An Experiment-Driven Performance Model of Stream Processing Operators in Fog Computing Environments

Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, Erik Elmroth

An Experiment-Driven Performance Model of Stream Processing Operators in Fog Computing Environments

HamidReza Arkian
Univ Rennes, Inria, CNRS, IRISA
hamidreza.arkian@irisa.fr

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
guillaume.pierre@irisa.fr

Johan Tordsson
Elastisys AB
johan.tordsson@elastisys.com

Erik Elmroth
Elastisys AB
erik.elmroth@elastisys.com

ABSTRACT

Data stream processing (DSP) is an interesting computation paradigm in geo-distributed infrastructures such as Fog computing because it allows one to decentralize the processing operations and move them close to the sources of data. However, any decomposition of DSP operators onto a geo-distributed environment with large and heterogeneous network latencies among its nodes can have significant impact on DSP performance. In this paper, we present a mathematical performance model for geo-distributed stream processing applications derived and validated by extensive experimental measurements. Using this model, we systematically investigate how different topological changes affect the performance of DSP applications running in a geo-distributed environment. In our experiments, the performance predictions derived from this model are correct within ±2% even in complex scenarios with heterogeneous network delays between every pair of nodes.

ACM Reference Format:
HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2019. An Experiment-Driven Performance Model of Stream Processing Operators in Fog Computing Environments. In *Proceedings of ACM SAC*. ACM, New York, NY, USA, 9 pages. https://doi.org/00000-00000

1 INTRODUCTION

Data stream processing is an attractive paradigm for analyzing real-time IoT-generated data in fog computing environments [7]. It combines a simple programming model with a distributed execution model that can be naturally mapped in geo-distributed environments. Although stream data processing engines were initially designed for powerful cluster environments, these properties motivate their increasing popularity in geo-distributed environments such as fog computing platforms [8, 22].

Understanding the performance of a geo-distributed stream processing application is a difficult challenge. Stream processing engines employ a variety of techniques and optimizations to decompose data processing as a potentially complex workflow of operators, each of which can possibly be distributed in multiple locations and connected with the rest of the system using heterogeneous networks [11]. Any configuration decision such as changing the replication factor or the placement of stream processing operators can have a significant impact on the resulting quality of service (QoS). Poor configuration choices may actually degrade performance compared to a basic single-site deployment [14, 21].

Numerous performance models have been proposed to capture the performance of stream processing engines in centralized [10, 14, 27] or geo-distributed [5, 9, 12] environments. These models are typically used to derive operator placement algorithms, and they are often evaluated with respect to the performance improvements provided by the placement strategy that derives from the model. In other terms, these works demonstrate that a proposed model enables better decisions than some chosen baseline, but they do not necessarily establish the accuracy of the model itself compared to the ground truth.

We propose a performance model for geo-distributed stream processing applications based on extensive experimental measurements, which allows us to explicitly assess the model's predictive accuracy rather than the quality of decisions derived from it. We first model the throughput performance of individual stream processing operators (*Map*, *Filter*, *Reduce*, etc.) with a varying number of operator replicas interconnected by networks with heterogeneous latencies. We then extend the model to take multiple data sources into account, and to support the *KeyBy* operator. After an initial calibration phase, our model can accurately predict the performance an application would experience if executing with a different replication and placement configuration of operators. In our experiments, the model delivers a predictive accuracy of ±2% even in complex scenarios with heterogeneous network performance between every pair of nodes. We show that performance models of individual operators may be easily composed with each other to capture the performance of simple workflows, and how to calibrate multiple models at minimum cost. We base our experiments on Apache Flink, but in principle the same model may be used with other stream processing engines.

This paper is organized as follows. Section 2 presents the background and related work. Section 3 discusses our methodology and experimental setup. Section 4 details our performance model. Finally, Section 5 evaluates the model and Section 6 concludes.

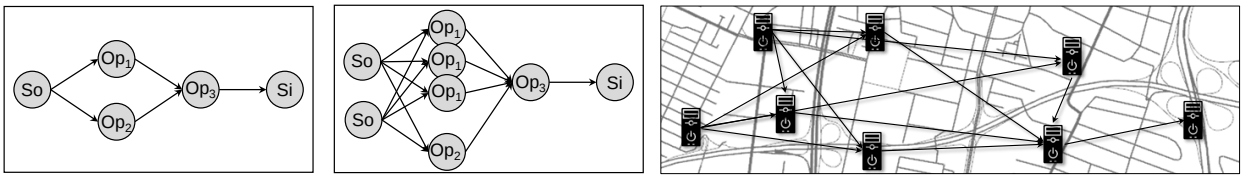
2 BACKGROUND

2.1 Stream processing in Fog

Stream processing was created to implement continuous data analytics tasks with low latency on unbounded input data streams [1]. Several stream processing engines (SPEs) have been proposed, including Apache Storm [25], Apache Spark [30] and Apache

ACM SAC, 2020, Brno, Czech Republic

ACM ISBN 99999-99999...$15.00
https://doi.org/00000-00000



(a) Logical graph of operators. (b) Replicated data source and operator. (c) Geo-distributed operator replicas in a Fog computing environment.

Figure 1: Three views of data processing workflows.

Flink [3], to execute stream processing applications in a scalable and efficient manner.

