Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures

Henriette Röger
University of Stuttgart
roegerhe@ipvs.uni-stuttgart.de

Sukanya Bhowmik
University of Stuttgart
bhowmisa@ipvs.uni-stuttgart.de

Kurt Rothermel
University of Stuttgart
kurt.rothermel@ipvs.uni-stuttgart.de

Abstract

Control mechanisms of stream processing applications (SPAs) that ensure latency bounds at minimal runtime cost mostly target a specific infrastructure, e.g., homogeneous nodes. With the growing popularity of the Internet of Things, fog, and edge computing, SPAs are more often distributed on heterogeneous infrastructures, triggering the need for a holistic SPA-control that still considers heterogeneity. We therefore combine individual control mechanisms via the latency-distribution problem that seeks to distribute latency budgets to individually managed components of distributed SPAs for a lightweight yet effective end-to-end control. To this end, we introduce a hierarchical control architecture, give a formal definition of the latency-distribution problem, and provide both an ILP formulation to find an optimal solution as well as a heuristic approach, thereby enabling the combination of individual control mechanisms into one SPA while ensuring global cost minimality. Our evaluations show that both solutions are effective—while the heuristic approach is only slightly more costly than the optimal ILP solution, it significantly reduces runtime and communication overhead.

CCS Concepts • **Information systems** → *Stream management*.

Keywords Stream Processing, Multi-provider Infrastructure, Fog Computing

ACM Reference Format:
Henriette Röger, Sukanya Bhowmik, and Kurt Rothermel. 2019. Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures. In *Proceedings of Middleware '19: Middleware '19: 20th International Middleware Conference (Middleware '19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361551

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361551

1 Introduction

The Internet of Things (IoT) requires continuous, timely processing of high volumes of data. Stream processing applications (SPAs) evaluate streams of data on the fly, thus enabling reactions to events and information gain with low latency [7, 12, 13, 18]. Latency-bounded SPAs process input streams within limited time, i.e., they are limited in the allowed end-to-end latency. To achieve this, they require a precise resource management to control queuing and communication latencies and at the same time keep the resource cost minimal. Due to the long running nature of SPAs with changing workload, the resource management needs to continuously find the best solution to provide the resources necessary to keep the latency bound and at the same time minimize resource cost by adapting the system's configuration accordingly. Particularly, for *distributed* SPAs that run on multiple nodes, this resource management requires SPA-wide coordination. Today, a variety of proactive *elasticity solutions* is available. They autonomously configure SPAs to keep latency bounds while minimizing the resource costs [3–7, 9, 13, 17–19, 28]. The authors of [23] give a detailed overview and classification of these elasticity solutions. Configuring the SPA comprises parallelization [19], placement [7] and adaptation of system parameters, e.g., buffer sizes [18]. In recent years, fog and edge computing have become a popular extension of cloud computing [2, 27]. Applications on fog and edge layers process data closer to the sources, which leads to lower communication latencies, more context related processing, and less bandwidth utilization than in cloud applications. Still, cloud computing provides high processing power and resource flexibility. To jointly benefit from low communication latencies, less network load, and privacy in edge and fog as well as computing power and flexibility in the cloud, SPAs are commonly deployed across edge, fog, and cloud, thus executing on a heterogeneous infrastructure. Cloud, fog, and edge nodes in such an infrastructure can further consist of individually controlled components and sublayers, hosted by different providers [1]. Situations that require SPAs with multiple providers include geographical distribution, legal or privacy constraints, and heterogeneous systems requirements [22]. Particularly, in IoT, we see high potential for these situations and thus expect modern SPAs

to be deployed across multiple providers and individually controlled components.

We name an SPA that fulfills the three properties — distributed, hosted on heterogeneous nodes, and infrastructure managed by multiple providers — DHM-SPA. Examples are world-wide web-applications, smart-grid applications monitoring different areas, digitalized factories, digital production control, and health-care applications.

Managing the end-to-end latency of DHM-SPAs at minimal resource cost is particularly challenging because the solution needs to consider different processing properties due to heterogeneity as well as different pricing models and manageability due to multiple providers. To manage a DHM-SPA, a single-model, i.e., one-size-fits-all approach, falls too short and might become a limitation because it cannot capture the complexity of the DHM-SPA. It oversimplifies, has high information requirements, or leads to non-optimal configurations. Furthermore, sometimes it is not possible to directly influence the configuration, e.g., if parts of the DHM-SPA are managed by different providers. However, ignoring heterogeneity and distribution can lead to a situation where the end-to-end latency-control is either not as effective as it could be or the cost is not optimal.

For DHM-SPAs, we therefore see the need to have a mechanism that *combines* different elasticity solutions that each optimally manage a part of the SPA. This enables a cost optimal latency compliant SPA-deployment while using the most eligible elasticity solution for each part of the SPA. The mechanism should ensure that, despite the individually managed components, the overall resource cost is minimal and the DHM-SPA maintains its end-to-end latency bound. So far, there is no integrated approach that considers the combination of different elasticity solutions in one SPA in a manner that minimizes global resource consumption, keeps the end-to-end latency bound, and enables each part of the SPA to be configured by the best-fitted elasticity approach.

Hence, in this paper, we tackle the problem of how to configure a latency-bounded DHM-SPA that is distributed over a heterogeneous infrastructure while combining multiple elasticity solutions so that the overall resource cost is minimized. A solution to this problem needs to fulfil the following requirements:

- Work with highly distributed infrastructures, hence enable as much *local* control as possible to enable fast elasticity and reduce communication overheads.
- Handle different kinds of elasticity solutions to align with the heterogeneous infrastructure.
- Support even black-box elasticity solutions, e.g., when a cloud provider does not exhibit its elasticity mechanism but provides cost and latency by Service Level Agreements (SLAs) only.

In this paper, we solve this configuration problem and thereby make the following contributions:

1. We provide an integrated approach to ensure end-to-end latency compliance for DHM-SPAs at minimal resource cost. The approach combines multiple elasticity solutions across a heterogeneous infrastructure.
2. We introduce a hierarchical architecture to optimally configure DHM-SPAs at runtime.
3. From the configuration problem, we derive the latency-distribution problem that relies on a divide-and-conquer mechanism.
4. We provide two effective solutions for the latency-distribution problem.
5. We show the effectiveness of our proposed solutions through extensive evaluations on various cost models and compare them with another state-of-the-art algorithm for *homogeneous* systems.

2 Latency-bounded Stream Processing Applications

Our approach, described in Sections 3, 4, and 5, shows how to optimally *combine* different mechanisms and models to provide latency compliant and cost minimal DHM-SPAs. This section gives the necessary background of commonly used mechanisms to control latency in an SPA and of the models that enable cost optimal use of these mechanisms.

2.1 Mechanisms to influence the latency of an SPA

SPAs consist of an *operator graph*. The operator graph is a directed acyclic graph where the nodes are the *operators* processing the input data streams and the edges are the communication channels that connect the operators and define the flow of information. Each operator usually has an input queue where new data items arrive and wait for processing. In a distributed SPA, operators run on different processing nodes, connected via network channels.

An SPA is latency-bounded if there is a given time limit between the point in time a data item arrives at the system, e.g., the input queue of the first operator, until it is available for the consumer, e.g., by placing it into the consumers input queue. If an operator requires multiple data items, in particular, for the detection of complex events in Complex Event Processing (CEP), the time limit is set from when the *last* data item of the pattern arrives in at the SPA until the output is available for the consumer.

