing to output the workflow execution plan with the optimal end-to-end latency. In Section 4.2, we proved that DAG* cannot scale in networks of 100s or 1000s sites. Flouris et al. [25] study complex event processing (CEP) operator placement across geo-distributed sites under the *push–pull* paradigm. Akili et al [5] optimize in-network CEP operator placement utilizing CEP-specific query rewritings over multiple queries sharing events and query sub-patterns. These optimizations have not been incorporated yet in typical DSPEs such as FlinkCEP [11] or NebulaStream [42]. NebulaStream [42, 58], EdgeWise [26] and Dart [40] are IoT DSPEs tailored for the heterogeneity and resource constraints of IoT networks. APEROL could complement these engines, acting as an optimization layer.

6 CONCLUSION AND FUTURE WORK

We have presented APEROL, a suite of parallel optimization algorithms for rapid, layered, in-network workflow execution in IoT settings. APEROL covers diverse heterogeneity and scale, exploring up to 2M plans/s on commodity hardware. APEROL's search space modeling is a foundation for new, parallel algorithms.

Interesting future work includes evolutionary search variations to further (besides the hybrids of Section 4.3) avoid local optima for algorithms such as Heuristic. A general direction would be to: (i) seed populations with the best e.g., Heuristic plans, (ii) apply mutations/crossovers on these plans (inheritance defined by hops in search space); (iii) run under an offspring/mutant budget, returning the best Pareto plan; and (iv) reuse APEROL's cost model as the fitness function for the evolutionary search.

ACKNOWLEDGMENTS

This work was supported by the EU Horizon projects CREXDATA (GA. 101092749) and DataGEMS (GA. 101188416).

REFERENCES

[1] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. 2015. "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed". Milan, Italy. https://hal.inria.fr/hal-01213938

[2] Ashvin Agrawal and Avrilia Floratou. 2018. Dhalion in Action: Automatic Management of Streaming Applications. *Proc. VLDB Endow.* 11, 12 (2018), 2050–2053. https://doi.org/10.14778/3229863.3236257

[3] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

[4] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

[5] Samira Akili, Steven Purtzel, and Matthias Weidlich. 2023. INEv: In-Network Evaluation for Event Stream Processing. *Proc. ACM Manag. Data* 1, 1, Article 101 (May 2023), 26 pages. https://doi.org/10.1145/3588955

[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[7] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. 2004. Contract-Based Load Management in Federated Distributed Systems. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, Robert Morris and Stefan Savage (Eds.). USENIX, 197–210. http://www.usenix.org/events/nsdi04/tech/balazinska.html

[8] Kaustubh Beedkar, Aurélien Bertrand, Haralampos Gavriilidis, Augusto José Fonseca, Zoi Kaoudi, Mingxi Liu, Volker Markl, Juri Petersen, Fábio Porto, Víctor Ribeiro, Mads Sejer Pedersen, Lucas Giusti Tavares, Michalis Vargiamis, and Chen Xu. 2025. Apache Wayang in Action: Enabling Data Systems Integration via a Unified Data Analytics Framework. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 35–38.

[9] Google Cloud Blog. [n.d.]. Extending Anthos to manage on–premises edge VMs: now generally available. https://cloud.google.com/blog/topics/anthos/extending-anthos-to-manage-on-premises-edge-vms-now-generally-available Accessed July 2025.

[10] Google Cloud Blog. [n.d.]. Small footprint, big impact: running cloud–connected Kubernetes at the edge. https://cloud.google.com/blog/topics/developers-practitioners/small-footprint-big-impact-running-cloud-connected-kubernetes-edge/ Accessed July 2025.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13. https://doi.org/10.1145/2806777.2806853

[12] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal operator placement for distributed stream processing applications. 69–80. https://doi.org/10.1145/2933267.2933312

[13] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1501–1514. https://doi.org/10.14778/3648160.3648186

[14] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2020. Governor: Operator Placement for a Unified Fog-Cloud Environment. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 631–634. https://doi.org/10.5441/002/EDBT.2020.81

[15] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1789–1792. https://doi.org/10.1109/IPDPSW.2016.220

[16] AWS IoT Greengrass Documentation. [n.d.]. Greengrass component basics. https://docs.aws.amazon.com/greengrass/v2/developerguide/greengrass-components.html Accessed July 2025.