SPEs allow programmers to express applications as a workflow of data transformations (*operators*) which execute over unbounded data streams. Workflows are organized as a directed acyclic graph where vertices represent *operators* and edges represent *data streams*. SPEs also introduce a variety of *stateless* or *stateful* operators to transform one or more input streams in one or more output streams. Stateless operators such as Map, Reduce, Filter and KeyBy produce output only based on individual input records: Map applies a user-provided function to every element in the stream; Reduce combines the elements of a keyed stream together; Filter evaluates each element with a user-provided predicate and outputs only those that satisfy the predicate; KeyBy logically splits a stream into disjoint partitions. In contrast, stateful operators produce their output from a sequence of inputs, and potentially maintain state to do so [24]. In this paper, we consider only stateless operators.

SPEs use diverse parallelization mechanisms for the execution of operators on available resources, such as pipelining, multiple threads of execution per operator (i.e., replication), and operator-parallel execution (i.e., distribution) [23].

We base our work on Apache Flink, but in principle it may apply to other SPEs as well. Apache Flink's execution model contains two types of processes: a master termed as *JobManager* (JM) and a number of workers called *TaskManagers* (TMs). The parallelism of Flink applications is determined by the degree of parallelism of streams and operators. Streams can be divided into logical *stream partitions* whereas operators can be split into *subtasks*. TaskManagers can execute subtasks over stream partitions independently from one another.

Stream processing has been well studied in the domain of Cloud computing [2]. However, stream processing engines, despite their technology evolution, still cannot handle all requirements of Fog computing scenarios. Indeed, they are designed to run on centralized clusters that are far from the Fog environment which comprises widely distributed fog nodes with heterogeneous inter-node network latencies. Figure 1 shows the difference between logical view of *workflows*, parallel execution of their *operators* and distribution of them in a Fog environment.

2.2 State of the art

Stream processing performance has been widely investigated under different assumptions and optimization goals, and many optimizations have been proposed for generic SPEs [11] or specific ones such as Apache Storm [15, 17] and Apache Spark [14, 26, 27].

Performance modeling of stream processing engines in cloud environments may be used to provide reliable estimates of the dataflow performance and resource utilization that is required in streaming applications, and thereby to map the operators on the available cloud resources [23]. However, cloud-related works do not take geo-distribution into account and therefore cannot be directly applied to fog computing environments.

A number of SPE schedulers aim to improve performance in geo-distributed environments using heuristics [4, 12], deep reinforcement learning [16] or static analysis [18]. Others use performance models to decide which operators should be placed at the edge or in a central cloud [9]. However, these systems do not try to *predict* the performance of stream processing applications in a wide range of possible configurations.

Cardellini et al. present a general formulation of geo-distributed stream processing replication and placement as an integer linear programming problem [5, 6]. Another work targets network usage minimization and propose a heuristic that models the applications as a system of springs where operators are bodies tied together by springs and the stream data-rate and network latency determine the stretching of the springs [20]. The network usage is indirectly minimized by finding the assignment that minimizes the overall elastic energy of this equivalent system. More recently, several model-based optimization heuristics additionally considered the heterogeneity of computing and networking resources [19].

These approaches are useful as they estimate the behavior and evaluate the performance implications of optimization techniques in geo-distributed environments. However, they have not been evaluated in a real geo-distributed environment, and their results are not based on experimental observations. In addition, the proposed models are commonly only intended for a specific optimization problem (e.g., operator placement). Therefore, there is a lack of systematic investigation on how different topological changes affect the performance of distributed stream processing engines and a verifiable (i.e., experimental) knowledge on how stream processing applications will perform in a geo-distributed Fog environment.

Note that we do not consider our model as a competitor of previously-proposed models which focus on the global behavior of an entire stream-processing application [5, 9, 12]. Rather we see it as a validated model of individual stream processing operators that may be easily integrated in the global models.

3 METHODOLOGY

This work is driven by experimental evaluations that allow us to derive a model from empirical observations, and to validate its accuracy against actual performance measurements.

We follow an iterative methodology to design a model that closely matches the empirical performance measurements of real stream processing systems. We start with a simple model capable of capturing stream processing performance in a simple situation. We then iteratively make the execution scenarios more complex,

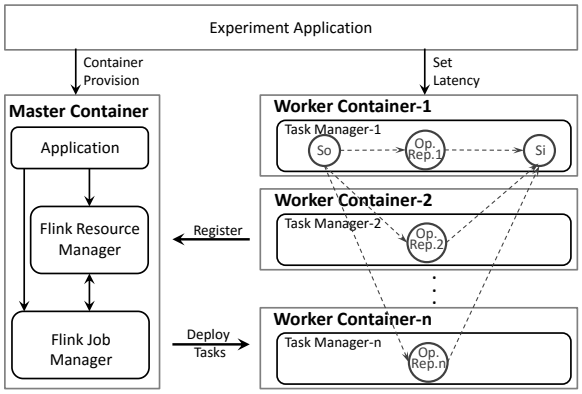


Figure 2: Experimental architecture.

criticize the model's accuracy, and refine the model to maintain the model's predictive power in increasingly complex situations.

3.1 Experimental environment

We conduct evaluations using a Dell PowerEdge R430 server equipped with two 8-core Intel Xeon E5-2620 v4 processors, providing 32 logical cores through hyper-threading, 64 GB of memory, and gigabit network connection. We use Apache Flink 1.7.0.

We deploy variable numbers of containers using Docker, where each container represents a separate node in a fog computing platform. Each emulated fog node executes a single TaskManager, and each TaskManager is configured with a single TaskSlot so it can execute only a single stream processing operator. We assume that the data sources and sinks are not performance bottlenecks. As shown in Figure 2, when deploying a stream processing workflow in such an infrastructure, Flink co-locates the data sources on the same TaskManager as the first operator in the workflow, and the data sinks on the same TaskManager as the last operator in the workflow.