Latency comes from *queuing, processing, and communication* efforts. To achieve a given latency bound, we can influence the SPA's *configuration — parallelism, placement and system parameters.*

Parallelism enables high throughput and low queuing latency. Most common is *data parallelism* that replicates *instances* of an operator onto multiple parallel processing nodes. Each instance processes a portion of the total data each. When using data parallelism, splitting the input stream,

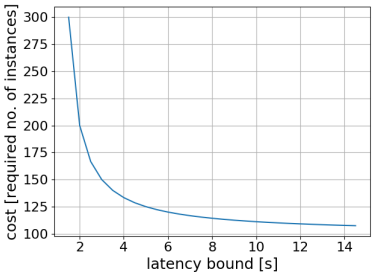

Figure 1. Smaller latency bounds need more instances.

management of distributed state and merging of results induce overhead. This overhead might diminish the positive effect on latency that parallelism shall achieve [26].

Placement assigns operator instances to processing nodes. An evenly balanced placement avoids overloaded nodes that have high queuing latencies. Further, placing operators on faster processing nodes reduces processing latency. To influence communication latency, a good placement places neighboring operators within a short distance. The degree to which placement reduces latency is limited, in particular, by the available infrastructure.

System parameters that influence latency are, e.g., acknowledgment frequencies, logging, and the output buffer size. The latter defines how many data items an operator assembles into one network package before it forwards them downstream [18]. A smaller output buffer leads to more network packages but small queuing latencies.

To sum up, parallelism, placement, and system parameters are techniques to control an SPA's latency. Lower latency comes with higher resource requirements, e.g., more or faster processing nodes to reduce queuing and processing latency, and vice versa. Finally, the resource requirements grow disproportionately with a latency decrease due to an increased overhead for data splitting and merging. A theoretical explanation can be given with queuing theory. It is depicted in Figure 1. It shows the number of instances (y-axis) given a latency bound (x-axis). For the plot, we use Equation 1 that is based on Littles Law for M/M/1 queues. We calculate the number of instances c with $c = \lceil \lambda_{is}/\lambda_{required} \rceil$ with an exponentially distributed arrival rate λ_{is} of 100 data items/s and a processing rate μ of 1 item/s. $\lambda_{required}$ is the average arrival rate *per instance* that would ensure processing within the given latency bound. We see that the lower the latency, the higher is the resource requirements for a further reduction.

$$\lambda_{required} = -1/latency + \mu \tag{1}$$

2.2 Model-Types for predictive Latency-bound management

While the former subsection described the mechanisms to influence an SPA's latency, in order to find cost-optimal and latency compliant SPA-configurations, we need to determine parameter values of these mechanisms. Example parameters are for parallelism a parallelization degree, for placement an operator-to-node assignment, and for system parameters an output batch size. A *configuration* of an SPA is then defined by these parameter values. We can use *models* that determine the necessary parameter values given information about the system such as workloads and processing rates. A model is the core of an elasticity solution and enables us to proactively calculate the parameter values for optimal configurations as opposed to, e.g., threshold-based configurations that monitor the system's workload and react to it, e.g., by adding a machine at an observed node-overload.

For an intuition of these models, in the following, we describe examples from literature. We focus on those models that can be used to calculate parameter values from system information such as workload and processing rates to achieve cost minimal, latency-bounded stream processing. However, they differ in whether they can be used to manage the latency of homogeneous or also of heterogeneous infrastructures as well as of single operators or multiple operators at once.

Models for **homogeneous** infrastructures assume that each processing node provides the same processing capacity. For example, a queuing-theory based model determines the number of service stations, i.e., the degree of parallelism, given the same service rate μ for each node.

In **heterogeneous** infrastructures, the model needs to distinguish the differences in processing capacities. The following example shows, why the model in this case also incorporates a placement problem. Consider a single, small processing node n. Now, the available latency decreases and requires lower queuing times, hence higher processing capacities. The model needs to reflect whether it is cheaper and latency compliant to add an additional smaller, hence slower, node, increasing the degree of parallelism, or to migrate the processing to one bigger, hence faster, node and turn off n. A migration decision itself can be costly and induce latencies, e.g., if state has to be migrated. The model thus needs to reflect these complex interdependencies. It can, for example, estimate the potential migration costs and latency spike.

Homogeneous infrastructures are common in cloud environments. We can rent multiple machines with the same configurations. Heterogeneous infrastructures are common in fog and edge processing.

Models for **single operators** only need to ensure the latency bound for the processing of this operator. Hence, the model determines the number of operator instances or necessary migrations. Models for **multiple operators** additionally balance the available latency between all operators.

An example for homogeneous, single operator control comes from Mayer et al. [19]. It uses queuing theory to keep queuing latencies below the latency bound. This works particularly well for M/M/1-queues. Another example comes from Balkesen et al. [3]. They predict the future workload of an operator and then divide the workload by the processing capacity for each node. A queuing-theory based model for

multiple operators in homogeneous infrastructures comes for example from Lohrmann [17]. They use Kingman's formula [11] to predict queuing latencies and thereof the required parallelization degrees. Zacheilas et al. [28] provide a model that uses a *learned* covariance matrix that describes the interdependence of end-to-end latency, arrival rates and number of instances. Given this covariance matrix, they determine the optimal parallelization degree by constructing a graph for possible configurations and their respective costs and apply a shortest path algorithm on this graph. A solution that considers heterogeneous nodes with placement and latency spikes from state migrations comes from Heinze et al. [7]. The publications of these models show that they are legible to keep latency bounds [7, 17, 19]. We now aim for a solution to *combine* these models within one DHM-SPA.

3 Cost optimal management of distributed SPAs

In this section, we present our problem statement and the conceptual parts of our solution to this problem.

3.1 Problem statement

Given a DHM-SPA, it is challenging to ensure the end-to-end latency. A single model type might not properly reflect the differences in the SPA's infrastructure. We further face the challenge to simultaneously find the optimal configuration for a DHM-SPA that can be distributed over three types of infrastructures which offer different levels of control. The first and second types are infrastructures with homogeneous or heterogeneous nodes that application owners can either directly configure because they rented the (virtual) machines or because they provide the infrastructure themselves. This level of control means, for example, that the owners can decide the placement of the operators, the degree of parallelism, the type of used machines. The third type are infrastructures that a provider controls and that an application owner influences via Service Level Agreements (SLAs) only. The application owner rents the complete service and has no internal control, e.g., how the degree of parallelism or the placement is defined or which type of machines is used. The parts of the SPA on infrastructures that are configurable directly by the application owner differ in the required model (c.f. Section 2). Those parts controlled by an external provider need a cost optimal SLA definition.

Hence, our problem statement is to find an optimal configuration for a DHM-SPA, where the parts might be managed by individual model types each and have different levels of control, such that the DHM-SPA keeps an end-to-end latency at minimal resource cost. In the following, we present our solution on how to handle this configuration problem. To this end, we formalize the latency-distribution problem and describe how we use *cost functions* to abstract over the differences in the models that jointly control the SPA .

3.2 Managing a distributed SPA with Control Units

The main idea of our solution is to encapsulate the heterogeneous parts of the SPA into independent components, and give each component a share of the overall latency. The component then manages its share according to the most applicable mechanism. The following two subsections give the details to this approach.

Available solutions that ensure end-to-end latency in a multi-operator SPA use one model for the complete application (c.f. [17, 28]). However, given our infrastructure, this might lead to suboptimal configurations due to the diverse model requirements or even not be possible at all due to the limitations of providers. Our goal is to have each part of the SPA managed with the best fitting model and to include SPA-components with different levels of control, e.g., those hosted by providers that do not exhibit their managing mechanisms to the customers.