[17] AWS IoT Greengrass Documentation. [n.d.]. Run a Docker container component. https://docs.aws.amazon.com/greengrass/v2/developerguide/run-docker-container.html Accessed July 2025.

[18] Apache Storm Documentation. [n.d.]. Apache Storm. https://storm.apache.org/index.html Accessed July 2025.

[19] Google Cloud Documentation. [n.d.]. Overview of connected deployments of Google Distributed Cloud. https://cloud.google.com/distributed-cloud/edge/latest/docs/overview Accessed July 2025.

[20] Microsoft Azure Documentation. [n.d.]. Module composition guidance for Azure IoT Edge. https://learn.microsoft.com/en-us/azure/iot-edge/module-composition Accessed July 2025.

[21] Microsoft Azure Documentation. [n.d.]. What is Azure IoT Edge. https://learn.microsoft.com/en-us/azure/iot-edge/about-iot-edge Accessed July 2025.

[22] Katerina Doka, Nikolaos Papailiou, Dimitrios Tsoumakos, Christos Mantas, and Nectarios Koziris. 2015. IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1451–1456. https://doi.org/10.1145/2723372.2735377

[23] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. 44, 2 (Aug. 2015), 11–16. https://doi.org/10.1145/2814710.2814713

[24] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10, 12 (2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[25] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, and Minos Garofalakis. 2020. Network-wide complex event processing over geographically distributed data sources. *Information Systems* 88 (2020), 101442. https://doi.org/10.1016/j.is.2019.101442

[26] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. 2019. EdgeWise: A Better Stream Processing Engine for the Edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 929–946. http://www.usenix.org/conference/atc19/presentation/fu

[27] Nikos Giatrakos, Elias Alevizos, Antonios Deligiannakis, Ralf Klinkenberg, and Alexander Artikis. 2023. Proactive Streaming Analytics at Scale: A Journey from the State-of-the-art to a Production Platform. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (Birmingham, United Kingdom) *(CIKM '23)*. Association for Computing Machinery, New York, NY, USA, 5204–5207. https://doi.org/10.1145/3583780.3615293

[28] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 2:1–2:16. https://doi.org/10.1145/2741948.2741968

[29] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296.

[30] Roman Heinrich, Carsten Binnig, Harald Kornmayer, and Manisha Luthra. 2024. Costream: Learned Cost Models for Operator Placement in Edge-Cloud Environments. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 96–109. https://doi.org/10.1109/ICDE60146.2024.00015

[31] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.* 12, 9 (2019), 1002–1015. https://doi.org/10.14778/3329772.3329777

[32] Jeong-Hyon Hwang, Sanghoon Cha, Ugur Çetintemel, and Stanley B. Zdonik. 2008. Borealis-R: a replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1303–1306. https://doi.org/10.1145/1376616.1376761

[33] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[34] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter R. Pietzuch. 2011. SQPR: Stream query planning with reuse. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 840–851. https://doi.org/10.1109/ICDE.2011.5767851

[35] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Bertty Contreras-Rojas, Rodrigo Pardo-Meza, Anis Troudi, and Sanjay Chawla. 2020. ML-based Cross-Platform Query Optimization. In *36th IEEE International Conference on Data Engineering, ICDE*

2020, Dallas, TX, USA, April 20-24, 2020. IEEE, 1489–1500.

[36] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[37] Sebastian Kruse, Zoi Kaoudi, Bertty Contreras-Rojas, Sanjay Chawla, Felix Naumann, and Jorge-Arnulfo Quiané-Ruiz. 2020. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *VLDB J.* 29, 6 (2020), 1287–1310. https://doi.org/10.1007/S00778-020-00612-X

[38] Sebastian Kruse, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Sanjay Chawla, Felix Naumann, and Bertty Contreras-Rojas. 2019. Optimizing Cross-Platform Data Movement. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1642–1645.

[39] Aljoscha Lepping, Hoang Mi Pham, Laura Mons, Balint Rueb, Ankit Chaudhary, Philipp Grulich, Steffen Zeuch, and Volker Markl. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *PVLDB* 16, 12 (2023).