We use the Linux `tc` (*"traffic control"*) command to emulate heterogeneous network latencies between the fog nodes. To produce realistic network latencies, we use a matrix of measured pairwise latencies between 16 European capital cities [29]. Figure 3 shows the selected cities and some examples of network latencies between them. When evaluating a system with n distributed task managers we sort cities by alphabetical order and reproduce the latencies between the first n cities.

In addition, we make the following assumptions:

- The fog nodes are geo-distributed, and the network latencies between them are heterogeneous. On the other hand, their individual processing capacities are identical.
- The processing times of the data sources and sinks are negligible compared to the processing times of the operators.
- In setups with multiple geo-distributed data sources, we assume that all sources produce the same volume of input data.

3.2 Performance metrics

In stream processing systems, throughput and latency are the two typical performance metrics that represent the quality of a deployment [13]. In this work we focus on the system's throughput, defined as the capacity of the system to ingest and process incoming data (e.g., produced by IoT devices).

For every test we use a data generator to generate a stream of 100,000 `Tuple2` input records, which are then fed to the chosen

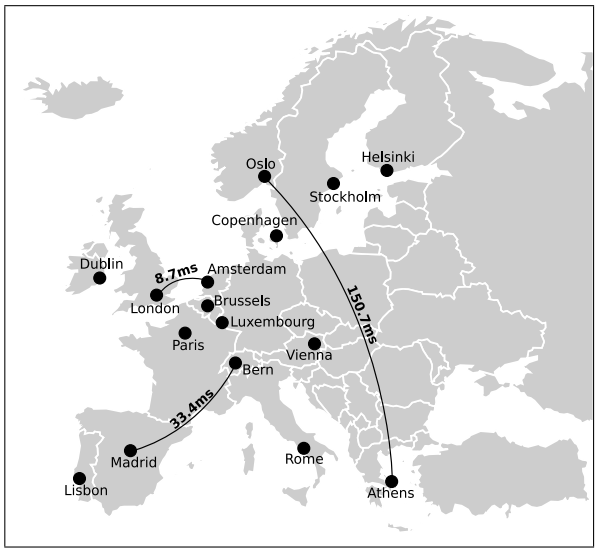


Figure 3: Selected European capital cities and some examples of network latencies between them.

operator and a data sink. To simulate the processing complexity of the operator's execution we use a simple call to the Fibonacci function $Fib(24)$. We run every test four times, and discard the results of the first run which is only used to warm-up the server's memory caches.

We evaluate system throughput by the time which is necessary to process this 100,000-record input. More precisely, we define two specific metrics which respectively capture the system's throughput at the operator and the workflow level.

Definition 1 — Processing Time (PT). We define the *Processing-Time* for each operator as the interval between the output of the first tuple from any instance of the previous operator in the workflow, and the output of the last tuple from the last instance of this operator. As illustrated in Figure 4(a), this means that we include the network latencies incurred by the data before reaching the concerned operator, but not the latencies incurred to reach the next operator in the workflow. When composing multiple operators together, this allows us to associates each inter-operator latency to a single operator.

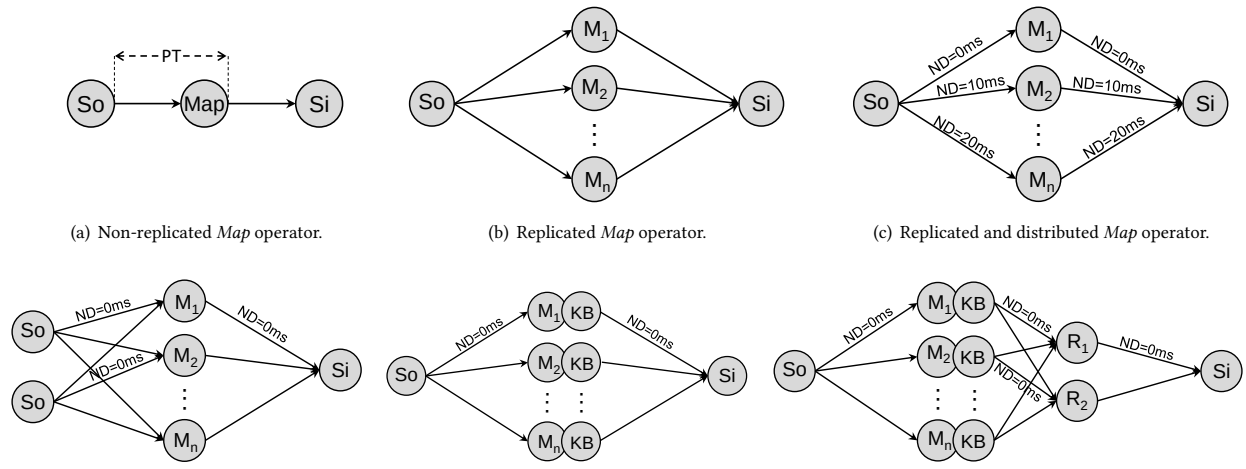
Definition 2 — Job Run Time (JRT). We define the *JobRunTime* of a workflow of operators as the interval between the input of the first tuple to the source operator of Flink, and the output of the last tuple from the sink operator.

4 PERFORMANCE MODEL DESIGN

We start by modeling a simple workflow which consists of one data source, one stream processing operator, and one data sink. Figure 4(a) represents this simple workflow. The operator initially executes on a single TaskManager (TM) (i.e., one fog node). We measure the time α for processing the entire input stream. This measure indicates the computation capacity of single machine. Table 1 shows the notations used in the performance model.