Hence, instead of finding a complete configuration for the SPA at once to ensure the end-to-end latency bound, we model the SPA as a graph of individually controlled *control units (CUs)*. This allows multiple control models for a single SPA but at the same time ensures end-to-end optimization. A CU comprises a subset of the SPA's operator that is managed together according to the best fitting model. A CU can cover a single operator — thus being manageable by those models that work best for the single-operator case — or multiple operators. The infrastructure of a CU can still be heterogeneous but is then jointly managed by a controller that is optimized for the node types the CU comprises. Given, e.g., a part of the SPA being hosted at a provider, this is seen as one CU.

With DHM-SPAs, we target SPAs that already consist of multiple parts, e.g., for legal, privacy, soft- and hardware constraints, and geographical distribution. We explicitly model them now as one SPA-graph and add CUs to reflect and make use of the partwise control in our model. Hence, the scope of this work is not to define the CUs but rather to explicitly exploit the given heterogeneity in infrastructure and ownership of the SPA. We refer to existing work for techniques to optimize the granularity of operator graphs [8, 21] . Figure 2 gives an example for CUs of an SPA which we describe in detail in the architecture Section 4.

The CUs form a DAG that is a higher-level abstraction of the original SPA's operator graph (c.f. Figure 2). Introducing the CUs frees us from managing all operators of the SPA in a complete configuration. We rather shift the problem to managing the CUs instead. A CU becomes the smallest unit of control on the operator graph. The CUs themselves implement the model that fits best to manage their operators. Using its model, a CU autonomously keeps an assigned latency bound in a cost minimal manner.

As we described in Section 2, the general requirement of each of the models is that they find those parameter values that provide a cost minimal SPA configuration that ensures

a latency bound. Further, a latency bound is a valid property of an SLA and thus enables us to include provider-hosted services into our SPA. Given this requirement, we now transfer the general management problem by simply finding the best latency bound for each CU, so that the overall cost is minimized and the end-to-end latency bound of the SPA is ensured. As each CU individually ensures a cost minimal configuration for its latency bound and this minimal cost-value depends on the assigned latency bound, we minimize the overall cost if we find the optimal distribution of the available latency onto CUs. We formalize this new problem as the *latency-distribution problem* in the following section.

3.3 The latency-distribution problem

The latency-distribution problem describes the challenge to assign each CU of a distributed SPA a latency budget so that the overall cost of the SPA is minimized and every path of the SPA is compliant to the end-to-end latency bound.

Each CU of the SPA gets a latency budget it has to keep. Keeping the budget induces costs. How high the cost for a budget is, is individual for each CU and depends, e.g., on the CU's workload, the infrastructure it is hosted on, and the pricing mechanisms for the resources of this CU. Hence, in order to be cost minimal, the latency-distribution problem needs to cover the different non-linear latency-cost behavior of each CU.

To ensure the end-to-end latency bound, the individual budgets of each CU need to be assigned so that the sum of budgets along each path of the SPA does not exceed the end-to-end latency bound. Recall that the CUs themselves form a graph, where each path of this graph needs to be compliant with the end-to-end latency constraint.

The *latency-distribution problem* is thus the problem of optimally distributing the available end-to-end latency onto the CUs of a DHM-SPA, so that the total resource cost is minimized. Equations 2 and 3 formalize it:

$$min \sum_{cu \in CU} cost(cu, latency) \tag{2}$$

$$\forall paths \sum_{cu \in path} latency(cu) \leq \text{e-to-e latency bound.} \tag{3}$$

To abstract over the heterogeneous models, we use cost function as shown in Equation 2. A cost function maps the latency budget of a CU to the cost that are required by the CU to keep this budget. For example, a latency budget of 5 ms requires a CU to provide 5 instances of an operator or the migration to one bigger machine from two smaller ones with the respective price. The cost function of this CU would return the cost for these five instances for a budget of 5 ms. Similarly, a cost function of a CU managed by a different provider would return the cost for a latency budget according to the providers pricing scheme.

$$Cost_{CU} : Latency \rightarrow Cost \tag{4}$$

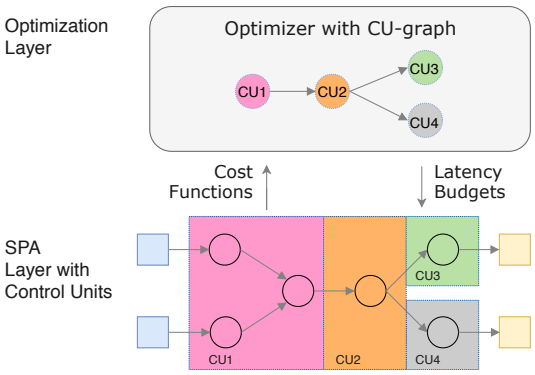

Figure 2. The Two-Layer Architecture with the SPA at the lower and the optimizer at the top layer.

Cost functions usually are convex, showing that disproportionally more resources are necessary the smaller the latency budget (c.f. Figure 1) . To derive a cost function from a control approach, it might, e.g., be necessary to provide a wrapper class that executes the algorithm of the control approach with the given latency value.

To show that the problem is NP-hard, we model it as a Knapsack problem: The set of possible items to pack are the possible latency budgets for CUs. The reward is the negative cost of a latency budget. The limited bag size is the maximum latency bound. However, this model is simplified as it assumes linear cost functions and discrete latency budgets for a discrete set of items to choose from, it ignores the condition of exactly one latency budget for a CU, and it prohibits different latency bounds for different paths which might be possible. Hence, the latency-distribution problem is even more complex.

4 System Model

Our solution has two layers (Figure 2). The figure demonstrates how an SPA in the lower layer is divided into four CUs. The different colors of the CUs represent the difference in control models. The upper layer contains the centralized optimizer that distributes the latency budgets.

4.1 SPA Layer and CU graph

The CUs of the SPA form a *CU graph* that is an abstraction of the SPA's operator graph. Recall that a CU is the smallest unit of control on the operator graph and each CU individually manages a latency budget. The most fine-grained division of the SPA into CUs is one CU per operator. A CU can belong to multiple paths of the CU graph., e.g., CU 2 in Figure 2.

4.2 Optimization Layer

At the Optimization Layer, given the CU graph, the optimizer divides the available end-to-end latency into individual latency budgets and assigns them to the CUs. To this end, it uses the algorithms from Section 5. The Optimization Layer only needs the CU graph and the cost function per CU to

optimize the assignments. With its global perspective, the optimizer can then find the cost-optimal latency distribution for the DHM-SPA.

4.3 Communication

For each CU, the optimizer has two options to access the cost function. The first option is a full transfer of the cost function to the optimizer. This option has low communication cost and is useful for simple cost functions, e.g., tables or simple models. To not limit our solution to these simple cases, the second option is a request/response mechanism that makes external cost-function calls. This option is useful, e.g., if there is a cloud provider that does not expose its cost function. Depending on the complexity of the cost function, a cost-function call can require non-linear processing time. If the CU is hosted by a public cloud provider, each cost-function call might include a fee. Hence, fewer cost-function calls are preferable when solving the latency-distribution problem.

5 Proposed Algorithms

This section presents two solutions to the latency-distribution problem. Section 5.1 describes an integer linear program (ILP) that solves the problem optimally. However, the latency-distribution problem is NP-hard. This complexity limits scalability of the ILP. We therefore provide a heuristic in Section 5.2. The heuristic exploits a greedy algorithm to solve the problem in linear runtime.

Both solutions use the following four principles for the cost function to find the optimal latency distribution:

1. The solutions assign an individual latency budget to each CU in the CU graph. The CU keeps this budget with its control model. For each path p in the CU graph, the path-wise *sum* of these latency budgets cannot exceed a maximum latency bound L_{max_p}.