[40] Pinchao Liu, Dilma Da Silva, and Liting Hu. 2021. DART: A Scalable and Adaptive Edge Stream Processing Engine. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 239–252. https://www.usenix.org/conference/atc21/presentation/liu

[41] Md. Redowan Mahmud, Samodha Pallewatta, Mohammad Goudarzi, and Rajkumar Buyya. 2022. iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments. *J. Syst. Softw.* 190 (2022), 111351. https://doi.org/10.1016/J.JSS.2022.111351

[42] Adrian Michalke, Aljoscha P. Lepping, Volker Markl, Ricardo Martinez, Nils L. Schubert, Lukas Schwerdtfeger, Taha Tekdogan, Steffen Zeuch, Ariane Ziehn, Christoph Falkensteiner, Kyle Krüger, Alexander Meyer, Tobias Röschl, and Svea Wilkending. 2025. NebulaStream: An Extensible, High-Performance Streaming Engine for Multi-Modal Edge Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 195–198.

[43] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/2814576.2814808

[44] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*. 49–49. https://doi.org/10.1109/ICDE.2006.105

[45] CREXDATA Project. [n.d.]. Weather Emergencies Use Case. https://crexdata.eu/weather-emergencies/ Accessed July 2025.

[46] Stamatia Rizou. 2014. *Concepts and algorithms for efficient distributed processing of data streams*. Ph.D. Dissertation. University of Stuttgart. http://elib.uni-stuttgart.de/opus/volltexte/2014/8835/

[47] M.A. Shah, J.M. Hellerstein, Sirish Chandrasekaran, and M.J. Franklin. 2003. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering*. 25–36. https://doi.org/10.1109/ICDE.2003.1260779

[48] Haichuan Shang and Masaru Kitsuregawa. 2013. Skyline operator on anti-correlated distributions. *Proc. VLDB Endow.* 6, 9 (July 2013), 649–660. https://doi.org/10.14778/2536360.2536365

[49] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4257. https://doi.org/10.1002/cpe.4257 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4257 e4257 cpe.4257.

[50] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4257.

[51] Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. 2012. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 829–840. https://doi.org/10.1145/2213836.2213963

[52] I.M. Sobol. 1998. On quasi-Monte Carlo integrations. *Mathematics and Computers in Simulation* 47, 2 (1998), 103–112. https://doi.org/10.1016/S0378-4754(98)00096-2

[53] George Stamatakis, Antonis Kontaxakis, Alkis Simitsis, Nikos Giatrakos, and Antonios Deligiannakis. 2022. SheerMP: Optimized Streaming Analytics-as-a-Service over Multi-site and Multi-platform Settings. In *EDBT*. 2:558–2:561. https://doi.org/10.48786/edbt.2022.50

[54] Errikos Streviniotis, Dimitrios Banelas, Nikos Giatrakos, and Antonios Deligiannakis. 2025. DAG*: A Novel A*-like Algorithm for Optimal Workflow Execution across IoT Platforms. In *Proceedings of the 41st International Conference on Data Engineering (ICDE'25)*. Hong Kong SAR, China.

[55] Marianna Tzortzi, Charalampos Kleitsikas, Agis Politis, Sotirios Niarchos, Katerina Doka, and Nectarios Koziris. 2023. Planning Workflow Executions over the Edge-to-Cloud Continuum. In *Algorithmic Aspects of Cloud Computing - 8th International Symposium, ALGOCLOUD 2023, Amsterdam, The Netherlands, September 5, 2023, Revised Selected Papers (Lecture Notes in Computer Science)*, Ioannis Chatzigiannakis and Ioannis Karydis (Eds.), Vol. 14053. Springer, 9–24. https://doi.org/10.1007/978-3-031-49361-4_1

[56] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 374–389. https://doi.org/10.1145/3132747.3132750

[57] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (2019), 516–530. https://doi.org/10.14778/3303753.3303758

[58] Steffen Zeuch, Ankit Chaudhary, Bonaventura Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *Conference on Innovative Data Systems Research (CIDR)*.