4.1 Modeling operator replication

If we decide to increase the number of TMs used to execute the stream processing operator as shown by Figure 4(b), the overall processing time should theoretically decrease proportionally to



(a) Non-replicated *Map* operator.
(b) Replicated *Map* operator.
(c) Replicated and distributed *Map* operator.
(d) Replicated and distributed *Map* operator with multiple data sources.
(e) Replicated and distributed *Map* operator followed by a *KeyBy* operator.
(f) Simple workflow with *Map*, *KeyBy*, and *Reduce* operators.

Figure 4: Different typologies that have been considered in the models.

Table 1: Notations used in the performance model.

Symbol	Description
Π_n	Processing Time of operator with n replicas.
α	Computation capacity of a single node.
β	SPE (Apache Flink) parallelization inefficiency.
γ	Effect of network delays.
ND_{max}	Maximum network delay between nodes.
$MAPE$	Mean absolute percentage error.
JRT	Overall JobRunTime of a workflow.

the number n of operator replicas. Equation 1 shows the initial version of our performance model:

$$\Pi_n = \frac{\alpha}{n} \tag{1}$$

However, when comparing this model with empirical performance measurements, we notice that the model does not offer an accurate representation of actual computation times. To make the model more accurate, we propose to introduce a parameter β which represents the overhead experienced by Flink when parallelizing execution. The second version of the model is thus:

$$\Pi_n = \frac{\alpha}{n^{\beta}} \tag{2}$$

where Π is the overall processing time of the selected operator, α is computation capacity of one single node, n is the number of replicas, and β represents the observed parallelization overhead. When fitting values α and β to the measured execution times, the model accurately predicts the performance of the SPE operator with any number of replicas (with a coefficient of determination $R^2 = 0.997$), as illustrated in Figure 5. Typical values for β in our experiments are $\beta \in [0.8, 0.9]$.

4.2 Modeling heterogeneous network delays

The model from Equation 2 works well for situations where all *TMs* run in a single cluster environment where communication performance between servers is uniform. However, in a fog computing environment we must expect to experience high

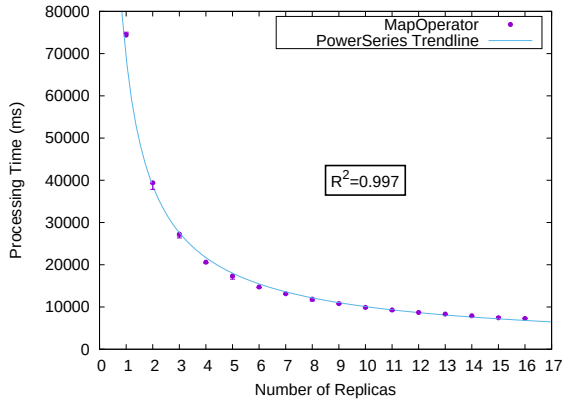


Figure 5: Effect of operator replication on processing time.

and heterogeneous network latencies between the servers. An example of such a topology is shown in Figure 4(c).

When we impose realistic network latencies between every pair of nodes, we observe that these latencies have an important effect on the overall system's performance, as illustrated in Figure 6. As the performance model from Equation 2 does not take network latencies into account, it performs poorly in this scenario.

To refine the model, we study the direct effect of network delay between two *TMs* (i.e., between the data source and one replica) on processing time. After fitting of the experimental results, we propose a linear model to represent the effects of network delay on the processing time in a system with two TMs:

$$\Pi_2 = a \times ND + b \tag{3}$$

where ND is the network delay between the two TMs, and Π_2 is processing time of the operator with two replicas. Also, a and b are two constants in the regression. Figure 6 shows the effect of different network delays between two *TMs*. Figure 7 shows the evolution of the Processing Time when varying both the number of replicas and the network delay between the data source and

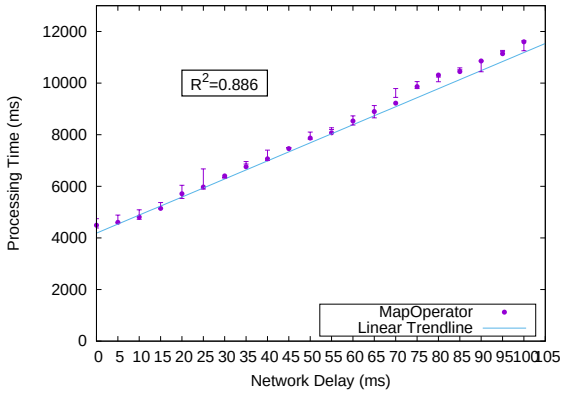


Figure 6: Effect of network delay on processing time.

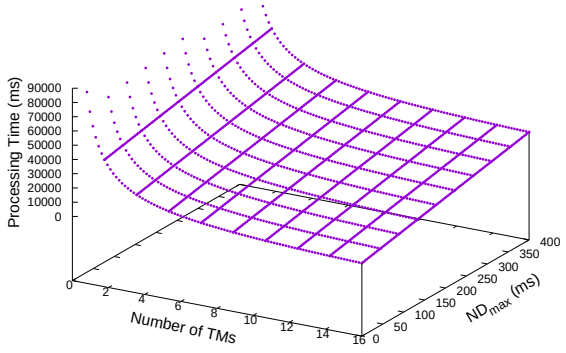


Figure 7: Effect of combination of number of replicas and network delay changes on processing time.

the operator replicas, in the situation of homogeneous network latencies between all the nodes.