2. Deterministic cost functions ensure that during the execution of a single optimization run, the cost for a latency budget value lat_{cu} is always the same. However, the SPA might experience changes over time (c.f. Section 2) which might affect the cost function. Examples include current inter-arrival times relevant for cost functions that use queueing-theory to determine the number of required processing nodes. In this realm, the inter-arrival time is an important parameter since a change in the inter-arrival time might affect the cost of a latency budget. Hence, the cost function remains deterministic during one optimization run but may update its parameters in-between runs to reflect these changes.

3. Minimal latency budget: Each CU requires a minimal latency budget to ensure a stable system. It is the minimal latency a CU can achieve. For example, even in case of zero queuing latency due to high parallelism and optimal placement, a minimal processing and communication latency always remains. Limited resources, e.g., in fog infrastructures, further increase the required minimal latency with only limited available processing capacity and restricted options for parallel processing.

4. Complete value domain: Both algorithms require cost functions for each CU, whose domain covers all possible latency budgets. The set of possible latency budgets is lower bounded by the required minimal latency budget $L_{min_{CU}}$ and upper bounded by the maximum latency bound L_{max}. We use the step size to get the set of discrete latency budgets from the continuous latency value by making the possible latency budgets multiples of the step size.

5.1 ILP based optimal solution

We formulate an ILP to find the optimal solution to the latency-distribution problem. Such an optimal solution assigns exactly one latency budget to each CU so that the total cost of the SPA is minimized and the path-wise sums of latency-budgets are below the end-to-end latency bound.

5.1.1 ILP Definition

Equations 5 to 8 formalize the ILP. The decision variables $x_{cu,l}$ represent all CU — latency budget combinations, i.e., $x \in \mathbb{B}^{L \times CU}$. $x_{cu,l}$ is 1, if the respective cu is assigned the latency budget l, or 0 otherwise. The coefficient $cost_{cu,l}$ is the cost of latency budget l at CU cu according to this CU's cost function. $\mathbb{CU}$ is the set of all CUs in the SPA, $\mathbb{P}$ is the set of all paths on the CU graph, $\mathbb{L}$ be the set of possible latency budgets defined as multiples of the step size between 0 and the latency bound L_{max}. This definition ensures discrete values for the latency budgets required for the ILP. The step size thereby indicates the granularity of the assignments. It is further possible to have arbitrarily small but fixed units for the latency budgets, e.g., milliseconds, seconds or minutes.

Equation 5 shows the cost-minimization objective that minimizes the total cost of the application. The cost functions are usually not linear and can therefore not be included directly into the objective function. Instead, we use the integer decision variables and the cost coefficients. Equations 6 to 8 denote the ILP's constraints. The first constraint, Equation 6, ensures that for each path, the sum of the latency budgets of the CU on that path cannot exceed the path's latency bound L_{max_p}. Each path can have an individual bound which is why we use the index p. Equation 7 ensures that each CU is assigned exactly one latency budget out of $\mathbb{L}$. Equation 8 is the integer-condition.

$$\min \sum_{cu \in \mathbb{CU}} \sum_{l \in \mathbb{L}} x_{cu,l} * cost_{cu}(l) \tag{5}$$

$$\sum_{cu \in \mathbb{P}} \sum_{l \in \mathbb{L}} x_{cu,l} * l \leq L_{max_p} \quad \forall p \in \mathbb{P} \tag{6}$$

$$\sum_{l \in \mathbb{L}} x_{cu,l} = 1 \quad \forall cu \in \mathbb{CU}, \tag{7}$$

$$x_{cu,l} \in \{0,1\} \quad \forall cu \in \mathbb{CU}, \forall l \in \mathbb{L} \tag{8}$$

5.1.2 Cost-function Calls by the ILP

A cost-function call, e.g., for a CU hosted by a public provider, can be seen as a Function as a Service, leading to *cost per call*. For instance, in [13], the algorithm needs to be executed multiple times to determine the required number of instances for different latency bounds. Hence, the number of cost-function calls is an important performance metric for a solution to the latency-distribution problem.

Given the ILP, a solver that finds an optimal solution for the ILP makes at initialization $|\mathbb{CU}| * |\mathbb{L}|$ cost-function calls and uses the results as the coefficients $cost_{cu}(l)$ in the objective function. $|\mathbb{CU}|$ is the number of CUs in the CU graph. $|\mathbb{L}|$ is the number of different latency budgets that the solver can assign to a CU. While $|\mathbb{CU}|$ is given by the SPA, the step size and the latency bound L_{max} defines $|\mathbb{L}|$. Through adjusting the step size, we can control the trade off between precision of the output and cost-function calls. For example given a latency bound of *2000ms*, a step size of 1 leads to $|\mathbb{L}| = 2001$; a step size of 100 to $|\mathbb{L}| = 21$. A longer step size reduces the number of cost-function calls, a smaller step size leads to a more precise, hence cheaper, result. An assignment of 168*ms* is possible using for step size one but can become a (less cost optimal) assignment of 200*ms* for step size 100.

5.2 Heuristic Solution

As the ILP is limited in scalability and needs a high number of cost-function calls, we additionally developed a heuristic. This heuristic implements a greedy strategy to quickly converge to a solution that is close to the optimal cost.

5.2.1 Greedy Algorithm

The greedy algorithm processes the CU graph pathwise. For each path, it initially distributes the available end-to-end latency evenly among all CUs of a path. It then optimizes this assignment stepwise. With each step, the greedy algorithm swaps a share of the latency assignment Lat_{CU} from one CU to another, where this share leads to reduced costs. The sum of Lat_{CU} of all CUs on a path thereby remains the same and does not exceed the end-to-end latency bound. The algorithm thus stepwise improves the total cost until it does not find an improving swap any more.

As in the ILP, the greedy algorithm needs for each CU a cost function with a domain over all possible latency budgets. Again, the step size controls the precision. Algorithm 1 gives the pseudocode. Its major steps are as follows:

a) Assign minimal latency to all CUs (L. 5)
b) While there are un-processed paths, select the path p_{min} with the smallest latency to distribute (L. 9)
c) process this path as follows:
   i Evenly distribute free latency (L. 11)
   ii Greedily optimize the assignment (L. 12)
   iii Mark CUs passive to forbid further changes and mark path as processed (L. 14)

Algorithm 1 Greedy Budget Assignment

```
1:  Graph g
2:  array of paths [1...P] ps ← paths_g
3:  array of cus [1...CU] cus ← cus_g
4:  for cu in cus do
5:      latency_cu ← getMinLatency(cu)
6:  end for
7:  p_left ← ps
8:  while size(p_left) > 0 do
9:      p ← p ∈ p_left : min(freeLat(p))
10:     cusActive ← getActiveCUs(p)
11:     distributeFreeLat(cusActive, freeLat(p))
12:     optimize(cusActive)
13:     for cu in cusActive do
14:         cu ← passive
15:     end for
16:     p_left ← p_left − p
17: end while

18: freeLat(path p):
19: assignedLat ←sum(latency_cu)∀cu ∈ p
20: freeLat ← L_p − assignedLat
21: return freeLat
```

Algorithm 2 Path-wise Optimization

```
1:  optimize(array of cus cus):
2:  cu_d ← cu ∈ cus : min(redCost_cu)
3:  cu_i ← cu ∈ cus : max(incGain_cu) and not cu_d
4:  while incGain_cu_i > redCost_cu_d do
5:      swapLatencyStep(cu_d, cu_i)
6:      cu_d ← cu ∈ cus : min(redCost_cu)
7:      cu_i ← cu ∈ cus : max(incGain_cu) and not cu_d
8:  end while

9:  swapLatencyStep(cu cu_d, cu cu_i):
10: incGain_cu_d ← redCost_cu_d
11: redCost_cu_i ← incGain_cu_i
12: latency_cu_d ←reduce(latency_cu_d)
13: latency_cu_i ←increase(latency_cu_i)
14: redCost_cu_d ←costFunction(reduce(latency_cu_d))
15: incGain_cu_i ←costFunction(increase(latency_cu_i))
```

In the following, we explain each of these steps:

a) Assign minimal latency. As initial assignment, each CU receives its minimal budget (Line 5). This budget is the minimal latency the CU can achieve, potentially at very high cost. If there is a path whose sum of these minimal assignments of the paths CUs exceeds the path's latency bound, the algorithm raises an exception—there is no solution for this latency bound. While each CU then has an assignment, it is not yet cost optimal. The greedy algorithm improves the total cost path-wise.