When the network delays between every pair of nodes are heterogeneous, we observed that the dominating factor is the greatest latency between the data source and any of the *TMs*. As illustrated in Figure 8, the reason for this behavior is that the overall processing time is determined by the slowest of all operator replicas, namely the one which experiences the greatest network latency. We therefore propose an updated version of the performance model as shown in Equation 4:

$$\Pi_n = \frac{\alpha}{n^{\beta}} + \gamma \times ND_{max} \tag{4}$$

where ND_{max} is the greatest observed network delay between the data source and any of the operator's *TMs*, and γ is a parameter which represents the impact of network delay on overall system performance. This updated model enables us to accurately estimate the changes of overall processing time of one specific operator when the number of replicas of that operator and/or network delays between the replicas and source will change. Typical values for γ in our experiments are $\gamma \in [50, 150]$.

4.3 Modeling multiple data sources

So far, we assumed that all data to be processed by the stream processing operator originated from a single source. However, in many situations the sources of data may be distributed, for instance in the case where the modeled operator receives its input from another replicated stream processing operator. Figure 4(d) shows an example of this scenario.

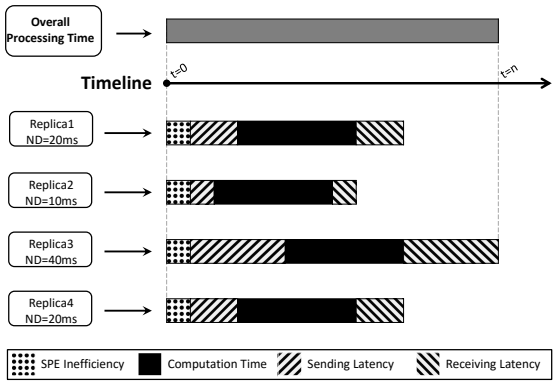


Figure 8: Influence of heterogeneous network latencies on processing time.

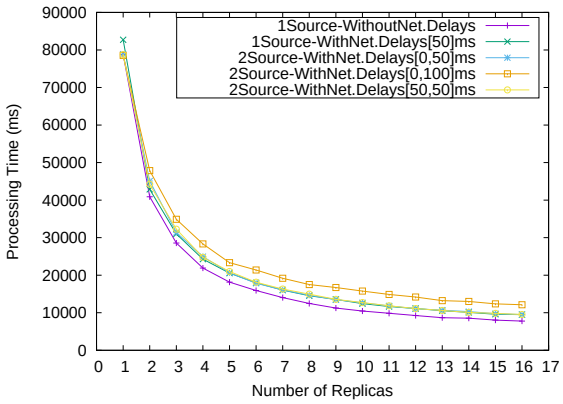


Figure 9: Effect of multiple sources on processing time.

As illustrated in Figure 9, the distribution of data sources does affect the system performance. However, an interesting observation is that increasing the number of sources with different network delays do not change the general pattern. We can therefore keep the same model as in Equation 4, by simply redefining ND_{max} as the greatest network delay between *any of the data sources* and any of the operator's *TM*.

4.4 Modeling the KeyBy operator

The model presented so far delivers accurate performance predictions for a large number of stateless stream processing operators (e.g., Map, Reduce, Filter), as we discuss in the next section. However, another frequently-used operator named *KeyBy* works differently. *KeyBy* is used to logically split a stream into disjoint partitions. This is useful for example to implement the *shuffle* operation between a Map and a Reduce operator. One example workflow with *KeyBy* is presented in Figure 4(e).

We discovered that, although *KeyBy* is exposed to application developers in exactly the same way as the other operators, it is in fact not implemented in Apache Flink as a standalone operator. Instead, it executes as an additional filter which is applied to the output of the preceding operator. It is therefore not necessary to model *KeyBy* as a separate operator. Instead, its processing time can be included when calibrating the model parameters of its preceding operator.

Once the resulting model has been calibrated to take into account the specificities of each stream processing application, it

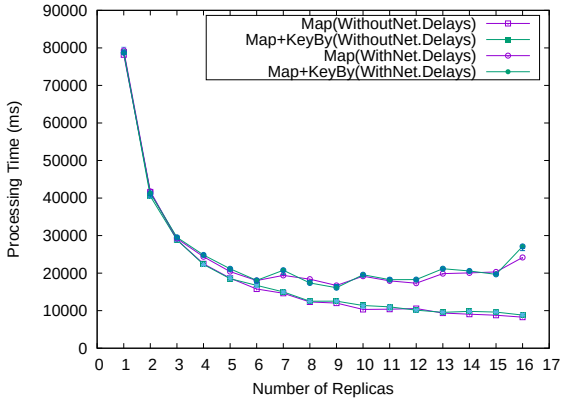


Figure 10: Effect of *KeyBy* on processing time.

can accurately predict the performance of the modeled application in a wide range of system deployment configurations. Figure 10 depicts the measured and modeled performance in two scenarios including the *KeyBy* operator (with and without heterogeneous network delays) while varying the number of *TMs*. Even in a complex scenario with network delays where every additional *TM* introduces a new instance of network latency, the model closely follows the actual measured performance.

4.5 Model calibration

To produce useful performance predictions, the performance model must be calibrated to match the characteristics of the application software as well as the underlying hardware. The model is fully parameterized with three parameters α, β and γ. To determine these three values in a unique manner, we normally need three experimental measurements gathered under different conditions. These measurements can be represented as a set of three equations with three unknown variables α, β and γ, which can then be resolved to determine the model's parameters.

However, in real-life conditions, obtaining three measurements may require time and unnecessary efforts. For example, after starting a stream processing application for the first time, it would be useful to start modeling the system's performance (even with some level of inaccuracy) using a single measurement, before additional measurements become available. However, with less than three measurements, it is impossible to determine all three parameters α, β and γ. Conversely, in case more than three measurements are available, there is usually no set of three parameters that perfectly matches all the measured data.

If a single measurement is available. In this situation we can only fit a single model parameter to the experimental data. We therefore give default values $\beta = 1$ and $\gamma = 0$, and only fit the value of α which captures the most important property of the stream processing operator (its individual computation complexity). This essentially simplifies the model back to its initial version from Equation 1 as follows.

$$\Pi_{m_1} = \frac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max} \Rightarrow \begin{pmatrix} \alpha = ? \\ \beta = default \rightarrow 1 \\ \gamma = default \rightarrow 0 \end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n} \quad (5)$$

The model does not capture complex scenarios such as heterogeneous network latencies, but it delivers reasonably good performance predictions for deployments with various numbers of *TMs*.

If two measurements are available. In this situation we can fit two parameters: either α and β, or α and γ. The remaining parameter simply keeps its default value. In our experiments we found that fitting α and γ gave slightly better results. Hence, we can change the model as follows:

$$\begin{Bmatrix} \Pi_{m_1} = \frac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max} \\ \Pi_{m_2} = \frac{\alpha}{{m_2}^{\beta}} + \gamma \times ND_{max} \end{Bmatrix} \Rightarrow \begin{pmatrix} \alpha = ? \\ \beta = default \rightarrow 1 \\ \gamma = ? \end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n} + \gamma \times ND_{max} \quad (6)$$

If three or more measurements are available. By having three measurements we can have three equations and the values of all three parameters consequently. Now we can use our completes model with all three parameters as follows.

$$\begin{Bmatrix} \Pi_{m_1} = \frac{\alpha}{{m_1}^{\beta}} + \gamma \times ND_{max} \\ \Pi_{m_2} = \frac{\alpha}{{m_2}^{\beta}} + \gamma \times ND_{max} \\ \Pi_{m_3} = \frac{\alpha}{{m_3}^{\beta}} + \gamma \times ND_{max} \end{Bmatrix} \Rightarrow \begin{pmatrix} \alpha = ? \\ \beta = ? \\ \gamma = ? \end{pmatrix} \Rightarrow \Pi_n = \frac{\alpha}{n^{\beta}} + \gamma \times ND_{max} \quad (7)$$

We make use of the non-linear least-square Levenberg-Marquardt algorithm [28] for identifying the set of values for α, β and γ which minimize the mean square error between the model and the measured data.

5 EVALUATION

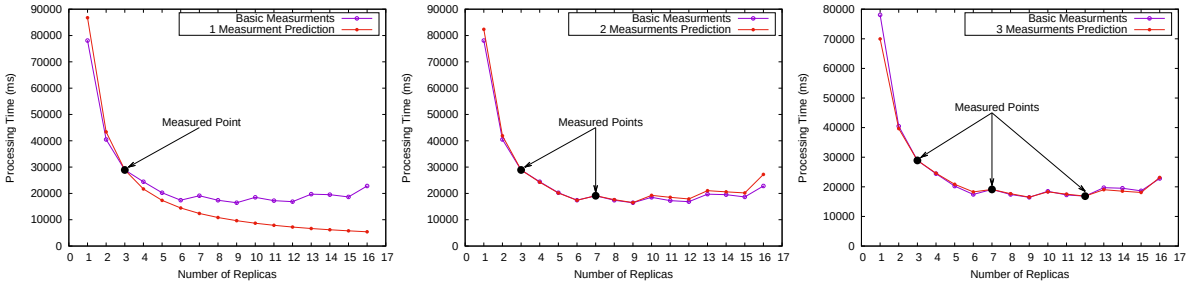
To evaluate the accuracy of this model we measured the actual performance over a large number of data points covering configurations with 1 to 16 *TMs* using heterogeneous network latencies between the nodes. We can thus compare the predictions issued by a model calibrated using a small number of these measurements, and the corresponding measured value. We evaluate the quality of the model's predictions by evaluating the *Mean Absolute Percentage Error* (MAPE) metric against the full set of measured performance values (where lower MAPE values indicate better performance):

$$MAPE_m = \frac{100}{n} \sum_{i=1}^{n} \frac{\left| \Pi_i^{actual} - \Pi_i^{predicted} \right|}{\Pi_i^{actual}} \quad (8)$$

5.1 Prediction accuracy

We first consider a model calibrated using a single measurement, and evaluate its MAPE while varying the number of *TMs*. We use only the Map operator here, and note that other stateless operators produce extremely similar results. Figure 11(a) shows that this simple model follows the general trend but fails to accurately capture the finer performance characteristics of the system. Its accuracy is $MAPE_{m_1} = 41.3\%$.

When using a model calibrated using two measurements the predictive power improves dramatically. Figure 11(b) shows that the model not only predicts the general trend much more accurately, but it also accurately predicts the variations that result



(a) With one measurement. (b) With two measurements. (c) With three measurements.

Figure 11: Quality of prediction based on the number of measurements.

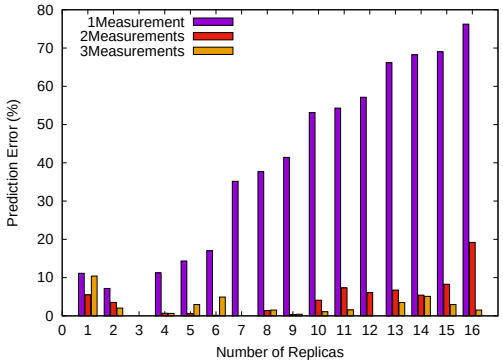


Figure 12: Prediction errors vs. the number of *TMs*.