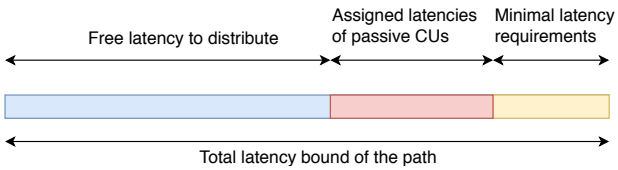

Figure 3. The *free latency* is the latency bound reduced by the minimally required and the already assigned latency.

Table 1. Example for greedy iterations with 3 CUs. Notation: (Lat_{CU}[sec], cost(Lat_{CU}) + $step$, cost($Lat_{CU} - step$)

Iteration	CU_1	CU_2	CU_3
Init	(5 sec,-1,**+2**)	(5 sec,-4,+5)	(5 sec ,**-5**,+6)
1	(4 sec,-2,**+3**)	(5 sec,**-4**,+5)	(6 sec,-1,+5)
2	(3 sec,-3,+8)	(4 sec,-2,+4)	(6 sec,-1,+5)

b) Find the next path to process. The algorithm processes next the path with the smallest latency value available to distribute. We name this latency value *free latency*. The free latency of a path is the latency bound of the path L_{max_p} minus already fixed assignments on this path (Figure 3, Line 11). These fixed assignments are the initial minimal latencies budgets (Line 5) and already fixed assignments of multi-path CUs. These multi-path CUs get their latency assignment when their first path is processed. When processing their other paths, their assignment is fix, the CUs thus *passive*. All CUs that have not yet been assigned a budget are *active* and will be considered in the path-wise processing.

The algorithm selects the path with the minimal free latency to avoid invalid assignments. An assignment is invalid for a path, if the sum of Lat_{CU} on the path exceeds the end-to-end latency bound. If the algorithm did not select the path with smallest free latency, it might assign latency budgets to multi-path CUs that exceed the free latency of another path, there resulting in an invalid assignment.

c) Path-wise Processing. Once the next path is selected, the algorithm first evenly distributes the free latency among the active CUs of the path, i.e., those CUs that have not been processed in other iterations before, (Line 10) and then greedily optimizes the assignment (Lines 12 to 14).

To evenly distribute the free latency, at first, the algorithm assigns each active CU, in addition to its minimal latency budget, an even share of the free latency. After this step, the complete latency is distributed among the CUs on the path but the assignment might not be optimal yet. Hence, the algorithm greedily optimizes the assignments (Line 12). As the latency budget is fully distributed after this step, a latency increase at one CU requires a latency decrease at another CU to fulfill the end-to-end latency bound. The algorithm therefore swaps assignments stepwise until it cannot further reduce the total cost of the path any.

Algorithm 2 details the greedy optimization. We use the example in Table 1 to illustrate the algorithm's steps. The example considers a path with three CUs, seconds as time unit and step size one. Each tuple shows the currently-assigned latency budget Lat_{CU}, the cost if this assignment *increases* by one step and the cost if this assignment *decreases* by one step. To ensure fast greedy steps and to limit the number of cost-function calls, we store this tuple for each CU. For readability, we show the respective deltas, not the actual cost value. In the first row, we see the initial assignment with the evenly distributed end-to-end latency (Line 11). For the greedy step, the algorithm selects the node with the biggest cost reduction when increasing latency (cu_i - CU to *increase* latency) and the node with the lowest cost increase when reducing latency (cu_d - CU to *reduce* latency) for one step (Line 2). In the example, it sets $cu_i = CU_3$ and $cu_d = CU_1$. If the cost reduction of cu_i is greater than the respective cost increase of cu_d, the CUs swap this latency step (Lines 9 et seq), in our example one second. In the example, CU_3 now has a budget of six sec and CU_1 a budget of four sec. The algorithm selects the next cu_i and cu_d until the stopping condition is met, i.e., if there is not cost reduction any more bigger than any resulting cost increase. Then, the path is fully processed. In our example, the stopping condition is true after the second swap.

There are cases when we can swap more than one latency step at once. This is possible if the bigger latency leads to the same cost at cu_d than a single step would, which can be the case if a cost function is not injective. Then, multiple latency values can have the same cost. An example is a step-wise defined function. With such a cost function, decreasing the latency of cu_d by a single step leads to some cost c. However, the system can, for the same cost c, provide an even lower latency and thus assigns a higher budget to the more expensive CU. Hence, we assign the smaller value to cu_d and respectively provide a higher increase of the latency budget of cu_i. This can reduce the overall cost further. We implemented this improvement for the Kingman-based cost function [11, 17], which has parallelization degrees as discrete steps and thus a non-injective cost function, and found in preliminary evaluations that this can improve the cost.

Once the stopping condition is met, the algorithm fixes the assignments and sets the CUs passive to be not affected by the processing of other paths they might belong to (Alg. 1, Line 14). Fixing the assignments reduces the runtime of the algorithm: Assume a bigger example with two paths, one with 10 CUs, another with 8 CUs, whereof 6 CUs are shared by both paths. After assigning the first paths with 10 CUs, the algorithm has to assign latency-bounds only to the two CUs not shared, instead of the full 8 CUs. Also, fixing the assignments guarantees that the algorithm converges.

The passive CU's budget is fixed when processing the other paths and considered when calculating the free latency as described before. In our example in Table 1, the budgets for CU_1, CU_2 and CU_3 would be fixed after the second iteration. If there was a path CU_2, CU_4, CU_5, only the CUs CU_4 and CU_5 are active and would be considered for further latency budget assignments.

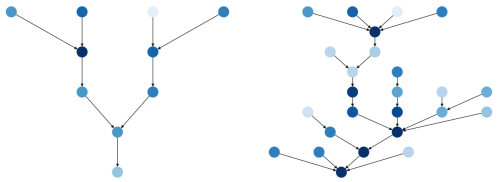

Figure 4. Examples for CU graphs. The darker the shade, the higher the workload.

After the assignments are fixed and CUs are flagged passive, the algorithm updates the free latency values of the un-processed paths and selects the path with the now minimal free latency as described in *b)* for the next iteration. When all paths are processed, the algorithm returns the final assignment.

5.2.2 Cost-Function Calls by the Greedy Algorithm

As cost-function calls can cost time and money, we aim to keep the number of cost-function calls low. The greedy algorithm at first evenly distributes the free latency budget among the active CUs of the path. This saves iterations that were required if we started from 0, making less cost-function calls. Afterwards, it calls the cost-function at each latency swap for cu_i and cu_d only. In Section 6, we show that the greedy algorithm makes significantly fewer calls than the ILP. To further reduce the number of cost-function calls, we provide a variation of the greedy algorithm that caches results of the cost-function calls.