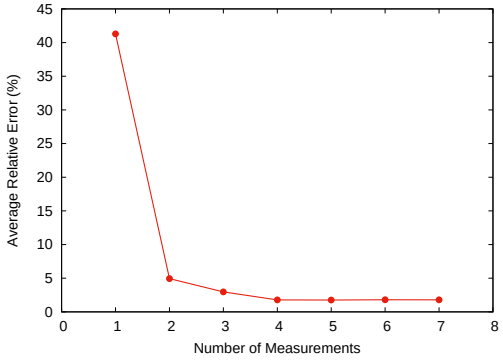


Figure 13: MAPE vs. the number of measurements.

from the fact that adding more *TMs* also adds new network latency values, which creates the fluctuations observed in the figure. This model has an accuracy $MAPE_{m_2} = 4.9\%$.

With three measurements, the error decreases further as all three parameters can be calibrated. In Figure 11(c) we can see that the model is extremely accurate, with $MAPE_{m_3} = 3.0\%$.

Figure 12 shows the prediction error of the three model versions for different numbers of *TMs*. We see that, with a single measurement based on three *TMs*, the model delivers predictions with less than 20% error for numbers of *TMs* close to the measured configuration (between 1 and 6 *TMs*). For greater numbers of *TMs*, the error grows up to 80% inaccuracy. The models based on two and three measurements are much accurate across the full range of numbers of *TMs*.

Figure 13 shows the MAPE metric for models based on different numbers of measurements. The model based on a single measurement exhibits an average error of 41%. Although this first model is fairly imprecise, it may already start delivering useful insights until additional measurements are available. With more measurements, the model becomes increasingly accurate. Four measurements yield an average inaccuracy of only 2%, after which additional measured data points do not improve the accuracy further. This level of precision is largely sufficient to take informed decisions about the future performance of the system in a wide range of potential situations.

5.2 Model Composition

Most stream processing applications are composed of more than a single operator. For such applications, it is necessary to build a separate model for each operator, and to compose multiple models together. We now show the feasibility of such composition.

Figure 4(f) depicts a simple workflow composed of three operators: *Map*, *KeyBy* and *Reduce*, which together implement the well-known MapReduce computation paradigm. The three operators are organized as a pipeline so intuitively the throughput of the entire pipeline should be determined by the operator with the highest Processing Time. Since the performance of *KeyBy* is integrated in that of the *Map* operator (as discussed in Section 4.4), we expect to compose the models of the *Map+KeyBy* and *Reduce* operators as follows:

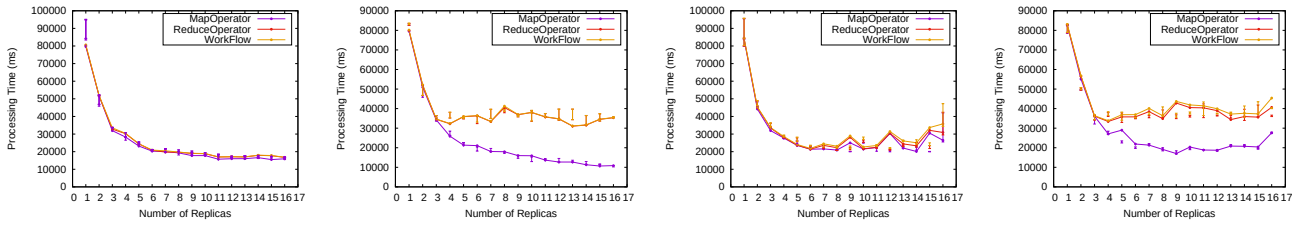
$$\Pi^{Workflow} = max\left(\Pi^{Map+KeyBy}, \Pi^{Reduce}\right) \quad (9)$$

Figures 14(a), 14(b), 14(c) and 14(d) show the *JRT* of the full workflow as well as the *PT* of each of its operators when using the same or different numbers of *TMs* for the operators, with or without inter-node network latencies. We observe that in all cases the *JRT* indeed remains very close from the maximum of the two *PTs*. Although we defer the question of model composition for more complex workflows to further work, these results show the potential of using our operator models as building blocks for global workflow performance modeling.

5.3 Parameter transfer

When modeling multiple operators which belong to the same or to different workflows, the need for three empirical performance measurements per model may delay the time by which all these models can deliver reasonable accuracy. We therefore propose to *transfer* parameters from one model to another.

In our models, the only parameter that is specific to a single operator is α, which captures the computation complexity of



(a) Same number of *TMs* for both operators, no network latencies.

(b) Different number of *TMs* for both operators, no network latencies.

(c) Same number of *TMs* for both operators, with network latencies.

(d) Different number of *TMs* for both operators, with network latencies.

Figure 14: Execution time of the full workflow and each of its operators in different operating conditions.

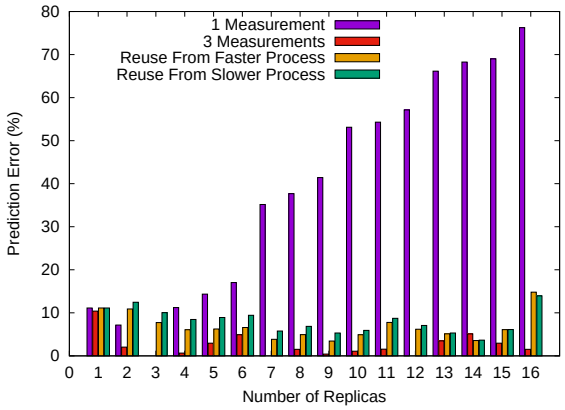


Figure 15: Prediction accuracy with parameter transfer.

the operator and the user-provided function it is configured to call. Reusing this parameter from one model to another (with a different computation complexity) would be highly unlikely to provide satisfactory accuracy. On the other hand, β and γ capture properties that are in principle independent from the nature of the computation carried out by the operator: β captures Flink's parallelization overhead, and γ captures the influence of network latency. This suggests that the values of β and γ that were calibrated to one operator might be reused for other operators using the same stream processing engine.

Figure 15 depicts the prediction errors of a model based on a single measurement to that of the same model where the β and γ values were transferred from another operator with a different computation complexity. We can see that the models with transferred parameters perform almost as well as a fully-calibrated model based on three actual measurements. This suggests that, after an initial value for β and γ has been calibrated for a first operator, the introduction of any new operator in the system may require only a single empirical measurement before we can build a first reasonably-accurate model for this operator.

6 CONCLUSION

Fog infrastructures allow the decentralization of data stream processing by moving the processing operators close to the data sources and/or the sinks. However, heterogeneous network characteristics make it difficult to understand the performance of stream processing engines in geo-distributed environments.

We presented a predictive performance model for Apache Flink operators that is backed by experimental measurements and evaluations. This model is very accurate with predictions ±2% of the actual values even in the presence of heterogeneous network latencies. Individual operator models can be composed together and, after the initial calibration of the first operator, a reasonably accurate model for other operators can be derived from a single measurement only.

We plan to extend this work to design operator placement algorithms which can guarantee a requested quality-of-service.

ACKNOWLEDGMENTS

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

REFERENCES

[1] H.C.M. Andrade et al. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics.* Cambridge University Press.
[2] L.F. Bittencourt et al. 2018. Scheduling in distributed systems: A cloud computing perspective. *Computer Science Review* 30 (2018).
[3] P. Carbone et al. 2015. Apache Flink : Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[4] V. Cardellini et al. 2015. Distributed QoS-aware scheduling in storm. In *Proc. DEBS'15.*
[5] V. Cardellini et al. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. *ACM SIGMETRICS Performance Evaluation Review* 44, 4 (2017).
[6] V. Cardellini et al. 2018. Decentralized self-adaptation for elastic Data Stream Processing. *Future Generation Computer Systems* 87 (2018).
[7] V. Cardellini et al. 2019. New Landscapes of the Data Stream Processing in the era of Fog Computing. *Future Generation Computer Systems* (2019).
[8] A. Da Silva Veith et al. 2018. Latency-Aware Placement of Data Stream Analytics on Edge Computing. In *Proc. ICSOC.*
[9] T. Elgamal et al. 2018. DROPLET: Distributed Operator Placement for IoT Applications Spanning Edge and Cloud Resources. In *Proc. IEEE CLOUD.*
[10] B. Gautam and A. Basava. 2019. Performance prediction of data streams on high-performance architecture. *Human-centric Computing and Information Sciences* 9, 2 (2019).
[11] M. Hirzel et al. 2014. A catalog of stream processing optimizations. *Comput. Surveys* 46, 4 (2014).
[12] Y. Huang et al. 2011. Operator Placement with QoS Constraints for Distributed Stream Processing. In *Proc. CNSM.*
[13] J. Karimov et al. 2018. Benchmarking Distributed Stream Processing Engines. In *Proc. ICDE.*
[14] J. Kroß and H. Krcmar. 2017. Model-Based Performance Evaluation of Batch and Stream Applications for Big Data. In *Proc. MASCOTS.*
[15] T. Li et al. 2016. Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing. *IEEE Trans. on Big Data* 2, 4 (2016).
[16] T. Li et al. 2018. Model-Free Control for Distributed Stream Data Processing using Deep Reinforcement Learning. In *Proc. VLDB Endow.*
[17] X. Liu and R. Buyya. 2019. Performance-Oriented Deployment of Streaming Applications on Cloud. *IEEE Trans. on Big Data* 5, 1 (2019).
[18] G. Mencagli et al. 2018. SpinStreams: a Static Optimization Tool for Data Stream Processing Applications. In *Proc. Middleware.*

[19] M. Nardelli et al. 2019. Efficient Operator Placement for Distributed Data Stream Processing Applications. *IEEE Trans. on Parallel and Distributed Systems* (2019).
[20] P. Pietzuch et al. 2006. Network-aware operator placement for stream processing systems. *Proc. ICDE*.
[21] H. Röger and R. Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *Comput. Surveys* 52, 2 (2019).
[22] E. Saurez et al. 2016. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proc. DEBS*.
[23] A. Shukla and Y. Simmhan. 2018. Model-driven scheduling for distributed stream processing systems. *J. Parallel and Distrib. Comput.* 117 (2018).
[24] Q. To et al. 2018. A Survey of State Management in Big Data Processing Systems. *The VLDB Journal* 27, 6 (2018).
[25] A. Toshniwal et al. 2014. Storm@Twitter. In *Proc. SIGMOD*.
[26] Shivaram V. et al. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proc. NSDI*.
[27] K. Wang and M.M.H. Khan. 2015. Performance Prediction for Apache Spark Platform. In *Proc. HPCC-CSS-ICESS '15*.
[28] Wikipedia. 2019. Levenberg–Marquardt algorithm. https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm.
[29] WonderNetwork. 2019. Global ping statistics. https://wondernetwork.com/pings.
[30] M. Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI*.