6 Performance Evaluations

In this section, we present the results of our performance evaluations. We evaluate runtime, calculated cost and the number of cost-function calls for the ILP and the greedy algorithm. We further show how our greedy algorithm performs compared to a state-of-the-art algorithm for homogeneous infrastructures.

6.1 Setup

This section describes the graph types, cost functions and parameters of the evaluations. We execute all evaluations on a 128 GB server with an Intel Xeon e5 2687w v3 3.1 GHz CPU with 40 cores. The ILP solver is the open-source *Coin-And-Branchcut* solver.

6.1.1 Graph Types

The operator graph of an SPA is usually a directed, tree-shaped acyclic graph (DAG) that routes data from sources, i.e., the leaves, to the consumer, i.e., the root node. The corresponding CU graph is thus also a DAG. As a CU can include multiple operators, we evaluate on relatively small graphs. For a realistic scenario, we execute our evaluations with graphs of 10 to 100 nodes. Note that the SPA behind a 100-nodes CU graph is already an application with 100 individually managed parts. Using up to 100 CUs, we show that the approach scales and is applicable for the fast growing trend for fog-/edge infrastructure. Our solution supports larger graphs as well in a similar manner. Figure 4 shows exemplary two of the CU graphs we use with 10 respective 25 nodes. The color intensity qualitatively represents the workload. We create each CU graph randomly and randomly add initial workload values $randint$. We then further simulate workload propagation throughout the graph with a selectivity of 0.5. The workload of a CU v is thus $randint + 0.5 * \sum_{u:(u,v)\in E^{max}} workload(u)$.

6.1.2 Cost Functions

We use four types of cost functions to show that our approach seamlessly works with multiple cost functions types.

1. A cost function for M/M/1 queues based on Little's Law (c.f. Equation 1).
2. The cost function used by Lohrmann et al. [17] shown in Equation 9. It uses Kingman's formula [11] and is applicable for G/G/1 queues, hence without any assumption about the distribution of arrival and service rates. p^* denotes the parallelization degree that is required for queuing latency $latency$. The respective cost per CU is the $p^* * cost(instance)$.
3. A simple linear cost function $(100 - CU_{degree} * latency)$ that returns a higher value for nodes with a higher input degree.
4. A general representative exponential cost function as in Equation 10.

$$latency = \left(\frac{\lambda/\mu^2}{p^* - \lambda * \mu}\right)\left(\frac{(c_A(p^*))^2 + c_S^2}{2}\right) \tag{9}$$

$$cost_{CU}(latency) = 10 * e^{(-1*(CU_{degree})*latency/100)} + 1 \tag{10}$$

The first two cost functions are applicable for single-operator CUs, the latter for single- and multi-operator CUs. If not stated otherwise, to show the multi-model feasibility of our solutions, we randomly assign the cost functions to the CUs so that each CU graph in the evaluations includes all four cost functions. Through evaluations with randomly assigned cost functions, we show that our solution is not limited to specific cost functions but applicable for SPAs that *combine* CUs with different cost functions.

6.1.3 Latency Bounds and Step Sizes

We set the latency bound of a graph as a multiple of the graph's node count, using a *bound factor*. For example, a bound factor of 2 and a graph with 10 nodes leads to a bound of 20, and for a graph with 50 nodes to a bound of 100.
The step size defines the granularity of the latency-budget assignment (c.f. Section 5). In our implementation, we initially use step size one if not stated otherwise.

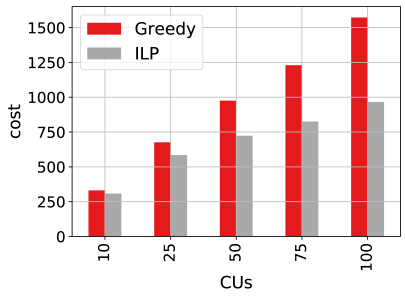

Figure 5. Cost

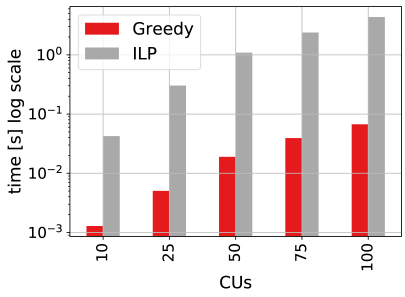

Figure 6. Algorithm Runtime

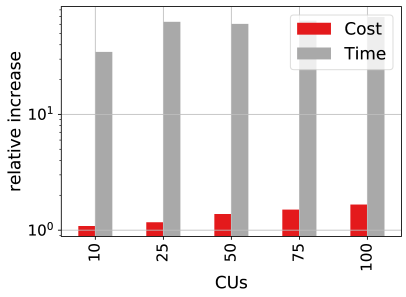

Figure 7. Cost, Runtime relative to ILP

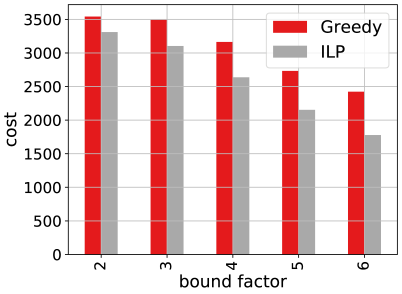

Figure 8. Cost for latency bounds

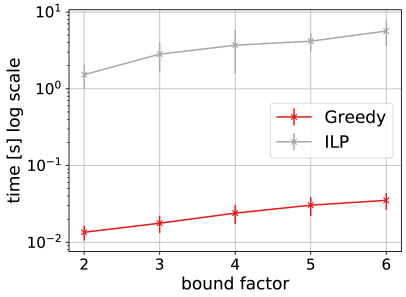

Figure 9. Runtime for latency bounds

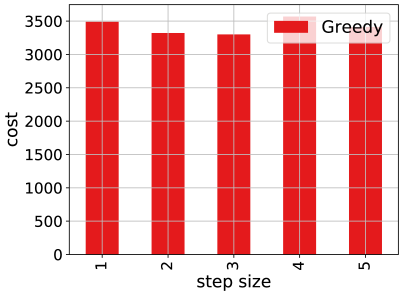

Figure 10. Cost for step sizes

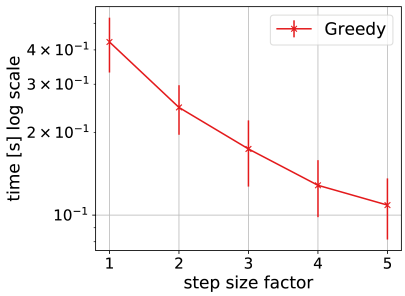

Figure 11. Runtime for step sizes

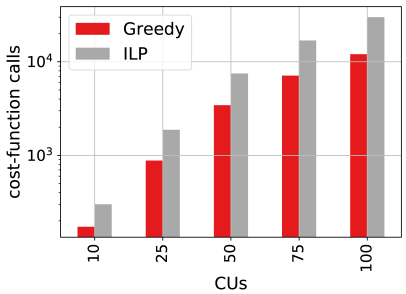

Figure 12. CF-Calls Greedy and ILP

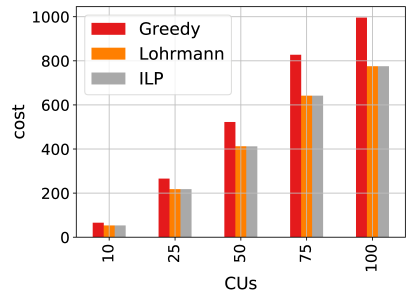

Figure 13. Cost Greedy, Lohrmann, ILP

6.2 Runtime and Cost Evaluation

We evaluate the two solutions for scalability, their performance for different step sizes, and for different latency bounds with the following three settings. Our performance measures are runtime and calculated cost and we evaluate with the settings as follows.

1. Graphs with 10, 25, 50, 75, 100 nodes, 50 randomly generated graphs per size
2. Step sizes 1-5, 50 graphs with 50 nodes
3. Bound factor 2-6, 50 graphs with 50 nodes

6.2.1 Evaluation Results

Figures 5 and 6 show the averaged global cost and the total logarithmic runtime of the algorithms for the different graph sizes. For both algorithms, the runtime increases with the number of CUs. Yet, it is sufficiently small to not indicate scalability issues. We see that in all configurations, the greedy solution leads to a slightly higher cost but decreases the runtime with orders of magnitudes compared to the ILP.

Figure 7 depicts the relative cost increase of the greedy solution to the optimal ILP solution and the runtime increase of the ILP solution to the greedy solution. We see that, with more CUs, the cost difference grows slightly but the runtime increase by the ILP is in the order of magnitudes higher than the cost increase.

Figures 8 and 9 show the performance of the algorithms for growing latency bound factors, i.e., less restrictive latency bounds. Figure 8 illustrates that the cost shrinks as expected with a higher latency bound and that both algorithms reflect this behavior. The runtime of both algorithms increases with a higher latency bound as seen in Figure 9 because a higher latency bound increases the search space $|\mathbb{L}|$ (c.f. Section 5).

Figures 10 and 11 show how the greedy algorithm performs with growing step sizes, i.e., more coarse grained latency assignments. Preliminary evaluations with both algorithms for *specific* cost functions indicated that with growing stepsizes, the ILP has an upward trend for the optimal costs as anticipated in Section 5. However, for the evaluations with multiple cost function and step sizes > 1, the ILP did not find a solution within a reasonable time (<1 hour per graph). As we focus on multi-model graphs, we ran the experiment with multiple cost functions again for the greedy algorithm only.

We see in the results hat the cost remains stable while the runtime decreases.

6.2.2 Discussion

For step size 1, both algorithms quickly find a solution. The greedy approach is faster than the ILP with only little worse cost. For a step size > 1, the ILP was not solvable for random cost functions in a reasonable time. With its lower runtime, the greedy approach is preferable, particularly, if the solution is executed frequently, or the step size is > 1 and cost functions differ. On the other hand, if the SPA is more cost-sensitive, and the step size is 1, the ILP solution is preferred.

6.3 Cost-function Calls

Each cost-function call induces latency and request-costs (c.f. Section 2.2). A small number of calls is important, particularly, in case of long runtimes of cost functions or fees per call, i.e. by a cloud provider. We therefore also evaluate the total cost-function calls of each solution. Figures 12 depicts in logarithmic scale how the cost-function calls increase with an increasing number of CUs. We see that the greedy approach continuously executes fewer cost-function calls than the ILP as anticipated in Section 5.2.2. If the cost-function calls are expensive — in time or in money — the greedy solutions is preferred. Enabling the caching mechanism (c.f. Section 5.2.2) can reduce the number of cost-function calls even further. If, however, the cost-functions are available locally and have short runtimes, i.e., little communication and processing overhead, the ILP provides the optimal result and is accordingly preferred.

6.4 Comparing our approach to a solution for homogeneous nodes

To see how our generic approach compares to a highly tailored one, we implemented the algorithm proposed by Lohrmann et al. [17]. It is tailored for a homogeneous infrastructure and solely uses parallelization as configuration mechanism to control latency. To determine the required degree of parallelism for each operator, the authors use queuing theory. In this section, we compare the performance of their solution to our greedy and ILP implementation. In Table 2 we compare their concepts and assumptions to ours.

To increase the precision, we modified the Lohrmanm algorithm in one point. The algorithm uses snapshots of arrival and service rates at each operator. From this snapshot, the original algorithm uses the measured c_A (covariation of arrivals) to compute the required parallelization degree (c.f. Equation 9). Yet, this value depends on the current parallelization degree at the moment of the snapshot. As the parallelization degree used in the equation changes throughout executing the algorithm, we use the more precise $c_{A(p)}$ that depends on the *new* degree. This modification makes the equation a cubic function that needs to be solved numerically which Lohrmann et al. claim to be future work.

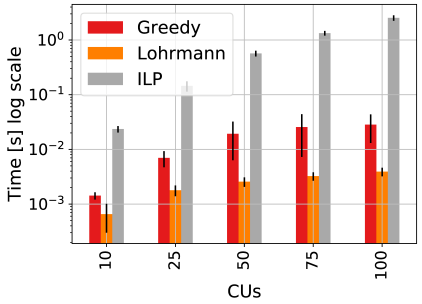

Figure 14. Runtime of all three solutions in log scale.

To compare the approaches, we need to match the cost function and the granularity of CUs. For each CU, we use the same cost function as Lohrmann's algorithm. Further, each CU of our architecture needs to represent exactly one operator, as Lohrmann's algorithm determines the parallelization degree operator-wise. Thus, in the comparison, the smallest unit of configuration a CU represents needs to have the same size. We, thus, need to limit our options drastically to match the strict assumptions of the Lohrmann algorithm.

Figures 13 and 14 show that Lohrmann's algorithm achieves almost the optimal cost compared to the ILP and outperforms both ILP and greedy in runtime. While the greedy algorithm is slightly slower than the Lohrmann algorithm, both finish even for 100 nodes within less than 0.1 seconds. Hence, we see this difference as insignificant. The reason for the better cost compared to our greedy solution is that Lohrmann's algorithm is closely tailored to SPAs on *homogeneous* nodes that influence latency via *parallelization* only, while we propose a highly abstract and generic solution. Modifying the parallelization degree stepwise, as Lohrmann's algorithm does, provides a more fine-grained control than controlling the latency value. However, controlling the parallelization degree poses those strict requirements on the infrastructure and configuration techniques that oppose our most important requirement — to inclusively configure a DHM-SPA on a heterogeneous infrastructure with multiple configuration techniques, different models to determine the configuration and no need for internal information from a CU such as the current parallelization degree.

6.5 Summary

We evaluate both solutions for runtime, cost, and cost-function calls. Both solve the latency-budget assignment problem in reasonable time. We see that in some cases, e.g., higher step sizes or expensive cost functions, the greedy algorithm outperforms the ILP. However, the ILP is preferable, e.g., when optimality is more important than runtime and cost-function calls are fast and cheap. We show that the greedy algorithm smoothly handles CU graphs that combine different cost functions. The ILP enables these combinations for step size one. We further saw that a less generic approach

Table 2. Comparing our approach to the alternative for homogeneous, parallelizable SPAs.

Property	Greedy	Lohrmann
Principle	Start with feasible assignment and make it cost optimal	Start with cost optimal assignment and make it feasible
Parameter to assign	Latency budget per CU	Parallelization degree per node
Greedy step size	Fixed	Until no more improvement
Configuration techniques	Any	Parallelization
Requirements	Cost function for each CU	Homogeneous infrastructure, arrival- and service rates from *global snapshot*. Independence of arrival- and processing times per item.
Level of detail	Summarize multiple operators into one CU if necessary (e.g., black box provider)	Individual configuration of each node.

can outperform our algorithms but poses much higher restrictions on the setup — thus being hardly applicable for DHM-SPAs.

7 Related Work

This section introduces autonomic control strategies to control latency of SPAs and continues with solutions to control cost and QoS across distributed infrastructures. Approaches that support autonomic latency compliance of SPAs usually provide *one* model for the complete SPA. Cardellini et al. [4] present a hybrid solution where local components request configuration changes that a global component approves in a globally optimal manner. However, the approval process induces additional latencies for each adaptation while our solution enables fast, individual and yet globally optimal adaptations by the CUs. Mencagli et al. [20] propose a game-theory based, decentralized approach where the SPA's nodes negotiate their degree of parallelization and an incentive mechanism makes the nodes shift from their local optimum towards a more globally optimal configuration. Lohrmann et al. [17] propose a central component that uses queuing theory to find the best parallelization degree for each operator in an SPA for a bounded end-to-end latency at minimal cost. Their solution, however, requires homogeneous nodes. Liu et al. [16] propose a framework that implements autonomic adaptation with a MAPE-loop. It defines and updates parallelization degrees solely from profiled and monitored data. Later they extend their work [15] to automatic cloud deploying with a bin-packing based plan. Russo et al. [24] recently proposed a solution for elasticity for SPAs on heterogeneous infrastructures using machine learning. However, they, too, require control over the complete application. To control end-to-end cost and bottleneck free processing already before runtime, Mencagli et al [21] propose a static analysis tool SpinStream. Besides fission, i.e., parallelization, they also consider merging of multiple operators into one (fusion). In our survey [23], we provide a more complete overview over parallelization of SPAs. Other approaches are available that tackle QoS and cost management across multi-provider and heterogeneous platforms. Liu and Shen [14] manage data storage across multiple cloud providers. As we do, they minimize the overall cost across multiple providers with an ILP and a heuristic but focus on data storage as resource. Wang et al. target multi-provider hybrid cloud environments [25] that include private and public infrastructures. They aim at cost optimal, QoS aware task-scheduling. A cost optimal solution first fully utilizes the private infrastructure and adds public resources when necessary and handles workload spikes within a required response time. Hung et al. [10] propose a job scheduling framework for geo-distributed jobs that include edge, fog and cloud processing. They handle the heterogeneity of network capacity and processing power and deal with dynamics at runtime. Ultimately, they aim at minimizing the average job response time by heuristically estimating the remaining time for the geo-distributed job.

8 Conclusion

In this paper, we solve the challenge of cost minimally keeping an end-to-end latency bound for SPAs that are hosted on a distributed, heterogeneous, and multi-provider infrastructure. To this end, we propose an architecture and two solutions. For our architecture, we model an SPA as a graph of individually managed control units (CUs). We formulate the latency-distribution problem to divide the end-to-end latency bound into individual budgets and assign them to the CUs in a cost minimizing manner. In doing so, each CU can implement the latency-control mechanism that fits best to the CUs specific infrastructure. As solutions to the latency-distribution problem, we propose an ILP formulation and a greedy algorithm. Evaluations show the effectiveness of both solutions. Finally, we compared our greedy approach to an existing solution that is restricted to homogeneous infrastructures. With this work, we enable to control the end-to-end latency of SPAs by the integration of different control mechanisms into one globally but lightweight managed system. This enhances stream processing in the emerging fields of IoT, edge and fog computing. For future work, we plan to additionally consider energy consumption and privacy constraints in the optimization and a mechanism to learn cost functions at runtime.

References

[1] Arif Ahmed, HamidReza Arkian, Davaadorj Battulga, Ali J. Fahs, Mozhdeh Farhadi, Dimitrios Giouroukis, Adrien Gougeon, Felipe Oliveira Gutierrez, Guillaume Pierre, Paulo R. Souza Jr, Mulugeta Ayalew Tamiru, and Li Wu. 2019. Fog Computing Applications: Taxonomy and Requirements. arXiv:cs.DC/1907.11621

[2] Iman Azimi, Arman Anzanpour, Amir M. Rahmani, Tapio Pahikkala, Marco Levorato, Pasi Liljeberg, and Nikil Dutt. 2017. HiCH: Hierarchical Fog-Assisted Computing Architecture for Healthcare IoT. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 174 (Sept. 2017), 20 pages. https://doi.org/10.1145/3126501

[3] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. 2013. Adaptive Input Admission and Management for Parallel Stream Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2488222.2488258

[4] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic Data Stream Processing. *Future Generation Computer Systems* 87 (Oct. 2018), 171–185. https://doi.org/10.1016/j.future.2018.05.025

[5] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2851141.2851148

[6] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software* 127 (2017), 302 – 319. https://doi.org/10.1016/j.jss.2016.08.037

[7] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proc. of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2611286.2611294

[8] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[9] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. 2016. Elastic Stream Processing for the Internet of Things. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 100–107. https://doi.org/10.1109/CLOUD.2016.0023

[10] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-area Analytics with Multiple Resources. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 12, 16 pages. https://doi.org/10.1145/3190508.3190528

[11] J. F. C. Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57, 4, 902–904. https://doi.org/10.1017/S0305004100036094

[12] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. 2018. P4CEP: Towards In-Network Complex Event Processing. In *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute@SIGCOMM 2018, Budapest, Hungary, August 20, 2018.*

[13] R. K. Kombi, N. Lumineau, and P. Lamarre. 2017. A Preventive Auto-Parallelization Approach for Elastic Stream Processing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS '17)*. 1532–1542. https://doi.org/10.1109/ICDCS.2017.253

[14] G. Liu and H. Shen. 2016. Minimum-Cost Cloud Storage Service Across Multiple Cloud Providers. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS '16)*. 129–138. https://doi.org/10.1109/ICDCS.2016.36

[15] Xunjun Liu and Rajkumar Buyya. 2019. Performance-Oriented Deployment of Streaming Applications on Cloud. *IEEE Transactions on Big Data* 5, 1 (March 2019), 46–59. https://doi.org/10.1109/TBDATA.2017.2720622

[16] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya. 2017. A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4, Article 24 (Nov. 2017), 33 pages. https://doi.org/10.1145/3132618

[17] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS '15)*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[18] Björn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele streaming: stream processing under QoS constraints at scale. *Cluster Computing* 17, 1 (mar 2014), 61–78. https://doi.org/10.1007/s10586-013-0281-8

[19] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal* 2, 4 (Aug 2015), 274–286. https://doi.org/10.1109/JIOT.2015.2397316

[20] Gabriele Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2, Article 13 (June 2016), 34 pages. https://doi.org/10.1145/2903146

[21] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. [n.d.]. SpinStreams: A Static Optimization Tool for Data Stream Processing Applications. In *Proceedings of the 19th International Middleware Conference* (2018) *(Middleware '18)*. ACM, 66–79. https://doi.org/10.1145/3274808.3274814 event-place: Rennes, France.

[22] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. 2012. A Federated Multi-cloud PaaS Infrastructure. In *2012 IEEE Fifth International Conference on Cloud Computing*. 392–399. https://doi.org/10.1109/CLOUD.2012.79

[23] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *Comput. Surveys* 52, 2, Article 36 (April 2019), 37 pages. https://doi.org/10.1145/3303849

[24] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources. In *Proc. of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/3328905.3329506

[25] Wei-Jen Wang, Yue-Shan Chang, Win-Tsung Lo, and Yi-Kang Lee. 2013. Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments. *The Journal of Supercomputing* 66, 2 (01 Nov 2013), 783–811. https://doi.org/10.1007/s11227-013-0890-2

[26] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. 2012. Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much?. In *Proc of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/2335484.2335515

[27] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky. 2014. Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing. In *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 325–329. https://doi.org/10.1109/CAMAD.2014.7033259

[28] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic Complex Event Processing Exploiting Prediction. In *Proc. of the 2015 IEEE International Conference on Big Data (BIG DATA '15)*. IEEE, Washington, DC, USA, 213–222. https://doi.org/10.1109/BigData.2015.7363758